
Learn Multibody Dynamics

Jason K. Moore

Feb 20, 2024

TABLE OF CONTENTS

1	Introduction	3
1.1	What You Will Learn	3
1.2	Prerequisites	3
1.3	Purpose	3
1.4	Choice of dynamics formalism	4
1.5	Choice of programming language	4
1.6	History	4
1.7	Acknowledgements	5
1.8	Tools Behind the Book	5
2	License	7
3	Install the Software	15
3.1	1) Miniconda	15
3.2	2) Create and Activate an Environment	15
3.3	3) Install Packages	16
3.4	4) Open Jupyter Notebook	16
3.5	Software Versions	16
4	Jupyter and Python	19
4.1	Learning Objectives	19
4.2	Introduction	19
4.3	The Jupyter Notebook	19
4.4	Python	21
4.5	Learn More	28
5	Sympy	29
5.1	Learning Objectives	29
5.2	Introduction	29
5.3	Import and Setup	30
5.4	Symbols	30
5.5	Undefined Functions	31
5.6	Symbolic Expressions	32
5.7	Printing	35
5.8	Differentiating	36
5.9	Evaluating Symbolic Expressions	38
5.10	Matrices	41
5.11	Solving Linear Systems	45
5.12	Simplification	48
5.13	Learn more	51

6 Orientation of Reference Frames	53
6.1 Learning Objectives	53
6.2 Reference Frames	53
6.3 Unit Vectors	54
6.4 Simple Orientations	54
6.5 Direction Cosine Matrices	57
6.6 Successive Orientations	58
6.7 SymPy Mechanics	60
6.8 Euler Angles	64
6.9 Alternatives for Representing Orientation	69
6.10 Learn more	69
7 Vectors	71
7.1 Learning Objectives	71
7.2 What is a vector?	72
7.3 Vector Functions	73
7.4 Addition	74
7.5 Scaling	76
7.6 Dot Product	77
7.7 Cross Product	80
7.8 Vectors Expressed in Multiple Reference Frames	82
7.9 Relative Position Among Points	83
8 Vector Differentiation	89
8.1 Learning Objectives	89
8.2 Partial Derivatives	90
8.3 Product Rule	92
8.4 Second Derivatives	93
8.5 Vector Functions of Time	93
9 Angular Kinematics	95
9.1 Learning Objectives	95
9.2 Introduction	96
9.3 Angular Velocity	96
9.4 Angular Velocity of Simple Orientations	99
9.5 Body Fixed Orientations	101
9.6 Time Derivatives of Vectors	103
9.7 Addition of Angular Velocity	105
9.8 Angular Acceleration	106
9.9 Addition of Angular Acceleration	108
10 Translational Kinematics	111
10.1 Learning Objectives	111
10.2 Introduction	111
10.3 Translational Velocity	111
10.4 Velocity Two Point Theorem	116
10.5 Velocity One Point Theorem	117
10.6 Translational Acceleration	119
10.7 Acceleration Two Point Theorem	119
10.8 Acceleration One Point Theorem	120
11 Holonomic Constraints	123
11.1 Learning Objectives	123
11.2 Four-Bar Linkage	124
11.3 Solving Holonomic Constraints	128

11.4	General Holonomic Constraints	130
11.5	Generalized Coordinates	132
11.6	Calculating Additional Kinematic Quantities	134
12	Nonholonomic Constraints	137
12.1	Learning Objectives	137
12.2	Motion Constraints	138
12.3	Chaplygin Sleigh	139
12.4	Rolling Without Slip	142
12.5	Kinematical Differential Equations	143
12.6	Choosing Generalized Speeds	144
12.7	Snakeboard	148
12.8	Degrees of Freedom	153
13	Mass Distribution	155
13.1	Learning Objectives	155
13.2	Particles and Rigid Bodies	156
13.3	Mass	156
13.4	Mass Center	157
13.5	Distribution of Mass	158
13.6	Inertia Matrix	163
13.7	Dyadics	164
13.8	Properties of Dyadics	167
13.9	Inertia Dyadic	167
13.10	Parallel Axis Theorem	170
13.11	Principal Axes and Moments of Inertia	171
13.12	Angular Momentum	173
14	Force, Moment, and Torque	175
14.1	Learning Objectives	175
14.2	Force	176
14.3	Bound and Free Vectors	176
14.4	Moment	176
14.5	Couple	178
14.6	Equivalence & Replacement	178
14.7	Specifying Forces and Torques	180
14.8	Equal & Opposite	181
14.9	Contributing and Noncontributing Forces	182
14.10	Gravity	182
14.11	Springs & Dampers	183
14.12	Friction	184
14.13	Aerodynamic Drag	185
14.14	Collision	186
15	Generalized Forces	189
15.1	Learning Objectives	189
15.2	Introduction	190
15.3	Partial Velocities	191
15.4	Nonholonomic Partial Velocities	193
15.5	Generalized Active Forces	194
15.6	Generalized Active Forces on a Rigid Body	197
15.7	Nonholonomic Generalized Active Forces	199
15.8	Generalized Inertia Forces	200
15.9	Nonholonomic Generalized Inertia Forces	202

16 Unconstrained Equations of Motion	203
16.1 Learning Objectives	203
16.2 Dynamical Differential Equations	204
16.3 Body Fixed Newton-Euler Equations	204
16.4 Equations of Motion	207
16.5 Example of Kane's Equations	208
16.6 Implicit and Explicit Form	213
17 Simulation and Visualization	215
17.1 Learning Objectives	215
17.2 Numerical Integration	216
17.3 Numerical Evaluation	216
17.4 Simulation	221
17.5 Plotting Simulation Trajectories	226
17.6 Integration with SciPy	230
17.7 Animation with Matplotlib	234
18 Three Dimensional Visualization	239
18.1 pythreejs	244
18.2 Creating a Scene	244
18.3 Transformation Matrices	245
18.4 Geometry and Mesh Definitions	249
18.5 Scene Setup	250
18.6 Animation Setup	251
18.7 Animated Interactive 3D Visualization	252
19 Equations of Motion with Nonholonomic Constraints	253
19.1 Learning Objectives	253
19.2 Introduction	254
19.3 Snakeboard Equations of Motion	254
19.4 Simulate the Snakeboard	262
19.5 Animate the Snakeboard	265
19.6 Calculating Dependent Speeds	268
20 Equations of Motion with Holonomic Constraints	271
20.1 Learning Objectives	271
20.2 Introduction	272
20.3 Four-bar Linkage Equations of Motion	272
20.4 Simulate without constraint enforcement	278
20.5 Animate the Motion	284
20.6 Correct Dependent Coordinates	286
20.7 Simulate Using a DAE Solver	288
21 Exposing Noncontributing Forces	293
21.1 Learning Objectives	293
21.2 Introduction	294
21.3 Double Pendulum Example	294
21.4 Apply Newton's Second Law Directly	296
21.5 Auxiliary Generalized Speeds	298
21.6 Auxiliary Generalized Active Forces	299
21.7 Auxiliary Generalized Inertia Forces	300
21.8 Augmented Dynamical Differential Equations	301
21.9 Compare Newton and Kane Results	302
22 Energy and Power	305

22.1	Learning Objectives	305
22.2	Introduction	306
22.3	Kinetic Energy	306
22.4	Potential Energy	306
22.5	Total Energy	307
22.6	Energetics of Jumping	307
22.7	Simulation Setup	313
22.8	Conservative Simulation	317
22.9	Conservative Simulation with Ground Spring	318
22.10	Nonconservative Simulation	319
22.11	Simulation with Passive Knee Torques	320
22.12	Simulation with Active Knee Torques	322
23	Equations of Motion with the Lagrange Method	325
23.1	Learning Objectives	325
23.2	Introduction	326
23.3	Inertial forces from kinetic energy	326
23.4	Conservative Forces	328
23.5	The Lagrange Method	328
23.6	Constrained equations of motion	331
23.7	Lagrange's vs Kane's	335
24	Unconstrained Equations of Motion with the TMT Method	337
24.1	Example Formulation	339
24.2	Create the TMT Components	342
24.3	Formulate the reduced equations of motion	344
24.4	Evaluate the equations of motion	346
25	Notation	347
25.1	General	347
25.2	Orientation of Reference Frames	347
25.3	Vectors and Vector Differentiation	348
25.4	Angular and Translational Kinematics	348
25.5	Constraints	349
25.6	Mass Distribution	349
25.7	Force, Moment, and Torque	349
25.8	Generalized Forces	350
25.9	Unconstrained Equations of Motion	350
25.10	Equations of Motion with Nonholonomic Constraints	351
25.11	Equations of Motion with Holonomic Constraints	351
25.12	Energy and Power	351
25.13	Lagrange's method	352
26	References	353
27	Prior Versions	355
28	Lecture Videos	357
Bibliography		359

Last Updated: Feb 20, 2024

Version: 0.2.dev0+d86f82e

This online book aims to teach multibody dynamics using interactive code woven into the text. It follows the organization and methods presented in [Kane1985] and can be thought of a retelling of many topics in the book. Each page can be downloaded as a [Python](#) script or [Jupyter](#) Notebook. The book is also [available in PDF format](#). This book is used primarily as a companion resource for TU Delft's [Multibody Dynamics](#) course taught by [Jason K. Moore](#) but it is designed to standalone from the course.

INTRODUCTION

1.1 What You Will Learn

- How to formulate the equations of motion for a set of interacting rigid bodies, i.e. a multibody system.
- How to manage and incorporate kinematic constraints.
- How to simulate a multibody system.
- How to visualize the motion of a multibody system in 2D and 3D.
- How to interpret the behavior of multibody systems.

1.2 Prerequisites

- Linear algebra
- Vector calculus
- Calculus based physics
- Statics
- Dynamics
- Introductory numerical methods
- Introductory scientific computing

1.3 Purpose

The goal of this text is to help you learn multibody dynamics via a mixture of computation and traditional mathematical presentation. Most existing textbooks on the subject are either purely mathematical and problems are solved by pencil and paper or there are computational elements that are tacked on rather than integrated. I hope to weave the two much more fluidly here so that you can learn the principles of multibody dynamics through computing.

This text is less about teaching deep theory in multibody dynamics and more about application and doing. After following the text and practicing, you should be able to correctly model, simulate, and visualize multibody dynamic systems so that you can use them as a tool to answer scientific questions and solve engineering problems.

1.4 Choice of dynamics formalism

To teach multibody dynamics, one must choose a formalism for notation and deriving the equations of motion. There are numerous methods for doing so, from Newton and Euler's to Lagrange and Hamilton's to Jain and Featherstone's. Here I use an approach primarily derived from Thomas R. Kane and David Levinson in their 1985 book "Dynamics, Theory and Application" [Kane1985]. The notation offers a precise way to track all of the nuances in multibody dynamics bookkeeping and a realization of the equations of motion that obviates having to introduce virtual motion concepts and that handles kinematic constraints without the need of Lagrange multipliers.

1.5 Choice of programming language

With the goal of teaching through computation, it means I need to also choose a programming language. There are many programming languages well suited to multibody dynamics computation, but I select Python for several reasons: 1) Python is open source and freely available for use, 2) Python is currently one, if not the, most popular programming language in the world, 3) the scientific libraries available in Python are voluminous and widely used in academia and industry, and 4) Python has SymPy which provides a foundation for computer aided-algebra and calculus.

1.6 History

The primary presentation of multibody dynamics in this text is based on the presentation I and my fellow graduate students received in the graduate Multibody Dynamics course taught by Mont Hubbard and the late Fidelis O. Eke at the University of California, Davis in the early 2000's. Profs. Hubbard and Eke taught the course from the late 80s or early 90s until they retired in 2013 (Prof. Hubbard) and 2016 (Prof. Eke). The 10-week course was based on Thomas R. Kane's and David A. Levinson's 1985 book "Dynamics, Theory and Application" and followed the book and companion computational materials closely. Prof. Eke was a PhD student of Thomas R. Kane at Stanford and Prof. Hubbard adopted Prof. Kane's approach to dynamics after moving to UC Davis from Stanford¹. I helped with Prof. Eke's 2015 course and taught the course in 2017 and 2019 at UC Davis and this text is a continuation of the notes and materials I developed based on Profs. Hubbard and Eke's notes and materials which now includes some elements of TU Delft's past multibody dynamics course.

When I took the UC Davis course in 2006 as a graduate student, I naively decided to derive and analyze the nonlinear and linear Carvallo-Whipple bicycle model [Meijaard2007] as my course project². Fortunately, another student visiting from Aachen University, Thomas Engelhardt, also choose the same model and his success finally helped me squash the bugs in my formulation. Luke Peterson, Gilbert Gede, and Angadh Nanjungud subsequently joined Hubbard and Eke's labs and with Luke's lead we were sucked into the world of open source scientific software. At that time, Python's use by scientists and engineers began to gain traction and we fortunately jumped on the bandwagon. We had become quite frustrated with the black box approach of the commercial software tools most engineers used at that time, this included the tool Autolev that was developed by Kane's collaborators for the automation of multibody dynamics modeling. To remedy this frustration, Luke wrote the first version of PyDy as a Google Summer of Code participant in 2009. Gilbert followed him by implementing a new version as SymPy Mechanics in 2011 also as a Google Summer of Code participant. We use Gilbert's, now modified and extended, implementation in this text. Combined with the power of SymPy and Jupyter Notebooks (IPython Notebooks back then), SymPy Mechanics provides a computational tool that is especially helpful for learning and teaching multibody dynamics. It is also equally useful for advanced modeling in research and industry.

I have stewarded and developed the software as well as taught and researched with it over the last decade with the help of a long list of contributors. This text is a presentation of the methods and lessons learned from over the years of doing multibody dynamics with open source Python software tools.

¹ The project is shared at <https://github.com/moorepants/MAE-223>

² Mont was working on a skateboard dynamics model in the late 70s and presented his model to an audience that included Thomas Kane. As the story goes, Prof. Kane approached Mont after the lecture to privately tell him his dynamics model was incorrect. Mont then took it upon himself to learn Kane's approach to dynamics so that his future models would be less likely to have such errors.

1.7 Acknowledgements

Wouter Wolfslag contributed the “Equations of Motion with the Lagrange Method” chapter, “Alternatives for Representing Orientation” section, and reviewed updates for version 0.2. Peter Stahlecker and Jan Heinen provided page-by-page review of the text while drafting version 0.1. Peter did the same for version 0.2. Arthur Ryman contributed edits to the first version. Their feedback has helped improve the text in many ways. We also thank the students of TU Delft’s Multibody Dynamics course who test the materials while learning.

These are the primary contributors to the SymPy Mechanics software presented in the text, in approximate order of first contribution:

- Dr. Luke Peterson, 2009
- Dr. Gilbert Gede, 2011
- Dr. Angadh Nanjangud, 2012
- Tarun Gaba, 2013
- Oliver Lee, 2013
- Dr. Chris Dembia, 2013
- Jim Crist, 2014
- Sahil Shekhawat, 2015
- James McMillan, 2016
- Nikhil Pappu, 2018
- Sudeep Sidhu, 2020
- Abhinav Kamath, 2020
- Timo Stienstra, 2022
- Dr. Sam Brockie, 2023

SymPy Mechanics is built on top of SymPy, whose [1000+ contributors](#) have also greatly helped SymPy Mechanics be what it is. Furthermore, the software sits on the top of a large ecosystem of open source software written by thousands and thousands of contributors who we owe for the solid foundation.

1.8 Tools Behind the Book

I write the contents in plain text using the [reStructuredText](#) markup language for processing by [Sphinx](#). The mathematics are rendered with [MathJax](#) in the HTML version. I use the [Jupyter Sphinx](#) extension which executes the code in each chapter as if it were a Jupyter notebook and embeds the Jupyter generated outputs into the resulting HTML page. The extension also converts each chapter into a Python script and Jupyter notebook for download. I use the [Material Sphinx Theme](#) and [sphinx-togglebutton](#) for the dropdown information boxes. I host the source for the book on [Github](#), where I use Github Actions to build the website and push it to a Github Pages host using [ghp-import](#). I use Github’s issue tracker and pull request tools to manage tasks and changes. The figures are drawn with a Wacom One tablet and the [Xournal++](#) application.

CHAPTER

TWO

LICENSE

The text, figures, and code is licensed under the Creative Commons Attribution 4.0 License (CC-BY 4.0). If you reuse the materials under the terms of the license you will also need to include the following citation to this work:

Moore, J. K., "Learn Multibody Dynamics", 2022, <https://moorepants.github.io/learn-multibody-dynamics/>

Text, figures, and code have been incorporated from the following resources and their licenses are included below:

1. Moore, Dembia, Crist, Nwanna, Milam, Wang, Gaba, Gede, McMurry. PyDy "Human Standing Tutorial", 2014, <https://github.com/pydy/pydy-tutorial-human-standing> CC-BY 4.0.

Attribution 4.0 International

=====

Creative Commons Corporation ("Creative Commons") is not a law firm and does not provide legal services or legal advice. Distribution of Creative Commons public licenses does not create a lawyer-client or other relationship. Creative Commons makes its licenses and related information available on an "as-is" basis. Creative Commons gives no warranties regarding its licenses, any material licensed under their terms and conditions, or any related information. Creative Commons disclaims all liability for damages resulting from their use to the fullest extent possible.

Using Creative Commons Public Licenses

Creative Commons public licenses provide a standard set of terms and conditions that creators and other rights holders may use to share original works of authorship and other material subject to copyright and certain other rights specified in the public license below. The following considerations are for informational purposes only, are not exhaustive, and do not form part of our licenses.

Considerations for licensors: Our public licenses are intended for use by those authorized to give the public permission to use material in ways otherwise restricted by copyright and certain other rights. Our licenses are irrevocable. Licensors should read and understand the terms and conditions of the license they choose before applying it. Licensors should also secure all rights necessary before applying our licenses so that the public can reuse the material as expected. Licensors should clearly mark any material not subject to the license. This includes other CC-licensed material, or material used under an exception or limitation to copyright. More considerations for licensors:

(continues on next page)

(continued from previous page)

wiki.creativecommons.org/Considerations_for_licensors

Considerations for the public: By using one of our public licenses, a licensor grants the public permission to use the licensed material under specified terms and conditions. If the licensor's permission is not necessary for any reason--for example, because of any applicable exception or limitation to copyright--then that use is not regulated by the license. Our licenses grant only permissions under copyright and certain other rights that a licensor has authority to grant. Use of the licensed material may still be restricted for other reasons, including because others have copyright or other rights in the material. A licensor may make special requests, such as asking that all changes be marked or described. Although not required by our licenses, you are encouraged to respect those requests where reasonable. More considerations for the public:

wiki.creativecommons.org/Considerations_for_licensees

Creative Commons Attribution 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

Section 1 -- Definitions.

- a. Adapted Material means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.
- b. Adapter's License means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.
- c. Copyright and Similar Rights means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b) (1)–(2) are not Copyright and Similar Rights.

(continues on next page)

(continued from previous page)

- d. Effective Technological Measures means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.
- e. Exceptions and Limitations means fair use, fair dealing, and/or any other exception or limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.
- f. Licensed Material means the artistic or literary work, database, or other material to which the Licenser applied this Public License.
- g. Licensed Rights means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licenser has authority to license.
- h. Licenser means the individual(s) or entity(ies) granting rights under this Public License.
- i. Share means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.
- j. Sui Generis Database Rights means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.
- k. You means the individual or entity exercising the Licensed Rights under this Public License. Your has a corresponding meaning.

Section 2 -- Scope.

- a. License grant.
 - 1. Subject to the terms and conditions of this Public License, the Licenser hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:
 - a. reproduce and Share the Licensed Material, in whole or in part; and
 - b. produce, reproduce, and Share Adapted Material.
 - 2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with

(continues on next page)

(continued from previous page)

- its terms and conditions.
3. Term. The term of this Public License is specified in Section 6(a).
 4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures. For purposes of this Public License, simply making modifications authorized by this Section 2(a) (4) never produces Adapted Material.
 5. Downstream recipients.
 - a. Offer from the Licensor -- Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.
 - b. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.
 6. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).
- b. Other rights.
1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.
 2. Patent and trademark rights are not licensed under this Public License.
 3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly

(continues on next page)

(continued from previous page)

reserves any right to collect such royalties.

Section 3 -- License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

a. Attribution.

1. If You Share the Licensed Material (including in modified form), You must:
 - a. retain the following if it is supplied by the Licenser with the Licensed Material:
 - i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licenser (including by pseudonym if designated);
 - ii. a copyright notice;
 - iii. a notice that refers to this Public License;
 - iv. a notice that refers to the disclaimer of warranties;
 - v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;
 - b. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and
 - c. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.
2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.
3. If requested by the Licenser, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.
4. If You Share Adapted Material You produce, the Adapter's License You apply must not prevent recipients of the Adapted Material from complying with this Public License.

Section 4 -- Sui Generis Database Rights.

(continues on next page)

(continued from previous page)

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

- a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;
- b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and
- c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 -- Disclaimer of Warranties and Limitation of Liability.

- a. UNLESS OTHERWISE SEPARATELY UNDERTAKEN BY THE LICENSOR, TO THE EXTENT POSSIBLE, THE LICENSOR OFFERS THE LICENSED MATERIAL AS-IS AND AS-AVAILABLE, AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE LICENSED MATERIAL, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHER. THIS INCLUDES, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OR ABSENCE OF ERRORS, WHETHER OR NOT KNOWN OR DISCOVERABLE. WHERE DISCLAIMERS OF WARRANTIES ARE NOT ALLOWED IN FULL OR IN PART, THIS DISCLAIMER MAY NOT APPLY TO YOU.
- b. TO THE EXTENT POSSIBLE, IN NO EVENT WILL THE LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY (INCLUDING, WITHOUT LIMITATION, NEGLIGENCE) OR OTHERWISE FOR ANY DIRECT, SPECIAL, INDIRECT, INCIDENTAL, CONSEQUENTIAL, PUNITIVE, EXEMPLARY, OR OTHER LOSSES, COSTS, EXPENSES, OR DAMAGES ARISING OUT OF THIS PUBLIC LICENSE OR USE OF THE LICENSED MATERIAL, EVEN IF THE LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH LOSSES, COSTS, EXPENSES, OR DAMAGES. WHERE A LIMITATION OF LIABILITY IS NOT ALLOWED IN FULL OR IN PART, THIS LIMITATION MAY NOT APPLY TO YOU.
- c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 -- Term and Termination.

- a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.
- b. Where Your right to use the Licensed Material has terminated under

(continues on next page)

(continued from previous page)

Section 6(a), it reinstates:

1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or
2. upon express reinstatement by the Licensor.

For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

- c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.
- d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7 -- Other Terms and Conditions.

- a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.
- b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 -- Interpretation.

- a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully be made without permission under this Public License.
- b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.
- c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.
- d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

=====

(continues on next page)

(continued from previous page)

Creative Commons is not a party to its public licenses. Notwithstanding, Creative Commons may elect to apply one of its public licenses to material it publishes and in those instances will be considered the "Licensor." The text of the Creative Commons public licenses is dedicated to the public domain under the CC0 Public Domain Dedication. Except for the limited purpose of indicating that material is shared under a Creative Commons public license or as otherwise permitted by the Creative Commons policies published at creativecommons.org/policies, Creative Commons does not authorize the use of the trademark "Creative Commons" or any other trademark or logo of Creative Commons without its prior written consent including, without limitation, in connection with any unauthorized modifications to any of its public licenses or any other arrangements, understandings, or agreements concerning use of licensed material. For the avoidance of doubt, this paragraph does not form part of the public licenses.

Creative Commons may be contacted at creativecommons.org.

INSTALL THE SOFTWARE

If you would like to run the software on your own computer, follow these instructions which recommend the use of a [Conda](#)-based installation process. The process below describes setting up a new base environment with the needed software installed.

3.1 1) Miniconda

Miniconda is a stripped down version of Anaconda so that you can install only what you desire. If you already have Miniconda (or Anaconda) on your computer, you can skip this step or delete your prior Miniconda (or Anaconda) folder on your computer to uninstall it. Download Miniconda for your operating system:

<https://docs.conda.io/en/latest/miniconda.html>

Install as a user, not an administrator, when asked. This will install the package manager conda and configure your computer to use the Python installed with Miniconda when you open a terminal or command prompt.

3.2 2) Create and Activate an Environment

Open either the terminal (Linux/Mac) or the (Anaconda) command prompt (Windows) and type the following series of commands followed each by the <enter> key to execute the commands.

Create the environment with:

```
conda create -c conda-forge -n learn-multibody-dynamics python=3.10
```

The `-c conda-forge` flag installs the packages from [Conda Forge](#). Conda Forge is a community maintained collection of compatible software packages and offers a larger number of packages than the default configuration.

Now activate the environment:

```
conda activate learn-multibody-dynamics
```

3.3 3) Install Packages

Now you can install the packages that are required for executing the code in this book with this command:

```
conda install -c conda-forge ipympl ipython jupyter notebook matplotlib numpy  
→pythreejs "scikits.odes" scipy "sympy>=1.11"
```

3.4 4) Open Jupyter Notebook

To check that everything works, type the command to open Jupyter:

```
jupyter notebook
```

Jupyter should open in your web browser and you should be able to run the scripts and notebooks found on the other pages.

3.5 Software Versions

This website was built with the following software versions:

```
import IPython
```

```
IPython.__version__
```

```
'8.15.0'
```

```
import jupyter_sphinx
```

```
jupyter_sphinx.__version__
```

```
'0.5.3'
```

```
import matplotlib
```

```
matplotlib.__version__
```

```
'3.7.3'
```

```
import notebook
```

```
notebook.__version__
```

```
'7.0.7'
```

```
import numpy
```

```
numpy.__version__
```

```
'1.24.4'
```

```
import platform
```

```
platform.python_version()
```

```
'3.10.13'
```

```
import pythreejs._version
pythreejs._version.__version__
```

```
'2.4.2'
```

```
import pkg_resources
pkg_resources.get_distribution("scikits.odes").version
```

```
'2.7.0'
```

```
import scipy
scipy.__version__
```

```
'1.11.4'
```

```
import sphinx
sphinx.__version__
```

```
'7.2.6'
```

```
import sphinx_material
sphinx_material.__version__
```

```
'0.0.36'
```

```
import sphinx_togglebutton
sphinx_togglebutton.__version__
```

```
'0.3.2'
```

```
import sympy
sympy.__version__
```

```
'1.11.1'
```


JUPYTER AND PYTHON

Note: You can download this example as a Python script: `jupyter-python.py` or Jupyter notebook: `jupyter-python.ipynb`.

4.1 Learning Objectives

After completing this chapter readers will be able to:

- Run the Jupyter notebook software
- Use magic commands in the Jupyter notebook
- Create basic data types in Python
- Create and use Python functions
- Import Python modules

4.2 Introduction

The following is a brief introduction to Python and how to use Python from a Jupyter Notebook. There is much more to learn than what is covered here. This is just enough to get you started for the purposes of this book. You will need to seek out the many excellent learning materials online to learn more; some are provided at the end of this chapter.

4.3 The Jupyter Notebook

The [Jupyter Notebook](#) is an application that lets you execute code to as well as display text, mathematics, digital media, and HTML-CSS-Javascript-based outputs. The displayed elements can be embedded directly or generated by the executed code. This makes the Jupyter Notebook well suited for communicating content that is linked and driven by code. It allows you to edit the code interactively. Each page of this book is a Jupyter Notebook and can be downloaded and executed on your computer. Jupyter can execute code from many programming languages. Different kernels are used for each language and a notebook can, in general, only have a single kernel. This book will use the Python 3 kernel.

4.3.1 Using the Notebook

To start the Jupyter notebook application open a terminal (Linux/Mac) or command prompt (Windows), navigate to a desired working directory then type the following command:

```
jupyter notebook
```

A new window will open in your web browser where you can open an existing notebook or start a new one. Notebooks are organized with cells. You may have a code cell for inputting commands followed by its result cell which contains the output of the code. You may also have a text cell that contains static content written in [Markdown](#). Markdown allows you to incorporate simple formatting and even things like mathematical equations using [LaTeX](#) notation, e.g. a^2 displays as a^2 . The cell type can be changed using a Jupyter drop-down menu.

There is the menu bar above for navigating a notebook but you will find the following keyboard shortcuts helpful:

- Enter : Create a new line with in cell
- Shift + Enter : Execute cell and advance to next cell
- Ctrl + Enter : Execute cell in place (do not advance to the next cell)
- Press `esc` (command mode) then `h` to display keyboard shortcuts

At times you might run code that gets stuck in an infinite loop or you might simply want to clear all your workspace variables and start over. To solve each of these problems you can click on the menu:

Kernel → Interrupt

then

Kernel → Restart

4.3.2 Magic Commands

These are special commands that only work in a Jupyter notebook or an IPython session, as opposed to the normal Python interpreter. Magic commands are preceded by a `%` or `%%`. You can list available magic commands but using the magic command `lsmagic`.

```
%lsmagic
```

Available line magics:

```
%alias %alias_magic %autoawait %autocall %automagic %autosave %bookmark %cat
→ %cd %clear %code_wrap %colors %conda %config %connect_info %cp %debug
→ %dhist %dirs %doctest_mode %ed %edit %env %gui %hist %history
→ %killbgscripts %ldir %less %lf %lk %ll %load %load_ext %loadpy %logoff
→ %logon %logstart %logstop %ls %lsmagic %lx %macro %magic %man
→ %matplotlib %mkdir %more %mv %notebook %page %pastebin %pdb %pdef %pdoc
→ %pfile %pinfo %pinfo2 %pip %popd %pprint %precision %prun %psearch
→ %psource %pushd %pwd %pycat %pylab %qtconsole %quickref %recall %rehashx
→ %reload_ext %rep %rerun %reset %reset_selective %rm %rmdir %run %save %sc
→ %set_env %store %sx %system %tb %time %timeit %unalias %unload_ext %who
→ %who_ls %whos %xdel %xmode
```

Available cell magics:

```
%%! %%HTML %%SVG %%bash %%capture %%code_wrap %%debug %%file %%html %
→ %%javascript %%js %%latex %%markdown %%perl %%prun %%pypy %%python %%python2_
→ %%python3 %%ruby %%script %%sh %%svg %%sx %%system %%time %%timeit %
→ %%writefile
```

(continues on next page)

(continued from previous page)

```
Automagic is ON, % prefix IS NOT needed for line magics.
```

For example `%whos` will show the variables available in your namespace:

```
a = 5
%whos
```

Variable	Type	Data/Info
a	int	5

4.3.3 Need Help?

In case you're lost help isn't far. The following commands should provide assistance.

`?` displays an overview of the features available when typing in code cells in Jupyter notebooks (the cells are parsed by the IPython Python interpreter when using the Python kernel):

```
?
```

A quick reference for the special commands in Jupyter code cells can be viewed with:

```
%quickref
```

For details about any Python object in the namespace, append a `?` to the variable or function (without `()`). For example, help for the `round()` function can be found like so:

```
round?
```

4.4 Python

Python has become one of the world's most popular programming languages. It is open source, free to use, and well suited for scientific and engineering programming needs. The following gives a brief introduction to the basics of Python.

4.4.1 Basic Data Types

Python has core builtin data types. The `type()` function shows you the type of any Python object. For example, here are the types of some integers, floating point numbers, and strings:

```
a = 5
b = 5.0
c = float(5)
d = 'dee'
e = 'e'
f = 2+3j
g = True

type(a), type(b), type(c), type(d), type(e), type(f), type(g)
```

```
(int, float, float, str, str, complex, bool)
```

4.4.2 Data Structures

Python offers several builtin data structures for grouping and organizing objects. Lists, tuples, and dictionaries are the most commonly used.

Lists

A list is a versatile container that holds objects in the order given. Lists are typically used to group similar items but may contain heterogeneous data types.

```
empty_list = []

string_list = ['lions', 'tigers', 'bears', 'sharks', 'hamsters']

int_list = [0, 1, 2, 3, 4]

int_list2 = list(range(5,10))

list_from_variables = [a, b, c, d, e]

list_of_lists = [empty_list,
                 string_list,
                 list_from_variables,
                 int_list,
                 int_list2]
```

Each of these can be displayed:

```
empty_list
```

```
[]
```

```
string_list
```

```
['lions', 'tigers', 'bears', 'sharks', 'hamsters']
```

```
int_list
```

```
[0, 1, 2, 3, 4]
```

```
int_list2
```

```
[5, 6, 7, 8, 9]
```

```
list_from_variables
```

```
[5, 5.0, 5.0, 'dee', 'e']
```

```
list_of_lists
```

```
[[],  
 ['lions', 'tigers', 'bears', 'sharks', 'hamsters'],  
 [5, 5.0, 5.0, 'dee', 'e'],  
 [0, 1, 2, 3, 4],  
 [5, 6, 7, 8, 9]]
```

Elements of a list are accessible by their index.

Warning: Beware that Python uses [zero-based numbering](#), i.e. the first index value is 0.

```
string_list[0]
```

```
'lions'
```

Slices can be used to extract a contiguous subset:

```
string_list[1:4]
```

```
['tigers', 'bears', 'sharks']
```

Or subset patterns. This extracts every 2nd element:

```
int_list[::-2]
```

```
[0, 2, 4]
```

To access an item in a nested list use successive square brackets:

```
list_of_lists[1][4]
```

```
'hamsters'
```

Lists are mutable, meaning after a list is created we can change, add, or remove elements. Here are several ways to modify a list:

```
int_list[2] = 222  
int_list.append(5)  
string_list.remove('lions')  
list_from_variables.extend(int_list)
```

Note that the existing lists have been modified in-place:

```
int_list
```

```
[0, 1, 222, 3, 4, 5]
```

```
string_list
```

```
['tigers', 'bears', 'sharks', 'hamsters']
```

```
list_from_variables
```

```
[5, 5.0, 5.0, 'dee', 'e', 0, 1, 222, 3, 4, 5]
```

Tuples

Tuples share similarities with lists. The primary difference between a list and tuple is that tuples are **not mutable**. A tuple is good for organizing related data that may be of different types. Note that tuples are defined with parentheses, (), rather than square brackets.

```
joe_blow = (32, 'tall', 'likes hats')
```

```
(32, 'tall', 'likes hats')
```

Indexing works the same as lists:

```
joe_blow[1]
```

```
'tall'
```

Unlike lists, tuples are immutable. They cannot be changed once defined. Trying some of the mutating methods of lists results in errors on tuples:

```
joe_blow.append('married')
```

```
-----  
AttributeError                                     Traceback (most recent call last)  
Cell In[24], line 1  
----> 1 joe_blow.append('married')  
  
AttributeError: 'tuple' object has no attribute 'append'
```

```
joe_blow[2] = 'not really a fan of hats'
```

```
-----  
TypeError                                         Traceback (most recent call last)  
Cell In[25], line 1  
----> 1 joe_blow[2] = 'not really a fan of hats'  
  
TypeError: 'tuple' object does not support item assignment
```

In Python, a function can return multiple values. These multiple outputs are packed into a tuple. Tuple unpacking assigns individual elements of a tuple to separate variables.

```
pets = ('elephant', 'cow', 'rock')
```

```
pet1, pet2, pet3 = pets
```

(continues on next page)

(continued from previous page)

pet1

'elephant'

A peculiar thing about tuples in Python is defining a single element tuple. Note the trailing comma. This is necessary for Python to know you want a one-element tuple.

tuple_with_one_item = pet1,

tuple_with_one_item

('elephant',)

Dictionaries

A dictionary is an unordered set of *key: value* pairs. Much like a language dictionary where you look up a *word* and get its *definition*, in a Python dictionary you look up a *key* and get its *value*.

Any immutable object can be used as a key, any object can be a value. For example, here are strings as both keys and values:

dictionary0 = {'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
dictionary0

{'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}

or integers can be used as keys:

dictionary1 = {1: 'value1', 2: 'value2', 3: 'value3'}
dictionary1

{1: 'value1', 2: 'value2', 3: 'value3'}

The keys and values can be extracted separately using `.keys()` and `.values()` and converting to a list:

list(dictionary1.keys())

[1, 2, 3]

list(dictionary1.values())

['value1', 'value2', 'value3']

Individual items can be extracted with square brackets and the key:

cylinder = {'mass': 50, 'base': 10, 'height': 100}
cylinder['mass']

50

The `zip()` function is a convenient way to help generate a dictionary. It takes sequence objects and combines them into a list of tuples. We can subsequently use the list of four-element tuples to create a dictionary.

```
keys = ['mass01', 'inertia01', 'mass02', 'inertia02']  
values = [10, 1, 50, 5]  
dict(zip(keys, values))
```

```
{'mass01': 10, 'inertia01': 1, 'mass02': 50, 'inertia02': 5}
```

4.4.3 Functions

Python does not use braces, {}, or end statements to separate blocks of code. Rather, code blocks are initialized with colon, :, and defined by their indentation. It is convention to use four spaces (not tabs) for each level of indentation. Functions are defined and used like so:

```
def abs_value(A):  
    if A < 0:  
        A = -A  
    return A  
  
abs_value(-100)
```

```
100
```

```
abs_value(123)
```

```
123
```

This function returns two results:

```
def long_div(dividend, divisor):  
    quotient = dividend // divisor # // : floor division  
    remainder = dividend % divisor # % : modulo  
    return quotient, remainder
```

Now you can use the function:

```
a = 430  
b = 25  
  
quo, rem = long_div(a, b)  
  
quo, rem
```

```
(17, 5)
```

`print()` and `.format()` can be used to make custom text to display:

```
msg = '{} divided {} is {} remainder {}'.format(a, b, quo, rem)  
print(msg)
```

```
430 divided 25 is 17 remainder 5
```

4.4.4 Modules

Modules add additional functionality not present in the default namespace of Python. Some modules are included with Python (builtin modules) and some are provided by other software packages and libraries you download and install. For example, the builtin `sys` module provides access to system-specific parameters and functions. You can check what Python version you are currently using by first importing the `sys` module and then accessing the `.version` variable:

```
import sys  
print(sys.version)
```

```
3.10.13 | packaged by conda-forge | (main, Dec 23 2023, 15:36:39) [GCC 12.3.0]
```

You can also import the `version` variable to have it included in the current namespace:

```
from sys import version  
print(version)
```

```
3.10.13 | packaged by conda-forge | (main, Dec 23 2023, 15:36:39) [GCC 12.3.0]
```

You will be using SymPy, NumPy, SciPy, and matplotlib further along in this book. These packages will consistently be imported like so:

```
import sympy as sm  
import numpy as np  
import scipy as sp  
import matplotlib.pyplot as plt
```

This will allow you to keep the namespaces separate so that there are no variable name clashes. For example, SymPy, NumPy, and SciPy all have trigonometric functions:

```
sm.cos(12.0)
```

0.843853958732492 (4.1)

```
np.cos(12.0)
```

```
0.8438539587324921
```

```
sp.cos(12.0)
```

```
0.8438539587324921
```

and there may be times when you want to use more than one version of `cos()` in a single namespace.

4.5 Learn More

4.5.1 More Jupyter

There are many introductory resources for learning to use Jupyter which can be found with search engines. As examples, this RealPython introduction is a good start (ignore the installation part, as you have it installed already from the instructions in this book):

<https://realPython.com/jupyter-notebook-introduction/>

This 7 minute video also gives the basics:

4.5.2 More Python

There are literally thousands of Python learning materials freely available on the web that fit many different needs. Here are a few recommendations for core Python for beginners:

- Allen Downey's book "ThinkPython": <https://greenteapress.com/wp/think-python-2e>
- Google's Python Class: <https://developers.google.com/edu/python/>
- The official Python tutorial: <https://docs.Python.org/3/tutorial>

Note: You can download this example as a Python script: `sympy.py` or Jupyter notebook: `sympy.ipynb`.

5.1 Learning Objectives

After completing this chapter readers will be able to:

- Write mathematical expressions with symbols and functions using SymPy.
- Print different forms of expressions and equations with SymPy.
- Differentiate mathematical expressions using SymPy.
- Evaluate mathematical expressions using SymPy.
- Create matrices and do linear algebra using SymPy.
- Solve a linear system of equations with SymPy.
- Simplify mathematical expressions with SymPy.

5.2 Introduction

SymPy is an open source, collaboratively developed computer algebra system (CAS) written in Python. It will be used extensively for manipulating symbolic expressions and equations. All of the mathematics needed to formulate the equations of motion of multibody systems can be done with pencil and paper, but the bookkeeping becomes extremely tedious and error prone for systems with even a small number of bodies. SymPy lets a computer handle the tedious aspects (e.g. differentiation or solving linear systems of equations) and reduces the errors one would encounter with pencil and paper. This chapter introduces SymPy and the primary SymPy features we will be using.

5.3 Import and Setup

I will import SymPy as follows throughout this book:

```
import sympy as sm
```

Since SymPy works with mathematical symbols it's nice to view SymPy objects in a format that is similar to the math in a textbook. Executing `init_printing()` at the beginning of your Jupyter Notebook will ensure that SymPy objects render as typeset mathematics. I use the `use_latex='mathjax'` argument here to disable math png image generation, but that keyword argument is not necessary.

```
sm.init_printing(use_latex='mathjax')
```

5.4 Symbols

Mathematical symbols are created with the `symbols()` function. A symbol a is created like so:

```
a = sm.symbols('a')  
a
```

$$a \tag{5.1}$$

This symbol object is of the `Symbol` type:

```
type(a)
```

```
sympy.core.symbol.Symbol
```

Multiple symbols can be created with one call to `symbols()` and SymPy recognizes common Greek symbols by their spelled-out name.

```
b, t, omega, Omega = sm.symbols('b, t, omega, Omega')  
b, t, omega, Omega
```

$$(b, t, \omega, \Omega) \tag{5.2}$$

Note that the argument provided to `symbols()` does not need to match the Python variable name it is assigned to. Using more verbose Python variable names may make code easier to read and understand, especially if there are many mathematical variables that you need to keep track of. Note that the subscripts are recognized too.

```
pivot_angle, w2 = sm.symbols('alpha1, omega2')  
pivot_angle, w2
```

$$(\alpha_1, \omega_2) \tag{5.3}$$

Exercise

Review the SymPy documentation and create symbols q_1, q_2, \dots, q_{10} with a very succinct call to `symbols()`.

Solution

```
sm.symbols('q1:11')
```

$$(q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}) \quad (5.4)$$

5.5 Undefined Functions

You will also work with undefined mathematical functions in addition to symbols. These will play an important role in setting up differential equations, where you typically don't know the function, but only its derivative(s). You can create arbitrary functions of variables. In this case, you make a function of t . First create the function name:

```
f = sm.Function('f')
```

```
f
```

This is of a type `sympy.core.function.UndefinedFunction`.

```
type(f)
```

```
sympy.core.function.UndefinedFunction
```

Now you can create functions of one or more variables like so:

```
f(t)
```

$$f(t) \quad (5.5)$$

Warning: Due to SymPy's internal implementations, the type of a function with its argument is not defined as expected:

```
type(f(t))
```

```
f
```

This can be confusing if you are checking types.

The same `UndefinedFunction` can be used to create multivariate functions:

```
f(a, b, omega, t)
```

$$f(a, b, \omega, t) \quad (5.6)$$

Exercise

Create a function $H(x, y, z)$.

Solution

```
x, y, z = sm.symbols('x, y, z')
sm.Function('H')(x, y, z)
```

$$H(x, y, z) \quad (5.7)$$

5.6 Symbolic Expressions

Now that you have mathematical variables and functions available, they can be used to construct mathematical expressions. The most basic way to construct expressions is with the standard Python operators `+`, `-`, `*`, `/`, and `**`. For example:

```
expr1 = a + b/omega**2
expr1
```

$$a + \frac{b}{\omega^2} \quad (5.8)$$

An expression will have the type `Add`, `Mul`, or `Pow`:

```
type(expr1)
```

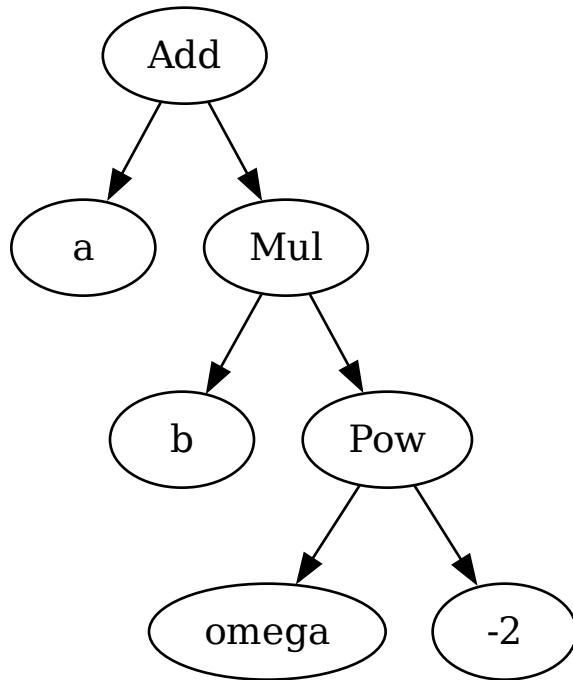
```
sympy.core.add.Add
```

This is because SymPy stores expressions behind the scenes as a `tree`. You can inspect this internal representation by using the `srepr()` function:

```
sm.srepr(expr1)
```

```
"Add(Symbol('a'), Mul(Symbol('b'), Pow(Symbol('omega'), Integer(-2))))"
```

This is a visual representation of the tree:



This representation is SymPy’s “true” representation of the symbolic expression. SymPy can display this expression in many other representations, for example the typeset mathematical expression you have already seen is one of those representations. This is important to know, because sometimes the expressions are displayed to you in a way that may be confusing and checking the `srepr()` version can help clear up misunderstandings. See the [manipulation section](#) of the SymPy tutorial for more information on this.

Undefined functions can also be used in expressions just like symbols:

```
expr2 = f(t) + a*omega
expr2
```

$$a\omega + f(t) \quad (5.9)$$

SymPy has a large number of elementary and special functions. See the SymPy [documentation on functions](#) for more information. For example, here is an expression that uses `sin`, `Abs`, and `sqrt()`:

```
expr3 = a*sm.sin(omega) + sm.Abs(f(t))/sm.sqrt(b)
expr3
```

$$a \sin(\omega) + \frac{|f(t)|}{\sqrt{b}} \quad (5.10)$$

Note that Python integers and floats can also be used when constructing expressions:

```
expr4 = 5*sm.sin(12) + sm.Abs(-1001)/sm.sqrt(89.2)
expr4
```

$$5 \sin(12) + 105.986768359379 \quad (5.11)$$

Warning: Be careful with numbers, as SymPy may not interpret them as expected. For example:

```
1/2*a
```

$$0.5a \quad (5.12)$$

Python does the division before it is multiplied by a , thus a floating point value is created. To fix this you can use the `S()` function to “sympify” numbers:

```
sm.S(1)/2*a
```

$$\frac{a}{2} \quad (5.13)$$

Or you can ensure the symbol comes first in the division operation:

```
a/2
```

$$\frac{a}{2} \quad (5.14)$$

Lastly, an expression of t :

```
expr5 = t*sm.sin(omega*f(t)) + f(t)/sm.sqrt(t)
expr5
```

$$t \sin(\omega f(t)) + \frac{f(t)}{\sqrt{t}} \quad (5.15)$$

Exercise

Create an expression for the normal distribution function:

$$\frac{1}{\sqrt{2\pi}\sigma} e^{\frac{(x-\mu)^2}{2\sigma^2}} \quad (5.16)$$

Solution

```
x, s, m = sm.symbols('x, sigma, mu')
sm.exp((x-m)**2/2/s**2)/sm.sqrt(2*sm.pi*s)
```

$$\frac{\sqrt{2}e^{\frac{(-\mu+x)^2}{2\sigma^2}}}{2\sqrt{\pi}\sqrt{\sigma}} \quad (5.17)$$

Notice that SymPy does some minor manipulation of the expression, but it is equivalent to the form shown in the prompt.

5.7 Printing

I introduced the `srepr()` form of SymPy expressions above and mentioned that expressions can have different representations. For the following `srepr()` form:

```
sm.srepr(expr3)
```

```
"Add(Mul(Symbol('a'), sin(Symbol('omega'))), Mul(Pow(Symbol('b'), Rational(-1, 2)),  
Abs(Function('f')(Symbol('t')))))"
```

There is also a standard representation accessed with the `repr()` function:

```
repr(expr3)
```

```
'a*sin(omega) + Abs(f(t))/sqrt(b)'
```

This form matches what you typically would type to create the expression and it returns a string. The `print()` function will display that string:

```
print(expr3)
```

```
a*sin(omega) + Abs(f(t))/sqrt(b)
```

SymPy also has a “pretty printer” (`pprint()`) that makes use of unicode symbols to provide a form that more closely resembles typeset math:

```
sm.pprint(expr3)
```

```
a·sin(ω) +  $\frac{|f(t)|}{\sqrt{b}}$ 
```

Lastly, the following lines show how SymPy expressions can be represented as LaTeX code using `sympy.printing.latex.latex()`. The double backslashes are present because double backslashes represent the escape character in Python strings.

```
sm.latex(expr3)
```

```
'a \sin{\omega} + \frac{|f(t)|}{\sqrt{b}}'
```

```
print(sm.latex(expr3))
```

```
a \sin{\left(\omega \right)} + \frac{\left|f{\left(t \right)}\right| \sqrt{b}}
```

Warning: When you are working with long expressions, which will be the case in this course, there is no need to print them to the screen. In fact, printing them to the screen make take a long time and fill your entire notebook with an unreadable mess.

Exercise

Print the normal distribution expression

$$\frac{1}{\sqrt{2\pi}\sigma} e^{\frac{(x-\mu)^2}{2\sigma^2}} \quad (5.18)$$

as a LaTeX string inside an equation environment.

Solution

```
x, s, m = sm.symbols('x, sigma, mu')
print(sm.latex(sm.exp((x-m)**2/2/s**2)/sm.sqrt(2*sm.pi*s),
mode='equation'))
```

```
\begin{equation}\frac{\sqrt{2} e^{\frac{(-\mu + x)^2}{2 \sigma^2}}}{\sqrt{\pi} \sqrt{\sigma}}\end{equation}
```

5.8 Differentiating

One of the most tedious tasks in formulating equations of motion is differentiating complex trigonometric expressions. SymPy can calculate derivatives effortlessly. The `diff()` SymPy function takes an undefined function or an expression and differentiates it with respect to the symbol provided as the second argument:

```
sm.diff(f(t), t)
```

$$\frac{d}{dt} f(t) \quad (5.19)$$

All functions and expressions also have a `.diff()` method which can be used like so (many SymPy functions exist as standalone functions and methods):

```
f(t).diff(t)
```

$$\frac{d}{dt} f(t) \quad (5.20)$$

`expr3` is a more complicated expression:

expr3

$$a \sin(\omega) + \frac{|f(t)|}{\sqrt{b}} \quad (5.21)$$

It can be differentiated, for example, with respect to b :

expr3.diff(b)

$$-\frac{|f(t)|}{2b^{\frac{3}{2}}} \quad (5.22)$$

You can also calculate partial derivatives with respect to successive variables. If you want to first differentiate with respect to b and then with respect to t as in the following operation:

$$\frac{\partial^2 h(a, \omega, t, b)}{\partial t \partial b} \quad (5.23)$$

where:

$$h(a, \omega, t, b) = a \sin(\omega) + \frac{|f(t)|}{\sqrt{b}} \quad (5.24)$$

then you can use successive arguments to `.diff()`:

expr3.diff(b, t)

$$-\frac{(\operatorname{re}(f(t)) \frac{d}{dt} \operatorname{re}(f(t)) + \operatorname{im}(f(t)) \frac{d}{dt} \operatorname{im}(f(t))) \operatorname{sign}(f(t))}{2b^{\frac{3}{2}} f(t)} \quad (5.25)$$

Note that the answer includes real and imaginary components and the `signum` function.

Warning: SymPy assumes all symbols are complex-valued unless told otherwise. You can attach assumptions to symbols to force them to be real, positive, negative, etc. For example, compare these three outputs:

```
h = sm.Function('h')
sm.Abs(h(t)).diff(t)
```

$$\frac{(\operatorname{re}(h(t)) \frac{d}{dt} \operatorname{re}(h(t)) + \operatorname{im}(h(t)) \frac{d}{dt} \operatorname{im}(h(t))) \operatorname{sign}(h(t))}{h(t)} \quad (5.26)$$

```
h = sm.Function('h', real=True)
sm.Abs(h(t)).diff(t)
```

$$\operatorname{sign}(h(t)) \frac{d}{dt} h(t) \quad (5.27)$$

```
h = sm.Function('h', real=True, positive=True)
sm.Abs(h(t)).diff(t)
```

$$\frac{d}{dt} h(t) \quad (5.28)$$

Sometimes you may need to add assumptions to variables, but in general it will not be necessary. Read more about assumptions in SymPy's [guide](#).

Exercise

Differentiate `expr5` above using this operator:

$$\frac{\partial^2}{\partial \omega \partial t} \quad (5.29)$$

Solution

First show `expr5`:

```
expr5
```

$$t \sin(\omega f(t)) + \frac{f(t)}{\sqrt{t}} \quad (5.30)$$

The twice partial derivative is:

```
expr5.diff(t, omega)
```

$$-\omega t f(t) \sin(\omega f(t)) \frac{d}{dt} f(t) + t \cos(\omega f(t)) \frac{d}{dt} f(t) + f(t) \cos(\omega f(t)) \quad (5.31)$$

or you can chain `.diff()` calls:

```
expr5.diff(t).diff(omega)
```

$$-\omega t f(t) \sin(\omega f(t)) \frac{d}{dt} f(t) + t \cos(\omega f(t)) \frac{d}{dt} f(t) + f(t) \cos(\omega f(t)) \quad (5.32)$$

5.9 Evaluating Symbolic Expressions

SymPy expressions can be evaluated numerically in several ways. The `xreplace()` method allows substitution of exact symbols or sub-expressions. First create a dictionary that maps symbols, functions or sub-expressions to the replacements:

```
rep1 = {omega: sm.pi/4, a: 2, f(t): -12, b: 25}
```

This dictionary can then be passed to `.xreplace()`:

```
expr3.xreplace(repl)
```

$$\sqrt{2} + \frac{12}{5} \quad (5.33)$$

Notice how the square root and fraction do not automatically reduce to their decimal equivalents. To do so, you must use the `evalf()` method. This method will evaluate an expression to an arbitrary number of decimal points. You provide the number of decimal places and the substitution dictionary to evaluate:

```
expr3.evalf(n=10, subs=repl)
```

$$3.814213562 \quad (5.34)$$

```
type(expr3.evalf(n=10, subs=repl))
```

```
sympy.core.numbers.Float
```

Note that this is a SymPy `Float` object, which is a special object that can have an arbitrary number of decimal places, for example here is the expression evaluated to 80 decimal places:

```
expr3.evalf(n=80, subs=repl)
```

$$3.814213562373095048801688724209698078569671875376948073176679737990732478462107 \quad (5.35)$$

To convert this to Python floating point number, use `float()`:

```
float(expr3.evalf(n=300, subs=repl))
```

$$3.81421356237309 \quad (5.36)$$

```
type(float(expr3.evalf(n=300, subs=repl)))
```

```
float
```

This value is a `machine precision` floating point value and can be used with standard Python functions that operate on floating point numbers.

To obtain machine precision floating point numbers directly and with more flexibility, it is better to use the `lambdify()` function to convert the expression to a Python function. When using `lambdify()`, all symbols and functions should be converted to numbers so first identify what symbols and functions make up the expression.

```
expr3
```

$$a \sin(\omega) + \frac{|f(t)|}{\sqrt{b}} \quad (5.37)$$

ω , a , $f(t)$, and b are all present in the expression. The first argument of `lambdify()` should be a sequence of all these symbols and functions and the second argument should be the expression.

```
eval_expr3 = sm.lambdify((omega, a, f(t), b), expr3)
```

`lambdify()` generates a Python function and, in this case, we store that function in the variable `eval_expr3`. You can see what the inputs and outputs of the function are with `help()`:

```
help(eval_expr3)
```

Help on function `_lambdifygenerated`:

```
_lambdifygenerated(omega, a, _Dummy_24, b)
  Created with lambdify. Signature:
    func(omega, a, f, b)

  Expression:
    a*sin(omega) + Abs(f(t))/sqrt(b)

  Source code:
    def _lambdifygenerated(omega, a, _Dummy_24, b):
        return a*sin(omega) + abs(_Dummy_24)/sqrt(b)

  Imported modules:
```

This function operates on and returns floating point values, for example:

```
eval_expr3(3.14/4, 2, -12, 25)
```

3.81365036221073 (5.38)

The type of `lambdify`'s return values will be `NumPy` floats.

```
type(eval_expr3(3.14/4, 2, -12, 25))
```

```
numpy.float64
```

These floats are interoperable with Python floats for single values (unlike `Sympy` Floats) but also support arrays of floats. For example:

```
eval_expr3(3.14/4, 2, -12, [25, 26, 27])
```

```
array([3.81365036, 3.76704398, 3.72305144])
```

```
type(eval_expr3(3.14/4, 2, -12, [25, 26, 27]))
```

```
numpy.ndarray
```

More on NumPy arrays of floats will be introduced in a later chapter.

Warning: Python and NumPy floats can be mixed, but avoid mixing SymPy Floats with either.

Note: This distinction between SymPy `Float` objects and regular Python and NumPy `float` objects is important. In this case, the Python float and the NumPy float are equivalent. The later will compute much faster because arbitrary precision is not required. In this book, you will almost always want to convert SymPy expressions to machine precision floating point numbers, so use `lambdify()`.

Exercise

Create a symbolic expression representing [Newton's Law of Universal Gravitation](#). Use `lambdify()` to evaluate the expression for two mass of 5.972E24 kg and 80 kg at a distance of 6371 km apart to find the gravitational force in Newtons.

Solution

```
G, m1, m2, r = sm.symbols('G, m1, m2, r')
F = G*m1*m2/r**2
eval_F = sm.lambdify((G, m1, m2, r), F)
eval_F(6.67430E-11, 5.972E24, 80, 6371E3)
```

785.597874097975 (5.39)

5.10 Matrices

SymPy supports matrices of expressions and linear algebra. Many of the operations needed in multibody dynamics are more succinctly formulated with matrices and linear algebra. Matrices can be created by passing nested lists to the `Matrix()` object. For example:

```
mat1 = sm.Matrix([[a, 2*a], [b/omega, f(t)]])
mat1
```

$$\begin{bmatrix} a & 2a \\ \frac{b}{\omega} & f(t) \end{bmatrix} \quad (5.40)$$

```
mat2 = sm.Matrix([[1, 2], [3, 4]])
mat2
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad (5.41)$$

All matrices are two dimensional and the number of rows and columns, in that order, are stored in the `.shape` attribute.

```
mat1.shape
```

$$(2, 2) \quad (5.42)$$

Individual elements of the matrix can be extracted with the bracket notation taking the row and column indices (remember Python indexes from 0):

```
mat1[0, 1]
```

$$2a \quad (5.43)$$

The slice notation can extract rows or columns:

```
mat1[0, 0:2]
```

$$\begin{bmatrix} a & 2a \end{bmatrix} \quad (5.44)$$

```
mat1[0:2, 1]
```

$$\begin{bmatrix} 2a \\ f(t) \end{bmatrix} \quad (5.45)$$

Matrix algebra can be performed. Matrices can be added:

```
mat1 + mat2
```

$$\begin{bmatrix} a + 1 & 2a + 2 \\ \frac{b}{\omega} + 3 & f(t) + 4 \end{bmatrix} \quad (5.46)$$

Both the `*` and the `@` operator perform matrix multiplication:

```
mat1*mat2
```

$$\begin{bmatrix} 7a & 10a \\ \frac{b}{\omega} + 3f(t) & \frac{2b}{\omega} + 4f(t) \end{bmatrix} \quad (5.47)$$

mat1@mat2

$$\begin{bmatrix} 7a & 10a \\ \frac{b}{\omega} + 3f(t) & \frac{2b}{\omega} + 4f(t) \end{bmatrix} \quad (5.48)$$

Element-by-element multiplication requires the `sympy.hadamard_product()` function:

sm.hadamard_product(mat1, mat2)

$$\begin{bmatrix} a & 4a \\ \frac{3b}{\omega} & 4f(t) \end{bmatrix} \quad (5.49)$$

Note that NumPy uses `*` for element-by-element multiplication and `@` for matrix multiplication, so to avoid possible confusion, use `@` for SymPy matrix multiplication.

Differentiation operates on each element of the matrix:

```
mat3 = sm.Matrix([expr1, expr2, expr3, expr4, expr5])
mat3
```

$$\begin{bmatrix} a + \frac{b}{\omega^2} \\ a\omega + f(t) \\ a \sin(\omega) + \frac{|f(t)|}{\sqrt{b}} \\ 5 \sin(12) + 105.986768359379 \\ t \sin(\omega f(t)) + \frac{f(t)}{\sqrt{t}} \end{bmatrix} \quad (5.50)$$

mat3.diff(a)

$$\begin{bmatrix} 1 \\ \omega \\ \sin(\omega) \\ 0 \\ 0 \end{bmatrix} \quad (5.51)$$

mat3.diff(t)

$$\begin{bmatrix} 0 \\ \frac{d}{dt}f(t) \\ \frac{(\operatorname{re}(f(t))\frac{d}{dt}\operatorname{re}(f(t))+\operatorname{im}(f(t))\frac{d}{dt}\operatorname{im}(f(t)))\operatorname{sign}(f(t))}{\sqrt{b}f(t)} \\ 0 \\ \omega t \cos(\omega f(t))\frac{d}{dt}f(t) + \sin(\omega f(t)) + \frac{\frac{d}{dt}f(t)}{\sqrt{t}} - \frac{f(t)}{2t^{\frac{3}{2}}} \end{bmatrix} \quad (5.52)$$

If you have column vectors \bar{v} and \bar{u} , the (i, j) entries of the Jacobian of \bar{v} with respect to the entries in vector \bar{u} are found with $\mathbf{J}_{ij} = \frac{\partial v_i}{\partial u_j}$. The `Jacobian` matrix of vector (column matrix) can be formed with the `jacobian()` method. This calculates the partial derivatives of each element in the vector with respect to a vector (or sequence) of variables.

```
mat4 = sm.Matrix([a, b, omega, t])
mat4
```

$$\begin{bmatrix} a \\ b \\ \omega \\ t \end{bmatrix} \quad (5.53)$$

```
mat3.jacobian(mat4)
```

$$\begin{bmatrix} 1 & \frac{1}{\omega^2} & -\frac{2b}{\omega^3} & 0 \\ \omega & 0 & a & \frac{d}{dt}f(t) \\ \sin(\omega) & -\frac{|f(t)|}{2b^{\frac{3}{2}}} & a \cos(\omega) & \frac{(\operatorname{re}(f(t)) \frac{d}{dt} \operatorname{re}(f(t)) + \operatorname{im}(f(t)) \frac{d}{dt} \operatorname{im}(f(t))) \operatorname{sign}(f(t))}{\sqrt{b}f(t)} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & tf(t) \cos(\omega f(t)) & \omega t \cos(\omega f(t)) \frac{d}{dt}f(t) + \sin(\omega f(t)) + \frac{\frac{d}{dt}f(t)}{\sqrt{t}} - \frac{f(t)}{2t^{\frac{3}{2}}} \end{bmatrix} \quad (5.54)$$

Exercise

Write your own function that produces a Jacobian given a column matrix of expressions. It should look like:

```
def jacobian(v, x):
    """Returns the Jacobian of the vector function v with respect to the
    vector of variables x."""
    # fill in your code here
    return J_v_x
```

Show that it gives the same solution as the above `.jacobian()` method. Do not use the `.jacobian()` method in your function.

Solution

```
def jacobian(v, x):
    """Returns the Jacobian of the vector function v with respect to the
    vector of variables x."""
    diffs = []
    for expr in v:
        for var in x:
            diffs.append(expr.diff(var))
    J_v_x = sm.Matrix(diffs).reshape(len(v), len(x))
    return J_v_x

jacobian(mat3, mat4)
```

$$\begin{bmatrix} 1 & \frac{1}{\omega^2} & -\frac{2b}{\omega^3} & 0 \\ \omega & 0 & a & \frac{d}{dt}f(t) \\ \sin(\omega) & -\frac{|f(t)|}{2b^{\frac{3}{2}}} & a \cos(\omega) & \frac{(\operatorname{re}(f(t)) \frac{d}{dt} \operatorname{re}(f(t)) + \operatorname{im}(f(t)) \frac{d}{dt} \operatorname{im}(f(t))) \operatorname{sign}(f(t))}{\sqrt{b}f(t)} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & tf(t) \cos(\omega f(t)) & \omega t \cos(\omega f(t)) \frac{d}{dt}f(t) + \sin(\omega f(t)) + \frac{\frac{d}{dt}f(t)}{\sqrt{t}} - \frac{f(t)}{2t^{\frac{3}{2}}} \end{bmatrix} \quad (5.55)$$

5.11 Solving Linear Systems

You'll need to solve linear systems of equations often in this book. SymPy offers a number of ways to do this, but the best way to do so if you know a set of equations are linear in specific variables is the method described below. First, you should confirm you have equations of this form:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n + b_1 &= 0 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n + b_2 &= 0 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n + b_n &= 0 \end{aligned} \quad (5.56)$$

These equations can be put into matrix form:

$$\mathbf{A}\bar{x} = \bar{b} \quad (5.57)$$

where:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}, \bar{x} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix}, \bar{b} = \begin{bmatrix} -b_1 \\ -b_2 \\ \dots \\ -b_n \end{bmatrix} \quad (5.58)$$

\bar{x} , the solution, is found with matrix inversion (if the matrix is invertible):

$$\bar{x} = \mathbf{A}^{-1}\bar{b} \quad (5.59)$$

Taking the inverse is not computationally efficient and potentially numerically inaccurate, so some form of Gaussian elimination should be used to solve the system.

To solve with SymPy, start with a column matrix of linear expressions:

```
a1, a2 = sm.symbols('a1, a2')

exprs = sm.Matrix([
    [a1*sm.sin(f(t))*sm.cos(2*f(t)) + a2 + omega/sm.log(f(t), t) + 100],
    [a1*omega**2 + f(t)*a2 + omega + f(t)**3],
])
exprs
```

$$\begin{bmatrix} a_1 \sin(f(t)) \cos(2f(t)) + a_2 + \frac{\omega \log(t)}{\log(f(t))} + 100 \\ a_1 \omega^2 + a_2 f(t) + \omega + f^3(t) \end{bmatrix} \quad (5.60)$$

Since we know these two expressions are linear in the a_1 and a_2 variables, the partial derivatives with respect to those two variables will return the linear coefficients. The \mathbf{A} matrix can be formed in one step with the `.jacobian()` method:

```
A = exprs.jacobian([a1, a2])
A
```

$$\begin{bmatrix} \sin(f(t)) \cos(2f(t)) & 1 \\ \omega^2 & f(t) \end{bmatrix} \quad (5.61)$$

The \bar{b} vector can be formed by setting $a_1 = a_2 = 0$, leaving the terms that are not linear in a_1 and a_2 .

```
b = -exprs.xreplace({a1: 0, a2: 0})
b
```

$$\begin{bmatrix} -\frac{\omega \log(t)}{\log(f(t))} - 100 \\ -\omega - f^3(t) \end{bmatrix} \quad (5.62)$$

The `inv()` method can compute the inverse of A to find the solution:

```
A.inv() @ b
```

$$\begin{bmatrix} -\omega - f^3(t) \\ -\frac{-\omega^2 + f(t) \sin(f(t)) \cos(2f(t))}{\omega^2(-\log(f(t)) - 100)} + \frac{(-\frac{\omega \log(t)}{\log(f(t))} - 100)f(t)}{-\omega^2 + f(t) \sin(f(t)) \cos(2f(t))} \\ -\frac{\omega^2(-\log(f(t)) - 100)}{-\omega^2 + f(t) \sin(f(t)) \cos(2f(t))} + \frac{(-\omega - f^3(t)) \sin(f(t)) \cos(2f(t))}{-\omega^2 + f(t) \sin(f(t)) \cos(2f(t))} \end{bmatrix} \quad (5.63)$$

But it is best to use the `LUsolve()` method to perform an LU decomposition Gaussian-Elimination to solve the system, especially as the dimension of A grows:

```
A.LUsolve(b)
```

$$\begin{bmatrix} -\frac{\omega^2(-\frac{\omega \log(t)}{\log(f(t))} - 100)}{\sin(f(t)) \cos(2f(t))} - \omega - f^3(t) \\ -\frac{\omega \log(t)}{\log(f(t))} - 100 - \frac{-\frac{\omega^2(-\frac{\omega \log(t)}{\log(f(t))} - 100)}{\sin(f(t)) \cos(2f(t))} - \omega - f^3(t)}{-\frac{\sin(f(t)) \cos(2f(t))}{\omega^2} + f(t)} \\ \frac{\sin(f(t)) \cos(2f(t))}{-\frac{\sin(f(t)) \cos(2f(t))}{\omega^2} + f(t)} \\ -\frac{\omega^2(-\frac{\omega \log(t)}{\log(f(t))} - 100)}{\sin(f(t)) \cos(2f(t))} - \omega - f^3(t) \\ -\frac{\omega^2}{\sin(f(t)) \cos(2f(t))} + f(t) \end{bmatrix} \quad (5.64)$$

Warning: This method of solving symbolic linear systems is fast, but it can give incorrect answers for:

1. expressions that are not actually linear in the variables the Jacobian is taken with respect to
2. A matrix entries that would evaluate to zero if simplified or specific numerical values are provided

So only use this method if you are sure your equations are linear and if your A matrix is made up of complex expressions, watch out for nan results after lambdify. `solve()` and `linsolve()` can also solve linear systems and they check for linearity and properties of the A matrix. The cost is that they can be extremely slow for large expressions (which we will have in this book).

Exercise

Solve the following equations for all of the L 's and then use `lambdify()` to evaluate the solution for $F_1 = 13$ and $F_2 = 32$.

$$\begin{aligned}
 -L_1 + L_2 - L_3/\sqrt{2} &= 0 \\
 L_3/\sqrt{2} + L_4 &= F_1 \\
 -L_2 - L_5/\sqrt{2} &= 0 \\
 L_5/\sqrt{2} &= F_2 \\
 L_5/\sqrt{2} + L_6 &= 0 \\
 -L_4 - L_5/\sqrt{2} &= 0
 \end{aligned} \tag{5.65}$$

Solution

```

L1, L2, L3, L4, L5, L6, F1, F2 = sm.symbols('L1, L2, L3, L4, L5, L6, F1, F2')

exprs = sm.Matrix([
    -L1 + L2 - L3/sm.sqrt(2),
    L3/sm.sqrt(2) + L4 - F1,
    -L2 - L5/sm.sqrt(2),
    L5/sm.sqrt(2) - F2,
    L5/sm.sqrt(2) + L6,
    -L4 - L5/sm.sqrt(2),
])
exprs

```

$$\begin{bmatrix}
 -L_1 + L_2 - \frac{\sqrt{2}L_3}{2} \\
 -F_1 + \frac{\sqrt{2}L_3}{2} + L_4 \\
 -L_2 - \frac{\sqrt{2}L_5}{2} \\
 -F_2 + \frac{\sqrt{2}L_5}{2} \\
 \frac{\sqrt{2}L_5}{2} + L_6 \\
 -L_4 - \frac{\sqrt{2}L_5}{2}
 \end{bmatrix} \tag{5.66}$$

```

unknowns = sm.Matrix([L1, L2, L3, L4, L5, L6])

coef_mat = exprs.jacobian(unknowns)
rhs = exprs.xreplace(dict(zip(unknowns, [0]*6)))

sol = coef_mat.LUsolve(rhs)

sm.Eq(unknowns, sol)

```

$$\begin{bmatrix}
 L_1 \\
 L_2 \\
 L_3 \\
 L_4 \\
 L_5 \\
 L_6
 \end{bmatrix} = \begin{bmatrix}
 F_1 + 2F_2 \\
 F_2 \\
 \sqrt{2}(-F_1 - F_2) \\
 F_2 \\
 -\sqrt{2}F_2 \\
 F_2
 \end{bmatrix} \tag{5.67}$$

```
eval_sol = sm.lambdify((F1, F2), sol)
eval_sol(13, 32)
```

```
array([[ 77.        ],
       [ 32.        ],
       [-63.63961031],
       [ 32.        ],
       [-45.254834  ],
       [ 32.        ]])
```

5.12 Simplification

The above result from `LUsolve()` is a bit complicated. Reproduced here:

```
a1, a2 = sm.symbols('a1, a2')
exprs = sm.Matrix([
    [a1*sm.sin(f(t))*sm.cos(2*f(t)) + a2 + omega/sm.log(f(t), t) + 100],
    [a1*omega**2 + f(t)*a2 + omega + f(t)**3],
])
A = exprs.jacobian([a1, a2])
b = -exprs.xreplace({a1: 0, a2: 0})
sol = A.LUsolve(b)
```

SymPy has some functionality for automatically simplifying symbolic expressions. The function `simplify()` will attempt to find a simpler version:

```
sm.simplify(sol)
```

$$\left[\frac{\frac{-\omega f(t) \log(t) + \omega \log(f(t)) + f^3(t) \log(f(t)) - 100 f(t) \log(f(t))}{(-\omega^2 + f(t) \sin(f(t)) \cos(2f(t))) \log(f(t))}}{\frac{-\omega^2 (\omega \log(t) + 100 \log(f(t))) + (\omega + f^3(t)) \log(f(t)) \sin(f(t)) \cos(2f(t))}{(\omega^2 - f(t) \sin(f(t)) \cos(2f(t))) \log(f(t))}} \right] \quad (5.68)$$

But you'll have the best luck at simplifying if you use simplification functions that target the type of expression you have. The `trigsimp()` function only attempts trigonometric simplifications, for example:

```
trig_expr = sm.cos(omega)**2 + sm.sin(omega)**2
trig_expr
```

$$\sin^2(\omega) + \cos^2(\omega) \quad (5.69)$$

```
sm.trigsimp(trig_expr)
```

Warning: Only attempt simplification on expressions that are several lines of text. Larger expressions become increasingly computationally intensive to simplify and there is generally no need to do so.

As mentioned earlier, SymPy represents expressions as trees. Symbolic expressions can also be represented as [directed acyclic graphs](#) that contain only one node for each unique expression (unlike SymPy's trees which may have the same expression in more than one node). These unique expressions, or “common subexpressions”, can be found with the `cse()` function. This function will provide a simpler form of the equations that minimizes the number of operations to compute the answer. We can count the number of basic operations (additions, multiplies, etc.) using `count_ops()`:

```
sm.count_ops(sol)
```

79 (5.71)

We can simplify with `cse()`:

```
substitutions, simplified = sm.cse(sol)
```

The `substitutions` variable contains a list of tuples, where each tuple has a new intermediate variable and the sub-expression it is equal to.

```
substitutions[0]
```

$(x_0, f(t))$ (5.72)

The `Eq()` class with tuple unpacking (*) can be used to display these tuples as equations:

```
sm.Eq(*substitutions[0])
```

$x_0 = f(t)$ (5.73)

```
sm.Eq(*substitutions[1])
```

$$x_1 = \frac{1}{\sin(x_0) \cos(2x_0)} \quad (5.74)$$

```
sm.Eq(*substitutions[2])
```

$$x_2 = \omega^2 x_1 \quad (5.75)$$

```
sm.Eq(*substitutions[4])
```

$$x_4 = \frac{-\omega - x_0^3 + x_2 x_3}{x_0 - x_2} \quad (5.76)$$

The `simplified` variable contains the simplified expression, made up of the intermediate variables.

```
simplified[0]
```

$$\begin{bmatrix} x_1 (-x_3 - x_4) \\ x_4 \end{bmatrix} \quad (5.77)$$

We can count the number of operations of the simplified version:

```
num_ops = sm.count_ops(simplified[0])
for sub in substitutions:
    num_ops += sm.count_ops(sub[1])
num_ops
```

$$22 \quad (5.78)$$

Exercise

`lambdify()` has an optional argument `cse=True|False` that applies common subexpression elimination internally to simplify the number of operations. Differentiate the `base_expr` with respect to `x` 10 times to generate a very long expression. Create two functions using `lambdify()`, one with `cse=True` and one with `cse=False`. Compare how long it takes to numerically evaluate the resulting functions using the `%timeit` magic.

```
a, b, c, x, y, z = sm.symbols('a, b, c, x, y, z')
base_expr = a*sm.sin(x*x + b*sm.cos(x*y) + c*sm.sin(x*z))
```

Solution

Differentiate 10 times:

```
long_expr = base_expr.diff(x, 10)
```

Create the numerical functions:

```
eval_long_expr = sm.lambdify((a, b, c, x, y, z), long_expr)
eval_long_expr_cse = sm.lambdify((a, b, c, x, y, z), long_expr, cse=True)
```

Now time each function:

```
%%timeit
eval_long_expr(1.0, 2.0, 3.0, 4.0, 5.0, 6.0)
```

```
261 µs ± 3.4 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```
%%timeit
eval_long_expr_cse(1.0, 2.0, 3.0, 4.0, 5.0, 6.0)
```

```
16.4 µs ± 38.3 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

5.13 Learn more

This section only scratches the surface of what SymPy can do. The presented concepts are the basic ones needed for this book, but getting more familiar with SymPy and what it can do will help. I recommend doing the [SymPy Tutorial](#). The “Gotchas” section is particularly helpful for common mistakes when using SymPy. The tutorial is part of the SymPy documentation <https://docs.sympy.org>, where you will find general information on SymPy.

The tutorial is also available on video:

If you want to ask a question about using SymPy (or search to see if someone else has asked your question), you can do so at the following places:

- [SymPy mailing list](#): Ask questions via email.
- [SymPy Github Discussions](#): Ask questions via Github.
- [Stackoverflow](#): Ask and search questions on the most popular coding Q&A website.

ORIENTATION OF REFERENCE FRAMES

Note: You can download this example as a Python script: `orientation.py` or Jupyter Notebook: `orientation.ipynb`.

```
import sympy as sm
sm.init_printing(use_latex='mathjax')
```

6.1 Learning Objectives

After completing this chapter readers will be able to:

- Define a reference frame with associated unit vectors.
- Define a direction cosine matrix between two oriented reference frames.
- Derive direction cosine matrices for simple rotations.
- Derive direction cosine matrices for successive rotations.
- Manage orientation and direction cosine matrices with SymPy.
- Rotate reference frames using Euler Angles.

6.2 Reference Frames

In the study of multibody dynamics, we are interested in observing motion of connected and interacting objects in three dimensional space. This observation necessitates the concept of a *frame of reference*, or reference frame. A reference frame is an abstraction which we define as the set of all points in *Euclidean space* that are carried by and fixed to the observer of any given state of motion. Practically speaking, it is useful to imagine your eye as an observer of motion. Your eye can orient itself in 3D space to view the motion of objects from any direction and the motion of objects will appear differently in the set of points associated with the reference frame attached to your eye depending on your eye's orientation.

It is important to note that a reference frame is not equivalent to a coordinate system. Any number of coordinate systems (e.g., Cartesian or spherical) can be used to describe the motion of points or objects in a reference frame. The coordinate system offers a system of measurement in a reference frame. We will characterize a reference frame by a *right-handed* set of mutually perpendicular unit vectors that can be used to describe its orientation relative to other reference frames and we will align a Cartesian coordinate system with the unit vectors to allow for easy measurement of points fixed or moving in the reference frame.

6.3 Unit Vectors

Vectors have a magnitude, direction, and sense (\pm) but notably not a position. Unit vectors have a magnitude of 1. Unit vectors can be fixed, orientation-wise, to a reference frame. For a reference frame named N we will define the three mutually perpendicular unit vectors as $\hat{n}_x, \hat{n}_y, \hat{n}_z$ where these right-handed cross products hold:

$$\begin{aligned}\hat{n}_x \times \hat{n}_y &= \hat{n}_z \\ \hat{n}_y \times \hat{n}_z &= \hat{n}_x \\ \hat{n}_z \times \hat{n}_x &= \hat{n}_y\end{aligned}\tag{6.1}$$

Note: Unit vectors will be designated using the “hat”, e.g. \hat{v} .

These unit vectors are fixed in the reference frame N . If a second reference frame A is defined, also with its set of right-handed mutually perpendicular unit vectors $\hat{a}_x, \hat{a}_y, \hat{a}_z$ then we can establish the relative orientation of these two reference frames based on the angles among the two frames’ unit vectors.

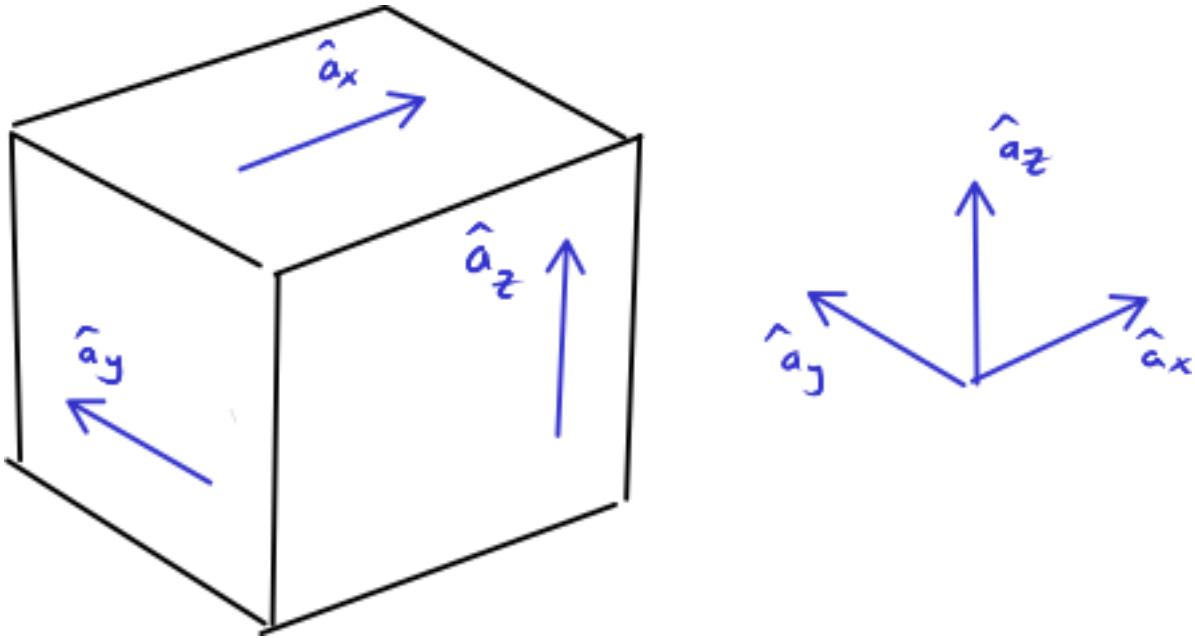


Fig. 6.1: The image on the left and right represent the same set of right-handed mutually perpendicular unit vectors. Vectors, in general, do not have a position and can be drawn anywhere in the reference frame. Drawing them with their tails coincident is simply done for convenience.

6.4 Simple Orientations

Starting with two reference frames N and A in which their sets of unit vectors are initially aligned, the A frame can then be simply oriented about the common parallel z unit vectors of the two frames. We then say “reference frame A is oriented with respect to reference frame N about the shared z unit vectors through an angle θ . A visual representation of this orientation looking from the direction of the positive z unit vector is:

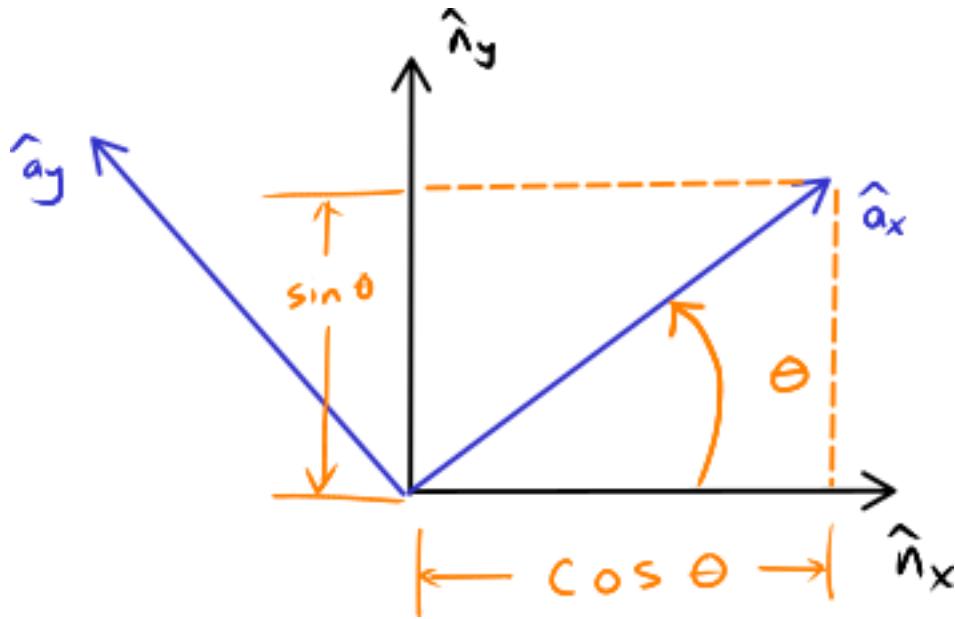


Fig. 6.2: View of the parallel xy planes of the simply oriented reference frames.

From the above figure these relationships between the \hat{a} and \hat{n} unit vectors can be deduced:

$$\begin{aligned}\hat{a}_x &= \cos \theta \hat{n}_x + \sin \theta \hat{n}_y + 0 \hat{n}_z \\ \hat{a}_y &= -\sin \theta \hat{n}_x + \cos \theta \hat{n}_y + 0 \hat{n}_z \\ \hat{a}_z &= 0 \hat{n}_x + 0 \hat{n}_y + 1 \hat{n}_z\end{aligned}\quad (6.2)$$

These equations can also be written in a matrix form:

$$\begin{bmatrix} \hat{a}_x \\ \hat{a}_y \\ \hat{a}_z \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \hat{n}_x \\ \hat{n}_y \\ \hat{n}_z \end{bmatrix} \quad (6.3)$$

This matrix uniquely describes the orientation between the two reference frames and so we give it its own variable:

$$\begin{bmatrix} \hat{a}_x \\ \hat{a}_y \\ \hat{a}_z \end{bmatrix} = {}^A\mathbf{C}^N \begin{bmatrix} \hat{n}_x \\ \hat{n}_y \\ \hat{n}_z \end{bmatrix} \quad (6.4)$$

This matrix ${}^A\mathbf{C}^N$ maps vectors expressed in the N frame to vectors expressed in the A frame. This matrix has an important property, which we will demonstrate with SymPy. Start by creating the matrix:

```
theta = sm.symbols('theta')

A_C_N = sm.Matrix([[sm.cos(theta), sm.sin(theta), 0],
                   [-sm.sin(theta), sm.cos(theta), 0],
                   [0, 0, 1]])
```

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (6.5)$$

If we'd like the inverse relationship between the two sets of unit vectors and ${}^A\mathbf{C}^N$ is invertible, then:

$$\begin{bmatrix} \hat{n}_x \\ \hat{n}_y \\ \hat{n}_z \end{bmatrix} = ({}^A\mathbf{C}^N)^{-1} \begin{bmatrix} \hat{a}_x \\ \hat{a}_y \\ \hat{a}_z \end{bmatrix} \quad (6.6)$$

SymPy can find this matrix inverse:

```
sm.trigsimp(A_C_N.inv())
```

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (6.7)$$

SymPy can also find the transpose of this matrix;

```
A_C_N.transpose()
```

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (6.8)$$

Notably, the inverse and the transpose are the same here. This indicates that this matrix is a special [orthogonal matrix](#). All matrices that describe the orientation between reference frames are orthogonal matrices. Following the notation convention, this holds:

$${}^N\mathbf{C}^A = ({}^A\mathbf{C}^N)^{-1} = ({}^A\mathbf{C}^N)^T \quad (6.9)$$

Exercise

Write ${}^A\mathbf{C}^N$ for simple rotations about both the shared \hat{n}_x and \hat{a}_x and shared \hat{n}_y and \hat{a}_y axes, rotating A with respect to N through angle θ .

Solution

For a x orientation:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & \sin \theta \\ 0 & -\sin \theta & \cos \theta \end{bmatrix} \quad (6.10)$$

For a y orientation:

$$\begin{bmatrix} \cos \theta & 0 & -\sin \theta \\ 0 & 1 & 0 \\ \sin \theta & 0 & \cos \theta \end{bmatrix} \quad (6.11)$$

6.5 Direction Cosine Matrices

If now A is oriented relative to N and the pairwise angles between each \hat{a} and \hat{n} mutually perpendicular unit vectors are measured, a matrix for an arbitrary orientation can be defined. For example, the figure below shows the three angles $\alpha_{xx}, \alpha_{xy}, \alpha_{xz}$ relating \hat{a}_x to each \hat{n} unit vector.

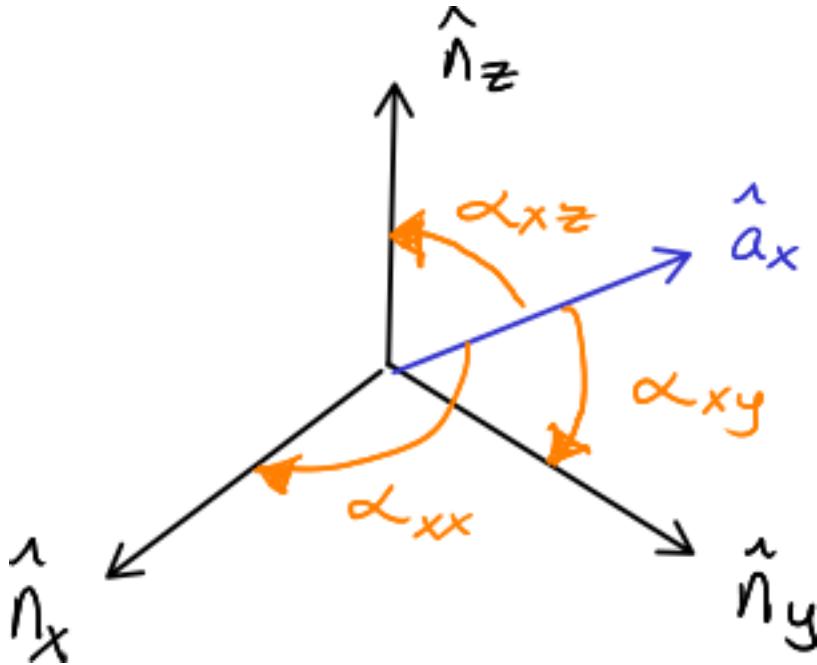


Fig. 6.3: Three angles relating \hat{a}_x to the unit vectors of N .

Similar to the simple example above, we can write these equations if the α_y and α_z angles relate the \hat{a}_y and \hat{a}_z unit vectors to those of N :

$$\begin{aligned}\hat{a}_x &= \cos \alpha_{xx} \hat{n}_x + \cos \alpha_{xy} \hat{n}_y + \cos \alpha_{xz} \hat{n}_z \\ \hat{a}_y &= \cos \alpha_{yx} \hat{n}_x + \cos \alpha_{yy} \hat{n}_y + \cos \alpha_{yz} \hat{n}_z \\ \hat{a}_z &= \cos \alpha_{zx} \hat{n}_x + \cos \alpha_{zy} \hat{n}_y + \cos \alpha_{zz} \hat{n}_z\end{aligned}\quad (6.12)$$

Since we are working with unit vectors the cosine of the angle between each pair of vectors is equivalent to the dot product between the two vectors, so this also holds:

$$\begin{aligned}\hat{a}_x &= (\hat{a}_x \cdot \hat{n}_x) \hat{n}_x + (\hat{a}_x \cdot \hat{n}_y) \hat{n}_y + (\hat{a}_x \cdot \hat{n}_z) \hat{n}_z \\ \hat{a}_y &= (\hat{a}_y \cdot \hat{n}_x) \hat{n}_x + (\hat{a}_y \cdot \hat{n}_y) \hat{n}_y + (\hat{a}_y \cdot \hat{n}_z) \hat{n}_z \\ \hat{a}_z &= (\hat{a}_z \cdot \hat{n}_x) \hat{n}_x + (\hat{a}_z \cdot \hat{n}_y) \hat{n}_y + (\hat{a}_z \cdot \hat{n}_z) \hat{n}_z\end{aligned}\quad (6.13)$$

Now the matrix relating the orientation of A with respect to N can be formed:

$$\begin{bmatrix} \hat{a}_x \\ \hat{a}_y \\ \hat{a}_z \end{bmatrix} = {}^A \mathbf{C}^N \begin{bmatrix} \hat{n}_x \\ \hat{n}_y \\ \hat{n}_z \end{bmatrix} \quad (6.14)$$

where

$${}^A \mathbf{C}^N = \begin{bmatrix} \hat{a}_x \cdot \hat{n}_x & \hat{a}_x \cdot \hat{n}_y & \hat{a}_x \cdot \hat{n}_z \\ \hat{a}_y \cdot \hat{n}_x & \hat{a}_y \cdot \hat{n}_y & \hat{a}_y \cdot \hat{n}_z \\ \hat{a}_z \cdot \hat{n}_x & \hat{a}_z \cdot \hat{n}_y & \hat{a}_z \cdot \hat{n}_z \end{bmatrix} \quad (6.15)$$

We call ${}^A\mathbf{C}^N$ the “direction cosine matrix” as a general description of the relative orientation of two reference frames. This matrix uniquely defines the relative orientation between reference frames N and A , it is invertible, and its inverse is equal to the transpose, as shown above in the simple example. The determinant of the matrix is also always 1, to ensure both associated frames are right-handed. The direction cosine matrix found in the prior section for a simple orientation is a specific case of this more general definition. The direction cosine matrix is also referred to as a “rotation matrix” or “orientation matrix” in some texts.

6.6 Successive Orientations

Successive orientations of a series of reference frames provides a convenient way to manage orientation among more than a single pair. Below, an additional auxiliary reference frame B is shown that is simply oriented with respect to A in the same way that A is from N above in the prior section.

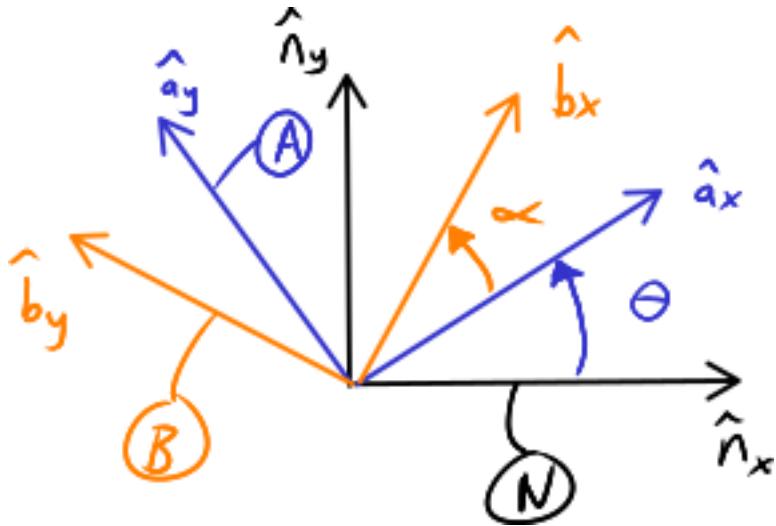


Fig. 6.4: Two successive simple orientations through angles θ and then α for frames A and B , respectively.

We know from the prior sections that we can define these two relationships between each pair of reference frames as follows:

$$\begin{bmatrix} \hat{a}_x \\ \hat{a}_y \\ \hat{a}_z \end{bmatrix} = {}^A\mathbf{C}^N \begin{bmatrix} \hat{n}_x \\ \hat{n}_y \\ \hat{n}_z \end{bmatrix} \quad (6.16)$$

$$\begin{bmatrix} \hat{b}_x \\ \hat{b}_y \\ \hat{b}_z \end{bmatrix} = {}^B\mathbf{C}^A \begin{bmatrix} \hat{a}_x \\ \hat{a}_y \\ \hat{a}_z \end{bmatrix} \quad (6.17)$$

Now, substitute (6.16) into (6.17) to get:

$$\begin{bmatrix} \hat{b}_x \\ \hat{b}_y \\ \hat{b}_z \end{bmatrix} = {}^B\mathbf{C}^{AA} \mathbf{C}^N \begin{bmatrix} \hat{n}_x \\ \hat{n}_y \\ \hat{n}_z \end{bmatrix} \quad (6.18)$$

showing that the direction cosine matrix between B and N results from matrix multiplying the intermediate direction cosine matrices.

$${}^B\mathbf{C}^N = {}^B\mathbf{C}^{AA} \mathbf{C}^N \quad (6.19)$$

This holds for any series of general three dimensional successive orientations and the relation is shown in the following theorem:

$${}^Z \mathbf{C}^A = {}^Z \mathbf{C}^{YY} \mathbf{C}^X \dots {}^C \mathbf{C}^{BB} \mathbf{C}^A \quad (6.20)$$

where frames A through Z are successively oriented.

Using Fig. 6.4 as an explicit example of this property, we start with the already defined ${}^A \mathbf{C}^N$:

A_C_N

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (6.21)$$

${}^B \mathbf{C}^A$ can then be defined similarly:

```
alpha = sm.symbols('alpha')

B_C_A = sm.Matrix([[sm.cos(alpha), sm.sin(alpha), 0],
                    [-sm.sin(alpha), sm.cos(alpha), 0],
                    [0, 0, 1]])

B_C_A
```

$$\begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (6.22)$$

Finally, ${}^B \mathbf{C}^N$ can be found by matrix multiplication:

$B_C_N = B_C_A * A_C_N$

$$\begin{bmatrix} -\sin(\alpha)\sin(\theta) + \cos(\alpha)\cos(\theta) & \sin(\alpha)\cos(\theta) + \sin(\theta)\cos(\alpha) & 0 \\ -\sin(\alpha)\cos(\theta) - \sin(\theta)\cos(\alpha) & -\sin(\alpha)\sin(\theta) + \cos(\alpha)\cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (6.23)$$

Simplifying these trigonometric expressions shows the expected result:

$sm.trigsimp(B_C_N)$

$$\begin{bmatrix} \cos(\alpha + \theta) & \sin(\alpha + \theta) & 0 \\ -\sin(\alpha + \theta) & \cos(\alpha + \theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (6.24)$$

Exercise

If you are given ${}^B \mathbf{C}^N$ and ${}^A \mathbf{C}^N$ from the prior example, how would you find ${}^A \mathbf{C}^B$?

Solution

$$\begin{aligned} {}^B\mathbf{C}^N &= {}^B\mathbf{C}^{AA}\mathbf{C}^N \\ {}^A\mathbf{C}^B &= \left({}^B\mathbf{C}^N ({}^A\mathbf{C}^N)^T \right)^T = {}^A\mathbf{C}^N ({}^B\mathbf{C}^N)^T \end{aligned} \quad (6.25)$$

6.7 SymPy Mechanics

As shown above, SymPy nicely handles the formulation of direction cosine matrices, but SymPy also offers a more useful tool for tracking orientation among reference frames. The `sympy.physics.mechanics` module includes numerous objects and functions that ease the bookkeeping and mental models needed to manage various aspects of multibody dynamics. We will import the module as in this text:

```
import sympy.physics.mechanics as me
```

```
class ReferenceFrame(me.ReferenceFrame):
    def __init__(self, *args, **kwargs):
        kwargs.pop('latexs', None)
        lab = args[0].lower()
        tex = r'\hat{{{}}}_{{{}}}'
        super(ReferenceFrame, self).__init__(*args,
                                             latex=(tex.format(lab, 'x'),
                                                    tex.format(lab, 'y'),
                                                    tex.format(lab, 'z')),
                                             **kwargs)
me.ReferenceFrame = ReferenceFrame
```

`sympy.physics.mechanics` includes a way to define and orient reference frames. To create a reference frame, use `ReferenceFrame` and provide a name for your frame as a string.

```
N = me.ReferenceFrame('N')
```

The right-handed mutually perpendicular unit vectors associated with a reference frame are accessed with the attributes `.x`, `.y`, and `.z`, like so:

```
N.x, N.y, N.z
```

$$(\hat{n}_x, \hat{n}_y, \hat{n}_z) \quad (6.26)$$

Using Fig. 6.4 again as an example, we can define all three reference frames by additionally creating *A* and *B*:

```
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')

N, A, B
```

(N, A, B)

We have already defined the direction cosine matrices for these two successive orientations. For example:

A_C_N

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (6.27)$$

relates A and N . `ReferenceFrame` objects can be oriented with respect to one another. The `orient_explicit()` method allows you to set the direction cosine matrix between two frames explicitly:

N.orient_explicit(A, A_C_N)

Warning: Note very carefully what version of the direction cosine matrix you pass to `.orient_explicit()`. Check its docstring with `N.orient_explicit?`.

Now you can ask for the direction cosine matrix of A with respect to N , i.e. ${}^A\mathbf{C}^N$, using the `dcm()` method:

A.dcm(N)

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (6.28)$$

The direction cosine matrix of N with respect to A is found by reversing the order of the arguments:

N.dcm(A)

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (6.29)$$

Exercise

Orient reference frame D with respect to F with a simple rotation about y through angle β and set this orientation with `orient_explicit()`.

Solution

```
beta = sm.symbols('beta')
D = me.ReferenceFrame('D')
F = me.ReferenceFrame('F')
```

(continues on next page)

(continued from previous page)

```
F_C_D = sm.Matrix([[sm.cos(beta), 0, -sm.sin(beta)],
                   [0, 1, 0],
                   [sm.sin(beta), 0, sm.cos(beta)]])

F.orient_explicit(D, F_C_D.transpose())

F.dcm(D)
```

$$\begin{bmatrix} \cos(\beta) & 0 & -\sin(\beta) \\ 0 & 1 & 0 \\ \sin(\beta) & 0 & \cos(\beta) \end{bmatrix} \quad (6.30)$$

`orient_explicit()` requires you to form the direction cosine matrix yourself, but there are also methods that relieve you of that necessity. For example, `orient_axis()` allows you to define simple orientations between reference frames more naturally. You provide the frame to orient from, the angle to orient through, and the vector to orient about and the correct direction cosine matrix will be formed. As an example, orient *B* with respect to *A* through α about \hat{a}_z by:

```
B.orient_axis(A, alpha, A.z)
```

Now the direction cosine matrix is automatically calculated and is returned with the `.dcm()` method:

```
B.dcm(A)
```

$$\begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (6.31)$$

The inverse is also defined on *A*:

```
A.dcm(B)
```

$$\begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (6.32)$$

So each pair of reference frames are aware of its orientation partner (or partners).

Now that we've established orientations between *N* and *A* and *A* and *B*, we might want to know the relationships between *B* and *N*. Remember that matrix multiplication of the two successive direction cosine matrices provides the answer:

```
sm.trigsimp(B.dcm(A) * A.dcm(N))
```

$$\begin{bmatrix} \cos(\alpha + \theta) & \sin(\alpha + \theta) & 0 \\ -\sin(\alpha + \theta) & \cos(\alpha + \theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (6.33)$$

But, the answer can also be found by calling `dcm()` with just the two reference frames in question, *B* and *N*. As long as there is a successive path of intermediate, or auxiliary, orientations between the two reference frames, this is sufficient for obtaining the desired direction cosine matrix and the matrix multiplication is handled internally for you:

```
sm.trigsimp(B.dcm(N))
```

$$\begin{bmatrix} \cos(\alpha + \theta) & \sin(\alpha + \theta) & 0 \\ -\sin(\alpha + \theta) & \cos(\alpha + \theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (6.34)$$

Lastly, recall the general definition of the direction cosine matrix. We showed that the dot product of pairs of unit vectors give the entries to the direction cosine matrix. `mechanics` has a `dot()` function that can calculate the dot product of two vectors. Using it on two of the unit vector pairs returns the expected direction cosine matrix entry:

```
sm.trigsimp(me.dot(B.x, N.x))
```

$$\cos(\alpha + \theta) \quad (6.35)$$

Exercise

Orient reference frame D with respect to C with a simple rotation through angle β about the shared $-y$ axis. Use the direction cosine matrix from this first orientation to set the orientation of reference frame E with respect to D . Show that both pairs of reference frames have the same relative orientations.

Solution

```
beta = sm.symbols('beta')

C = me.ReferenceFrame('C')
D = me.ReferenceFrame('D')
E = me.ReferenceFrame('E')

D.orient_axis(C, beta, -C.y)

D.dcm(C)
```

$$\begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{bmatrix} \quad (6.36)$$

```
E.orient_explicit(D, C.dcm(D))
E.dcm(D)
```

$$\begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{bmatrix} \quad (6.37)$$

6.8 Euler Angles

The camera stabilization gimbal shown in Fig. 6.5 has three revolute joints that orient the camera D relative to the handgrip frame A .

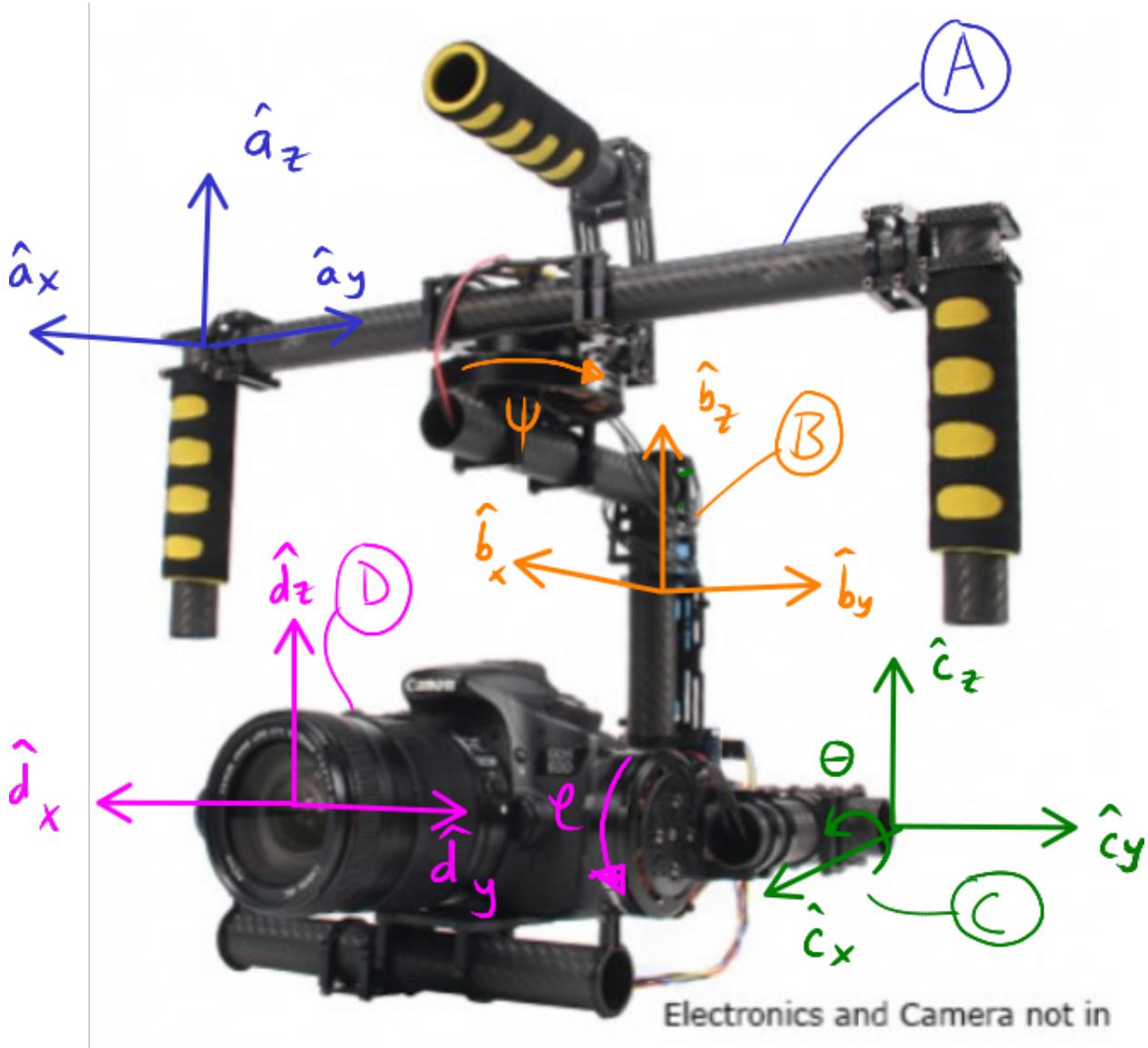


Fig. 6.5: Four reference frames labeled on the Turnigy Pro Steady Hand Camera Gimbal. *Image copyright HobbyKing, used under fair use for educational purposes.*

If we introduce two additional auxiliary reference frames, B and C , attached to the intermediate camera frame members, we can use three successive simple orientations to go from A to D . We can formulate the direction cosine matrices for the reference frames using the same technique for the successive simple orientations shown in [Successive Orientations](#), but now our sequence of three orientations will enable us to orient D in any way possible relative to A in three dimensional space.

Watch this video to get a sense of the orientation axes for each intermediate auxiliary reference frame:

We first orient B with respect to A about the shared z unit vector through the angle ψ , as shown below:

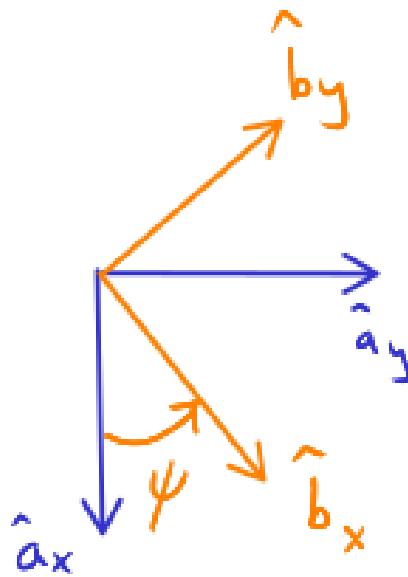


Fig. 6.6: View of the A and B x - y plane showing the orientation of B relative to A about z through angle ψ .

In SymPy, use `ReferenceFrame` to establish the relative orientation:

```
psi = sm.symbols('psi')

A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')

B.orient_axis(A, psi, A.z)

B.dcm(A)
```

$$\begin{bmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (6.38)$$

Now orient C with respect to B about their shared x unit vector through angle θ .

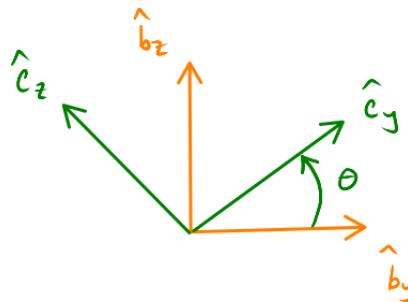


Fig. 6.7: View of the B and C y - z plane showing the orientation of C relative to B about x through angle θ .

```
theta = sm.symbols('theta')
```

(continues on next page)

(continued from previous page)

```
C = me.ReferenceFrame('C')
C.orient_axis(B, theta, B.x)
C.dcm(B)
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) \\ 0 & -\sin(\theta) & \cos(\theta) \end{bmatrix} \quad (6.39)$$

Finally, orient the camera D with respect to C about their shared y unit vector through the angle ϕ .

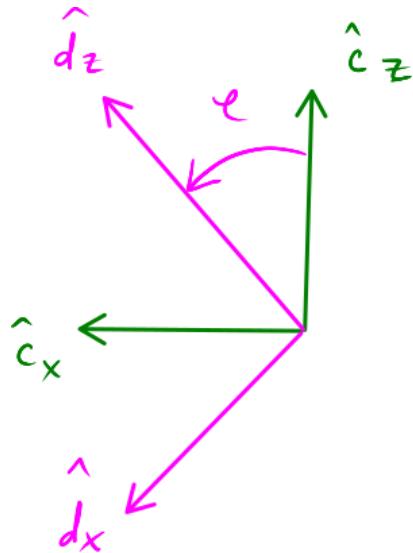


Fig. 6.8: View of the C and D x - z plane showing the orientation of D relative to C about y through angle φ .

```
phi = sm.symbols('varphi')
D = me.ReferenceFrame('D')
D.orient_axis(C, phi, C.y)
D.dcm(C)
```

$$\begin{bmatrix} \cos(\varphi) & 0 & -\sin(\varphi) \\ 0 & 1 & 0 \\ \sin(\varphi) & 0 & \cos(\varphi) \end{bmatrix} \quad (6.40)$$

With all of the intermediate orientations defined, when can now ask for the relationship ${}^D\mathbf{C}^A$ of the camera D relative to the handgrip frame A :

```
D.dcm(A)
```

$$\begin{bmatrix} -\sin(\psi)\sin(\theta)\sin(\varphi) + \cos(\psi)\cos(\varphi) & \sin(\psi)\cos(\varphi) + \sin(\theta)\sin(\varphi)\cos(\psi) & -\sin(\varphi)\cos(\theta) \\ -\sin(\psi)\cos(\theta) & \cos(\psi)\cos(\theta) & \sin(\theta) \\ \sin(\psi)\sin(\theta)\cos(\varphi) + \sin(\varphi)\cos(\psi) & \sin(\psi)\sin(\varphi) - \sin(\theta)\cos(\psi)\cos(\varphi) & \cos(\theta)\cos(\varphi) \end{bmatrix} \quad (6.41)$$

With these three successive orientations the camera can be rotated arbitrarily relative to the handgrip frame. These successive z - x - y orientations are a standard way of describing the orientation of two reference frames and are referred to as [Euler Angles](#)¹.

There are 12 valid sets of successive orientations that can arbitrarily orient one reference frame with respect to another. These are the six “Proper Euler Angles”:

$$z\text{-}x\text{-}z, x\text{-}y\text{-}x, y\text{-}z\text{-}y, z\text{-}y\text{-}z, x\text{-}z\text{-}x, y\text{-}x\text{-}y \quad (6.42)$$

and the six “Tait-Bryan Angles”:

$$x\text{-}y\text{-}z, y\text{-}z\text{-}x, z\text{-}x\text{-}y, x\text{-}z\text{-}y, z\text{-}y\text{-}x, y\text{-}x\text{-}z \quad (6.43)$$

Different sets can be more or less suitable for the kinematic nature of the system you are describing. We will also refer to these 12 possible orientation sets as “body fixed orientations”. As we will soon see, a rigid body and a reference frame are synonymous from an orientation perspective and each successive orientation rotates about a shared unit vector fixed in both of the reference frames (or bodies), thus “body fixed orientations”. The method `orient_body_fixed()` can be used to establish the relationship between A and D without the need to create auxiliary reference frames B and C :

```
A = me.ReferenceFrame('A')
D = me.ReferenceFrame('D')

D.orient_body_fixed(A, (psi, theta, phi), 'zxy')

D.dcm(A)
```

$$\begin{bmatrix} -\sin(\psi)\sin(\theta)\sin(\varphi) + \cos(\psi)\cos(\varphi) & \sin(\psi)\cos(\varphi) + \sin(\theta)\sin(\varphi)\cos(\psi) & -\sin(\varphi)\cos(\theta) \\ -\sin(\psi)\cos(\theta) & \cos(\psi)\cos(\theta) & \sin(\theta) \\ \sin(\psi)\sin(\theta)\cos(\varphi) + \sin(\varphi)\cos(\psi) & \sin(\psi)\sin(\varphi) - \sin(\theta)\cos(\psi)\cos(\varphi) & \cos(\theta)\cos(\varphi) \end{bmatrix} \quad (6.44)$$

Exercise

[Euler](#) discovered 6 of the 12 orientation sets. One of these sets is shown in this figure:

¹ Technically, this set of angles for the gimbal are one of the 6 Tait-Bryan angles, but “Euler Angles” is used as a general term to describe both Tait-Bryan angles and “proper Euler angles”.

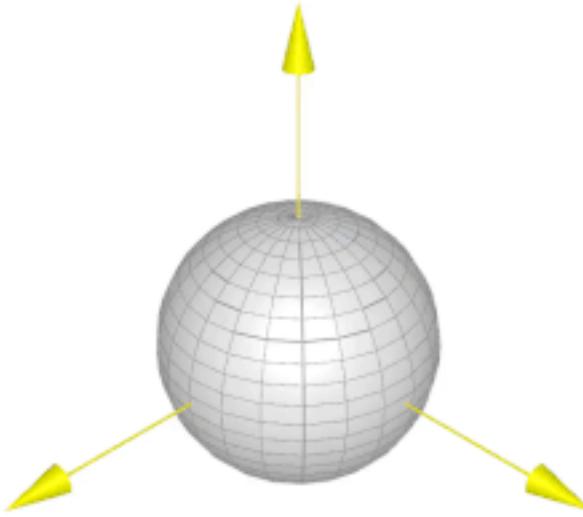


Fig. 6.9: An orientation through Euler angles with frame A (yellow), B (red), C (green), and D (blue). The rightward yellow arrow is the x direction, leftward yellow arrow is the y direction, and upward yellow arrow is the z direction. All frames' unit vectors are aligned before being oriented.

Take the acute angles between A and B to be ψ , B and C to be θ , and C and D to be φ . Determine what Euler angle set this is and then calculate ${}^D\mathbf{C}^A$ using `orient_axis()` and then with `orient_body_fixed()` showing that you get the same result.

Solution

The Euler angle set is z - x - z .

```
psi, theta, phi = sm.symbols('psi, theta, varphi')
```

With `orient_axis()`:

```
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
C = me.ReferenceFrame('C')
D = me.ReferenceFrame('D')

B.orient_axis(A, psi, A.z)
C.orient_axis(B, theta, B.x)
D.orient_axis(C, phi, C.z)
```

(continues on next page)

(continued from previous page)

D.dcm(A)

$$\begin{bmatrix} -\sin(\psi)\sin(\varphi)\cos(\theta) + \cos(\psi)\cos(\varphi) & \sin(\psi)\cos(\varphi) + \sin(\varphi)\cos(\psi)\cos(\theta) & \sin(\theta)\sin(\varphi) \\ -\sin(\psi)\cos(\theta)\cos(\varphi) - \sin(\varphi)\cos(\psi) & -\sin(\psi)\sin(\varphi) + \cos(\psi)\cos(\theta)\cos(\varphi) & \sin(\theta)\cos(\varphi) \\ \sin(\psi)\sin(\theta) & -\sin(\theta)\cos(\psi) & \cos(\theta) \end{bmatrix} \quad (6.45)$$

With `orient_body_fixed()`:

```
A = me.ReferenceFrame('A')
D = me.ReferenceFrame('D')

D.orient_body_fixed(A, (psi, theta, phi), 'zxz')

D.dcm(A)
```

$$\begin{bmatrix} -\sin(\psi)\sin(\varphi)\cos(\theta) + \cos(\psi)\cos(\varphi) & \sin(\psi)\cos(\varphi) + \sin(\varphi)\cos(\psi)\cos(\theta) & \sin(\theta)\sin(\varphi) \\ -\sin(\psi)\cos(\theta)\cos(\varphi) - \sin(\varphi)\cos(\psi) & -\sin(\psi)\sin(\varphi) + \cos(\psi)\cos(\theta)\cos(\varphi) & \sin(\theta)\cos(\varphi) \\ \sin(\psi)\sin(\theta) & -\sin(\theta)\cos(\psi) & \cos(\theta) \end{bmatrix} \quad (6.46)$$

6.9 Alternatives for Representing Orientation

In the previous section, Euler-angles were used to encode the orientation of a frame or body. There are many alternative approaches to representing orientations. Three such representations, which will be used throughout this book, were already introduced:

- **Euler-angles** themselves, which provides a minimal representation (only 3 numbers), and a relatively straightforward way to compute the change in orientation from the angular velocity (see [Angular Kinematics](#)).
- the **direction cosine matrix**, which allow easy rotations or vectors and consecutive rotations, both via matrix multiplication,
- the **axis-angle representation** (used in the `orient_axis()` method), which is often an intuitive way to describe the orientation for manual input, and is useful when the axis of rotation is fixed.

Each representation also has downsides. For example, the direction cosine matrix consists of nine elements; more to keep track of than three Euler angles. Furthermore, not all combinations of nine elements form a valid direction cosine matrix, so we have to be careful to check and enforce validity when writing code.

6.10 Learn more

One more frequently used approach to representing orientations is based on so called **quaternions**. Quaternions are like imaginary numbers, but with three imaginary constants: i , j and k . These act as described by the rule

$$i^2 = j^2 = k^2 = ijk = -1. \quad (6.47)$$

A general quaternion q can thus be written in terms of its components q_0, q_i, q_j, q_k which are real numbers:

$$q = q_0 + q_i i + q_j j + q_k k \quad (6.48)$$

The `orient_quaternion()` method enables orienting a reference frame using a quaternion in sympy:

```
N = me.ReferenceFrame('N')
A = me.ReferenceFrame('A')

q_0, q_i, q_j, q_k = sm.symbols('q_0 q_i q_j q_k')
q = (q_0, q_i, q_j, q_k)
A.orient_quaternion(N, q)
A.dcm(N)
```

$$\begin{bmatrix} q_0^2 + q_i^2 - q_j^2 - q_k^2 & 2q_0q_k + 2q_iq_j & -2q_0q_j + 2q_iq_k \\ -2q_0q_k + 2q_iq_j & q_0^2 - q_i^2 + q_j^2 - q_k^2 & 2q_0q_i + 2q_jq_k \\ 2q_0q_j + 2q_iq_k & -2q_0q_i + 2q_jq_k & q_0^2 - q_i^2 - q_j^2 + q_k^2 \end{bmatrix} \quad (6.49)$$

A rotation of an angle θ around a unit vector \hat{e} can be converted to a quaternion representation by having $q_0 = \cos(\frac{\theta}{2})$, and the other components equal to a factor $\sin(\frac{\theta}{2})$ times the components of the axis of rotation \hat{e} . For example, if the rotation axis is \hat{n}_x , we get:

```
q = (sm.cos(theta/2), sm.sin(theta/2), 0, 0)
A.orient_quaternion(N, q)
sm.trigsimp(A.dcm(N))
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) \\ 0 & -\sin(\theta) & \cos(\theta) \end{bmatrix} \quad (6.50)$$

The length of a quaternion is the square root of the sum of the squares of its components. For a quaternion representing an orientation, this length must always be 1.

It turns out that the multiplication rules for (unit) quaternions provide an efficient way to compose multiple rotations, and to numerically integrate the orientation when given an angular velocity. Due to the interpretation related to the angle and axis representation, it is also a somewhat intuitive representation. However, the integration algorithm needs to take an additional step to ensure the quaternion always has unit length.

The representation of orientations in general, turns out to be related to an area of mathematics called Lie-groups. The theory of Lie-groups has further applications to the mechanics and control of multibody systems. An example application is finding a general method for simplifying the equations for symmetric systems, so this can be done more easily and to more systems. The Lie-group theory is not used in this book. Instead, the interested reader can look up the [3D rotation group](#) as a starting point for further study.

VECTORS

Note: You can download this example as a Python script: `vectors.py` or Jupyter Notebook: `vectors.ipynb`.

```
import sympy as sm
import sympy.physics.mechanics as me
sm.init_printing(use_latex='mathjax')

class ReferenceFrame(me.ReferenceFrame):

    def __init__(self, *args, **kwargs):
        kwargs.pop('latexs', None)

        lab = args[0].lower()
        tex = r'\hat{{}}_{{}}_{{}}'

        super(ReferenceFrame, self).__init__(*args,
                                             latexs=(tex.format(lab, 'x'),
                                                      tex.format(lab, 'y'),
                                                      tex.format(lab, 'z')),
                                             **kwargs)
me.ReferenceFrame = ReferenceFrame
```

7.1 Learning Objectives

After completing this chapter readers will be able to:

- State the properties of a vectors.
- Determine what scalars a vector is a function of.
- Add, subtract, scale, negate, normalize vectors.
- Dot and cross vectors with each other.
- Express vectors in different reference frames.
- Define vectors with components expressed in different reference frames.
- Create position vectors between points using reference frame unit vectors.

7.2 What is a vector?

Vectors have three characteristics:

1. magnitude
2. orientation
3. sense

The direction the vector points is derived from both the orientation and the sense. Vectors are equal when all three characteristics are the same.

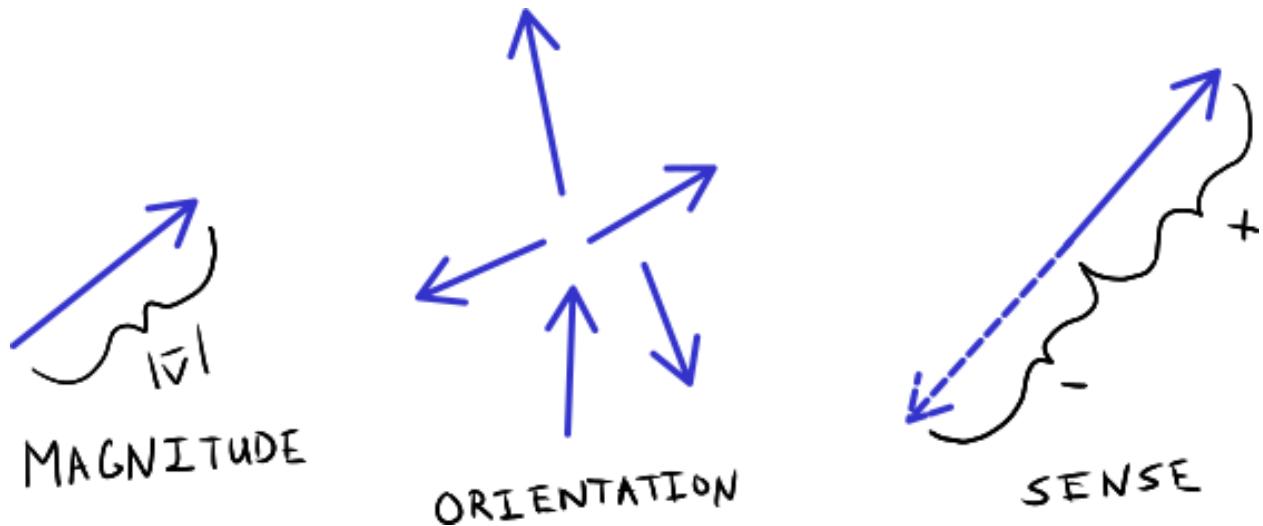


Fig. 7.1: Three characteristics of vectors: magnitude, orientation, and sense.

Note: In this text we will distinguish scalar variables, e.g. v , from vectors by including a bar over the top of the symbol, e.g. \bar{v} . Vectors will be drawn as follows:

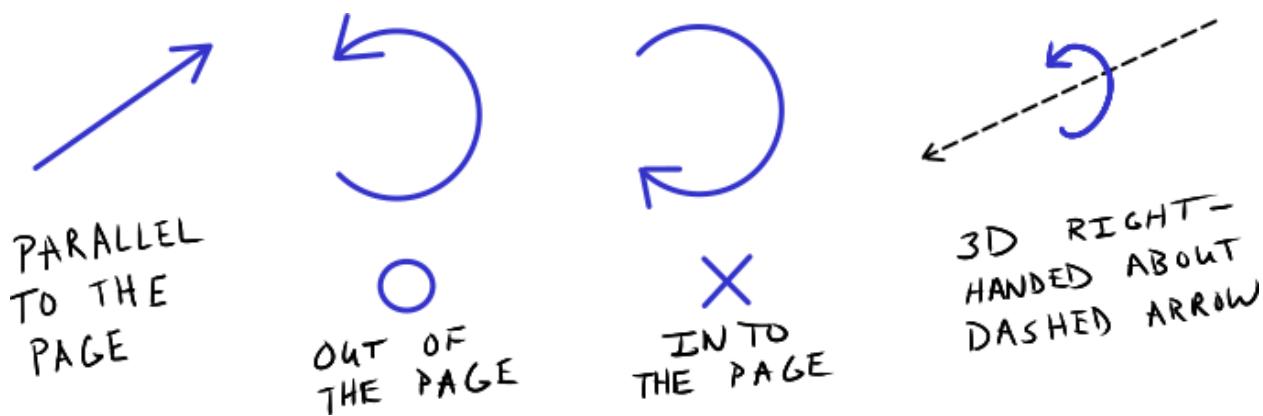


Fig. 7.2: Various ways vectors will be drawn in figures.



Fig. 7.3: See [right-hand rule](#) for a refresher on right handed systems.

Right_hand_rule_simple.png: The original uploader was Schorsch2 at German Wikipedia.derivative work: Wizard191, Public domain, via Wikimedia Commons

Vectors have these mathematical properties:

- scalar multiplicative: $\bar{v} = \lambda \bar{u}$ where λ can only change the magnitude and the sense of the vector, i.e. \bar{v} and \bar{u} have the same orientation
- commutative: $\bar{u} + \bar{v} = \bar{v} + \bar{u}$
- distributive: $\lambda(\bar{u} + \bar{v}) = \lambda\bar{u} + \lambda\bar{v}$
- associative: $(\bar{u} + \bar{v}) + \bar{w} = \bar{u} + (\bar{v} + \bar{w})$

Unit vectors are vectors with a magnitude of 1. If the magnitude of \bar{v} is 1, then we indicate this with \hat{v} . Any vector has an associated unit vector with the same orientation and sense, found by:

$$\hat{u} = \frac{\bar{u}}{|\bar{u}|} \quad (7.1)$$

where $|\bar{u}|$ is the [Euclidean norm](#) (2-norm), or magnitude, of the vector \bar{u} .

7.3 Vector Functions

Vectors can be functions of scalar variables. If a change in scalar variable q changes the magnitude and/or direction of \bar{v} when observed from A , \bar{v} is a vector function of q in A . It is possible that \bar{v} may not be a vector function of scalar variable q when observed from another reference frame, i.e. the function dependency of a vector on a scalar depends on the reference frame it is observed from.

Let vector \bar{v} be a function of n scalars q_1, q_2, \dots, q_n in A . If we introduce $\hat{a}_x, \hat{a}_y, \hat{a}_z$ as a set of mutually perpendicular unit vectors fixed in A , then these unit vectors are constant when observed from A . There are then three unique scalar

functions v_x, v_y, v_z of q_1, q_2, \dots, q_n such that:

$$\bar{v} = v_x \hat{a}_x + v_y \hat{a}_y + v_z \hat{a}_z \quad (7.2)$$

$v_x \hat{a}_x$ is called the \hat{a}_x component of \bar{v} and v_x is called measure number of \bar{v} . Since the components are mutually perpendicular the measure number can also be found from the dot product of \bar{v} and the respective unit vector:

$$\bar{v} = (\bar{v} \cdot \hat{a}_x) \hat{a}_x + (\bar{v} \cdot \hat{a}_y) \hat{a}_y + (\bar{v} \cdot \hat{a}_z) \hat{a}_z \quad (7.3)$$

which is the projection of \bar{v} onto each unit vector. When written this way we can say that \bar{v} is expressed in A . See sections 1.1-1.3 in [Kane1985] for a more general explanation.

7.4 Addition

When we add vector \bar{b} to vector \bar{a} , the result is a vector that starts at the tail of \bar{a} and ends at the tip of \bar{b} :

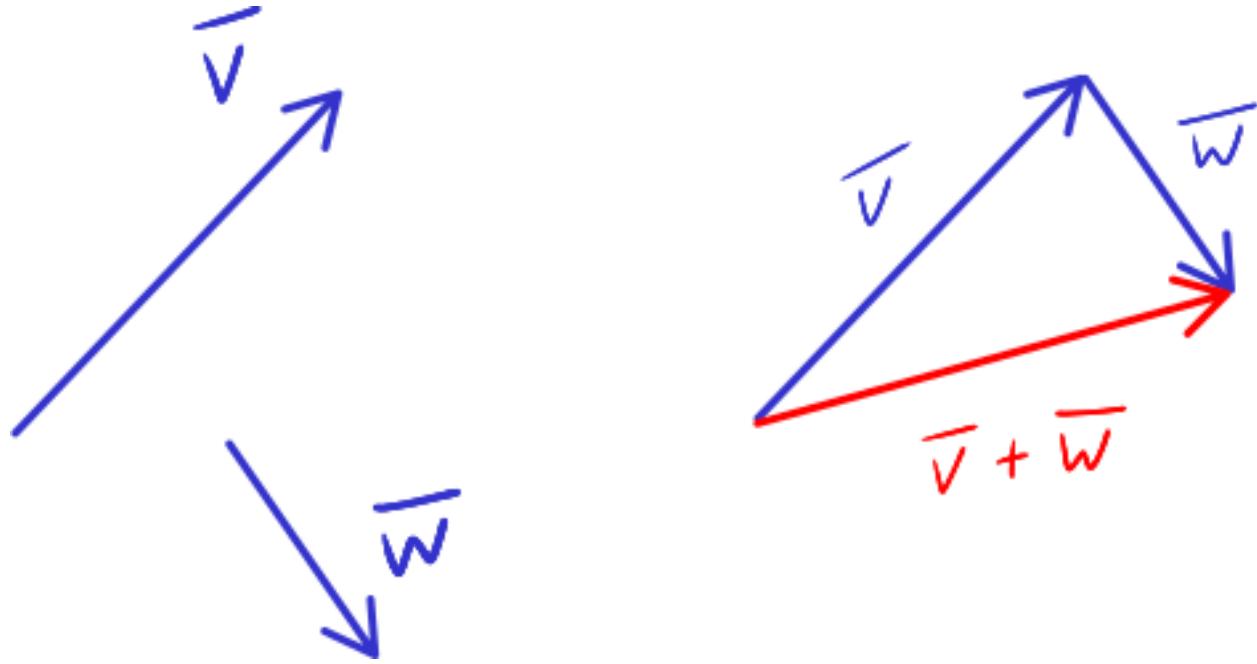


Fig. 7.4: Graphical vector addition

Vectors in SymPy Mechanics are created by first introducing a reference frame and using its associated unit vectors to construct vectors of arbitrary magnitude and direction.

```
N = me.ReferenceFrame('N')
```

Now introduce some scalar variables:

```
a, b, c, d, e, f = sm.symbols('a, b, c, d, e, f')
```

The simplest 3D non-unit vector is made up of a single component:

```
v = a*N.x
```

$$a\hat{n}_x \quad (7.4)$$

A, possible more familiar, column matrix form of a vector is accessed with the `to_matrix()`.

```
v.to_matrix(N)
```

$$\begin{bmatrix} a \\ 0 \\ 0 \end{bmatrix} \quad (7.5)$$

Fully 3D and arbitrary vectors can be created by providing a measure number for each unit vector of N :

```
w = a*N.x + b*N.y + c*N.z
w
```

$$a\hat{n}_x + b\hat{n}_y + c\hat{n}_z \quad (7.6)$$

And the associated column matrix form:

```
w.to_matrix(N)
```

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \quad (7.7)$$

Vector addition works by adding the measure numbers of each common component:

$$\begin{aligned} \bar{w} &= a\hat{n}_x + b\hat{n}_y + c\hat{n}_z \\ \bar{x} &= d\hat{n}_x + e\hat{n}_y + f\hat{n}_z \\ \bar{w} + \bar{x} &= (a+d)\hat{n}_x + (b+e)\hat{n}_y + (c+f)\hat{n}_z \end{aligned} \quad (7.8)$$

Sympy Mechanics vectors work as expected:

```
x = d*N.x + e*N.y + f*N.z
x
```

$$d\hat{n}_x + e\hat{n}_y + f\hat{n}_z \quad (7.9)$$

```
w + x
```

$$(a+d)\hat{n}_x + (b+e)\hat{n}_y + (c+f)\hat{n}_z \quad (7.10)$$

7.5 Scaling

Multiplying a vector by a scalar changes its magnitude, but not its orientation. Scaling by a negative number changes a vector's magnitude and reverses its sense (rotates it by π radians).

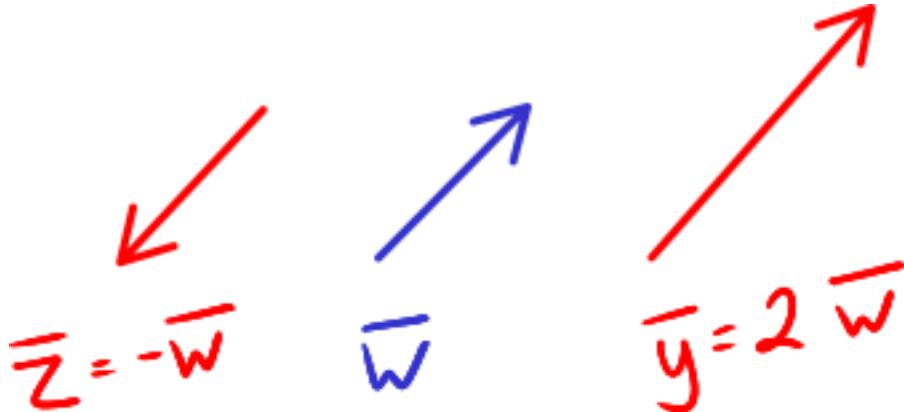


Fig. 7.5: Vector scaling

$$\begin{aligned} y &= 2 * w \\ y \end{aligned}$$

$$2a\hat{n}_x + 2b\hat{n}_y + 2c\hat{n}_z \quad (7.11)$$

$$\begin{aligned} z &= -w \\ z \end{aligned}$$

$$-a\hat{n}_x - b\hat{n}_y - c\hat{n}_z \quad (7.12)$$

Exercise

Create three vectors that lie in the xy plane of reference frame N where each vector is:

1. of length l that is at an angle of $\frac{\pi}{4}$ degrees from the \hat{n}_x unit vector.
2. of length 10 and is in the $-\hat{n}_y$ direction
3. of length l and is θ radians from the \hat{n}_y unit vector.

Finally, add vectors from 1 and 2 and subtract 5 times the third vector.

Hint: SymPy has fundamental constants and trigonometric functions, for example `sm.tan`, `sm.pi`.

Solution

```
N = me.ReferenceFrame('N')
l, theta = sm.symbols('l, theta')
```

```
v1 = l*sm.cos(sm.pi/4)*N.x + l*sm.sin(sm.pi/4)*N.y
v1
```

$$\frac{\sqrt{2}l}{2}\hat{n}_x + \frac{\sqrt{2}l}{2}\hat{n}_y \quad (7.13)$$

```
v2 = -10*N.y
v2
```

$$-10\hat{n}_y \quad (7.14)$$

```
v3 = -l*sm.sin(theta)*N.x + l*sm.cos(theta)*N.y
v3
```

$$-l \sin(\theta)\hat{n}_x + l \cos(\theta)\hat{n}_y \quad (7.15)$$

```
v1 + v2 - 5*v3
```

$$(5l \sin(\theta) + \frac{\sqrt{2}l}{2})\hat{n}_x + (-5l \cos(\theta) + \frac{\sqrt{2}l}{2} - 10)\hat{n}_y \quad (7.16)$$

7.6 Dot Product

The [dot product](#), which yields a scalar quantity, is defined as:

$$\bar{v} \cdot \bar{w} = |\bar{v}||\bar{w}| \cos \theta \quad (7.17)$$

where θ is the angle between the two vectors. For arbitrary measure numbers this results in the following:

$$\begin{aligned} \bar{v} &= v_x \hat{n}_x + v_y \hat{n}_y + v_z \hat{n}_z \\ \bar{w} &= w_x \hat{n}_x + w_y \hat{n}_y + w_z \hat{n}_z \\ \bar{v} \cdot \bar{w} &= v_x w_x + v_y w_y + v_z w_z \end{aligned} \quad (7.18)$$

The dot product has these properties:

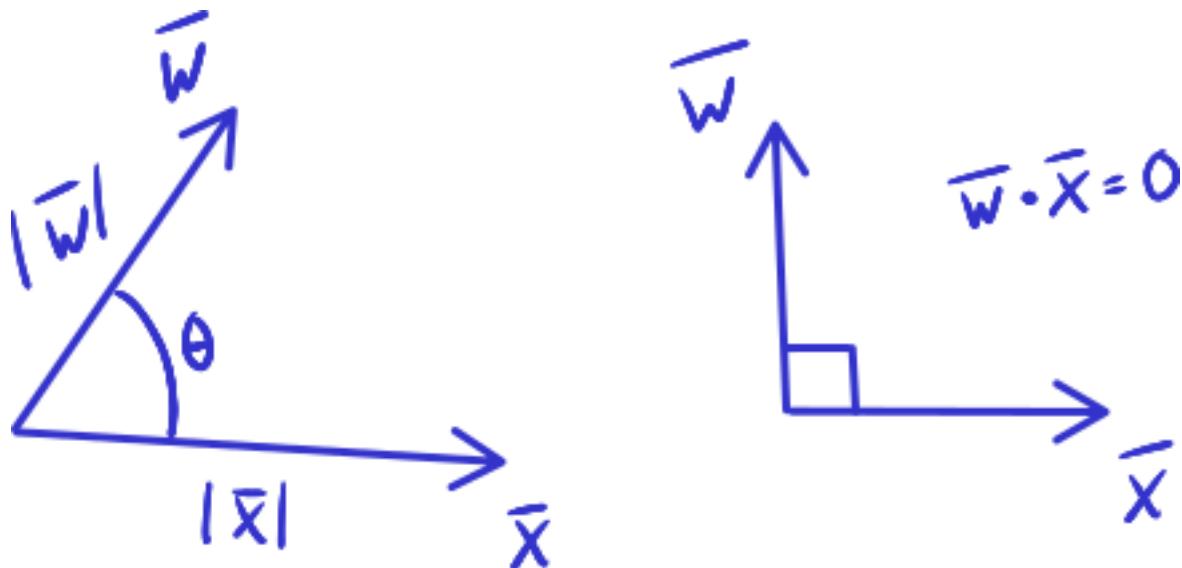


Fig. 7.6: Vector dot product

- You can pull out scalars: $c\bar{u} \cdot d\bar{v} = cd(\bar{u} \cdot \bar{v})$
- Order does not matter (commutative multiplication): $\bar{u} \cdot \bar{v} = \bar{v} \cdot \bar{u}$
- You can distribute: $\bar{u} \cdot (\bar{v} + \bar{w}) = \bar{u} \cdot \bar{v} + \bar{u} \cdot \bar{w}$

The dot product is often used to determine:

- the angle between two vectors: $\theta = \arccos \frac{\bar{a} \cdot \bar{b}}{|\bar{a}| |\bar{b}|}$
- a vector's magnitude: $|\bar{v}| = \sqrt{\bar{v} \cdot \bar{v}}$
- the length of a vector along a direction of another vector \hat{u} (called the projection): $\text{proj}_{\hat{u}} \bar{v} = \bar{v} \cdot \hat{u}$
- if two vectors are perpendicular: $\bar{v} \cdot \bar{w} = 0$ if and only if $\bar{v} \perp \bar{w}$
- Compute power: $P = \bar{F} \cdot \bar{v}$, where \bar{F} is a force vector and \bar{v} is the velocity of the point the force is acting on.

Also, dot products are used to convert a vector equation into a scalar equation by “dotting” an entire equation with a vector.

```
N = me.ReferenceFrame('N')
w = a*N.x + b*N.y + c*N.z
x = d*N.x + e*N.y + f*N.z
```

The `dot()` function calculates the dot product:

```
me.dot(w, x)
```

$$ad + be + cf \quad (7.19)$$

The method form is equivalent:

```
w.dot(x)
```

$$ad + be + cf \quad (7.20)$$

You can compute a unit vector \hat{w} in the same direction as \bar{w} with the `normalize()` method:

```
w.normalize()
```

$$\frac{a}{\sqrt{a^2 + b^2 + c^2}}\hat{n}_x + \frac{b}{\sqrt{a^2 + b^2 + c^2}}\hat{n}_y + \frac{c}{\sqrt{a^2 + b^2 + c^2}}\hat{n}_z \quad (7.21)$$

Exercise

Write your own function that normalizes an arbitrary vector and show that it gives the same result as `w.normalize()`.

Solution

```
def normalize(vector):
    return vector/sm.sqrt(me.dot(vector, vector))

normalize(w)
```

$$\frac{a}{\sqrt{a^2 + b^2 + c^2}}\hat{n}_x + \frac{b}{\sqrt{a^2 + b^2 + c^2}}\hat{n}_y + \frac{c}{\sqrt{a^2 + b^2 + c^2}}\hat{n}_z \quad (7.22)$$

SymPy Mechanics vectors also have a method `magnitude()` which is helpful:

```
w.magnitude()
```

$$\sqrt{a^2 + b^2 + c^2} \quad (7.23)$$

```
w/w.magnitude()
```

$$\frac{a}{\sqrt{a^2 + b^2 + c^2}}\hat{n}_x + \frac{b}{\sqrt{a^2 + b^2 + c^2}}\hat{n}_y + \frac{c}{\sqrt{a^2 + b^2 + c^2}}\hat{n}_z \quad (7.24)$$

Exercise

Given the vectors $\bar{v}_1 = a\hat{n}_x + b\hat{n}_y + a\hat{n}_z$ and $\bar{v}_2 = b\hat{n}_x + a\hat{n}_y + b\hat{n}_z$ find the angle between the two vectors using the dot product.

Solution

```
N = me.ReferenceFrame('N')
v1 = a*N.x + b*N.y + a*N.z
v2 = b*N.x + a*N.y + b*N.z
```

```
sm.acos(v1.dot(v2) / (v1.magnitude() * v2.magnitude()))
```

$$\cos\left(\frac{3ab}{\sqrt{a^2+2b^2}\sqrt{2a^2+b^2}}\right) \quad (7.25)$$

7.7 Cross Product

The **cross product**, which yields a vector quantity, is defined as:

$$\bar{v} \times \bar{w} = |\bar{v}|\bar{w}|\sin\theta\hat{u} \quad (7.26)$$

where θ is the angle between the two vectors, and \hat{u} is the unit vector perpendicular to both \bar{v} and \bar{w} whose sense is given by the right-hand rule. For arbitrary measure numbers this results in the following:

$$\begin{aligned} \bar{v} &= v_x\hat{n}_x + v_y\hat{n}_y + v_z\hat{n}_z \\ \bar{w} &= w_x\hat{n}_x + w_y\hat{n}_y + w_z\hat{n}_z \\ \bar{v} \times \bar{w} &= (v_yw_z - v_zw_y)\hat{n}_x + (v_zw_x - v_xw_z)\hat{n}_y + (v_xw_y - v_yw_x)\hat{n}_z \end{aligned} \quad (7.27)$$

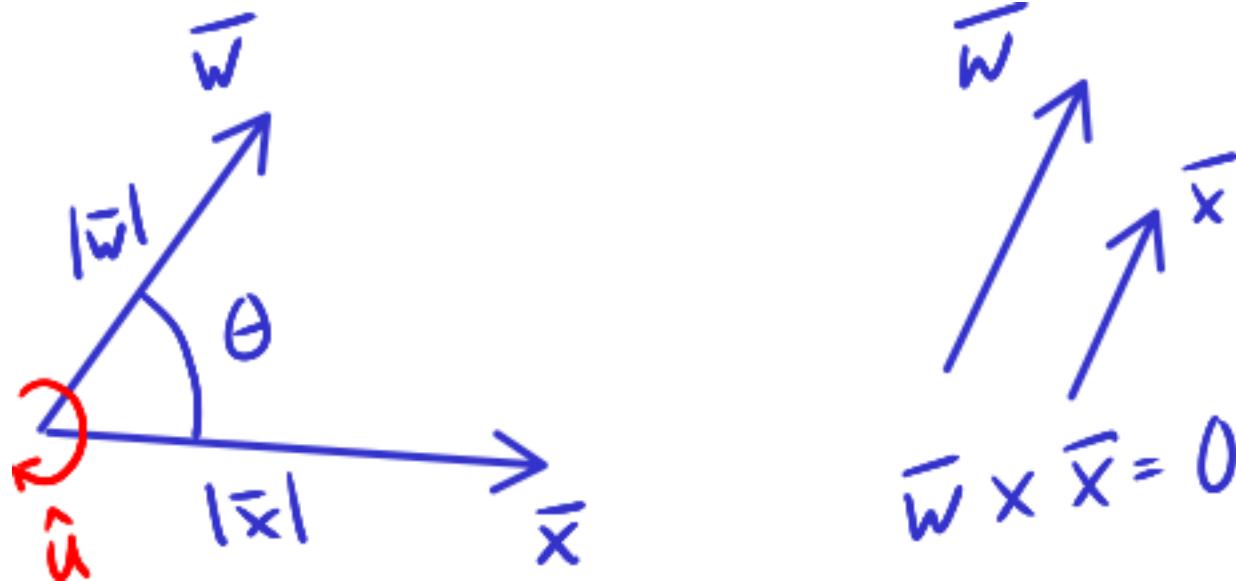


Fig. 7.7: Vector cross product

Some properties of cross products are:

- Crossing a vector with itself “cancels” it: $\bar{a} \times \bar{a} = \bar{0}$

- You can pull out scalars: $c\bar{a} \times d\bar{b} = cd(\bar{a} \times \bar{b})$
- Order **DOES** matter (because of the right-hand rule): $\bar{a} \times \bar{b} = -\bar{b} \times \bar{a}$
- You can distribute: $\bar{a} \times (\bar{b} + \bar{c}) = \bar{a} \times \bar{b} + \bar{a} \times \bar{c}$
- They are **NOT** associative: $\bar{a} \times (\bar{b} \times \bar{c}) \neq (\bar{a} \times \bar{b}) \times \bar{c}$

The cross product is used to:

- obtain a vector/direction perpendicular to two other vectors
- determine if two vectors are parallel: $\bar{v} \times \bar{w} = \bar{0}$ if $\bar{v} \parallel \bar{w}$
- compute moments: $\bar{r} \times \bar{F}$
- compute the area of a triangle

SymPy Mechanics can calculate cross products with the `cross()` function:

```
N = me.ReferenceFrame('N')
w = a*N.x + b*N.y + c*N.z
w
```

$$a\hat{n}_x + b\hat{n}_y + c\hat{n}_z \quad (7.28)$$

```
x = d*N.x + e*N.y + f*N.z
x
```

$$d\hat{n}_x + e\hat{n}_y + f\hat{n}_z \quad (7.29)$$

```
me.cross(w, x)
```

$$(bf - ce)\hat{n}_x + (-af + cd)\hat{n}_y + (ae - bd)\hat{n}_z \quad (7.30)$$

The method form is equivalent:

```
w.cross(x)
```

$$(bf - ce)\hat{n}_x + (-af + cd)\hat{n}_y + (ae - bd)\hat{n}_z \quad (7.31)$$

Exercise

Given three points located in reference frame N by:

$$\begin{aligned} \bar{p}_1 &= 23\hat{n}_x - 12\hat{n}_y \\ \bar{p}_2 &= 16\hat{n}_x + 2\hat{n}_y - 4\hat{n}_z \\ \bar{p}_3 &= \hat{n}_x + 14\hat{n}_z \end{aligned} \quad (7.32)$$

Find the area of the triangle bounded by these three points using the cross product.

Hint: Search online for the relationship of the cross product to triangle area.

Solution

```
N = me.ReferenceFrame('N')
p1 = 23*N.x - 12*N.y
p2 = 16*N.x + 2*N.y - 4*N.z
p3 = N.x + 14*N.z

me.cross(p2 - p1, p3 - p1).magnitude() / 2
```

$$\sqrt{36077} \quad (7.33)$$

7.8 Vectors Expressed in Multiple Reference Frames

The notation of vectors represented by a scalar measure numbers associated with unit vectors becomes quite useful when you need to describe vectors with components in multiple reference frames. Utilizing unit vectors fixed in various frames is rather natural, with no need to think about direction cosine matrices.

```
N = me.ReferenceFrame('N')
A = me.ReferenceFrame('A')

a, b, theta = sm.symbols('a, b, theta')

v = a*A.x + b*N.y
v
```

$$a\hat{a}_x + b\hat{n}_y \quad (7.34)$$

All of the previously described operations work as expected:

```
v + v
```

$$2a\hat{a}_x + 2b\hat{n}_y \quad (7.35)$$

If an orientation is established between the two reference frames, the direction cosine transformations are handled for you and can be used to naturally express the vector in either reference frame using the `express()`.

```
A.orient_axis(N, theta, N.z)

v.express(N)
```

$$a \cos(\theta) \hat{n}_x + (a \sin(\theta) + b) \hat{n}_y \quad (7.36)$$

```
v.express(A)
```

$$(a + b \sin(\theta)) \hat{a}_x + b \cos(\theta) \hat{a}_y \quad (7.37)$$

7.9 Relative Position Among Points

Take for example the [balanced-arm lamp](#), which has multiple articulated joints configured in a way to balance the weight of the lamp in any configuration. Here are two examples:

With those lamps in mind, [Fig. 7.10](#) shows a diagram of a similar desk lamp with all necessary configuration information present. The base N is fixed to the desk. The first linkage A is oriented with respect to N by a z - x body fixed orientation through angles q_1 and q_2 . Point P_1 is fixed in N and is located at the center of the base. Linkage A is defined by points P_1 and P_2 which are separated by length l_1 along the \hat{a}_z direction. Linkage B orients simply with respect to A about $\hat{a}_x = \hat{b}_x$ through angle q_3 and point P_3 is l_2 from P_2 along \hat{b}_z . Lastly, the lamp head C orients relative to B by a x - z body fixed orientation through angles q_4 and q_5 . The center of the light bulb P_4 is located relative to P_3 by the distances l_3 along \hat{c}_z and l_4 along $-\hat{c}_y$.

We will use the following notation for vectors that indicate the relative position between two points:

$$\bar{r}^{P_2/P_1} \quad (7.38)$$

which reads as the “position vector from P_1 to P_2 ” or the “position vector of P_2 with respect to P_1 ”. The tail of the vector is at P_1 and the tip is at P_2 .

Exercise

Reread the [Vector Functions](#) section and answer the following questions:

1. Is \bar{r}^{P_2/P_1} vector function of q_1 and q_2 in N ?
2. Is \bar{r}^{P_2/P_1} vector function of q_1 and q_1 in A ?
3. Is \bar{r}^{P_2/P_1} vector function of q_3 and q_4 in N ?
4. Is \bar{r}^{P_3/P_2} vector function of q_1 and q_2 in N ?

Solution

See below how to use `.free_symbols()` to check your answers.

We can now write position vectors between pairs of points as we move from the base of the lamp to the light bulb. We'll do so with SymPy Mechanics. First create the necessary symbols and reference frames.



Fig. 7.8: Balanced-arm desk lamp.

Flickr user “renaissance chambara”, cropped by uploader, CC BY 2.0 <https://creativecommons.org/licenses/by/2.0>, via Wikimedia Commons



Fig. 7.9: Example of a huge balance-arm lamp in Rotterdam at the Schouwburgplein.
GraphyArchy, CC BY-SA 4.0 <https://creativecommons.org/licenses/by-sa/4.0>, via Wikimedia Commons

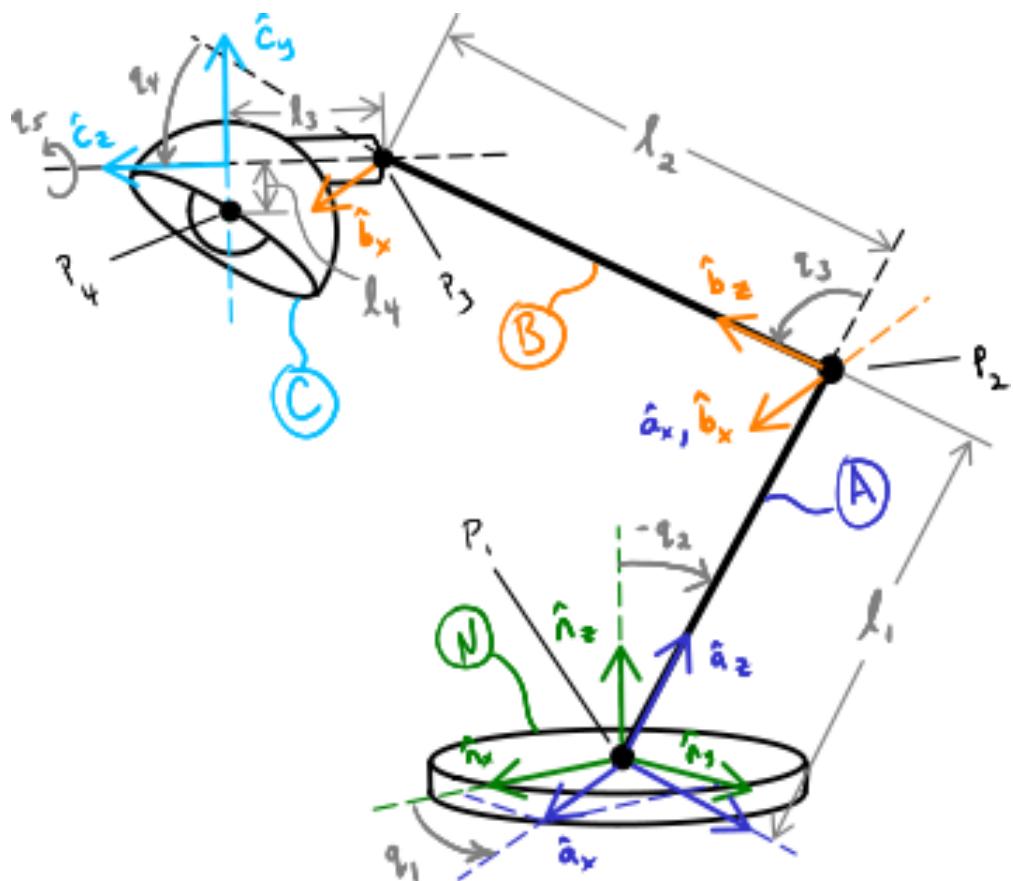


Fig. 7.10: Configuration diagram of a balanced-arm desk lamp.

```

q1, q2, q3, q4, q5 = sm.symbols('q1, q2, q3, q4, q5')
l1, l2, l3, l4 = sm.symbols('l1, l2, l3, l4')
N = me.ReferenceFrame('N')
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
C = me.ReferenceFrame('C')

```

Now establish the orientations, starting with A 's orientation relative to N :

```
A.orient_body_fixed(N, (q1, q2, 0), 'Zxz')
```

Note: Notice that the unneeded third simple orientation angle was set to zero.

Then B 's orientation relative to A :

```
B.orient_axis(A, q3, A.x)
```

And finally C 's orientation relative to B :

```
C.orient_body_fixed(B, (q4, q5, 0), 'Xzx')
```

We can now create position vectors between pairs of points in the most convenient frame to do so, i.e. the reference frame in which both points lie on a line parallel to an existing unit vector. The intermediate vectors that connect P_1 to P_2 , P_2 to P_3 , and P_3 to P_4 are:

```

R_P1_P2 = l1*A.z
R_P2_P3 = l2*B.z
R_P3_P4 = l3*C.z - l4*C.y

```

The position vector from P_1 to P_4 is then found by vector addition:

```

R_P1_P4 = R_P1_P2 + R_P2_P3 + R_P3_P4
R_P1_P4

```

$$l_1\hat{a}_z + l_2\hat{b}_z - l_4\hat{c}_y + l_3\hat{c}_z \quad (7.39)$$

To convince you of the utility of our vector notation, have a look at what \bar{r}^{P_4/P_1} looks like if expressed completely in the N frame:

```
R_P1_P4.express(N)
```

$$(l_1 \sin(q_1) \sin(q_2) + l_2 (\sin(q_1) \sin(q_2) \cos(q_3) + \sin(q_1) \sin(q_3) \cos(q_2)) + l_3 (-(\sin(q_1) \sin(q_2) \sin(q_3) - \sin(q_1) \cos(q_2) \cos(q_3))) \quad (7.40)$$

If you have properly established your orientations and position vectors, SymPy Mechanics can help you determine the answers to the previous exercise. Expressing \bar{r}^{P_2/P_1} in N can show us which scalar variables that vector function depends on in N .

```
R_P1_P2.express(N)
```

$$l_1 \sin(q_1) \sin(q_2) \hat{n}_x - l_1 \sin(q_2) \cos(q_1) \hat{n}_y + l_1 \cos(q_2) \hat{n}_z \quad (7.41)$$

By inspection, we see the variables are l_1, q_1, q_2 . The `free_symbols()` function can extract these scalars directly:

```
R_P1_P2.free_symbols(N)
```

$$\{l_1, q_1, q_2\} \quad (7.42)$$

Warning: `free_symbols()` shows all SymPy `Symbol` objects, but will not show `Function()` objects. In the next chapter we will introduce a way to do the same thing when functions of time are present in your vector expressions.

Similarly, other vector functions can be inspected:

```
R_P1_P2.free_symbols(A)
```

$$\{l_1\} \quad (7.43)$$

```
R_P1_P4.free_symbols(N)
```

$$\{l_1, l_2, l_3, l_4, q_1, q_2, q_3, q_4, q_5\} \quad (7.44)$$

VECTOR DIFFERENTIATION

Note: You can download this example as a Python script: differentiation.py or Jupyter Notebook: differentiation.ipynb.

```
import sympy as sm
import sympy.physics.mechanics as me
sm.init_printing(use_latex='mathjax')

class ReferenceFrame(me.ReferenceFrame):
    def __init__(self, *args, **kwargs):
        kwargs.pop('latexs', None)
        lab = args[0].lower()
        tex = r'\hat{{}}_{{}}_{{}}'
        super(ReferenceFrame, self).__init__(*args,
                                             latexs=(tex.format(lab, 'x'),
                                                      tex.format(lab, 'y'),
                                                      tex.format(lab, 'z')),
                                             **kwargs)
me.ReferenceFrame = ReferenceFrame
```

8.1 Learning Objectives

After completing this chapter readers will be able to:

- Calculate the partial derivative of a vector with respect to any variable when viewed from any reference frame.
- Use the product rule to find the relationship of changing measure numbers and changing unit vectors.
- Explain the difference in expressing a vector in a reference frame and taking the derivative of the vector when observed from the reference frame.
- Calculate second partial derivatives.
- Calculate time derivatives of vector functions.

8.2 Partial Derivatives

If a vector \bar{v} is a function of n scalar variables q_1, q_2, \dots, q_n in reference frame A then the first partial derivatives of \bar{v} in A with respect to q_r where $r = 1 \dots n$ can be formed by applying the product rule of differentiation and taking into account that the mutually perpendicular unit vectors fixed in A do not change when observed from A . The partial derivatives are then:

$${}^A \frac{\partial \bar{v}}{\partial q_r} = \sum_{i=1}^3 \frac{\partial v_i}{\partial q_r} \hat{a}_i \text{ for } r = 1 \dots n \quad (8.1)$$

where v_i are the measure numbers of \bar{v} expressed in A associated with the mutually perpendicular unit vectors $\hat{a}_1, \hat{a}_2, \hat{a}_3$.

If $\bar{v} = v_x \hat{a}_x + v_y \hat{a}_y + v_z \hat{a}_z$ the above definition expands to:

$${}^A \frac{\partial \bar{v}}{\partial q_r} = \frac{\partial v_x}{\partial q_r} \hat{a}_x + \frac{\partial v_y}{\partial q_r} \hat{a}_y + \frac{\partial v_z}{\partial q_r} \hat{a}_z \text{ for } r = 1 \dots n \quad (8.2)$$

Many of the vectors we will work with in multibody dynamics will be a function of a single variable, most often time t . If that is the case, the partial derivative reduces to a single variate derivative:

$${}^A \frac{d\bar{v}}{dt} := \sum_{i=1}^3 \frac{dv_i}{dt} \hat{a}_i \quad (8.3)$$

Warning: A derivative written as $\frac{\partial \bar{v}}{\partial q_r}$ is meaningless because no reference frame is indicated. The derivative is dependent on which reference frame the change is observed from, so without a reference frame, the derivative cannot be calculated. This is not the case for partial derivatives of scalar expressions, as no reference frame is involved.

The above definition implies that a vector must be expressed in the reference frame one is observing the change from before calculating the partial derivatives of the scalar measure numbers. For example, here is a vector that is expressed with unit vectors from three different reference frames:

```
alpha, beta = sm.symbols('alpha, beta')
a, b, c, d, e, f = sm.symbols('a, b, c, d, e, f')

A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
C = me.ReferenceFrame('C')

B.orient_axis(A, alpha, A.x)
C.orient_axis(B, beta, B.y)

v = a*A.x + b*A.y + c*B.x + d*B.y + e*C.x + f*C.y
v
```

$$a\hat{a}_x + b\hat{a}_y + c\hat{b}_x + d\hat{b}_y + e\hat{c}_x + f\hat{c}_y \quad (8.4)$$

To calculate $\frac{^A \partial \bar{v}}{\partial \alpha}$ we first need to project the vector \bar{v} onto the unit vectors of A and take the partial derivative of those measure numbers with respect to α . The dot product provides the projection and the resulting scalar is differentiated:

```
dvdaAx = v.dot(A.x).diff(alpha)
dvdaAx
```

$$0 \quad (8.5)$$

```
dvddalphaAy = v.dot(A.y).diff(alpha)
dvddalphaAy
```

$$-d \sin(\alpha) + e \sin(\beta) \cos(\alpha) - f \sin(\alpha) \quad (8.6)$$

```
dvddalphaAz = v.dot(A.z).diff(alpha)
dvddalphaAz
```

$$d \cos(\alpha) + e \sin(\alpha) \sin(\beta) + f \cos(\alpha) \quad (8.7)$$

We can then construct the vector ${}^A\bar{v}$ from the new measure numbers know that the A unit vectors are fixed:

```
dvddalphaA = dvddalphaAx*A.x + dvddalphaAy*A.y + dvddalphaAz*A.z
dvddalphaA
```

$$(-d \sin(\alpha) + e \sin(\beta) \cos(\alpha) - f \sin(\alpha))\hat{a}_y + (d \cos(\alpha) + e \sin(\alpha) \sin(\beta) + f \cos(\alpha))\hat{a}_z \quad (8.8)$$

Sympy Mechanics vectors have a special `diff()` method that manages taking partial derivatives from different reference frames. For the vector `.diff()` method you provide first the variable α followed by the reference frame you are observing from:

```
v.diff(alpha, A)
```

$$(-d \sin(\alpha) + e \sin(\beta) \cos(\alpha) - f \sin(\alpha))\hat{a}_y + (d \cos(\alpha) + e \sin(\alpha) \sin(\beta) + f \cos(\alpha))\hat{a}_z \quad (8.9)$$

This gives the identical result as our manually constructed partial derivative above.

Exercise

Calculate ${}^B\bar{v}$ manually and with `diff()` and show the results are the same.

Solution

```
dvdeBx = v.dot(B.x).diff(e)
dvdeBy = v.dot(B.y).diff(e)
dvdeBz = v.dot(B.z).diff(e)
dvdeBx*B.x + dvdeBy*B.y + dvdeBz*B.z
```

$$\cos(\beta)\hat{b}_x - \sin(\beta)\hat{b}_z \quad (8.10)$$

```
v.diff(e, B).express(B)
```

$$\cos(\beta)\hat{b}_x - \sin(\beta)\hat{b}_z \quad (8.11)$$

Warning: What's the difference in `.express()` and `.diff()`?

Any vector can be “expressed” in any reference frame. To express a vector in a reference frame means to project it onto the three mutually perpendicular unit vectors fixed in the reference frame and then to rewrite the vector in terms of measure numbers associated with those three unit vectors using the relevant direction cosine matrix entries. This has nothing to do with differentiation.

We can also take the derivative of a vector when viewed from a specific reference frame. To do so, we observe how the vector changes when viewed from the reference frame and formulate that derivative. Once the derivative is taken, we can express the new vector in any reference frame we desire.

Expressing a vector in a reference frame and taking a derivative of a vector when observed from a reference frame are two different things! Try not to get tripped up by this important distinction.

8.3 Product Rule

Consider again vector $\bar{v} = v_x\hat{a}_x + v_y\hat{a}_y + v_z\hat{a}_z$. Previously, only the measure numbers of this vector were scalar functions of q_r . Now consider a reference frame N that is oriented relative to A such that the relative orientation also depends on q_r . This means, that when observed from N , the unit vectors $\hat{a}_x, \hat{a}_y, \hat{a}_z$ may be a function of q_r . With both the measure numbers and unit vectors dependent on q_r the derivative of \bar{v} in N requires the use of the product rule when taking the partial derivative. For example:

$$\frac{^N\partial\bar{v}}{\partial q_r} = \frac{^N\partial v_x}{\partial q_r}\hat{a}_x + v_x\frac{^N\partial\hat{a}_x}{\partial q_r} + \frac{^N\partial v_y}{\partial q_r}\hat{a}_y + v_y\frac{^N\partial\hat{a}_y}{\partial q_r} + \frac{^N\partial v_z}{\partial q_r}\hat{a}_z + v_z\frac{^N\partial\hat{a}_z}{\partial q_r} \quad (8.12)$$

The three similar terms with scalar derivatives have the same interpretation of the ones in the prior section.

$$\frac{^N\partial v_x}{\partial q_r}\hat{a}_x, \frac{^N\partial v_y}{\partial q_r}\hat{a}_y, \frac{^N\partial v_z}{\partial q_r}\hat{a}_z \quad (8.13)$$

But the part with unit vector derivatives is more interesting. The partial derivative of a unit vector depends on how it changes. But unit vectors do not change in length, only in orientation.

$$v_x\frac{^N\partial\hat{a}_x}{\partial q_r}, v_y\frac{^N\partial\hat{a}_y}{\partial q_r}, v_z\frac{^N\partial\hat{a}_z}{\partial q_r} \quad (8.14)$$

You will learn in the next chapter how to interpret and use these terms to simplify the calculations of common derivatives. But for now, just be aware of the nature of this partial derivative in N .

The product rule also applies to the dot and cross products:

$$\begin{aligned}\frac{\partial}{\partial q_r}(\bar{v} \cdot \bar{w}) &= \frac{\partial \bar{v}}{\partial q_r} \cdot \bar{w} + \bar{v} \cdot \frac{\partial \bar{w}}{\partial q_r} \\ \frac{\partial}{\partial q_r}(\bar{v} \times \bar{w}) &= \frac{\partial \bar{v}}{\partial q_r} \times \bar{w} + \bar{v} \times \frac{\partial \bar{w}}{\partial q_r}\end{aligned}\quad (8.15)$$

and generalizes to any series of products. Let $G = f_1 \cdots f_n$ be a series of products, then:

$$\frac{\partial G}{\partial q_r} = \frac{\partial f_1}{\partial q_r} \cdot f_2 \cdots f_n + f_1 \cdot \frac{\partial f_2}{\partial q_r} \cdot f_3 \cdots f_n + \cdots + f_1 \cdots f_{n-1} \cdot \frac{\partial f_n}{\partial q_r} \quad (8.16)$$

8.4 Second Derivatives

$\frac{^A\partial \bar{v}}{\partial q_r}$ is also a vector and, just like \bar{v} , may be a vector function. We can thus calculate the second partial derivative with respect to q_s where $s = 1 \dots n$. This second partial derivative need not be taken with respect to the same reference frame as the first partial derivative. If we first differentiate with when viewed from A and then when viewed from B , the second partial derivative is:

$$\frac{^B\partial}{\partial q_s} \left(\frac{^A\partial \bar{v}}{\partial q_r} \right) \quad (8.17)$$

Second partials in different reference frames do not necessarily commute:

$$\frac{^B\partial}{\partial q_s} \left(\frac{^A\partial \bar{v}}{\partial q_r} \right) \neq \frac{^A\partial}{\partial q_r} \left(\frac{^B\partial \bar{v}}{\partial q_s} \right) \quad (8.18)$$

If the reference frames of each partial derivative are the same, then mixed partials do commute.

8.5 Vector Functions of Time

In multibody dynamics we are primarily concerned with how motion changes with respect to time t and our vectors and measure numbers will often be implicit functions of time, i.e. $q_r(t)$. When that is the case the chain rule can be used to take total derivatives:

$$\frac{^A d\bar{v}}{dt} = \sum_{i=1}^n \frac{^A\partial \bar{v}}{\partial q_r(t)} \frac{dq_r(t)}{dt} + \frac{^A\partial \bar{v}}{\partial t} \text{ where } r = 1, \dots, n \quad (8.19)$$

Note: We will typically use the “dot” notation for time derivatives, i.e. $\frac{dq}{dt}$ as \dot{q} and $\frac{d^2q}{dt^2}$ as \ddot{q} and so on.

In SymPy Mechanics, scalar functions of time can be created like so:

```
t = sm.symbols('t')
q_of = sm.Function('q')

q = q_of(t)
q
```

$$q(t) \quad (8.20)$$

And these scalar functions can be differentiated:

```
q.diff(t)
```

$$\frac{d}{dt}q(t) \quad (8.21)$$

SymPy Mechanics provides the convenience function `dynamicssymbols()` to create scalar functions of time just like `symbols()`:

```
q1, q2, q3 = me.dynamicssymbols('q1, q2, q3')
q1, q2, q3
```

$$(q_1(t), q_2(t), q_3(t)) \quad (8.22)$$

The time variable used in `q1`, `q2`, `q3` can be accessed like so:

```
t = me.dynamicssymbols._t
```

SymPy Mechanics also provide a special printing function `init_vprinting()` which enables the dot notation on functions of time:

```
me.init_vprinting(use_latex='mathjax')
q1.diff(t), q2.diff(t, 2), q3.diff(t, 3)
```

$$(\dot{q}_1, \ddot{q}_2, \ddot{q}_3) \quad (8.23)$$

Now these scalar functions of time can be used to formulate vectors:

```
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
B.orient_body_fixed(A, (q1, q2, q3), 'ZXZ')
v = q1*A.x + q2*A.y + t**2*A.z
v
```

$$q_1\hat{a}_x + q_2\hat{a}_y + t^2\hat{a}_z \quad (8.24)$$

And the time derivative can be found with:

```
v.diff(t, A)
```

$$\dot{q}_1\hat{a}_x + \dot{q}_2\hat{a}_y + 2t\hat{a}_z \quad (8.25)$$

Lastly, vectors have a `dt()` method that calculates time derivatives when viewed from a reference frame, saving a few characters of typing:

```
v.dt(A)
```

$$\dot{q}_1\hat{a}_x + \dot{q}_2\hat{a}_y + 2t\hat{a}_z \quad (8.26)$$

We will use time derivatives in the next chapters to formulate velocity and acceleration.

ANGULAR KINEMATICS

Note: You can download this example as a Python script: angular.py or Jupyter Notebook: angular.ipynb.

```
import sympy as sm
import sympy.physics.mechanics as me
me.init_vprinting(use_latex='mathjax')

class ReferenceFrame(me.ReferenceFrame):

    def __init__(self, *args, **kwargs):
        kwargs.pop('latexs', None)

        lab = args[0].lower()
        tex = r'\hat{{\{}{\}}}_{{}}'

        super(ReferenceFrame, self).__init__(*args,
                                             latexs=(tex.format(lab, 'x'),
                                                     tex.format(lab, 'y'),
                                                     tex.format(lab, 'z')),
                                             **kwargs)
me.ReferenceFrame = ReferenceFrame
```

9.1 Learning Objectives

After completing this chapter readers will be able to:

- apply the definition of angular velocity
- calculate the angular velocity of simple rotations
- choose Euler angles for a rotating reference frame
- calculate the angular velocity of references frames described by successive simple rotations
- derive the time derivative of a vector in terms of angular velocities
- calculate the angular acceleration of a reference frame
- calculate the angular acceleration of successive rotations

9.2 Introduction

To apply Euler's Laws of Motion to a multibody system we will need to determine how the angular momentum of each rigid body changes with time. This requires that we specify the angular kinematics of each body in the system: typically both angular velocity and angular acceleration. Assuming that a reference frame is fixed to a rigid body, we will start by finding the angular kinematics of a single reference frame and then use the properties of *Successive Orientations* to find the angular kinematics of a set of connected reference frames.

In the video below, a small T-handle is shown spinning in low Earth orbit gravity onboard the International Space Station. This single rigid body has an orientation, angular velocity, and angular acceleration at any given instance of time.

The T-handle exhibits unintuitive motion, reversing back and forth periodically. This phenomena is commonly referred to as the “Dzhanibekov effect” and Euler's Laws of Motion predict the behavior, which we will investigate in later chapters. For now, we will learn how to specify the angular kinematics of a reference frame in motion, such as one fixed to this T-handle.

9.3 Angular Velocity

In Ch. *Orientation of Reference Frames* we learned that reference frames can be oriented relative to each other. If the relative orientation of two reference frames change with respect to time, then we can calculate the angular velocity ${}^A\bar{\omega}^B$ of reference frame B when observed from reference frame A . If $\hat{b}_x, \hat{b}_y, \hat{b}_z$ are right handed mutually perpendicular unit vectors fixed in B then the angular velocity of B in A is defined as ([Kane1985], pg. 16):

$${}^A\bar{\omega}^B := \left(\frac{^A d\hat{b}_y}{dt} \cdot \hat{b}_z \right) \hat{b}_x + \left(\frac{^A d\hat{b}_z}{dt} \cdot \hat{b}_x \right) \hat{b}_y + \left(\frac{^A d\hat{b}_x}{dt} \cdot \hat{b}_y \right) \hat{b}_z. \quad (9.1)$$

Warning: Don't confuse the left and right superscripts on direction cosine matrices and angular velocities. ${}^B\mathbf{C}^A$ describes the orientation of B rotated with respect to A and the mapping of vectors in A to vectors expressed in B . Whereas ${}^A\bar{\omega}^B$ describes the angular velocity of B when observed from A .

If B is oriented with respect to A and mutually perpendicular unit vectors $\hat{a}_x, \hat{a}_y, \hat{a}_z$ are fixed in A then there are these general relationships among the unit vectors of each frame (see *Direction Cosine Matrices*):

$$\begin{aligned} \hat{b}_x &= c_{xx}\hat{a}_x + c_{xy}\hat{a}_y + c_{xz}\hat{a}_z \\ \hat{b}_y &= c_{yx}\hat{a}_x + c_{yy}\hat{a}_y + c_{yz}\hat{a}_z \\ \hat{b}_z &= c_{zx}\hat{a}_x + c_{zy}\hat{a}_y + c_{zz}\hat{a}_z \end{aligned} \quad (9.2)$$

We can create these equations in SymPy to demonstrate how to work with the definition of angular velocity. Start by first creating the direction cosine matrix with time varying elements:

```
cxx, cyy, czz = me.dynamicsymbols('c_{xx}, c_{yy}, c_{zz}')
cxy, cxz, cyx = me.dynamicsymbols('c_{xy}, c_{xz}, c_{yx}')
cyz, czx, czy = me.dynamicsymbols('c_{yz}, c_{zx}, c_{zy}')

B_C_A = sm.Matrix([[cxx, cxy, cxz],
                  [cyx, cyy, cyz],
                  [czx, czy, czz]])
```

and establish the orientation using `orient_explicit()`:

Warning: Remember this method takes the transpose of the direction cosine matrix.

```
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
B.orient_explicit(A, B_C_A.transpose())
B.dcm(A)
```

$$\begin{bmatrix} c_{xx} & c_{xy} & c_{xz} \\ c_{yx} & c_{yy} & c_{yz} \\ c_{zx} & c_{zy} & c_{zz} \end{bmatrix} \quad (9.3)$$

This now let's us write the B unit vectors in terms of the A unit vectors:

```
B.x.express(A)
```

$$c_{xx}\hat{a}_x + c_{xy}\hat{a}_y + c_{xz}\hat{a}_z \quad (9.4)$$

```
B.y.express(A)
```

$$c_{yx}\hat{a}_x + c_{yy}\hat{a}_y + c_{yz}\hat{a}_z \quad (9.5)$$

```
B.z.express(A)
```

$$c_{zx}\hat{a}_x + c_{zy}\hat{a}_y + c_{zz}\hat{a}_z \quad (9.6)$$

Recalling the definition of angular velocity above, each of the measure numbers of the angular velocity is calculated by dotting the derivative of a B unit vector in A with a unit vector in B . $\frac{^A\dot{b}_y}{dt}$ is for example:

```
B.y.express(A).dt(A)
```

$$\dot{c}_{yx}\hat{a}_x + \dot{c}_{yy}\hat{a}_y + \dot{c}_{yz}\hat{a}_z \quad (9.7)$$

Each of the measure numbers of ${}^A\bar{\omega}^B$ are then:

```
mnx = me.dot(B.y.express(A).dt(A), B.z)
mnx
```

$$c_{zx}\dot{c}_{yx} + c_{zy}\dot{c}_{yy} + c_{zz}\dot{c}_{yz} \quad (9.8)$$

```
mn_y = me.dot(B.z.express(A).dt(A), B.x)
mn_y
```

$$c_{xx}\dot{c}_{zx} + c_{xy}\dot{c}_{zy} + c_{xz}\dot{c}_{zz} \quad (9.9)$$

```
mn_z = me.dot(B.x.express(A).dt(A), B.y)
mn_z
```

$$c_{yx}\dot{c}_{xx} + c_{yy}\dot{c}_{xy} + c_{yz}\dot{c}_{xz} \quad (9.10)$$

The angular velocity vector for an arbitrary direction cosine matrix is then:

```
A_w_B = mn_x*B.x + mn_y*B.y + mn_z*B.z
A_w_B
```

$$(c_{zx}\dot{c}_{yx} + c_{zy}\dot{c}_{yy} + c_{zz}\dot{c}_{yz})\hat{b}_x + (c_{xx}\dot{c}_{zx} + c_{xy}\dot{c}_{zy} + c_{xz}\dot{c}_{zz})\hat{b}_y + (c_{yx}\dot{c}_{xx} + c_{yy}\dot{c}_{xy} + c_{yz}\dot{c}_{xz})\hat{b}_z \quad (9.11)$$

If you know the direction cosine matrix and the derivative of its entries with respect to time, the angular velocity can be directly calculated with the above equation.

Exercise

At one instance of time, the direction cosine matrix is:

$$B\mathbf{C}^A = \begin{bmatrix} \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{2} & \frac{\sqrt{6}}{4} \\ -\frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \\ \frac{\sqrt{2}}{4} & -\frac{\sqrt{2}}{2} & \frac{\sqrt{6}}{4} \end{bmatrix} \quad (9.12)$$

and the time derivatives of the entries of the direction cosine matrix are:

$$\frac{d^B\mathbf{C}^A}{dt} = \begin{bmatrix} -\frac{\sqrt{6}}{2} - \frac{3\sqrt{2}}{4} & -\frac{\sqrt{6}}{4} + \frac{3\sqrt{2}}{2} & -\frac{3\sqrt{6}}{4} + \sqrt{2} \\ -1 & -\frac{1}{2} & -\sqrt{3} \\ -\frac{\sqrt{6}}{2} + \frac{3\sqrt{2}}{4} & -\frac{\sqrt{6}}{4} + \frac{3\sqrt{2}}{2} & \frac{3\sqrt{6}}{4} \end{bmatrix} \quad (9.13)$$

apply the definition of angular velocity to find ${}^A\bar{\omega}^B$.

Solution

Define the two matrices:

```
B_C_A = sm.Matrix([
    [ sm.sqrt(2)/4, sm.sqrt(2)/2, sm.sqrt(6)/4 ],
    [ -sm.sqrt(3)/2, 0, sm.S(1)/2 ],
    [ sm.sqrt(2)/4, -sm.sqrt(2)/2, sm.sqrt(6)/4 ]
])
B_C_A
```

$$\begin{bmatrix} \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{2} & \frac{\sqrt{6}}{4} \\ -\frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \\ \frac{\sqrt{2}}{4} & -\frac{\sqrt{2}}{2} & \frac{\sqrt{6}}{4} \end{bmatrix} \quad (9.14)$$

```
B_C_A_dt = sm.Matrix([
    [-sm.sqrt(6)/2 - 3*sm.sqrt(2)/4, -sm.sqrt(6)/4 + 3*sm.sqrt(2)/2, -3*sm.sqrt(6)/4 +
     + sm.sqrt(2)], [
        -1, -sm.S(1)/2, -sm.
     -sm.sqrt(3)], [
        -sm.sqrt(6)/2 + 3*sm.sqrt(2)/4, -sm.sqrt(6)/4 + 3*sm.sqrt(2)/2, 3*sm.
     -sm.sqrt(6)/4]
    ])
B_C_A_dt
```

$$\begin{bmatrix} -\frac{\sqrt{6}}{2} - \frac{3\sqrt{2}}{4} & -\frac{\sqrt{6}}{4} + \frac{3\sqrt{2}}{2} & -\frac{3\sqrt{6}}{4} + \sqrt{2} \\ -1 & -\frac{1}{2} & -\sqrt{3} \\ -\frac{\sqrt{6}}{2} + \frac{3\sqrt{2}}{4} & -\frac{\sqrt{6}}{4} + \frac{3\sqrt{2}}{2} & \frac{3\sqrt{6}}{4} \end{bmatrix} \quad (9.15)$$

Recognizing the pattern in the definition of angular velocity, rows of each matrix can be matrix multiplied to arrive at the correct measure number:

```
mnx = (B_C_A[2, :]*B_C_A_dt[1, :].transpose())[0, 0]
mny = (B_C_A[0, :]*B_C_A_dt[2, :].transpose())[0, 0]
mnz = (B_C_A[1, :]*B_C_A_dt[0, :].transpose())[0, 0]

A_w_B = mnx*B.x + mny*B.y + mnz*B.z
```

`simplify()` applies `simplify()` to each measure number of a vector:

```
A_w_B.simplify()
```

$$-\frac{3\sqrt{2}}{4}\hat{b}_x + \left(3 - \frac{\sqrt{3}}{2}\right)\hat{b}_y + \frac{5\sqrt{2}}{4}\hat{b}_z \quad (9.16)$$

9.4 Angular Velocity of Simple Orientations

For a simple orientation of B with respect to A about the z axis through θ the direction cosine matrix is:

```
theta = me.dynamicsymbols('theta')

B_C_A = sm.Matrix([[sm.cos(theta), sm.sin(theta), 0],
                   [-sm.sin(theta), sm.cos(theta), 0],
                   [0, 0, 1]])

B_C_A
```

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (9.17)$$

Applying the definition of angular velocity as before, the angular velocity of B in A is:

```
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
B.orient_explicit(A, B_C_A.transpose())

mnx = me.dot(B.y.express(A).dt(A), B.z)
mny = me.dot(B.z.express(A).dt(A), B.x)
mnz = me.dot(B.x.express(A).dt(A), B.y)

A_w_B = mnx*B.x + mny*B.y + mnz*B.z
A_w_B
```

$$(\sin^2(\theta)\dot{\theta} + \cos^2(\theta)\dot{\theta})\hat{b}_z \quad (9.18)$$

This can be simplified with a trigonometric identity. We can do this with `simplify()` which applies `simplify()` to each measure number of a vector:

```
A_w_B.simplify()
```

$$\dot{\theta}\hat{b}_z \quad (9.19)$$

The angular velocity of a simple orientation is simply the time rate of change of θ about $\hat{b}_z = \hat{a}_z$, the axis of the simple orientation. SymPy Mechanics offers the `ang_vel_in()` method for automatically calculating the angular velocity if a direction cosine matrix exists between the two reference frames:

```
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
B.orient_axis(A, theta, A.z)
B.ang_vel_in(A)
```

$$\dot{\theta}\hat{a}_z \quad (9.20)$$

A simple orientation and associated simple angular velocity can be formulated for any arbitrary orientation axis vector, not just one of the three mutually perpendicular unit vectors as shown above. There is a simple angular velocity between two reference frames A and B if there exists a single unit vector \hat{k} which is fixed in both A and B for some finite time. If this is the case, then ${}^A\bar{\omega}^B = \omega\hat{k}$ where ω is the time rate of change of the angle θ between a line fixed in A and another line fixed in B both of which are perpendicular to the orientation axis \hat{k} . We call $\omega = \dot{\theta}$ the angular speed of B in A . `orient_axis()` can take any arbitrary vector fixed in A and B to establish the orientation:

```
theta = me.dynamicsymbols('theta')

A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
B.orient_axis(A, theta, A.x + A.y)
B.ang_vel_in(A)
```

$$\frac{\sqrt{2}\dot{\theta}}{2}\hat{a}_x + \frac{\sqrt{2}\dot{\theta}}{2}\hat{a}_y \quad (9.21)$$

The angular speed is then:

```
B.ang_vel_in(A).magnitude()
```

$$\sqrt{\dot{\theta}^2} \quad (9.22)$$

Note: This result should be $|\dot{\theta}|$. This is a bug in SymPy, see <https://github.com/sympy/sympy/issues/23173> for more info. This generally will not cause issues, but for certain equation of motion derivations it could, so beware.

9.5 Body Fixed Orientations

If you establish a Euler z - x - z orientation with angles ψ, θ, φ respectively, then the angular velocity vector is:

```
psi, theta, phi = me.dynamicsymbols('psi, theta, varphi')

A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
B.orient_body_fixed(A, (psi, theta, phi), 'ZXZ')

mnx = me.dot(B.y.express(A).dt(A), B.z)
mny = me.dot(B.z.express(A).dt(A), B.x)
mnz = me.dot(B.x.express(A).dt(A), B.y)

A_w_B = mnx*B.x + mny*B.y + mnz*B.z
A_w_B.simplify()
```

$$(\sin(\theta)\sin(\varphi)\dot{\psi} + \cos(\varphi)\dot{\theta})\hat{b}_x + (\sin(\theta)\cos(\varphi)\dot{\psi} - \sin(\varphi)\dot{\theta})\hat{b}_y + (\cos(\theta)\dot{\psi} + \dot{\varphi})\hat{b}_z \quad (9.23)$$

The method `ang_vel_in()` does this same calculation and gives the same result:

```
B.ang_vel_in(A)
```

$$(\sin(\theta)\sin(\varphi)\dot{\psi} + \cos(\varphi)\dot{\theta})\hat{b}_x + (\sin(\theta)\cos(\varphi)\dot{\psi} - \sin(\varphi)\dot{\theta})\hat{b}_y + (\cos(\theta)\dot{\psi} + \dot{\varphi})\hat{b}_z \quad (9.24)$$

Exercise

Calculate the angular velocity of the T-handle T with respect to the space station N if \hat{t}_z is parallel to the spin axis, \hat{t}_y is parallel with the handle axis, and \hat{t}_x is normal to the plane made by the “T” and follows from the right hand rule. Select Euler angles that avoid **gimbal lock**. Hint: Read “Loss of degree of freedom with Euler angles” in the [gimbal lock article](#).

Solution

```
psi, theta, phi = me.dynamicsymbols('psi, theta, varphi')

N = me.ReferenceFrame('N')
T = me.ReferenceFrame('T')
T.orient_body_fixed(N, (psi, theta, phi), 'xyz')
```

To check whether the x - y - z body fixed rotation angles we chose are suitable for the observed motion in the video we first estimate the likely bounds of motion in terms of multiples of $\pi/2$. For our Euler angles this seems reasonable:

$$\begin{aligned} 0 &\leq \psi \leq \pi \\ -\pi/2 &\leq \theta \leq \pi/2 \\ -\infty &\leq \varphi \leq \infty \end{aligned} \tag{9.25}$$

Now we can check the direction cosine matrix at the limits of ψ and θ to see if they reduce the direction cosine matrix to a form that indicates gimbal lock.

```
sm.trigsimp(T.dcm(N).xreplace({psi: 0}))
```

$$\begin{bmatrix} \cos(\theta) \cos(\varphi) & \sin(\varphi) & -\sin(\theta) \cos(\varphi) \\ -\sin(\varphi) \cos(\theta) & \cos(\varphi) & \sin(\theta) \sin(\varphi) \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \tag{9.26}$$

```
sm.trigsimp(T.dcm(N).xreplace({psi: sm.pi}))
```

$$\begin{bmatrix} \cos(\theta) \cos(\varphi) & -\sin(\varphi) & \sin(\theta) \cos(\varphi) \\ -\sin(\varphi) \cos(\theta) & -\cos(\varphi) & -\sin(\theta) \sin(\varphi) \\ \sin(\theta) & 0 & -\cos(\theta) \end{bmatrix} \tag{9.27}$$

These first matrices show that we can still orient the handle if ψ is at its limits.

```
sm.trigsimp(T.dcm(N).xreplace({theta: -sm.pi/2}))
```

$$\begin{bmatrix} 0 & -\sin(\psi - \varphi) & \cos(\psi - \varphi) \\ 0 & \cos(\psi - \varphi) & \sin(\psi - \varphi) \\ -1 & 0 & 0 \end{bmatrix} \tag{9.28}$$

```
sm.trigsimp(T.dcm(N).xreplace({theta: sm.pi/2}))
```

$$\begin{bmatrix} 0 & \sin(\psi + \varphi) & -\cos(\psi + \varphi) \\ 0 & \cos(\psi + \varphi) & \sin(\psi + \varphi) \\ 1 & 0 & 0 \end{bmatrix} \tag{9.29}$$

These second set of matrices show that gimbal lock can occur if θ reaches its limits. But for the observed motion this limit shouldn't ever be reached. So we can use this Euler angle set to model the T-handle for the observed motion without worry of gimbal lock.

9.6 Time Derivatives of Vectors

Using the definition of angular velocity one can show ([Kane1985], pg. 17) that the time derivative of a unit vector **fixed in B** is related to B 's angular velocity by the following theorem:

$$\frac{^A d\hat{b}_x}{dt} = {}^A \bar{\omega}^B \times \hat{b}_x \quad (9.30)$$

This indicates that the time derivative is always normal to the unit vector because the magnitude of the unit vector is constant and the derivative scales with the magnitude of the angular velocity:

$$\frac{^A d\hat{b}_x}{dt} = |{}^A \bar{\omega}^B| \left({}^A \hat{\omega}^B \times \hat{b}_x \right) \quad (9.31)$$

Now if vector $\bar{v} = v\hat{b}_x$ and v is constant with respect to time we can infer:

$$\frac{^A d\bar{v}}{dt} = v({}^A \bar{\omega}^B \times \hat{b}_x) = {}^A \bar{\omega}^B \times v\hat{b}_x = {}^A \bar{\omega}^B \times \bar{v} \quad (9.32)$$

Eq. (9.30) extends to any vector **fixed in B** and observed from A , making the time derivative equal to the cross product of the angular velocity of B in A with the vector.

Now, if \bar{u} is a vector that is **not fixed in B** we return to the product rule in Section *Product Rule* and first express \bar{u} in B :

$$\bar{u} = u_1 \hat{b}_x + u_2 \hat{b}_y + u_3 \hat{b}_z \quad (9.33)$$

Taking the derivative in another reference frame A by applying the product rule and applying the above theorems let's us arrive at this new theorem:

$$\begin{aligned} \frac{^A d\bar{u}}{dt} &= \dot{u}_1 \hat{b}_x + \dot{u}_2 \hat{b}_y + \dot{u}_3 \hat{b}_z + u_1 \frac{^A d\hat{b}_x}{dt} + u_2 \frac{^A d\hat{b}_y}{dt} + u_3 \frac{^A d\hat{b}_z}{dt} \\ \frac{^A d\bar{u}}{dt} &= \frac{^B d\bar{u}}{dt} + u_1 {}^A \bar{\omega}^B \times \hat{b}_x + u_2 {}^A \bar{\omega}^B \times \hat{b}_y + u_3 {}^A \bar{\omega}^B \times \hat{b}_z \\ \frac{^A d\bar{u}}{dt} &= \frac{^B d\bar{u}}{dt} + {}^A \bar{\omega}^B \times \bar{u} \end{aligned} \quad (9.34)$$

Eq. (9.34) is a powerful equation because it lets us differentiate any vector if we know how it changes in a rotating reference frame relative to the reference frame we are observing the change from.

We can show that Eq. (9.34) holds with an example. Take a z - x orientation and an arbitrary vector that is not fixed in B :

```
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
B.orient_body_fixed(A, (psi, theta, 0), 'Zxz')

u1, u2, u3 = me.dynamicsymbols('u1, u2, u3')

u = u1*B.x + u2*B.y + u3*B.z
```

$$u_1 \hat{b}_x + u_2 \hat{b}_y + u_3 \hat{b}_z \quad (9.35)$$

As we learned in the last chapter we can express the vector in A and then take the time derivative of the measure numbers to arrive at $\frac{^A d\bar{u}}{dt}$:

u.express(A)

$$(u_1 \cos(\psi) - u_2 \sin(\psi) \cos(\theta) + u_3 \sin(\psi) \sin(\theta))\hat{a}_x + (u_1 \sin(\psi) + u_2 \cos(\psi) \cos(\theta) - u_3 \sin(\theta) \cos(\psi))\hat{a}_y + (u_2 \sin(\theta) + u_3 \cos(\theta))\hat{a}_z \quad (9.36)$$

u.express(A).dt(A)

$$(-u_1 \sin(\psi)\dot{\psi} + u_2 \sin(\psi) \sin(\theta)\dot{\theta} - u_2 \cos(\psi) \cos(\theta)\dot{\psi} + u_3 \sin(\psi) \cos(\theta)\dot{\theta} + u_3 \sin(\theta) \cos(\psi)\dot{\psi} + \sin(\psi) \sin(\theta)\dot{u}_3 - \sin(\psi) \cos(\theta)\dot{u}_2) \hat{a}_x + \dots \quad (9.37)$$

But applying the theorem above we can find the derivative with a cross product. The nice aspect of this formulation is there is no need to express the vector in A . First $\frac{^B d\bar{u}}{dt}$:

u.dt(B)

$$\dot{u}_1 \hat{b}_x + \dot{u}_2 \hat{b}_y + \dot{u}_3 \hat{b}_z \quad (9.38)$$

and then ${}^A \bar{\omega}^B$:

A_w_B = B.ang_vel_in(A)
A_w_B

$$\theta \hat{b}_x + \sin(\theta) \dot{\psi} \hat{b}_y + \cos(\theta) \dot{\psi} \hat{b}_z \quad (9.39)$$

$\frac{^A d\bar{u}}{dt}$ is then:

u.dt(B) + me.cross(A_w_B, u)

$$(-u_2 \cos(\theta)\dot{\psi} + u_3 \sin(\theta)\dot{\psi} + \dot{u}_1) \hat{b}_x + (u_1 \cos(\theta)\dot{\psi} - u_3 \dot{\theta} + \dot{u}_2) \hat{b}_y + (-u_1 \sin(\theta)\dot{\psi} + u_2 \dot{\theta} + \dot{u}_3) \hat{b}_z \quad (9.40)$$

which is a relatively simple form of the derivative when expressed in the rotating reference frame.

We can show that the first result is equivalent by expressing in B and simplifying:

u.express(A).dt(A).express(B).simplify()

$$(-u_2 \cos(\theta)\dot{\psi} + u_3 \sin(\theta)\dot{\psi} + \dot{u}_1) \hat{b}_x + (u_1 \cos(\theta)\dot{\psi} - u_3 \dot{\theta} + \dot{u}_2) \hat{b}_y + (-u_1 \sin(\theta)\dot{\psi} + u_2 \dot{\theta} + \dot{u}_3) \hat{b}_z \quad (9.41)$$

Exercise

Show that `.dt()` uses the theorem Eq. (9.34) internally.

Solution

u.dt (A)

$$(-u_2 \cos(\theta)\dot{\psi} + u_3 \sin(\theta)\dot{\psi} + \dot{u}_1)\hat{b}_x + (u_1 \cos(\theta)\dot{\psi} - u_3 \dot{\theta} + \dot{u}_2)\hat{b}_y + (-u_1 \sin(\theta)\dot{\psi} + u_2 \dot{\theta} + \dot{u}_3)\hat{b}_z \quad (9.42)$$

u.dt (B) + me.cross(A_w_B, u)

$$(-u_2 \cos(\theta)\dot{\psi} + u_3 \sin(\theta)\dot{\psi} + \dot{u}_1)\hat{b}_x + (u_1 \cos(\theta)\dot{\psi} - u_3 \dot{\theta} + \dot{u}_2)\hat{b}_y + (-u_1 \sin(\theta)\dot{\psi} + u_2 \dot{\theta} + \dot{u}_3)\hat{b}_z \quad (9.43)$$

9.7 Addition of Angular Velocity

Similar to the relationship in direction cosine matrices of successive orientations (Sec. *Successive Orientations*), there is a relationship among the angular velocities of successively oriented reference frames ([Kane1985], pg. 24) but it relies on the addition of vectors instead of multiplication of matrices. The theorem is:

$${}^A\bar{\omega}^Z = {}^A\bar{\omega}^B + {}^B\bar{\omega}^C + \dots + {}^Y\bar{\omega}^Z \quad (9.44)$$

We can demonstrate this by creating three simple orientations for a Euler *y-x-y* orientation:

```
psi, theta, phi = me.dynamicsymbols('psi, theta, varphi')

A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
C = me.ReferenceFrame('C')
D = me.ReferenceFrame('D')

B.orient_axis(A, psi, A.y)
C.orient_axis(B, theta, B.x)
D.orient_axis(C, phi, C.y)
```

The simple angular velocity of each successive orientation is shown:

```
A_w_B = B.ang_vel_in(A)
A_w_B
```

$$\dot{\psi}\hat{a}_y \quad (9.45)$$

```
B_w_C = C.ang_vel_in(B)
B_w_C
```

$$\dot{\theta}\hat{b}_x \quad (9.46)$$

```
C_w_D = D.ang_vel_in(C)
C_w_D
```

$$\dot{\varphi}\hat{c}_y \quad (9.47)$$

Summing the successive angular velocities gives the compact result:

```
A_w_D = A_w_B + B_w_C + C_w_D
A_w_D
```

$$\dot{\psi}\hat{a}_y + \dot{\theta}\hat{b}_x + \dot{\varphi}\hat{c}_y \quad (9.48)$$

Similarly, we can skip the auxiliary frames and form the relationship between A and D directly and calculate ${}^A\bar{\omega}^D$:

```
A2 = me.ReferenceFrame('A')
D2 = me.ReferenceFrame('D')
D2.orient_body_fixed(A2, (psi, theta, phi), 'YXY')
D2.ang_vel_in(A2).simplify()
```

$$(\sin(\theta)\sin(\varphi)\dot{\psi} + \cos(\varphi)\dot{\theta})\hat{d}_x + (\cos(\theta)\dot{\psi} + \dot{\varphi})\hat{d}_y + (-\sin(\theta)\cos(\varphi)\dot{\psi} + \sin(\varphi)\dot{\theta})\hat{d}_z \quad (9.49)$$

If we express our prior result in D we see the results are the same:

```
A_w_D.express(D)
```

$$(\sin(\theta)\sin(\varphi)\dot{\psi} + \cos(\varphi)\dot{\theta})\hat{d}_x + (\cos(\theta)\dot{\psi} + \dot{\varphi})\hat{d}_y + (-\sin(\theta)\cos(\varphi)\dot{\psi} + \sin(\varphi)\dot{\theta})\hat{d}_z \quad (9.50)$$

9.8 Angular Acceleration

The angular acceleration of B when observed from A is defined as:

$${}^A\bar{\alpha}^B := \frac{^A d}{dt} {}^A\bar{\omega}^B \quad (9.51)$$

${}^A\bar{\omega}^B$ is simply a vector so we can time differentiate it with respect to frame A . Using Eq. (9.34) we can write:

$$\frac{^A d}{dt} {}^A\bar{\omega}^B = \frac{^B d}{dt} {}^A\bar{\omega}^B + {}^A\bar{\omega}^B \times {}^A\bar{\omega}^B \quad (9.52)$$

and since ${}^A\bar{\omega}^B \times {}^A\bar{\omega}^B = 0$:

$$\frac{^A d}{dt} {}^A\bar{\omega}^B = \frac{^B d}{dt} {}^A\bar{\omega}^B \quad (9.53)$$

which is rather convenient.

With SymPy Mechanics ${}^A\bar{\alpha}^B$ is found automatically with `ang_acc_in()` if the orientations are established. For a simple orientation:

```

theta = me.dynamicsymbols('theta')

A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
B.orient_axis(A, theta, A.z)
B.ang_acc_in(A)

```

$$\ddot{\theta}\hat{a}_z \quad (9.54)$$

Similarly we can calculate the derivative manually:

```
B.ang_vel_in(A).dt(A)
```

$$\ddot{\theta}\hat{a}_z \quad (9.55)$$

and see that that Eq. (9.53) holds:

```
B.ang_vel_in(A).dt(B)
```

$$\ddot{\theta}\hat{a}_z \quad (9.56)$$

For a body fixed orientation we get:

```

psi, theta, phi = me.dynamicsymbols('psi, theta, varphi')

A = me.ReferenceFrame('A')
D = me.ReferenceFrame('D')
D.orient_body_fixed(A, (psi, theta, phi), 'YXY')

D.ang_acc_in(A).simplify()

```

$$(\sin(\theta)\sin(\varphi)\ddot{\psi} + \sin(\theta)\cos(\varphi)\dot{\psi}\dot{\varphi} + \sin(\varphi)\cos(\theta)\dot{\psi}\dot{\theta} - \sin(\varphi)\dot{\theta}\dot{\varphi} + \cos(\varphi)\ddot{\theta})\hat{d}_x + (-\sin(\theta)\dot{\psi}\dot{\theta} + \cos(\theta)\ddot{\psi} + \ddot{\varphi})\hat{d}_y + (\sin(\theta)\sin(\varphi)\ddot{\psi} + \sin(\theta)\cos(\varphi)\dot{\psi}\dot{\varphi} + \sin(\varphi)\cos(\theta)\dot{\psi}\dot{\theta} - \sin(\varphi)\dot{\theta}\dot{\varphi} + \cos(\varphi)\ddot{\theta})\hat{d}_x + (-\sin(\theta)\dot{\psi}\dot{\theta} + \cos(\theta)\ddot{\psi} + \ddot{\varphi})\hat{d}_y + (\sin(\theta)\sin(\varphi)\ddot{\psi} + \sin(\theta)\cos(\varphi)\dot{\psi}\dot{\varphi} + \sin(\varphi)\cos(\theta)\dot{\psi}\dot{\theta} - \sin(\varphi)\dot{\theta}\dot{\varphi} + \cos(\varphi)\ddot{\theta})\hat{d}_x + (-\sin(\theta)\dot{\psi}\dot{\theta} + \cos(\theta)\ddot{\psi} + \ddot{\varphi})\hat{d}_y \quad (9.57)$$

and with manual derivatives of the measure numbers:

```
D.ang_vel_in(A).dt(A).simplify()
```

$$(\sin(\theta)\sin(\varphi)\ddot{\psi} + \sin(\theta)\cos(\varphi)\dot{\psi}\dot{\varphi} + \sin(\varphi)\cos(\theta)\dot{\psi}\dot{\theta} - \sin(\varphi)\dot{\theta}\dot{\varphi} + \cos(\varphi)\ddot{\theta})\hat{d}_x + (-\sin(\theta)\dot{\psi}\dot{\theta} + \cos(\theta)\ddot{\psi} + \ddot{\varphi})\hat{d}_y + (\sin(\theta)\sin(\varphi)\ddot{\psi} + \sin(\theta)\cos(\varphi)\dot{\psi}\dot{\varphi} + \sin(\varphi)\cos(\theta)\dot{\psi}\dot{\theta} - \sin(\varphi)\dot{\theta}\dot{\varphi} + \cos(\varphi)\ddot{\theta})\hat{d}_x + (-\sin(\theta)\dot{\psi}\dot{\theta} + \cos(\theta)\ddot{\psi} + \ddot{\varphi})\hat{d}_y + (\sin(\theta)\sin(\varphi)\ddot{\psi} + \sin(\theta)\cos(\varphi)\dot{\psi}\dot{\varphi} + \sin(\varphi)\cos(\theta)\dot{\psi}\dot{\theta} - \sin(\varphi)\dot{\theta}\dot{\varphi} + \cos(\varphi)\ddot{\theta})\hat{d}_x + (-\sin(\theta)\dot{\psi}\dot{\theta} + \cos(\theta)\ddot{\psi} + \ddot{\varphi})\hat{d}_y \quad (9.58)$$

```
D.ang_vel_in(A).dt(D).simplify()
```

$$(\sin(\theta)\sin(\varphi)\ddot{\psi} + \sin(\theta)\cos(\varphi)\dot{\psi}\dot{\varphi} + \sin(\varphi)\cos(\theta)\dot{\psi}\dot{\theta} - \sin(\varphi)\dot{\theta}\dot{\varphi} + \cos(\varphi)\ddot{\theta})\hat{d}_x + (-\sin(\theta)\dot{\psi}\dot{\theta} + \cos(\theta)\ddot{\psi} + \ddot{\varphi})\hat{d}_y + (\sin(\theta)\sin(\varphi)\ddot{\psi} + \sin(\theta)\cos(\varphi)\dot{\psi}\dot{\varphi} + \sin(\varphi)\cos(\theta)\dot{\psi}\dot{\theta} - \sin(\varphi)\dot{\theta}\dot{\varphi} + \cos(\varphi)\ddot{\theta})\hat{d}_x + (-\sin(\theta)\dot{\psi}\dot{\theta} + \cos(\theta)\ddot{\psi} + \ddot{\varphi})\hat{d}_y + (\sin(\theta)\sin(\varphi)\ddot{\psi} + \sin(\theta)\cos(\varphi)\dot{\psi}\dot{\varphi} + \sin(\varphi)\cos(\theta)\dot{\psi}\dot{\theta} - \sin(\varphi)\dot{\theta}\dot{\varphi} + \cos(\varphi)\ddot{\theta})\hat{d}_x + (-\sin(\theta)\dot{\psi}\dot{\theta} + \cos(\theta)\ddot{\psi} + \ddot{\varphi})\hat{d}_y \quad (9.59)$$

Note the equivalence regardless of the frame the change in velocity is observed from.

9.9 Addition of Angular Acceleration

The calculation of angular acceleration is relatively simple due to the equivalence when observed from different reference frames, but the addition of angular velocities explained in Sec. [Addition of Angular Velocity](#) does not extend to angular accelerations. Adding successive angular accelerations does not result in a valid total angular acceleration.

$${}^A\bar{\alpha}^Z \neq {}^A\bar{\alpha}^B + {}^B\bar{\alpha}^C + \dots + {}^Y\bar{\alpha}^Z \quad (9.60)$$

We can show by example that an equality in Eq. (9.60) will not hold. Coming back to the successive orientations that form a y - x - y Euler rotation, we can test the relationship.

```
psi, theta, phi = me.dynamicsymbols('psi, theta, varphi')

A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
C = me.ReferenceFrame('C')
D = me.ReferenceFrame('D')

B.orient_axis(A, psi, A.y)
C.orient_axis(B, theta, B.x)
D.orient_axis(C, phi, C.y)
```

The simple angular acceleration of each successive orientation is shown:

```
A_alp_B = B.ang_acc_in(A)
A_alp_B
```

$$\ddot{\psi}\hat{a}_y \quad (9.61)$$

```
B_alp_C = C.ang_acc_in(B)
B_alp_C
```

$$\ddot{\theta}\hat{b}_x \quad (9.62)$$

```
C_alp_D = D.ang_acc_in(C)
C_alp_D
```

$$\ddot{\varphi}\hat{c}_y \quad (9.63)$$

Summing the successive angular accelerations and expressing the resulting vector in the body fixed reference frame D gives this result:

```
A_alp_D = A_alp_B + B_alp_C + C_alp_D
A_alp_D.express(D).simplify()
```

$$(\sin(\theta)\sin(\varphi)\ddot{\psi} + \cos(\varphi)\ddot{\theta})\hat{d}_x + (\cos(\theta)\ddot{\psi} + \ddot{\varphi})\hat{d}_y + (-\sin(\theta)\cos(\varphi)\ddot{\psi} + \sin(\varphi)\ddot{\theta})\hat{d}_z \quad (9.64)$$

which is not equal to the correct, more complex, result:

```
D.ang_vel_in(A).dt(A).express(D).simplify()
```

$$(\sin(\theta)\sin(\varphi)\ddot{\psi} + \sin(\theta)\cos(\varphi)\dot{\psi}\dot{\varphi} + \sin(\varphi)\cos(\theta)\dot{\psi}\dot{\theta} - \sin(\varphi)\dot{\theta}\dot{\varphi} + \cos(\varphi)\ddot{\theta})\hat{d}_x + (-\sin(\theta)\dot{\psi}\dot{\theta} + \cos(\theta)\ddot{\psi} + \ddot{\varphi})\hat{d}_y + (\sin(\theta)\sin(\varphi)\ddot{\psi} + \sin(\theta)\cos(\varphi)\dot{\psi}\dot{\varphi} + \sin(\varphi)\cos(\theta)\dot{\psi}\dot{\theta} - \sin(\varphi)\dot{\theta}\dot{\varphi} + \cos(\varphi)\ddot{\theta})\hat{d}_z \quad (9.65)$$

Angular accelerations derived from successive orientations require an explicit differentiation of the associated angular velocity vector. There unfortunately is no theorem that simplifies this calculation as we see with orientation and angular velocity.

TRANSLATIONAL KINEMATICS

Note: You can download this example as a Python script: `translational.py` or Jupyter Notebook: `translational.ipynb`.

10.1 Learning Objectives

After completing this chapter readers will be able to:

- calculate the velocity and acceleration of a point in a multibody system
- apply the one and two point theorems to calculate velocity and acceleration of points

10.2 Introduction

In multibody dynamics, we are going to need to calculate the translation velocities and accelerations of points. We will learn that the acceleration of the mass centers of the bodies in a multibody system will be a primary ingredient in forming Newton's Second Law of motion $\bar{F} = m\bar{a}$. This chapter will equip you to calculate the relative translational velocities and accelerations of points in a system.

10.3 Translational Velocity

If a point P is moving with respect to a point O that is fixed in reference frame A the translational velocity vector of point P is defined as:

$${}^A\bar{v}^P := \frac{{}^A d\bar{r}^{P/O}}{dt} \quad (10.1)$$

We also know from Eq. (9.34) that the time derivative of any vector can be written in terms of the angular velocity of the associated reference frames, so:

$$\begin{aligned} {}^A\bar{v}^P &= \frac{{}^A d\bar{r}^{P/O}}{dt} \\ &= \frac{{}^B d\bar{r}^{P/O}}{dt} + {}^A\bar{\omega}^B \times \bar{r}^{P/O} \\ &= {}^B\bar{v}^P + {}^A\bar{\omega}^B \times \bar{r}^{P/O} \end{aligned} \quad (10.2)$$



Fig. 10.1: Kinetic sculpture “Two Turning Vertical Rectangles” (1971) in Rotterdam/The Netherlands (FOP) by George Rickey. https://nl.wikipedia.org/wiki/Two_Turning_Vertical_Rectangles

K.Siereveld, Public domain, via Wikimedia Commons

This formulation will allow us to utilize different reference frames to simplify velocity calculations. Take for example this piece of kinetic art that now stands in Rotterdam:

User <https://www.reddit.com/user/stravalnak> posted this video of the sculpture to Reddit during the 2022 storm Eunice: and it looks very dangerous. It would be interesting to know the velocity and acceleration of various points on this sculpture. First, we sketch a configuration diagram:

Now let's use SymPy Mechanics to calculate Eq. (10.2) for this example.

```
import sympy as sm
import sympy.physics.mechanics as me
me.init_vprinting(use_latex='mathjax')
```

```
class ReferenceFrame(me.ReferenceFrame):
    def __init__(self, *args, **kwargs):
        kwargs.pop('latexs', None)
        lab = args[0].lower()
        tex = r'\hat{{\{}{\}}}_{{}}'
        super(ReferenceFrame, self).__init__(*args,
                                             latexs=(tex.format(lab, 'x'),
                                                      tex.format(lab, 'y'),
                                                      tex.format(lab, 'z')),
                                             **kwargs)
me.ReferenceFrame = ReferenceFrame
```

Set up the orientations:

```
alpha, beta = me.dynamicsymbols('alpha, beta')

N = me.ReferenceFrame('N')
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')

A.orient_axis(N, alpha, N.z)
B.orient_axis(A, beta, A.x)
```

Write out the position vectors to P , S , and Q :

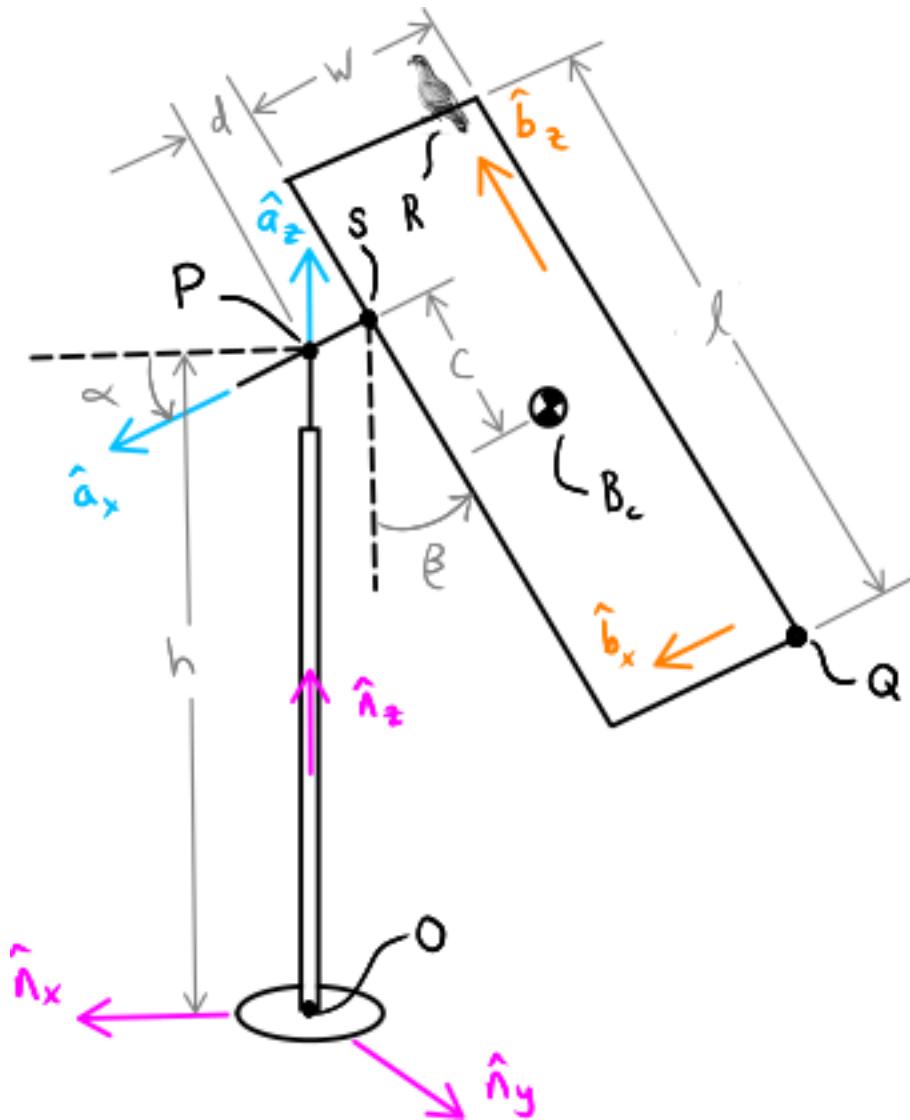


Fig. 10.2: Sketch of one of the two plates mounted on the rotating T-support. Reference frames N , A , and B are shown. Also note the pigeon trying to walk across one edge of the plate at point R .

Pigeon SVG from <https://freesvg.org/vector-clip-art-of-homing-pigeon> Public Domain

```

h, d, w, c, l = sm.symbols('h, d, w, c, l')

r_O_P = h*N.z
r_P_S = -d*A.x
r_S_Q = -w*B.x - (c + l/2)*B.z

r_O_P, r_P_S, r_S_Q

```

$$\left(h\hat{n}_z, -d\hat{a}_x, -w\hat{b}_x + \left(-c - \frac{l}{2}\right)\hat{b}_z \right) \quad (10.3)$$

Now calculate:

$${}^N\bar{v}^S = {}^A\bar{v}^S + {}^N\bar{\omega}^A \times \bar{r}^{S/O} \quad (10.4)$$

S is not moving when observed from A so:

```
(r_O_P + r_P_S).dt(A)
```

$$0 \quad (10.5)$$

The second term does have a value and can be found with these two components:

```
A.ang_vel_in(N)
```

$$\dot{\alpha}\hat{n}_z \quad (10.6)$$

```
me.cross(A.ang_vel_in(N), r_O_P + r_P_S)
```

$$-d\dot{\alpha}\hat{a}_y \quad (10.7)$$

giving ${}^N\bar{v}^S$:

```

N_v_S = (r_O_P + r_P_S).dt(A) + me.cross(A.ang_vel_in(N), r_O_P + r_P_S)
N_v_S

```

$$-d\dot{\alpha}\hat{a}_y \quad (10.8)$$

Similarly for point Q :

```
(r_O_P + r_P_S + r_S_Q).dt(B)
```

$$-h \sin(\alpha)\dot{\beta}\hat{n}_x + h \cos(\alpha)\dot{\beta}\hat{n}_y \quad (10.9)$$

```
me.cross(B.ang_vel_in(N), r_O_P + r_P_S + r_S_Q)
```

$$h \sin(\alpha) \dot{\beta} \hat{n}_x - h \cos(\alpha) \dot{\beta} \hat{n}_y - d \dot{\alpha} \hat{a}_y + \left(-c - \frac{l}{2} \right) \sin(\beta) \dot{\alpha} \hat{b}_x + (-w \cos(\beta) \dot{\alpha} - \left(-c - \frac{l}{2} \right) \dot{\beta}) \hat{b}_y + w \sin(\beta) \dot{\alpha} \hat{b}_z \quad (10.10)$$

```
N_v_Q = (r_O_P + r_P_S + r_S_Q).dt(B) + me.cross(B.ang_vel_in(N), r_O_P + r_P_S + r_S_Q)
```

$$-d \dot{\alpha} \hat{a}_y + \left(-c - \frac{l}{2} \right) \sin(\beta) \dot{\alpha} \hat{b}_x + (-w \cos(\beta) \dot{\alpha} - \left(-c - \frac{l}{2} \right) \dot{\beta}) \hat{b}_y + w \sin(\beta) \dot{\alpha} \hat{b}_z \quad (10.11)$$

SymPy Mechanics provides the `Point` object that simplifies working with position vectors. Start by creating points and setting relative positions among points with `set_pos()`.

```
O = me.Point('O')
P = me.Point('P')
S = me.Point('S')
Q = me.Point('Q')

P.set_pos(O, h*N.z)
S.set_pos(P, -d*A.x)
Q.set_pos(S, -w*B.x - (c + 1/2)*B.z)
```

Once relative positions among points are established you can request the position vector between any pair of points that are connected by the `set_pos()` statements, for example $\bar{r}^{Q/O}$ is:

```
Q.pos_from(O)
```

$$-w \hat{b}_x + \left(-c - \frac{l}{2} \right) \hat{b}_z - d \hat{a}_x + h \hat{n}_z \quad (10.12)$$

Also, once the position vectors are established, velocities can be calculated. You will always explicitly need to set the velocity of at least one point. In our case, we can set ${}^N\bar{v}^O = 0$ with `set_vel()`:

```
O.set_vel(N, 0)
```

Note: SymPy Mechanics has no way of knowing whether the sculpture is fixed on the road or floating around with some constant speed. All the relative velocities of the various points would not be changed in those two scenarios. Hence, at least the speed of one point must be specified.

Now the velocity in N for any point that is connected to O by the prior `set_pos()` statements can be found with the `vel()` method:

```
Q.vel(N)
```

$$\left(-c - \frac{l}{2} \right) \sin(\beta) \dot{\alpha} \hat{b}_x + (-w \cos(\beta) \dot{\alpha} - \left(-c - \frac{l}{2} \right) \dot{\beta}) \hat{b}_y + w \sin(\beta) \dot{\alpha} \hat{b}_z - d \dot{\alpha} \hat{a}_y \quad (10.13)$$

Warning: `vel()` method will calculate velocities naively, i.e. not necessarily give the simplest form.

10.4 Velocity Two Point Theorem

If there are two points P and S fixed in a reference frame A and you know the angular velocity ${}^N\bar{\omega}^A$ and the velocity ${}^N\bar{v}^P$ then ${}^N\bar{v}^S$ can be calculated if the vector $\bar{r}^{S/P}$, which is fixed in A , is known. The following theorem provides a convenient formulation:

$$\begin{aligned} {}^N\bar{v}^S &= \frac{{}^N d\bar{r}^{S/O}}{dt} \\ &= \frac{{}^N d(\bar{r}^{P/O} + \bar{r}^{S/P})}{dt} \\ &= {}^N\bar{v}^P + \frac{{}^N d\bar{r}^{S/P}}{dt} \\ &= {}^N\bar{v}^P + {}^N\bar{\omega}^A \times \bar{r}^{S/P} \end{aligned} \tag{10.14}$$

For our example kinetic sculpture, both O and P are fixed in N , so ${}^N\bar{v}^P = 0$:

`N_v_P = 0 * N.z`

Only the cross product then needs to be formed:

`N_v_S = N_v_P + me.cross(A.ang_vel_in(N), S.pos_from(P))`
`N_v_S`

$$-d\dot{\alpha}\hat{a}_y \tag{10.15}$$

Using pairs of points both fixed in the same reference frame and Eq. (10.14) gives a compact result.

Point objects have the `v2pt_theory()` method for applying the above equation given the other point fixed in the same frame, the frame you want the velocity in, and the frame both points are fixed in. The velocity of P is set to zero using `set_vel()` first to ensure we start with a known velocity.

`P.set_vel(N, 0)`
`S.v2pt_theory(P, N, A)`

$$-d\dot{\alpha}\hat{a}_y \tag{10.16}$$

Note that when you call `v2pt_theory()` it also sets the velocity of point S to this version of the velocity vector:

`S.vel(N)`

$$-d\dot{\alpha}\hat{a}_y \tag{10.17}$$

Both points S and Q are fixed in reference frame B and we just calculated ${}^N\bar{v}^S$, so we can use the two point theorem to find the velocity of Q in a similar fashion by applying:

$${}^N\bar{v}^Q = {}^N\bar{v}^S + {}^N\bar{\omega}^B \times \bar{r}^{Q/S} \quad (10.18)$$

First, using the manual calculation:

```
N_v_Q = N_v_S + me.cross(B.ang_vel_in(N), Q.pos_from(S))
```

$$-d\dot{\alpha}\hat{a}_y + \left(-c - \frac{l}{2}\right) \sin(\beta)\dot{\alpha}\hat{b}_x + (-w \cos(\beta)\dot{\alpha} - \left(-c - \frac{l}{2}\right)\dot{\beta})\hat{b}_y + w \sin(\beta)\dot{\alpha}\hat{b}_z \quad (10.19)$$

and then with the `v2pt_theory()`:

```
Q.v2pt_theory(S, N, B)
```

$$-d\dot{\alpha}\hat{a}_y + \left(-c - \frac{l}{2}\right) \sin(\beta)\dot{\alpha}\hat{b}_x + (-w \cos(\beta)\dot{\alpha} - \left(-c - \frac{l}{2}\right)\dot{\beta})\hat{b}_y + w \sin(\beta)\dot{\alpha}\hat{b}_z \quad (10.20)$$

Exercise

Calculate the velocity of the center of mass of the plate B_c using the two point theorem.

Solution

```
Bc = me.Point('B_c')
Bc.set_pos(S, -c*B.z - w/2*A.x)
Bc.v2pt_theory(S, N, B)
```

$$(-d\dot{\alpha} - \frac{w\dot{\alpha}}{2})\hat{a}_y - c \sin(\beta)\dot{\alpha}\hat{b}_x + c\dot{\beta}\hat{b}_y \quad (10.21)$$

10.5 Velocity One Point Theorem

If you are interested in the velocity of a point R that is moving in a reference frame B and you know the velocity of a point S fixed in B then the velocity of R is the sum of its velocity when observed from B and the velocity of a point fixed in B at R at that instant of time. Put into mathematical terms we get:

$${}^N\bar{v}^R = {}^B\bar{v}^R + {}^N\bar{v}^T \quad (10.22)$$

where point T is a point that coincides with R at that instant.

Combined with the two point theorem for T , you can write:

$${}^N\bar{v}^R = {}^B\bar{v}^R + {}^N\bar{v}^S + {}^N\bar{\omega}^B \times \bar{r}^{R/S} \quad (10.23)$$

In our kinetic sculpture example, if the pigeon R is walking at a distance s in the \hat{b}_x direction from the upper right corner, then we can calculate the velocity of the pigeon when observed from the N reference frame. First establish the position of R :

```
s = me.dynamicsymbols('s')
t = me.dynamicsymbols._t

R = me.Point('R')
R.set_pos(Q, l*B.z + s*B.x)
```

The velocity of the pigeon when observed from B is:

```
B_v_R = s.diff(t)*B.x
B_v_R
```

$$\dot{s}\hat{b}_x \quad (10.24)$$

Now the other terms:

```
r_S_R = R.pos_from(S)
r_S_R
```

$$(-w + s)\hat{b}_x + \left(-c + \frac{l}{2}\right)\hat{b}_z \quad (10.25)$$

```
N_v_T = N_v_S + me.cross(B.ang_vel_in(N), r_S_R)
N_v_T
```

$$-d\dot{\alpha}\hat{a}_y + \left(-c + \frac{l}{2}\right)\sin(\beta)\dot{\alpha}\hat{b}_x + \left(-\left(-c + \frac{l}{2}\right)\dot{\beta} + (-w + s)\cos(\beta)\dot{\alpha}\right)\hat{b}_y - (-w + s)\sin(\beta)\dot{\alpha}\hat{b}_z \quad (10.26)$$

And finally the velocity of the pigeon when observed from N :

```
N_v_R = B_v_R + N_v_T
N_v_R
```

$$\left(\left(-c + \frac{l}{2}\right)\sin(\beta)\dot{\alpha} + \dot{s}\right)\hat{b}_x + \left(-\left(-c + \frac{l}{2}\right)\dot{\beta} + (-w + s)\cos(\beta)\dot{\alpha}\right)\hat{b}_y - (-w + s)\sin(\beta)\dot{\alpha}\hat{b}_z - d\dot{\alpha}\hat{a}_y \quad (10.27)$$

There is a method `v1pt_theory()` that does this calculation. It does require that the point S 's, in our case, velocity is fixed in B before making the computation:

```
S.set_vel(B, 0)
R.v1pt_theory(S, N, B)
```

$$\left(\left(-c + \frac{l}{2}\right)\sin(\beta)\dot{\alpha} + \dot{s}\right)\hat{b}_x + \left(-\left(-c + \frac{l}{2}\right)\dot{\beta} + (-w + s)\cos(\beta)\dot{\alpha}\right)\hat{b}_y - (-w + s)\sin(\beta)\dot{\alpha}\hat{b}_z - d\dot{\alpha}\hat{a}_y \quad (10.28)$$

10.6 Translational Acceleration

The acceleration of point P in reference frame A is defined as

$${}^A\bar{a}^P := \frac{{}^A d^A \bar{v}^P}{dt} \quad (10.29)$$

Using SymPy Mechanics, the acceleration of a point in a reference frame can be calculated with `acc()`:

```
S.acc(N)
```

$$d\dot{\alpha}^2 \hat{a}_x - d\ddot{\alpha} \hat{a}_y \quad (10.30)$$

10.7 Acceleration Two Point Theorem

The two point theorem above has a corollary for acceleration. Starting with the velocity theorem:

$${}^N \bar{v}^S = {}^N \bar{v}^P + {}^N \bar{\omega}^A \times \bar{r}^{S/P} \quad (10.31)$$

the acceleration can be found by applying the definition of acceleration:

$$\begin{aligned} {}^N \bar{a}^S &= \frac{{}^N d({}^N \bar{v}^P)}{dt} + \frac{{}^N d({}^N \bar{\omega}^A \times \bar{r}^{S/P})}{dt} \\ &= {}^N \bar{a}^P + \frac{{}^N d({}^N \bar{\omega}^A)}{dt} \times \bar{r}^{S/P} + {}^N \bar{\omega}^A \times \frac{{}^N d(\bar{r}^{S/P})}{dt} \\ &= {}^N \bar{a}^P + {}^N \bar{\alpha}^A \times \bar{r}^{S/P} + {}^N \bar{\omega}^A \times \left({}^N \bar{\omega}^A \times \bar{r}^{S/P} \right) \end{aligned} \quad (10.32)$$

This presentation of the acceleration shows the tangential component of acceleration:

$${}^N \bar{\alpha}^A \times \bar{r}^{S/P} \quad (10.33)$$

${}^N \bar{\alpha}^A$ can be calculated with `ang_acc_in()`:

```
me.cross(A.ang_acc_in(N), S.pos_from(P))
```

$$-d\ddot{\alpha} \hat{a}_y \quad (10.34)$$

And this presentation also shows the radial component of acceleration:

$${}^N \bar{\omega}^A \times \left({}^N \bar{\omega}^A \times \bar{r}^{S/P} \right) \quad (10.35)$$

which can also be calculated using the methods of with `Point` and `ReferenceFrame`:

```
me.cross(A.ang_vel_in(N), me.cross(A.ang_vel_in(N), S.pos_from(P)))
```

$$d\dot{\alpha}^2 \hat{a}_x \quad (10.36)$$

Lastly, `a2pt_theory()` calculates the acceleration using this theorem with:

S.a2pt_theory(P, N, A)

$$d\dot{\alpha}^2 \hat{a}_x - d\ddot{\alpha} \hat{a}_y \quad (10.37)$$

where S and P are fixed in A and the velocity is desired in N .

Exercise

Calculate the acceleration of point Q with the two point theorem.

Solution

Q.a2pt_theory(S, N, B)

$$d\dot{\alpha}^2 \hat{a}_x - d\ddot{\alpha} \hat{a}_y + (w \sin^2(\beta) \dot{\alpha}^2 + \left(-c - \frac{l}{2}\right) (\sin(\beta) \ddot{\alpha} + \cos(\beta) \dot{\alpha} \dot{\beta}) - \left(-w \cos(\beta) \dot{\alpha} - \left(-c - \frac{l}{2}\right) \dot{\beta}\right) \cos(\beta) \dot{\alpha}) \hat{b}_x + (-w (-\sin(\beta) \dot{\alpha}^2 - \dot{\alpha} \dot{\beta} \cos(\beta) + \dot{\beta}^2) + \left(-c - \frac{l}{2}\right) \dot{\beta} \cos(\beta) \dot{\alpha}) \hat{b}_y \quad (10.38)$$

10.8 Acceleration One Point Theorem

The velocity one point theorem also can be time differentiated to see its acceleration form. Starting with the expanded one point theorem for velocity:

$${}^N \bar{v}^R = {}^B \bar{v}^R + {}^N \bar{v}^S + {}^N \bar{\omega}^B \times \bar{r}^{R/S} \quad (10.39)$$

and taking the time derivative in the frame N the corollary formula for acceleration can be derived:

$$\begin{aligned} {}^N \bar{a}^R &= \frac{N d {}^B \bar{v}^R}{dt} + \frac{N d {}^N \bar{v}^S}{dt} + \frac{N d {}^N \bar{\omega}^B \times \bar{r}^{R/S}}{dt} \\ &= \frac{N d {}^N \bar{v}^R}{dt} + {}^N \bar{\omega}^B \times {}^N \bar{v}^R + {}^N \bar{a}^S + \frac{N d {}^N \bar{\omega}^B}{dt} \times \bar{r}^{R/S} + {}^N \bar{\omega}^B \times \frac{N d \bar{r}^{R/S}}{dt} \\ &= {}^B \bar{a}^R + {}^N \bar{\omega}^B \times {}^B \bar{v}^R + {}^N \bar{a}^S + {}^N \bar{\alpha}^B \times \bar{r}^{R/S} + {}^N \bar{\omega}^B \times \left({}^B \bar{v}^T + {}^N \bar{\omega}^B \times \bar{r}^{R/S}\right) \\ &= {}^B \bar{a}^R + 2 {}^N \bar{\omega}^B \times {}^B \bar{v}^R + {}^N \bar{a}^S + {}^N \bar{\alpha}^B \times \bar{r}^{R/S} + {}^N \bar{\omega}^B \times \left({}^N \bar{\omega}^B \times \bar{r}^{R/S}\right) \end{aligned} \quad (10.40)$$

One of my dynamics professors, Dean Karnopp, liked to call this equation the “five term beast”, as it is about the nastiest equation that shows up in dynamics. Looking carefully at this form, the result of the two point theorem is embedded, so this is equivalent to:

$${}^N \bar{a}^R = {}^B \bar{a}^R + {}^N \bar{a}^T + 2 {}^N \bar{\omega}^B \times {}^B \bar{v}^R \quad (10.41)$$

where T is again the point fixed at R in this instant of time. The term $2 {}^N \bar{\omega}^B \times {}^B \bar{v}^R$ is the **Coriolis acceleration** that arises from R moving in the rotating frame B .

The three terms in Eq. (10.41) can be calculated for our pigeon like so:

```
B_a_R = R.acc(B)
B_a_R
```

$$\ddot{s}\hat{b}_x \quad (10.42)$$

```
N_a_T = R.a2pt_theory(S, N, B)
N_a_T
```

$$d\dot{\alpha}^2 \hat{a}_x - d\ddot{\alpha} \hat{a}_y + \left(-c + \frac{l}{2} \right) \left(\sin(\beta) \ddot{\alpha} + \cos(\beta) \dot{\alpha} \dot{\beta} \right) - (-w + s) \sin^2(\beta) \dot{\alpha}^2 - \left(-c + \frac{l}{2} \right) \dot{\beta} + (-w + s) \cos(\beta) \dot{\alpha} \cos(\beta) \dot{\alpha} \dot{\beta} \quad (10.43)$$

```
2 * me.cross(B.ang_vel_in(N), R.vel(B))
```

$$2 \cos(\beta) \dot{\alpha} \dot{s} \hat{b}_y - 2 \sin(\beta) \dot{\alpha} \dot{s} \hat{b}_z \quad (10.44)$$

The `a1pt_theory()` method can also be used to make this calculation:

```
R.a1pt_theory(S, N, B)
```

$$\left(-c + \frac{l}{2} \right) \left(\sin(\beta) \ddot{\alpha} + \cos(\beta) \dot{\alpha} \dot{\beta} \right) - (-w + s) \sin^2(\beta) \dot{\alpha}^2 - \left(-c + \frac{l}{2} \right) \dot{\beta} + (-w + s) \cos(\beta) \dot{\alpha} \cos(\beta) \dot{\alpha} \dot{\beta} + \left(-c + \frac{l}{2} \right) \cos(\beta) \dot{\alpha} + \ddot{s} \hat{b}_x + \left(-c + \frac{l}{2} \right) \dot{\beta} \quad (10.45)$$

The acceleration of the pigeon when viewed from N is no flapping matter.

HOLONOMIC CONSTRAINTS

Note: You can download this example as a Python script: configuration.py or Jupyter Notebook: configuration.ipynb.

```
import sympy as sm
import sympy.physics.mechanics as me
me.init_vprinting(use_latex='mathjax')

class ReferenceFrame(me.ReferenceFrame):
    def __init__(self, *args, **kwargs):
        kwargs.pop('latexs', None)
        lab = args[0].lower()
        tex = r'\hat{{}}_{{}}'
        super(ReferenceFrame, self).__init__(*args,
                                             latexs=(tex.format(lab, 'x'),
                                                      tex.format(lab, 'y'),
                                                      tex.format(lab, 'z')),
                                             **kwargs)
me.ReferenceFrame = ReferenceFrame
```

11.1 Learning Objectives

After completing this chapter readers will be able to:

- derive and specify the configuration constraints (holonomic constraints) equations for a system of connected rigid bodies
- numerically solve a set of holonomic constraints for the dependent coordinates
- apply a point configuration constraints as a general approach to constraining a system
- calculate the number of generalized coordinates
- choose generalized coordinates
- calculate velocities when holonomic constraints are present

11.2 Four-Bar Linkage

Consider the linkage shown below:

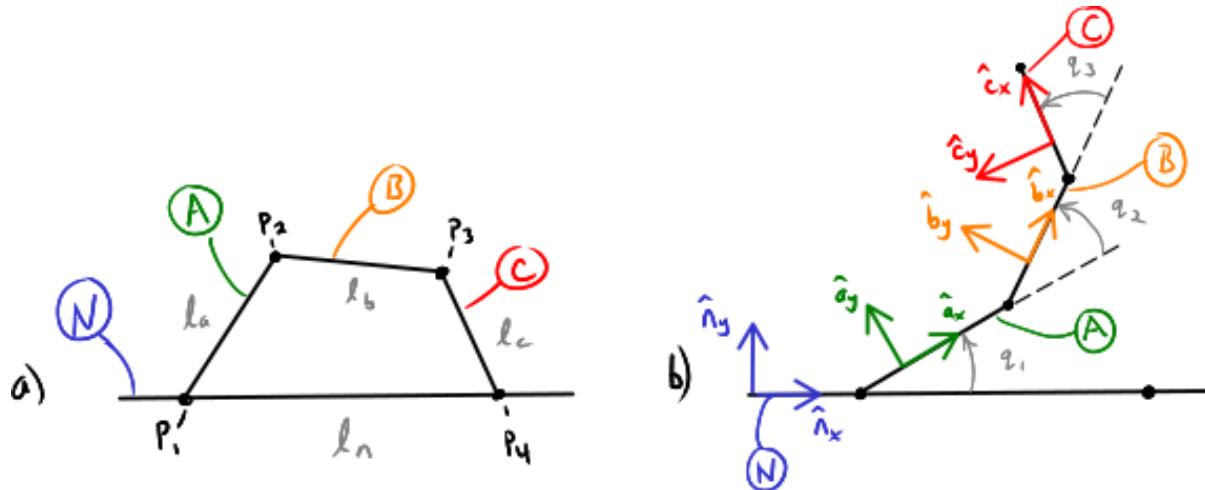


Fig. 11.1: a) Shows four links in a plane A, B, C , and N with respective lengths l_a, l_b, l_c, l_n connected in a closed loop at points P_1, P_2, P_3, P_4 . b) Shows the same linkage that has been separated at point P_4 to make it an open chain of links.

This is a planar **four-bar linkage** with reference frames N, A, B, C attached to each bar. Four bar linkages are used in a wide variety of mechanisms. One you may be familiar with is this rear suspension on a mountain bicycle:

Depending on the length of the links, different motion types are possible. Fig. 11.3 shows some of the possible motions.

A four bar linkage is an example of a *closed kinematic loop*. The case of Fig. 11.1 there are two vector paths to point P_4 from P_1 :

$$\begin{aligned}\bar{r}^{P_4/P_1} &= l_n \hat{n}_x \\ \bar{r}^{P_4/P_1} &= \bar{r}^{P_2/P_1} + \bar{r}^{P_3/P_2} + \bar{r}^{P_4/P_3} = l_a \hat{a}_x + l_b \hat{b}_x + l_c \hat{c}_x\end{aligned}\quad (11.1)$$

For the loop to close, the two vector paths must equate. We can resolve this by disconnecting the loop at some location, P_4 in our case, and forming the *open loop* vector equations to points that should coincide. Keep in mind that we assume that the lengths are constant and the angles change with time.

Setup the variables, reference frames, and points:

```
q1, q2, q3 = me.dynamicsymbols('q1, q2, q3')
l_a, l_b, l_c, l_n = sm.symbols('l_a, l_b, l_c, l_n')

N = me.ReferenceFrame('N')
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
C = me.ReferenceFrame('C')

A.orient_axis(N, q1, N.z)
B.orient_axis(A, q2, A.z)
C.orient_axis(B, q3, B.z)

P1 = me.Point('P1')
P2 = me.Point('P2')
P3 = me.Point('P3')
P4 = me.Point('P4')
```



Fig. 11.2: Four bar linkage shown in blue, red, orange, and green used in the rear suspension mechanism of a mountain bicycle.

Cartemere, CC BY-SA 3.0 <https://creativecommons.org/licenses/by-sa/3.0>, via Wikimedia Commons

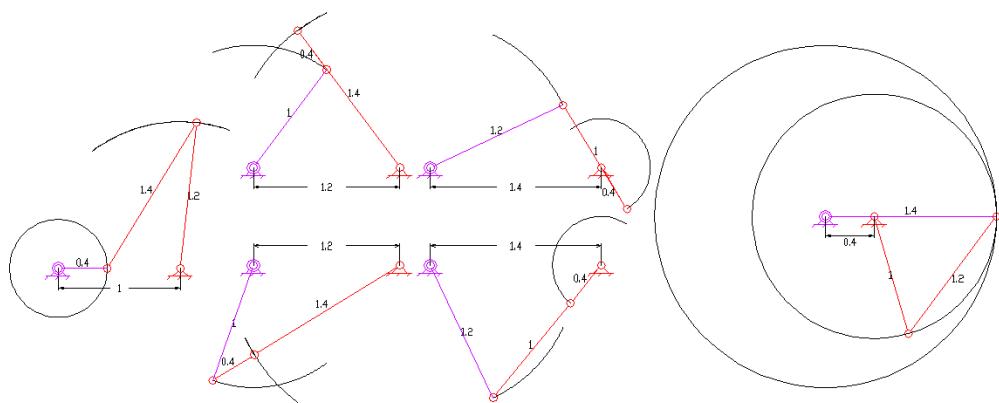


Fig. 11.3: Pasimi, CC BY-SA 4.0 <https://creativecommons.org/licenses/by-sa/4.0>, via Wikimedia Commons

SymPy Mechanics will warn you if you try to establish a closed loop among a set of points and you should not do that because functions that use points have no way to know which vector path you desire to use. Instead you will establish positions among points on one open leg of the chain:

```
P2.set_pos(P1, la*A.x)
P3.set_pos(P2, lb*B.x)
P4.set_pos(P3, lc*C.x)

P4.pos_from(P1)
```

$$l_c \hat{c}_x + l_b \hat{b}_x + l_a \hat{a}_x \quad (11.2)$$

Now, declare a vector for the other path to P_4 :

```
r_P1_P4 = ln*N.x
```

With both vector paths written, we can form the left hand side of the following equation:

$$\bar{r}^{P_4/P_1} - \left(\bar{r}^{P_2/P_1} + \bar{r}^{P_3/P_2} + \bar{r}^{P_4/P_3} \right) = 0 \quad (11.3)$$

Using `pos_from()` for the open loop leg made of points and the additional vector:

```
loop = P4.pos_from(P1) - r_P1_P4
loop
```

$$l_c \hat{c}_x + l_b \hat{b}_x + l_a \hat{a}_x - l_n \hat{n}_x \quad (11.4)$$

This “loop” vector expression must equate to zero for our linkage to always be a closed loop. We have a planar mechanism, so we can extract two scalar equations associated with a pair of unit vectors in the plane of the mechanism. We can pick any two non-parallel unit vectors to express the components in, with the intuitive choice being \hat{n}_x and \hat{y} .

```
fhx = sm.trigsimp(loop.dot(N.x))
fhx
```

$$l_a \cos(q_1) + l_b \cos(q_1 + q_2) + l_c \cos(q_1 + q_2 + q_3) - l_n \quad (11.5)$$

```
fhy = sm.trigsimp(loop.dot(N.y))
fhy
```

$$l_a \sin(q_1) + l_b \sin(q_1 + q_2) + l_c \sin(q_1 + q_2 + q_3) \quad (11.6)$$

For the loop to close, these two expressions must equal zero for all values q_1, q_2, q_3 . These are two nonlinear equations in three time varying variables called coordinates. The solution can be found if we solve for two of the time varying variables. For example, q_2 and q_3 can be solved for in terms of q_1 . We would then say that q_2 and q_3 depend on q_1 . These two equations are called holonomic constraints, or configuration constraints, because they constrain the kinematic configuration to be a loop. Holonomic constraints take the form of a real valued vector function:

$$\bar{f}_h(q_1, \dots, q_N, t) = 0 \text{ where } \bar{f}_h \in \mathbb{R}^M \quad (11.7)$$

N is number of coordinates that you have used to describe the system and M is the number of scalar constraint equations.

Warning: Holonomic constraints are defined strictly as equations that are function of the N time varying coordinates. It is true that these equations are only valid for a limited set of ranges for the constants in the equations, e.g. the lengths of the bars, but the range and combination constraints on the constants are not what we are considering here. Secondly, Eq. (11.7) does not represent inequality constraints. A coordinate may be constrained to a specific range, e.g. $-\pi < q_1 < \pi$, but these are not holonomic constraints in the sense defined here. Inequality constraints are generally dealt with using collision models to capture the real dynamics of forcefully limiting motion.

The four-bar linkage constraints are functions of configuration variables: time varying angles and distances. In our case the constraint equations are:

$$\bar{f}_h(q_1, q_2, q_3) = 0 \text{ where } \bar{f}_h \in \mathbb{R}^2 \quad (11.8)$$

and $N = 3$ and $M = 2$.

In SymPy, we'll typically form this column vector as so:

```
fh = sm.Matrix([f hx, f hy])
fh
```

$$\begin{bmatrix} l_a \cos(q_1) + l_b \cos(q_1 + q_2) + l_c \cos(q_1 + q_2 + q_3) - l_n \\ l_a \sin(q_1) + l_b \sin(q_1 + q_2) + l_c \sin(q_1 + q_2 + q_3) \end{bmatrix} \quad (11.9)$$

Exercise

Watt's Linkage is a four-bar linkage that can generate almost straight line motion of the center point of the middle coupler link. Write the holonomic constraints for the Watt's Linkage. The coupler link has a length of $2a$, the left and right links have length b . Make the vertical distance between the fixed points of the left and right lengths $2a$ and the horizontal distance $(2 - 1/20)b$. Use the same reference frame and angle definitions as the four-bar linkage above.

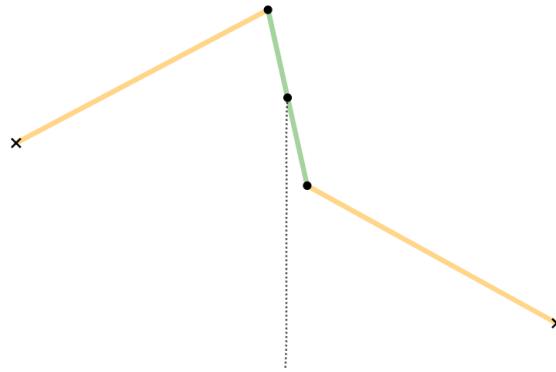


Fig. 11.4: Arglin Kampling, CC BY-SA 4.0 <https://creativecommons.org/licenses/by-sa/4.0>, via Wikimedia Commons

Solution

```

q1, q2, q3 = me.dynamicsymbols('q1, q2, q3')
a, b = sm.symbols('a, b')

N = me.ReferenceFrame('N')
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
C = me.ReferenceFrame('C')

A.orient_axis(N, q1, N.z)
B.orient_axis(A, q2, A.z)
C.orient_axis(B, q3, B.z)

P1 = me.Point('P1')
P2 = me.Point('P2')
P3 = me.Point('P3')
P4 = me.Point('P4')

P2.set_pos(P1, b*A.x)
P3.set_pos(P2, 2*a*B.x)
P4.set_pos(P3, b*C.x)

P4.pos_from(P1)

r_P1_P4 = (2 - sm.S(1)/20)*b*N.x - 2*a*N.y

loop = P4.pos_from(P1) - r_P1_P4

fh_watts = sm.trigsimp(sm.Matrix([loop.dot(N.x), loop.dot(N.y)]))
fh_watts
  
```

$$\begin{bmatrix} 2a \cos(q_1 + q_2) + b \cos(q_1 + q_2 + q_3) + b \cos(q_1) - \frac{39b}{20} \\ 2a \sin(q_1 + q_2) + 2a + b \sin(q_1 + q_2 + q_3) + b \sin(q_1) \end{bmatrix} \quad (11.10)$$

11.3 Solving Holonomic Constraints

Only the simplest of holonomic constraint equations may be solved symbolically due to their nonlinear nature, so you will in general need to solve them numerically. In [Equations of Motion with Holonomic Constraints](#) we will show how to solve them for simulation purposes, but for now SymPy's `nsolve()` can be used to numerically solve the equations. If we choose q_2 and q_3 to be the dependent coordinates, we need to select numerical values for all other variables. Note that not all link length combinations result in a valid linkage geometry. Starting with the replacements,

```

import math # provides pi as a float

repl = {
    la: 1.0,
    lb: 4.0,
    lc: 3.0,
    ln: 5.0,
  
```

(continues on next page)

(continued from previous page)

```

q1: 30.0/180.0*math.pi, # 30 degrees in radians
}
repl

```

$$\{l_a : 1.0, l_b : 4.0, l_c : 3.0, l_n : 5.0, q_1 : 0.523598775598299\} \quad (11.11)$$

we can then formulate the constraint equations such that only q_2 and q_3 are variables:

```
fh.xreplace(repl)
```

$$\left[\begin{array}{l} 4.0 \cos(q_2 + 0.523598775598299) + 3.0 \cos(q_2 + q_3 + 0.523598775598299) - 4.13397459621556 \\ 4.0 \sin(q_2 + 0.523598775598299) + 3.0 \sin(q_2 + q_3 + 0.523598775598299) + 0.5 \end{array} \right] \quad (11.12)$$

Generally, there may be multiple numerical solutions for the unknowns and the underlying algorithms require a guess to return a specific result. If we make an educated guess for the unknowns, then we can find the specific solution with `nsolve()`:

```

q2_guess = -75.0/180.0*math.pi # -75 degrees in radians
q3_guess = 100.0/180.0*math.pi # 100 degrees in radians

sol = sm.nsolve(fh.xreplace(repl), (q2, q3), (q2_guess, q3_guess))
sol/math.pi*180.0 # to degrees

```

$$\left[\begin{array}{l} -79.9561178980214 \\ 108.613175851763 \end{array} \right] \quad (11.13)$$

Exercise

Find the angles of the remaining links in Watt's Linkage if the middle linkage is rotated clockwise 5 degrees, $a = 1$, and $b = 4$.

Solution

The angle relative to vertical of the middle link is $3\pi/2 - (q_1 + q_2)$, which we can use to solve for q_2 .

```

repl = {
    a: 1.0,
    b: 4.0,
    q2: 3.0*math.pi/2.0 - 5.0/180.0*math.pi - q1,
}
repl

```

$$\{a : 1.0, b : 4.0, q_2 : 4.62512251778497 - q_1\} \quad (11.14)$$

```
fh_watts.xreplace(repl)
```

$$\begin{bmatrix} 4.0 \cos(q_3 + 4.62512251778497) + 4.0 \cos(q_1) - 7.97431148549532 \\ 4.0 \sin(q_3 + 4.62512251778497) + 4.0 \sin(q_1) + 0.00761060381650891 \end{bmatrix} \quad (11.15)$$

```
q1_guess = 10.0/180.0*math.pi
q3_guess = 100.0/180.0*math.pi

sol = sm.nsolve(fh_watts.xreplace(repl), (q1, q3), (q1_guess, q3_guess))
sol/math.pi*180.0 # to degrees
```

$$\begin{bmatrix} 4.53780194767253 \\ 90.3528330377729 \end{bmatrix} \quad (11.16)$$

11.4 General Holonomic Constraints

If you consider a set of v points, P_1, P_2, \dots, P_v that can move unconstrained in Euclidean 3D space, then one would need $3v$ constraint equations to fix the points (fully constrain the motion) in that Euclidean space. For the four points in the four-bar linkage, we would then need $3(4) = 12$ constraints to lock all the points fully in place. The figure below will be used to illustrate the general idea of constraining the configuration of the four bar linkage.

Starting with a), there are the four points in 3D Euclidean space that are free to move. Moving to b), each of the four points can be then constrained to be in a plane with:

$$\begin{aligned} \bar{r}^{P_1/O} \cdot \hat{n}_z &= 0 \\ \bar{r}^{P_2/O} \cdot \hat{n}_z &= 0 \\ \bar{r}^{P_3/O} \cdot \hat{n}_z &= 0 \\ \bar{r}^{P_4/O} \cdot \hat{n}_z &= 0 \end{aligned} \quad (11.17)$$

where O is a point fixed in N . This applies four constraints leaving 8 coordinates for the planar location of the points. Now at c) we constrain the points with:

$$\begin{aligned} |\bar{r}^{P_2/P_1}| &= l_a \\ |\bar{r}^{P_3/P_2}| &= l_b \\ |\bar{r}^{P_4/P_3}| &= l_c \\ |\bar{r}^{P_4/P_1}| &= l_n \end{aligned} \quad (11.18)$$

These four constraint equations keep the points within the specified distances from each other leaving 4 coordinates free. In d) point P_1 is fixed relative to O with 2 scalar constraints:

$$\begin{aligned} \bar{r}^{P_1/O} \cdot \hat{n}_x &= 0 \\ \bar{r}^{P_1/O} \cdot \hat{n}_y &= 0 \end{aligned} \quad (11.19)$$

Finally in e), P_4 is constrained with the single scalar:

$$\bar{r}^{P_4/P_1} \cdot \hat{n}_y = 0 \quad (11.20)$$

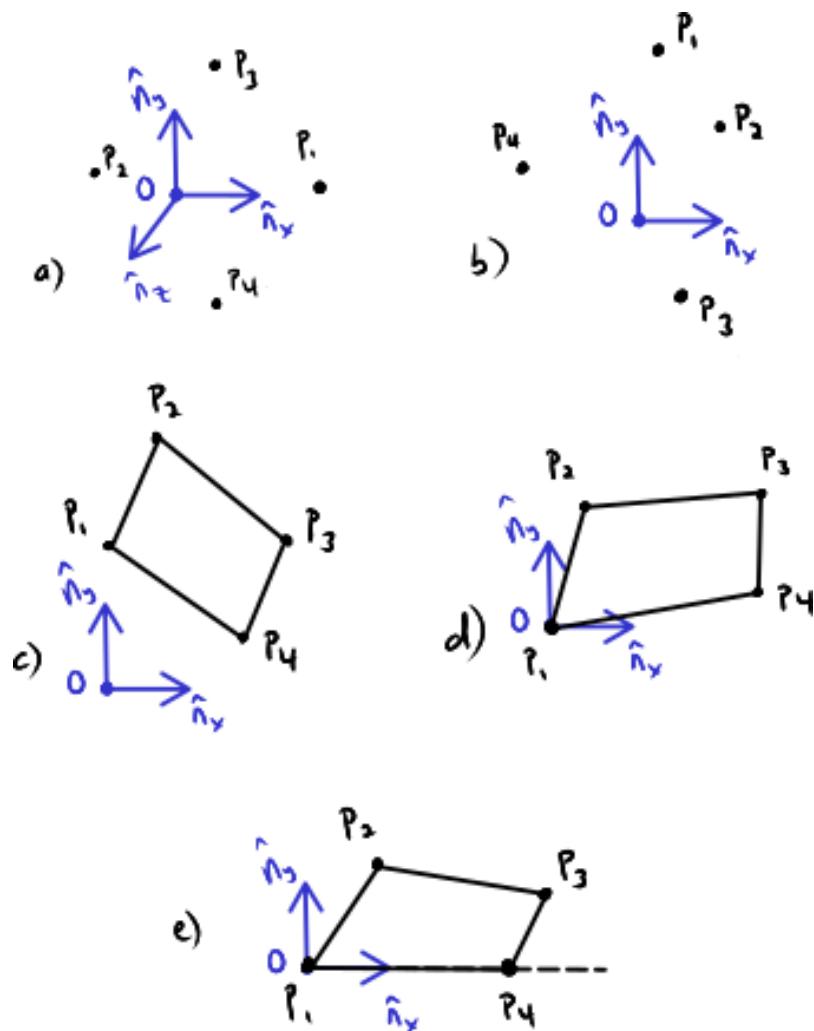


Fig. 11.5: a) Four points in 3D space, b) four points constrained to 2D space, c) points are fixed to adjacent points by a fixed length, d) the first point is fixed at O in two dimensions, e) the fourth point is fixed in the y coordinate relative to O .

Notice that we did not need $\bar{r}^{P_4/P_1} \cdot \hat{n}_x = 0$, because (11.18) ensures the x coordinate of P_4 is in the correct location.

These 11 constraints leave a single free coordinate to describe the orientation of A , B , and C in N . When we originally sketched Fig. 11.1 most of these constraints were implied, i.e. we drew a planar mechanism with points P_1 and P_4 fixed in N , but formally there are 12 coordinates needed to locate the four points and 11 constraints that constrain them to have the configuration of a four-bar linkage.

A general holonomic constraint for a set of v points with Cartesian coordinates is then ([Kane1985] pg. 35):

$$f_h(x_1, y_1, z_1, \dots, x_v, y_v, z_v, t) = 0 \quad (11.21)$$

We include t as it is possible that the constraint is an explicit function of time (instead of only implicit, as seen above in the four-bar linkage example).

11.5 Generalized Coordinates

If a set of v points are constrained with M holonomic constraints then only n of the Cartesian coordinates are independent of each other. The number of independent coordinates is then defined as ([Kane1985] pg. 37):

$$n := 3v - M \quad (11.22)$$

These n independent Cartesian coordinates can also be expressed as n functions of time $q_1(t), q_2(t), \dots, q_n(t)$ in such a way that the constraint equations are always satisfied. These functions $q_1(t), q_2(t), \dots, q_n(t)$ are called *generalized coordinates* and it is possible to find n independent coordinates that minimize the number of explicit constraint equations needed to describe the system's configuration at all times t . These generalized coordinates are typically determined by inspection of the system and there is a bit of an art to choosing the best set. But you can always fall back to the formal process of constraining each relevant point. If you describe your system with $N \leq 3v$ coordinates then:

$$n := N - M \quad (11.23)$$

Take this simple pendulum with points O and P as an example:

If the pendulum length l is constant and the orientation between A and N can change, then the location of P relative to O can be described with the Cartesian coordinates x and y . It should be clear that x and y depend on each other for this system. The constraint relationship between those two coordinates is:

$$x^2 + y^2 = l^2 \quad (11.24)$$

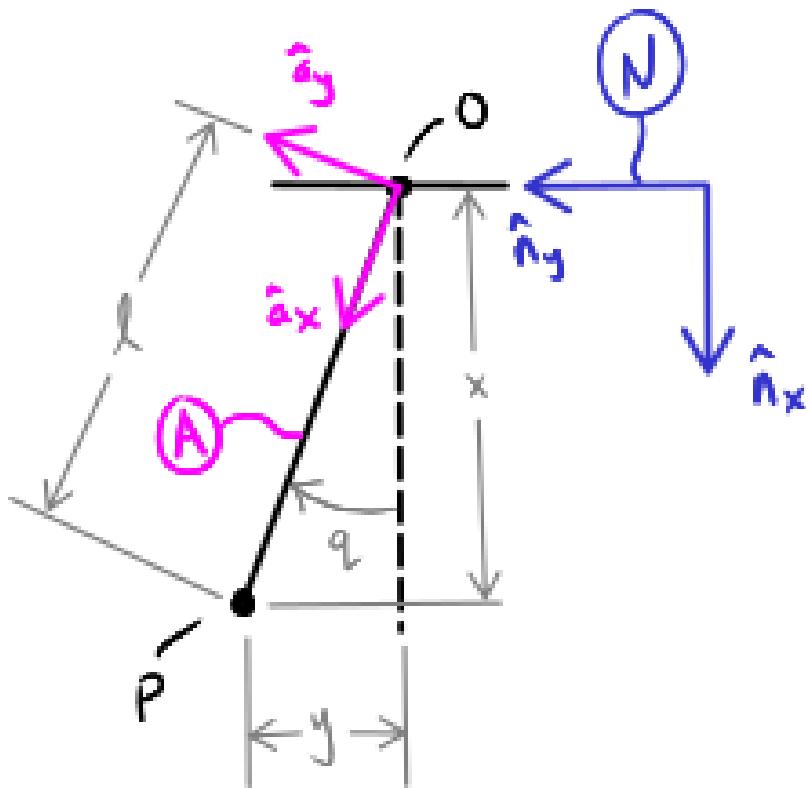
This implies that only one coordinate is independent, i.e. $n = 1$. More formally, the two points give $3v = 3(2) = 6$ and there are 2 constraints for the planar motion of each point, 2 constraints fixing O in N , and 1 constraint fixing the distance from O to P , making $M = 5$ and thus confirming our intuition $n = 6 - 5 = 1$.

But there may be functions of time that relieve us from having to consider Eq. (11.24). For example, these two coordinates can also be written as functions of the angle q :

$$\begin{aligned} x &= l \cos q \\ y &= l \sin q \end{aligned} \quad (11.25)$$

and if we describe the configuration with only q , the constraint is implicitly satisfied. q is then a generalized coordinate because it satisfies $n = 1$ and we do not have to explicitly define a constraint equation.

Now, let's return to the four-bar linkage example in Fig. 11.1 and think about what the generalized coordinates of this system are. We know, at least intuitively, that $n = 1$ for the four bar linkage. The four-bar linkage in Fig. 11.1 is described in a way that assumes a number of constraints are fulfilled, such as Eqs. (11.17) and (11.19), so we do not have to formally consider them.



Exercise

Are q_1, q_2, q_3 generalized coordinates of the four-bar linkage? If not, why?

Solution

Any one of the q_1, q_2, q_3 can be a generalized coordinate, but only one. The other two are dependent due to the two constraints. We started with three coordinates q_1, q_2, q_3 describing the open chain P_1 to P_2 to P_3 to P_4 . Then we have two scalar constraint equations, leaving $n = 1$. Thus we can choose q_1 , q_2 , or q_3 to be the independent generalized coordinate.

If we take the general approach, starting with four unconstrained points, we need 11 constraints to describe the system, but if we select generalized coordinates to describe the system we only need 2 constraint equations (Eq. (11.8))! This simplifies the mathematical problem description and, as we will later see, is essential for obtaining the simplest forms of the equations of motion of a multibody system.

11.6 Calculating Additional Kinematic Quantities

You will often need to calculate velocities and accelerations of points and reference frames of systems with holonomic constraints. Due to the differentiation chain rule, velocities will be linear in the time derivatives of the coordinates and accelerations will be linear in the double time derivatives of the coordinates. Our holonomic constraints dictate that there is no relative motion between points or reference frames, implying that the relevant positions, velocities, and accelerations will all equate to zero.

Start by setting up the points for the four-bar linkage again:

```
P1 = me.Point('P1')
P2 = me.Point('P2')
P3 = me.Point('P3')
P4 = me.Point('P4')
P2.set_pos(P1, la*A.x)
P3.set_pos(P2, lb*B.x)
P4.set_pos(P3, lc*C.x)
```

In the four-bar linkage, ${}^N\bar{v}^{P_4}$ must be zero. We can calculate the unconstrained velocity like so:

```
P1.set_vel(N, 0)
P4.vel(N)
```

$$l_c(\dot{q}_1 + \dot{q}_2 + \dot{q}_3)\hat{c}_y + l_b(\dot{q}_1 + \dot{q}_2)\hat{b}_y + l_a\dot{q}_1\hat{a}_y \quad (11.26)$$

The scalar velocity constraints can be formed in a similar fashion as the configuration constraints:

$$\begin{aligned} {}^N\bar{v}^{P_4} \cdot \hat{n}_x &= 0 \\ {}^N\bar{v}^{P_4} \cdot \hat{n}_y &= 0 \end{aligned} \quad (11.27)$$

```
sm.trigsimp(P4.vel(N).dot(N.x))
```

$$-l_a \sin(q_1)\dot{q}_1 - l_b(\dot{q}_1 + \dot{q}_2) \sin(q_1 + q_2) - l_c(\dot{q}_1 + \dot{q}_2 + \dot{q}_3) \sin(q_1 + q_2 + q_3) \quad (11.28)$$

```
sm.trigsimp(P4.vel(N).dot(N.y))
```

$$l_a \cos(q_1)\dot{q}_1 + l_b(\dot{q}_1 + \dot{q}_2) \cos(q_1 + q_2) + l_c(\dot{q}_1 + \dot{q}_2 + \dot{q}_3) \cos(q_1 + q_2 + q_3) \quad (11.29)$$

Notice that this is identical to taking the time derivative of the constraint vector function \bar{f}_h :

```
t = me.dynamicsymbols._t
fhd = fh.diff(t)
fhd
```

$$\begin{bmatrix} -l_a \sin(q_1)\dot{q}_1 - l_b(\dot{q}_1 + \dot{q}_2) \sin(q_1 + q_2) - l_c(\dot{q}_1 + \dot{q}_2 + \dot{q}_3) \sin(q_1 + q_2 + q_3) \\ l_a \cos(q_1)\dot{q}_1 + l_b(\dot{q}_1 + \dot{q}_2) \cos(q_1 + q_2) + l_c(\dot{q}_1 + \dot{q}_2 + \dot{q}_3) \cos(q_1 + q_2 + q_3) \end{bmatrix} \quad (11.30)$$

We can see that the expressions are linear in \dot{q}_1 , \dot{q}_2 and \dot{q}_3 . If we select \dot{q}_2 and \dot{q}_3 to be dependent, we can solve the linear system $\mathbf{A}\bar{x} = \bar{b}$ for those variables using the technique shown in [Solving Linear Systems](#). First we define a column vector holding the dependent variables:

```
x = sm.Matrix([q2.diff(t), q3.diff(t)])
```

$$\begin{bmatrix} \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} \quad (11.31)$$

then extract the linear terms:

```
A = fhd.jacobian(x)
```

$$\begin{bmatrix} -l_b \sin(q_1 + q_2) - l_c \sin(q_1 + q_2 + q_3) & -l_c \sin(q_1 + q_2 + q_3) \\ l_b \cos(q_1 + q_2) + l_c \cos(q_1 + q_2 + q_3) & l_c \cos(q_1 + q_2 + q_3) \end{bmatrix} \quad (11.32)$$

find the terms not linear in the dependent variables:

```
b = -fhd.xreplace({q2.diff(t): 0, q3.diff(t): 0})
```

$$\begin{bmatrix} l_a \sin(q_1) \dot{q}_1 + l_b \sin(q_1 + q_2) \dot{q}_1 + l_c \sin(q_1 + q_2 + q_3) \dot{q}_1 \\ -l_a \cos(q_1) \dot{q}_1 - l_b \cos(q_1 + q_2) \dot{q}_1 - l_c \cos(q_1 + q_2 + q_3) \dot{q}_1 \end{bmatrix} \quad (11.33)$$

and finally solve for the dependent variables:

```
x_sol = sm.simplify(A.LUsolve(b))
```

$$\begin{bmatrix} -\left(\frac{l_a \sin(q_2)}{\tan(q_3)} + l_a \cos(q_2) + l_b\right) \dot{q}_1 \\ \frac{l_a(l_b \sin(q_2) + l_c \sin(q_2 + q_3)) \dot{q}_1}{l_b l_c \sin(q_3)} \end{bmatrix} \quad (11.34)$$

Now we can write any velocity strictly in terms of the independent speed \dot{q}_1 and all of the other coordinates. `free_dynamicsymbols()` shows us what coordinates and their time derivatives present an any vector:

```
P4.vel(N).free_dynamicsymbols(N)
```

$$\{q_1, q_2, q_3, \dot{q}_1, \dot{q}_2, \dot{q}_3\} \quad (11.35)$$

Using the results in `x_sol` above we can write the velocity in terms of only the independent \dot{q}_1 :

$${}^N\bar{v}^A = v_x(\dot{q}_1, q_1, q_2, q_3) \hat{n}_x + v_y(\dot{q}_1, q_1, q_2, q_3) \hat{n}_y + v_z(\dot{q}_1, q_1, q_2, q_3) \hat{n}_z \quad (11.36)$$

Making the substitutions gives the desired result:

```
qd_dep_repl = {
    q2.diff(t): x_sol[0, 0],
    q3.diff(t): x_sol[1, 0],
}
qd_dep_repl
```

$$\left\{ \dot{q}_2 : -\frac{\left(\frac{l_a \sin(q_2)}{\tan(q_3)} + l_a \cos(q_2) + l_b\right) \dot{q}_1}{l_b}, \dot{q}_3 : \frac{l_a (l_b \sin(q_2) + l_c \sin(q_2 + q_3)) \dot{q}_1}{l_b l_c \sin(q_3)} \right\} \quad (11.37)$$

```
P4.vel(N).xreplace(qd_dep_repl)
```

$$l_c \left(\frac{l_a (l_b \sin(q_2) + l_c \sin(q_2 + q_3)) \dot{q}_1}{l_b l_c \sin(q_3)} + \dot{q}_1 - \frac{\left(\frac{l_a \sin(q_2)}{\tan(q_3)} + l_a \cos(q_2) + l_b\right) \dot{q}_1}{l_b} \right) \hat{c}_y + l_b \left(\dot{q}_1 - \frac{\left(\frac{l_a \sin(q_2)}{\tan(q_3)} + l_a \cos(q_2) + l_b\right) \dot{q}_1}{l_b} \right) \hat{b}_y \quad (11.38)$$

```
P4.vel(N).xreplace(qd_dep_repl).free_dynamicsymbols(N)
```

$$\{q_1, q_2, q_3, \dot{q}_1\} \quad (11.39)$$

The holonomic constraints will have to be solved numerically as described in *Solving Holonomic Constraints*, but once done only the independent \dot{q}_1 is needed.

NONHOLONOMIC CONSTRAINTS

Note: You can download this example as a Python script: `motion.py` or Jupyter Notebook: `motion.ipynb`.

```
import sympy as sm
import sympy.physics.mechanics as me
me.init_vprinting(use_latex='mathjax')

class ReferenceFrame(me.ReferenceFrame):
    def __init__(self, *args, **kwargs):
        kwargs.pop('latexs', None)
        lab = args[0].lower()
        tex = r'\hat{{}}_{{}}_{{}}'
        super(ReferenceFrame, self).__init__(*args,
                                             latexs=(tex.format(lab, 'x'),
                                                      tex.format(lab, 'y'),
                                                      tex.format(lab, 'z')),
                                             **kwargs)
me.ReferenceFrame = ReferenceFrame
```

12.1 Learning Objectives

After completing this chapter readers will be able to:

- Formulate nonholonomic constraints to constrain motion.
- Determine if a nonholonomic constraint is essential and not simply a differentiated holonomic constraint.
- Formulate a nonholonomic constraint for rolling without slip.
- Define kinematical differential equations and solve them to put in first order form.
- Select different choices of generalized speeds.
- Solve for the dependent generalized speeds in terms of the independent generalized speeds.
- Calculate the degrees of freedom of a multibody system.

12.2 Motion Constraints

In *Holonomic Constraints*, we discussed constraints on the configuration of a system. Configuration only concerns where points are and how reference frames are oriented. In this chapter, we will consider constraints on the motion of a system. Motion concerns how points and reference frames move. Take parallel parking a car as a motivating example.

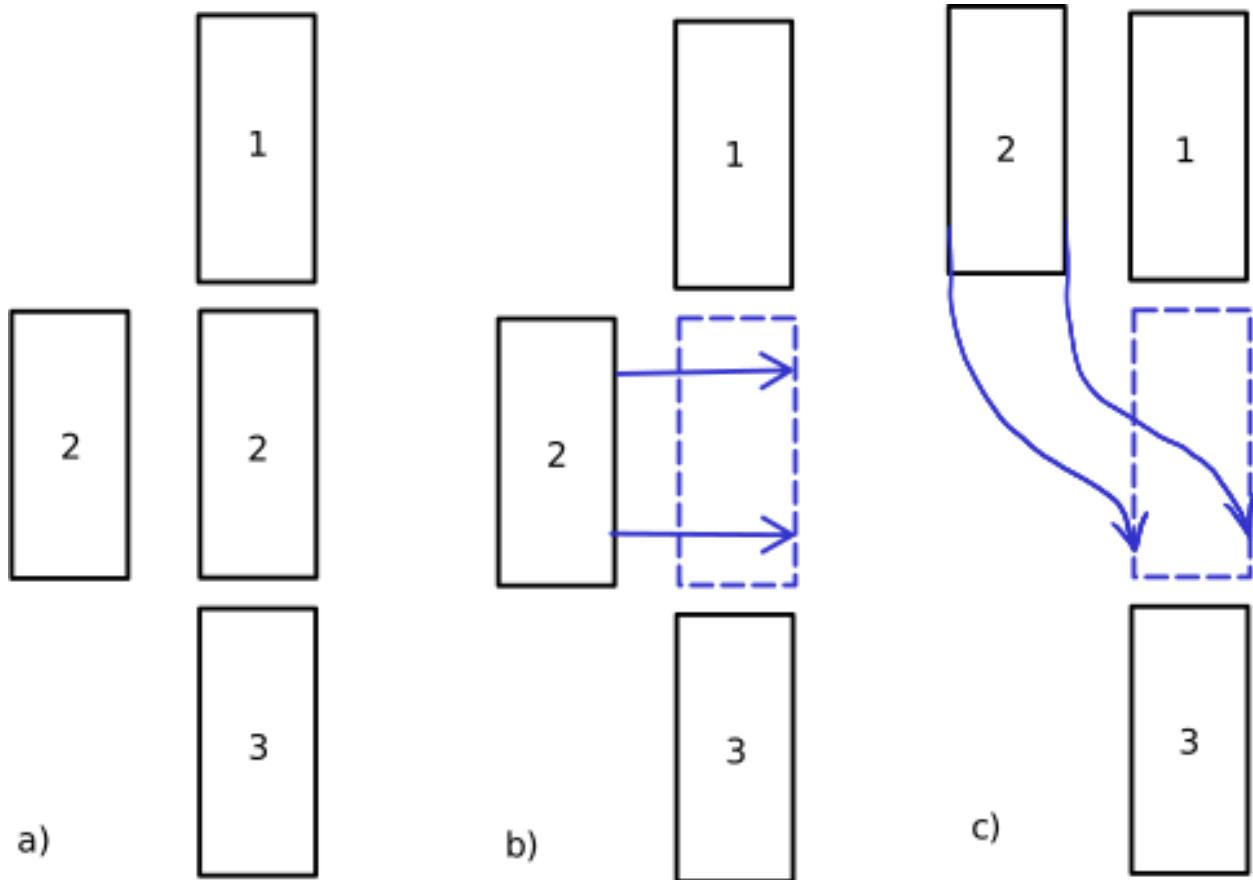


Fig. 12.1: a) two positions (or configurations) of car 2 relative to cars 1 and 3, b) simplest motion to move car 2 into an empty spot between cars 1 and 3, c) actual motion to move car 2 into the empty spot

We know that car 2 can be in either the left or right location in a), i.e. the car's configuration permits either location. But the motion scenario in b) is not possible. A car cannot move from the left configuration to the right configuration by simply sliding directly to the right (see the note below if you question this). Although, this surely would be nice if we could. A car has wheels and only the front wheels can be steered, so the scenario in c) is a viable motion for the car to end up in the correct final configuration. The car has to *move* in a specific way to get from one configuration to another. This implies that we have some kind of constraint on the motion but not the configuration. Constraints such as these are called *nonholonomic constraints* and they take the form:

$$\begin{aligned} \bar{f}_n(\dot{\bar{q}}, \bar{q}, t) &= 0 \\ \text{where} \\ \bar{f}_n &\in \mathbb{R}^m \\ \bar{q} &= [q_1, \dots, q_n]^T \in \mathbb{R}^n \end{aligned} \tag{12.1}$$

The m constraints involve the time derivatives of the generalized coordinates and arise from scalar equations derived from velocities.

Note: We could find a very strong person to push the car sideways, overcoming the very high resisting friction force. It is important to note that any constraint is just a model of a physical phenomena. We know that if we push hard enough and low enough that the car's lateral motion is not constrained. Also, if the car were on ice, then the nonholonomic constraint would be a poor modeling decision.

12.3 Chaplygin Sleigh

Take the simple example of the [Chaplygin Sleigh](#), sketched out in Fig. 12.2. A sleigh can slide along a flat plane, but can only move in the direction it is pointing, much like the wheels of the car above. This system is described by three generalized coordinates x, y, θ . For the motion to only occur along its body fixed \hat{a}_x direction, the component of velocity in the body fixed \hat{a}_y direction must equal zero at all times.

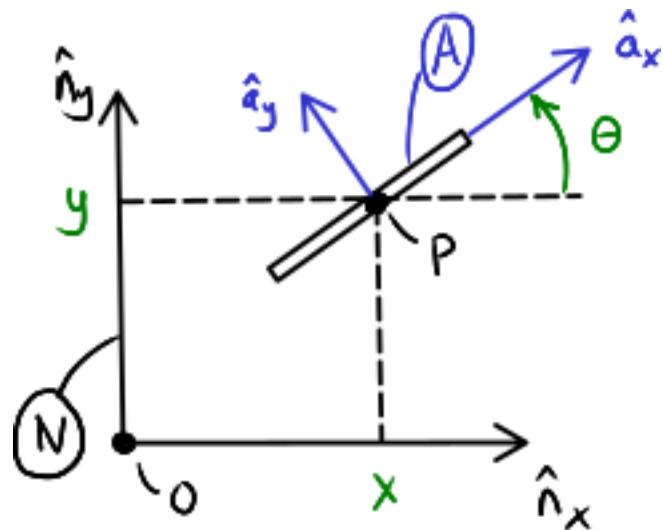


Fig. 12.2: Configuration diagram of a Chaplygin Sleigh. The rectangle A represents a sleigh moving on a plane. Point P represents the center of the sleigh.

Using SymPy Mechanics we can find the velocity of P and express it in the A reference frame:

```

x, y, theta = me.dynamicsymbols('x, y, theta')

N = me.ReferenceFrame('N')
A = me.ReferenceFrame('A')

A.orient_axis(N, theta, N.z)

O = me.Point('O')
P = me.Point('P')

P.set_pos(O, x*N.x + y*N.y)
O.set_vel(N, 0)

P.vel(N).express(A)

```

$$(\sin(\theta)\dot{y} + \cos(\theta)\dot{x})\hat{a}_x + (-\sin(\theta)\dot{x} + \cos(\theta)\dot{y})\hat{a}_y \quad (12.2)$$

The single scalar nonholonomic constraint then takes this form:

$${}^N\bar{v}^P \cdot \hat{a}_y = 0 \quad (12.3)$$

because there can be no velocity component in the \hat{a}_y direction. With SymPy, this is:

```
fn = P.vel(N).dot(A.y)
fn
```

$$-\sin(\theta)\dot{x} + \cos(\theta)\dot{y} \quad (12.4)$$

How do we know that this is, in fact, a nonholonomic constraint and not simply the time derivative of a holonomic constraint?

Recall one of the four-bar linkage holonomic constraints arising from Eq. (11.3) and time differentiate it:

```
t = me.dynamicsymbols._t
q1, q2, q3 = me.dynamicsymbols('q1, q2, q3')
l_a, l_b, l_c, l_n = sm.symbols('l_a, l_b, l_c, l_n')
f hx = l_a*sm.cos(q1) + l_b*sm.cos(q1 + q2) + l_c*sm.cos(q1 + q2 + q3) - l_n
sm.trigsimp(f hx.diff(t))
```

$$-l_a \sin(q_1)\dot{q}_1 - l_b (\dot{q}_1 + \dot{q}_2) \sin(q_1 + q_2) - l_c (\dot{q}_1 + \dot{q}_2 + \dot{q}_3) \sin(q_1 + q_2 + q_3) \quad (12.5)$$

This looks like a nonholonomic constraint, i.e. it has time derivatives of the coordinates, but we know that if we integrate this equation with respect to time we can retrieve the original holonomic constraint, so it really isn't a nonholonomic constraint even though it looks like one.

So if we can integrate f_n with respect to time and we arrive at a function of only the generalized coordinates and time, then we do not have a nonholonomic constraint, but a holonomic constraint in disguise. Unfortunately, it is not generally possible to integrate f_n so we can check the integrability of f_n indirectly.

If f_n of the sleigh was the time derivative of a holonomic constraint f_h then it must be able to be expressed in this form:

$$f_n = \frac{df_h}{dt} = \frac{\partial f_h}{\partial x} \frac{dx}{dt} + \frac{\partial f_h}{\partial y} \frac{dy}{dt} + \frac{\partial f_h}{\partial \theta} \frac{d\theta}{dt} + \frac{\partial f_h}{\partial t} \quad (12.6)$$

and a [condition of integrability](#) is that the mixed partial derivatives must commute. By inspection of f_n we see that we can extract the partial derivatives by collecting the coefficients. SymPy's `coeff()` can extract the linear coefficients for us:

```
dfdx = fn.coeff(x.diff(t))
dfdy = fn.coeff(y.diff(t))
dfdth = fn.coeff(theta.diff(t))

dfdx, dfdy, dfdth
```

$$(-\sin(\theta), \cos(\theta), 0) \quad (12.7)$$

Each pair of mixed partials can be calculated. For example $\frac{\partial^2 f_h}{\partial y \partial x}$ and $\frac{\partial^2 f_h}{\partial x \partial y}$:

```
dfdx.diff(y), dfdy.diff(x)
```

$$(0, 0) \quad (12.8)$$

and the other two pairs:

```
dfdx.diff(theta), dfdth.diff(x)
```

$$(-\cos(\theta), 0) \quad (12.9)$$

```
dfdy.diff(theta), dfdth.diff(y)
```

$$(-\sin(\theta), 0) \quad (12.10)$$

We see that for the last two pairs, the mixed partials do not commute. This proves that f_n is not integrable and is thus an essential nonholonomic constraint that is not a holonomic constraint in disguise.

Exercise

Check whether the mixed partials of the time derivative of the four-bar linkage constraints commute.

Solution

```
fnx = fhx.diff(t)
dfdq1 = fnx.diff(q1)
dfdq2 = fnx.diff(q2)
dfdq3 = fnx.diff(q3)
```

All of the mixed partials are the same:

```
dfdq1.diff(q2) - dfdq2.diff(q1)
```

$$0 \quad (12.11)$$

```
dfdq2.diff(q3) - dfdq3.diff(q2)
```

$$0 \quad (12.12)$$

$$\text{dfdq3.diff}(q1) - \text{dfdq1.diff}(q3)$$

$$0 \quad (12.13)$$

All of the mixed partials are the same so this is a holonomic constraint in disguise.

12.4 Rolling Without Slip

It is quite common to make the modeling assumption that a wheel rolls without slip. A wheel best provides its beneficial properties of rolling and propulsion by ensuring that the friction between the wheel and the surface it rolls on is sufficiently high. This avoids relative motion between a point fixed on the wheel and a point fixed on the surface located at the wheel-surface contact location at any given time. This nature can be modeled by a motion constraint. The key to developing the constraint to ensure there is no relative slip velocity is to identify the correct two points, calculate the velocity of those points, and specify that the relative velocity is zero.

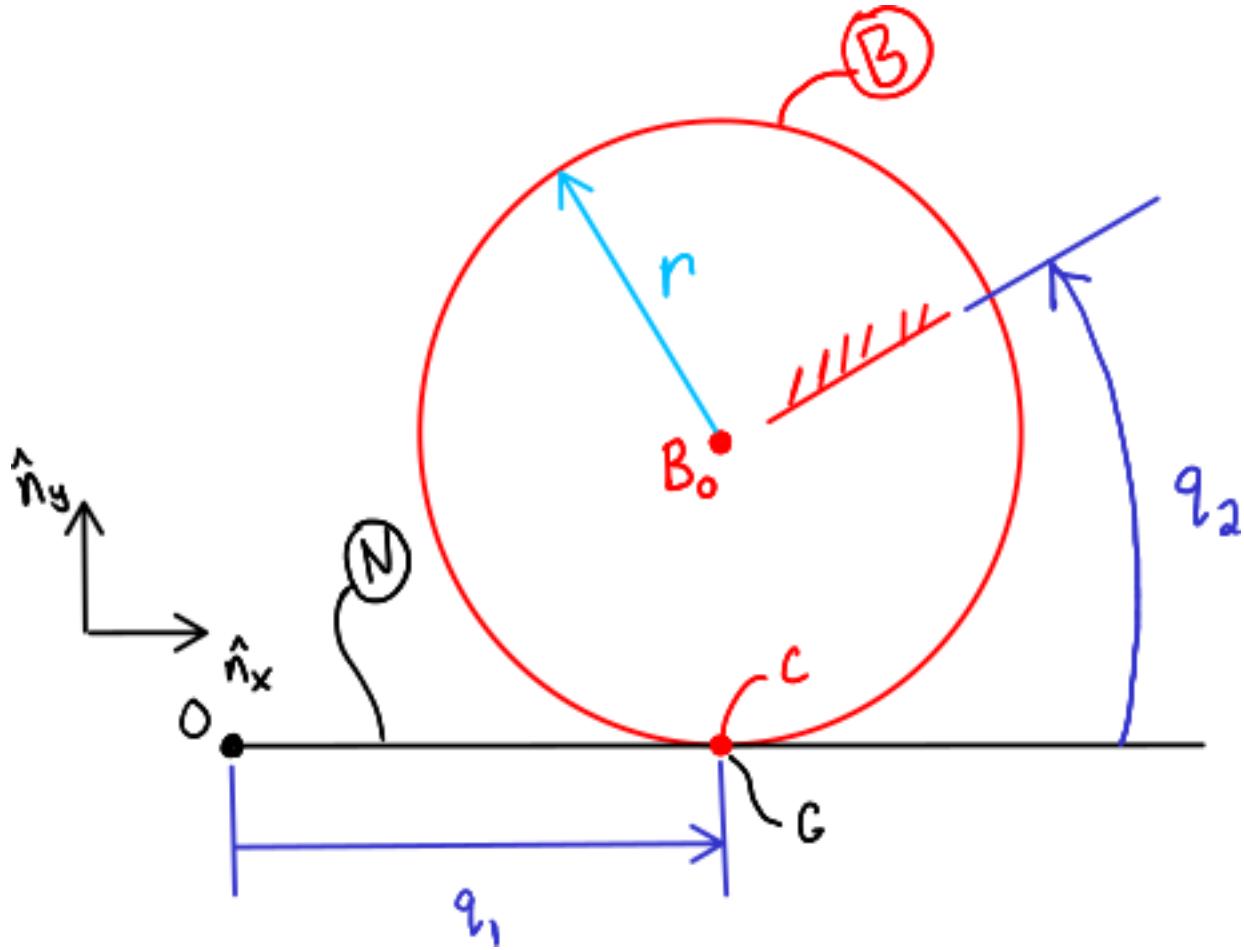


Fig. 12.3: A 2D disc B rolling on a motionless plane N .

For example, when a 2D disc B rolls without slip over a motionless plane N (Fig. 12.3), the velocity of a point C fixed in B at the contact point with the plane must be zero to ensure no slip when observed from the plane's reference frame.

We can state this mathematically as:

$${}^N\bar{v}^C = 0 \quad (12.14)$$

One must be careful about calculating this velocity and recognizing that there are numerous points of possible interest at the same wheel-plane contact location. You may consider these points, for example:

- A point B_C that moves in the plane N which is *always* located at the wheel-plane contact location. The coordinate q_1 tracks this point in the figure.
- A point G_C that is fixed in the wheel which follows a [cycloid](#) curve as it rolls along.
- A point G that is fixed in the plane which is located at the wheel-plane contact point at any given instance of time.
- A point C that is fixed in the wheel which is located at the wheel-plane contact point at any given instance of time.

A motion constraint that ensures rolling without slip, can only be formed by considering the last two points. The vector constraint equation is:

$${}^N\bar{v}^C - {}^N\bar{v}^G = 0 \quad (12.15)$$

Point G is fixed in N so it has no velocity in N :

$${}^N\bar{v}^G = 0 \quad (12.16)$$

Point C is fixed in B . To determine its velocity, take B_o to be the wheel center which is also fixed in B . Since both points are fixed in B we can apply the two point velocity theorem.

$${}^N\bar{v}^C = {}^N\bar{v}^{B_o} + {}^N\bar{\omega}^B \times \bar{r}^{C/B_o} \quad (12.17)$$

We can then use two generalized coordinates to describe the position q_1 (from O fixed in N) and rotation q_2 of the wheel. The velocity of the wheel center is then:

$${}^N\bar{v}^{B_o} = \dot{q}_1 \hat{n}_x \quad (12.18)$$

The cross product terms are found with the radius of the wheel with r and the angular velocity to give the velocity of C :

$$\begin{aligned} {}^N\bar{v}^C &= \dot{q}_1 \hat{n}_x - \dot{q}_2 \hat{n}_z \times -r \hat{n}_y \\ {}^N\bar{v}^C &= \dot{q}_1 \hat{n}_x - \dot{q}_2 r \hat{n}_x \end{aligned} \quad (12.19)$$

Applying the motion constraint and knowing that ${}^N\bar{v}^G = 0$ gives us this scalar constraint equation:

$$\dot{q}_1 - \dot{q}_2 r = 0 \quad (12.20)$$

This is a scalar constraint equation that ensures rolling without slip and involves the time derivatives of the coordinates. It is integrable and thus actually a holonomic constraint, i.e. $q_1 - q_2 r = 0$. General rolling without slip in three dimensions will be nonholonomic. Take care to calculate the relative velocities of the two points fixed in each of the bodies in rolling contact that are located at the contact point at that *instance of time*.

12.5 Kinematical Differential Equations

In Eq. (12.1) we show the form of the nonholonomic constraints in terms of \dot{q} . Newton's and Euler's Second Laws of motion will require calculation of acceleration and angular acceleration respectively. These laws of motion are second order differential equations because it involves second time derivatives of distances and angles. Any second order differential equation can be equivalently represented by two first order differential equations by introducing a new variable for any first derivative terms. We are working towards writing the equations of motion of a multibody system, which will

be differential equations that are most useful for simulation when in a first order form. To do this, we now introduce the variables $\bar{u} = [u_1, \dots, u_n]^T$ and define them as linear functions of the time derivatives of the generalized coordinates $\dot{q}_1, \dots, \dot{q}_n$. These variables are called *generalized speeds*. They take the form:

$$\bar{u} := \mathbf{Y}_k(\bar{q}, t)\dot{\bar{q}} + \bar{z}_k(\bar{q}, t) \quad (12.21)$$

\bar{u} must be chosen such that \mathbf{Y}_k is invertible. If it is, then we solve for $\dot{\bar{q}}$ we can write these first order differential equations as such:

$$\dot{\bar{q}} = \mathbf{Y}_k^{-1}(\bar{u} - \bar{z}_k) \quad (12.22)$$

Eq. (12.22) are called the *kinematical differential equations*.

The most common, and always valid, choice of generalized speeds is:

$$\bar{u} = \mathbf{I}\dot{\bar{q}} \quad (12.23)$$

where \mathbf{I} is the identity matrix. This results in $u_i = \dot{q}_i$ for $i = 1, \dots, n$.

Now that we have introduced generalized speeds, the nonholonomic constraints can then be written as:

$$\begin{aligned} \bar{f}_n(\bar{u}, \bar{q}, t) &= 0 \\ \text{where} \\ \bar{f}_n &\in \mathbb{R}^m \\ \bar{u} &= [u_1, \dots, u_n]^T \in \mathbb{R}^n \\ \bar{q} &= [q_1, \dots, q_n]^T \in \mathbb{R}^n \end{aligned} \quad (12.24)$$

12.6 Choosing Generalized Speeds

There are many possible choices for generalized speed and you are free to select them as you please, as long as they fit the form of equation (12.21) and \mathbf{Y}_k is invertible. Some selections of generalized speeds can reduce the complexity of important velocity expressions and if selected carefully may reduce the complexity of the equations of motion we will derive in a later chapters (see [Mitiguy1996] for examples). To see some examples of selecting generalized speeds, take for example the angular velocity of a reference frame which is oriented with a z - x - y body fixed orientation:

```
q1, q2, q3 = me.dynamicsymbols('q1, q2, q3')
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
B.orient_body_fixed(A, (q1, q2, q3), 'ZXY')
A_w_B = B.ang_vel_in(A).simplify()
A_w_B
```

$$(-\sin(q_3)\cos(q_2)\dot{q}_1 + \cos(q_3)\dot{q}_2)\hat{b}_x + (\sin(q_2)\dot{q}_1 + \dot{q}_3)\hat{b}_y + (\sin(q_3)\dot{q}_2 + \cos(q_2)\cos(q_3)\dot{q}_1)\hat{b}_z \quad (12.25)$$

12.6.1 Choice 1

If we choose the simplest definition for the u 's, i.e. $u_1 = \dot{q}_1$, $u_2 = \dot{q}_2$, and $u_3 = \dot{q}_3$, the angular velocity takes this form:

```
u1, u2, u3 = me.dynamicsymbols('u1, u2, u3')
t = me.dynamicsymbols._t
qdot = sm.Matrix([q1.diff(t), q2.diff(t), q3.diff(t)])
u = sm.Matrix([u1, u2, u3])

A_w_B = A_w_B.xreplace(dict(zip(qdot, u)))
A_w_B
```

$$(-u_1 \sin(q_3) \cos(q_2) + u_2 \cos(q_3))\hat{b}_x + (u_1 \sin(q_2) + u_3)\hat{b}_y + (u_1 \cos(q_2) \cos(q_3) + u_2 \sin(q_3))\hat{b}_z \quad (12.26)$$

```
Yk_plus_zk = qdot
Yk_plus_zk
```

$$\begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} \quad (12.27)$$

Recall from [Solving Linear Systems](#) that the Jacobian is a simple way to extract the coefficients of linear terms into a coefficient matrix for a system of linear equations. In this case, we see that this results in the identity matrix.

```
Yk = Yk_plus_zk.jacobian(qdot)
Yk
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (12.28)$$

Now find \bar{z}_k by setting the time derivatives of the generalized coordinates to zero:

```
qd_zero_repl = dict(zip(qdot, sm.zeros(3, 1)))
qd_zero_repl
```

$$\{\dot{q}_1 : 0, \dot{q}_2 : 0, \dot{q}_3 : 0\} \quad (12.29)$$

```
zk = Yk_plus_zk.xreplace(qd_zero_repl)
zk
```

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad (12.30)$$

The linear equation can be solved for the \dot{q} 's, (Eq. (12.22)):

```
sm.Eq(qdot, Yk.LUsolve(u - zk))
```

$$\begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \quad (12.31)$$

12.6.2 Choice 2

Another valid choice is to set the u 's equal to each measure number of the angular velocity expressed in B :

$$\begin{aligned} u_1 &= {}^A\bar{\omega}^B \cdot \hat{b}_x \\ u_2 &= {}^A\bar{\omega}^B \cdot \hat{b}_y \\ u_3 &= {}^A\bar{\omega}^B \cdot \hat{b}_z \end{aligned} \quad (12.32)$$

so that:

$${}^A\bar{\omega}^B = u_1\hat{b}_x + u_2\hat{b}_y + u_3\hat{b}_z \quad (12.33)$$

```
A_w_B = B.ang_vel_in(A).simplify()
A_w_B
```

$$(-\sin(q_3)\cos(q_2)\dot{q}_1 + \cos(q_3)\dot{q}_2)\hat{b}_x + (\sin(q_2)\dot{q}_1 + \dot{q}_3)\hat{b}_y + (\sin(q_3)\dot{q}_2 + \cos(q_2)\cos(q_3)\dot{q}_1)\hat{b}_z \quad (12.34)$$

```
u1_expr = A_w_B.dot(B.x)
u2_expr = A_w_B.dot(B.y)
u3_expr = A_w_B.dot(B.z)

Yk_plus_zk = sm.Matrix([u1_expr, u2_expr, u3_expr])
Yk_plus_zk
```

$$\begin{bmatrix} -\sin(q_3)\cos(q_2)\dot{q}_1 + \cos(q_3)\dot{q}_2 \\ \sin(q_2)\dot{q}_1 + \dot{q}_3 \\ \sin(q_3)\dot{q}_2 + \cos(q_2)\cos(q_3)\dot{q}_1 \end{bmatrix} \quad (12.35)$$

```
Yk = Yk_plus_zk.jacobian(qdot)
Yk
```

$$\begin{bmatrix} -\sin(q_3)\cos(q_2) & \cos(q_3) & 0 \\ \sin(q_2) & 0 & 1 \\ \cos(q_2)\cos(q_3) & \sin(q_3) & 0 \end{bmatrix} \quad (12.36)$$

```
zk = Yk_plus_zk.xreplace(qd_zero_repl)
zk
```

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad (12.37)$$

Now we form:

```
sm.Eq(qdot, sm.trigsimp(Yk.LUsolve(u - zk)))
```

$$\begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = \begin{bmatrix} \frac{-u_1 \sin(q_3) - u_3 \cos(q_3)}{\cos(q_2)} \\ \frac{u_1 \cos(2q_3) + u_1 + u_3 \sin(2q_3)}{2 \cos(q_3)} \\ u_1 \sin(q_3) \tan(q_2) + u_2 - u_3 \cos(q_3) \tan(q_2) \end{bmatrix} \quad (12.38)$$

Note: Notice how the kinematical differential equations are not valid when q_2 or q_3 are even multiples of $\pi/2$. If your system must orient through these values, you'll need to select a different body fixed rotation or an orientation method that isn't susceptible to these issues.

12.6.3 Choice 3

Another valid choice is to set the u 's equal to each measure number of the angular velocity expressed in A :

$$\begin{aligned} u_1 &= {}^A\bar{\omega}^B \cdot \hat{a}_x \\ u_2 &= {}^A\bar{\omega}^B \cdot \hat{a}_y \\ u_3 &= {}^A\bar{\omega}^B \cdot \hat{a}_z \end{aligned} \quad (12.39)$$

so that:

$${}^A\bar{\omega}^B = u_1 \hat{a}_x + u_2 \hat{a}_y + u_3 \hat{a}_z \quad (12.40)$$

```
A_w_B = B.ang_vel_in(A).express(A).simplify()
A_w_B
```

$$(-\sin(q_1) \cos(q_2) \dot{q}_3 + \cos(q_1) \dot{q}_2) \hat{a}_x + (\sin(q_1) \dot{q}_2 + \cos(q_1) \cos(q_2) \dot{q}_3) \hat{a}_y + (\sin(q_2) \dot{q}_3 + \dot{q}_1) \hat{a}_z \quad (12.41)$$

```
u1_expr = A_w_B.dot(A.x)
u2_expr = A_w_B.dot(A.y)
u3_expr = A_w_B.dot(A.z)

Yk_plus_zk = sm.Matrix([u1_expr, u2_expr, u3_expr])
Yk_plus_zk
```

$$\begin{bmatrix} -\sin(q_1)\cos(q_2)\dot{q}_3 + \cos(q_1)\dot{q}_2 \\ \sin(q_1)\dot{q}_2 + \cos(q_1)\cos(q_2)\dot{q}_3 \\ \sin(q_2)\dot{q}_3 + \dot{q}_1 \end{bmatrix} \quad (12.42)$$

```
Yk = Yk_plus_zk.jacobian(qdot)
Yk
```

$$\begin{bmatrix} 0 & \cos(q_1) & -\sin(q_1)\cos(q_2) \\ 0 & \sin(q_1) & \cos(q_1)\cos(q_2) \\ 1 & 0 & \sin(q_2) \end{bmatrix} \quad (12.43)$$

```
zk = Yk_plus_zk.xreplace(qd_zero_repl)
zk
```

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad (12.44)$$

```
sm.Eq(qdot, sm.trigsimp(Yk.LUsolve(u - zk)))
```

$$\begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = \begin{bmatrix} (u_1 \sin(q_1) - u_2 \cos(q_1)) \tan(q_2) + u_3 \\ u_1 \cos(q_1) + u_2 \sin(q_1) \\ -\frac{u_1 \sin(q_1) - u_2 \cos(q_1)}{\cos(q_2)} \end{bmatrix} \quad (12.45)$$

12.7 Snakeboard

A **snakeboard** is a variation on a skateboard that can be propelled via nonholonomic locomotion [Ostrowski1994]. Similar to the Chaplygin Sleigh, the wheels can generally only travel in the direction they are pointed. This classic video from 1993 shows how to propel the board:

Fig. 12.4 shows what a real Snakeboard looks like and Fig. 12.5 shows a configuration diagram.

Start by defining the time varying variables and constants:

```
q1, q2, q3, q4, q5 = me.dynamicsymbols('q1, q2, q3, q4, q5')
l = sm.symbols('l')
```

The reference frames are all simple rotations about the axis normal to the plane:

```
N = me.ReferenceFrame('N')
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
C = me.ReferenceFrame('C')
```

(continues on next page)



Fig. 12.4: Example of a snakeboard that shows the two footpads each with attached truck and pair of wheels that are connected by the coupler.

Николайков Вячеслав, CC BY-SA 3.0, via Wikimedia Commons

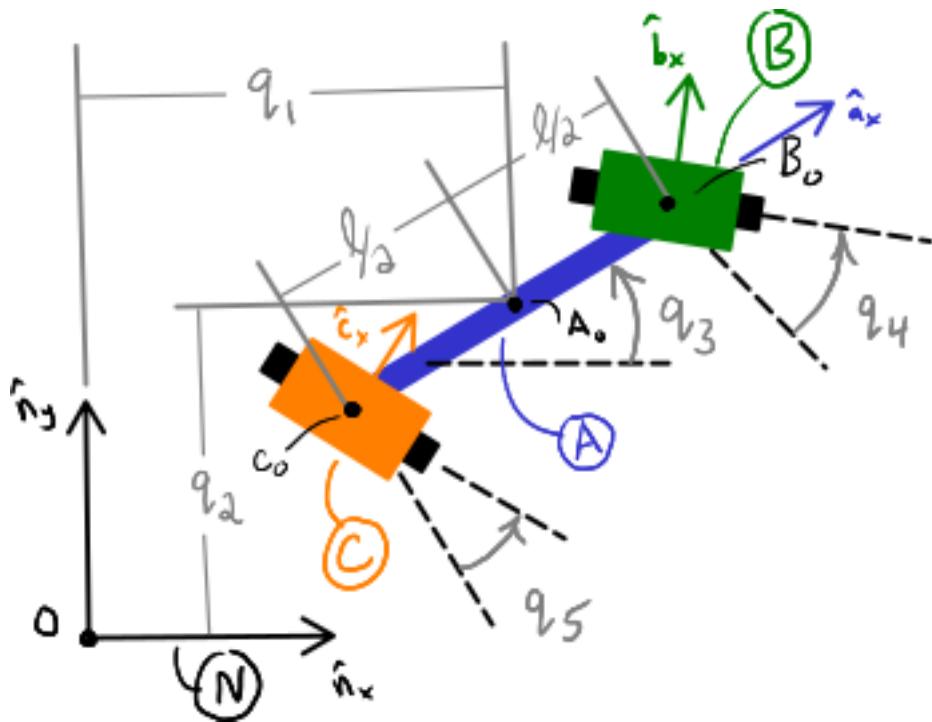


Fig. 12.5: Configuration diagram of a planar Snakeboard model.

(continued from previous page)

```
A.orient_axis(N, q3, N.z)
B.orient_axis(A, q4, A.z)
C.orient_axis(A, q5, A.z)
```

The angular velocities of each reference frame are then:

```
A.ang_vel_in(N)
```

$$\dot{q}_3 \hat{n}_z \quad (12.46)$$

```
B.ang_vel_in(N)
```

$$\dot{q}_4 \hat{a}_z + \dot{q}_3 \hat{n}_z \quad (12.47)$$

```
C.ang_vel_in(N)
```

$$\dot{q}_5 \hat{a}_z + \dot{q}_3 \hat{n}_z \quad (12.48)$$

Establish the position vectors among the points:

```
O = me.Point('O')
Ao = me.Point('A_o')
Bo = me.Point('B_o')
Co = me.Point('C_o')

Ao.set_pos(O, q1*N.x + q2*N.y)
Bo.set_pos(Ao, 1/2*A.x)
Co.set_pos(Ao, -1/2*A.x)
```

The velocity of A_o in N is a simple time derivative:

```
O.set_vel(N, 0)
Ao.vel(N)
```

$$\dot{q}_1 \hat{n}_x + \dot{q}_2 \hat{n}_y \quad (12.49)$$

The two point theorem is handy for computing the other two velocities:

```
Bo.v2pt_theory(Ao, N, A)
```

$$\dot{q}_1 \hat{n}_x + \dot{q}_2 \hat{n}_y + \frac{l \dot{q}_3}{2} \hat{a}_y \quad (12.50)$$

```
Co.v2pt_theory(Ao, N, A)
```

$$\dot{q}_1 \hat{n}_x + \dot{q}_2 \hat{n}_y - \frac{l\dot{q}_3}{2} \hat{a}_y \quad (12.51)$$

The unit vectors of B and C are aligned with the wheels of the Snakeboard. This lets us impose that there is no velocity in the direction normal to the wheel's rolling direction by taking dot products with the respectively reference frames' y direction unit vector to form the two nonholonomic constraints:

$$\begin{aligned} {}^A\bar{v}^{Bo} \cdot \hat{b}_y &= 0 \\ {}^A\bar{v}^{Co} \cdot \hat{c}_y &= 0 \end{aligned} \quad (12.52)$$

```
fn = sm.Matrix([Bo.vel(N).dot(B.y),
               Co.vel(N).dot(C.y)])
fn = sm.trigsimp(fn)
fn
```

$$\begin{bmatrix} \frac{l \cos(q_4)\dot{q}_3}{2} - \sin(q_3 + q_4)\dot{q}_1 + \cos(q_3 + q_4)\dot{q}_2 \\ -\frac{l \cos(q_5)\dot{q}_3}{2} - \sin(q_3 + q_5)\dot{q}_1 + \cos(q_3 + q_5)\dot{q}_2 \end{bmatrix} \quad (12.53)$$

Now we introduce some generalized speeds. By inspection of the above constraint equations, we can see that defining a generalized speed equal to $\frac{l\dot{q}_3}{2}$ can simplify the equations a bit. So define these generalized speeds:

$$\begin{aligned} u_i &= \dot{q}_i \text{ for } i = 1, 2, 4, 5 \\ u_3 &= \frac{l\dot{q}_3}{2} \end{aligned} \quad (12.54)$$

Now replace all of the time derivatives of the generalized coordinates with the generalized speeds. We use `subs()` here because the replacement isn't an exact replacement (in the sense of `xreplace()`).

```
u1, u2, u3, u4, u5 = me.dynamicsymbols('u1, u2, u3, u4, u5')

u_repl = {
    q1.diff(): u1,
    q2.diff(): u2,
    1*q3.diff()/2: u3,
    q4.diff(): u4,
    q5.diff(): u5
}

fn = fn.subs(u_repl)
fn
```

$$\begin{bmatrix} -u_1 \sin(q_3 + q_4) + u_2 \cos(q_3 + q_4) + u_3 \cos(q_4) \\ -u_1 \sin(q_3 + q_5) + u_2 \cos(q_3 + q_5) - u_3 \cos(q_5) \end{bmatrix} \quad (12.55)$$

These nonholonomic constraints take this form:

$$\bar{f}_n(u_1, u_2, u_3, q_3, q_4, q_5) = 0 \text{ where } \bar{f}_n \in \mathbb{R}^2 \quad (12.56)$$

We now have two equations with three unknown generalized speeds. Note that all of the generalized coordinates are not present in these constraints which is common. We can solve for two of the generalized speeds in terms of the third.

So we select two as dependent generalized speeds and one as an independent generalized speed. Because nonholonomic constraints are derived from measure numbers of velocity vectors, the nonholonomic constraints are always linear in the generalized speeds. If we introduce \bar{u}_s as a vector of independent generalized speeds and \bar{u}_r as a vector of dependent generalized speeds, the nonholonomic constraints can be written as:

$$\bar{f}_n(\bar{u}_s, \bar{u}_r, \bar{q}, t) = \mathbf{A}_r(\bar{q}, t)\bar{u}_r + \mathbf{A}_s(\bar{q}, t)\bar{u}_s + \bar{b}_{rs}(\bar{q}, t) = 0 \quad (12.57)$$

or

$$\begin{aligned} \bar{u}_r &= \mathbf{A}_r^{-1}(-\mathbf{A}_s\bar{u}_s - \bar{b}_{rs}) \\ \bar{u}_r &= \mathbf{A}_n\bar{u}_s + \bar{b}_n \end{aligned} \quad (12.58)$$

For the Snakeboard let's choose $\bar{u}_s = [u_3, u_4, u_5]^T$ as the independent generalized speeds and $\bar{u}_r = [u_1, u_2]^T$ as the dependent generalized speeds.

```
us = sm.Matrix([u3, u4, u5])
ur = sm.Matrix([u1, u2])
```

\mathbf{A}_r are the linear coefficients of \bar{u}_r so:

```
Ar = fn.jacobian(ur)
Ar
```

$$\begin{bmatrix} -\sin(q_3 + q_4) & \cos(q_3 + q_4) \\ -\sin(q_3 + q_5) & \cos(q_3 + q_5) \end{bmatrix} \quad (12.59)$$

\mathbf{A}_s are the linear coefficients of \bar{u}_s so:

```
As = fn.jacobian(us)
As
```

$$\begin{bmatrix} \cos(q_4) & 0 & 0 \\ -\cos(q_5) & 0 & 0 \end{bmatrix} \quad (12.60)$$

\bar{b}_{rs} remains when $\bar{u} = 0$:

```
brs = fn.xreplace(dict(zip([u1, u2, u3, u4, u5], [0, 0, 0, 0, 0])))
brs
```

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad (12.61)$$

\mathbf{A}_n and \bar{b}_n are formed by solving the linear system:

```
An = Ar.LUsolve(-As)
An = sm.simplify(An)
An
```

$$\begin{bmatrix} \frac{\cos(q_3 - q_4 + q_5)}{2} + \frac{\cos(q_3 + q_4 - q_5)}{2} + \cos(q_3 + q_4 + q_5) & 0 & 0 \\ \frac{\sin(q_3 - q_4 + q_5)}{2} + \frac{\sin(q_3 + q_4 - q_5)}{2} + \sin(q_3 + q_4 + q_5) & 0 & 0 \end{bmatrix} \quad (12.62)$$

```
bn = Ar.LUsolve(-brs)
bn
```

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad (12.63)$$

We now have the $m = 2$ dependent generalized speeds $\bar{u}_r = [u_1, u_2]^T$ written as functions of the $n=1$ independent generalized speeds $\bar{u}_s = [u_3]$:

```
sm.Eq(ur, An*us + bn)
```

$$\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} \left(\frac{\cos(q_3 - q_4 + q_5)}{2} + \frac{\cos(q_3 + q_4 - q_5)}{2} + \cos(q_3 + q_4 + q_5) \right) u_3 \\ \frac{\sin(q_4 - q_5)}{\sin(q_4 - q_5)} \\ \left(\frac{\sin(q_3 - q_4 + q_5)}{2} + \frac{\sin(q_3 + q_4 - q_5)}{2} + \sin(q_3 + q_4 + q_5) \right) u_3 \end{bmatrix} \quad (12.64)$$

12.8 Degrees of Freedom

For simple nonholonomic systems observed in a reference frame A , such as the Chaplygin Sleigh or the Snakeboard, the *degrees of freedom* in A are equal to the number of independent generalized speeds. The number of degrees of freedom p is defined as:

$$p := n - m \quad (12.65)$$

where n is the number of generalized coordinates and m are the number of nonholonomic constraints (and thus dependent generalized speeds). If there are no nonholonomic constraints, the system is a holonomic system in A and $p = n$ making the number of degrees of freedom equal to the number of generalized coordinates.

Exercise

What are the number of degrees of freedom for the Chaplygin Sleigh, Snakeboard, and Four-bar linkage?

Solution

The Chaplygin Sleigh has $n = 3$ generalized coordinates x, y, θ and $m = 1$ nonholonomic constraints. The degrees of freedom are then $p = 3 - 1 = 2$.

The Snakeboard has $n = 5$ generalized coordinates and $m = 2$ nonholonomic constraints. The degrees of freedom are then $p = 5 - 2 = 3$.

We described the four-bar linkage with $N = 3$ coordinates and there were $M = 2$ holonomic constraints leaving us with $n = N - M = 3 - 2 = 1$ generalized coordinates. There are no nonholonomic constraints so $m = 0$. This means that there $p = n - m = 1 - 0 = 1$ degrees of freedom.

It is not always easy to visualize the degrees of freedom of a nonholonomic system when thinking of its motion, but for holonomic systems thought experiments where you vary one or two generalized coordinates at a time can help you visualize the motion.

If you have a holonomic system (no nonholonomic constraints) the degrees of freedom are equal to the number of generalized coordinates. Nonholonomic systems (those with non-integrable motion constraints) have fewer degrees of freedom than the number of generalized coordinates.

MASS DISTRIBUTION

Note: You can download this example as a Python script: `mass.py` or Jupyter Notebook: `mass.ipynb`.

13.1 Learning Objectives

After completing this chapter readers will be able to:

- calculate the mass, mass center, and inertia of a set of particles
- use inertia vectors to find inertia scalars of a set of particles
- formulate an inertia matrix for a set of particles
- use a dyadic to manipulate 2nd order tensors in multiple reference frames
- calculate the inertia dyadic of a set of particles
- apply the parallel axis theorem
- calculate the principal moments of inertia and the principal axes
- calculate angular momentum of a rigid body

```
import sympy as sm
import sympy.physics.mechanics as me
me.init_vprinting(use_latex='mathjax')
```

```
class ReferenceFrame(me.ReferenceFrame):  
  
    def __init__(self, *args, **kwargs):  
  
        kwargs.pop('latexs', None)  
  
        lab = args[0].lower()  
        tex = r'\hat{{}}_{{}}_{{}}'  
  
        super(ReferenceFrame, self).__init__(*args,  
                                            latexs=(tex.format(lab, 'x'),  
                                                    tex.format(lab, 'y'),  
                                                    tex.format(lab, 'z')),  
                                            **kwargs)  
me.ReferenceFrame = ReferenceFrame
```

In the prior chapters, we have developed the tools to formulate the kinematics of points and reference frames. The kinematics are the first of three essential parts needed to form the equations of motion of a multibody system; the other two being the mass distribution of and the forces acting on the system.

When a point is associated with a particle of mass m or a reference frame is associated with a rigid body that has some mass distribution, [Newton's](#) and [Euler's](#) second laws of motion show that the time rate of change of the linear and angular momenta must be equal to the forces and torques acting on the particle or rigid body, respectively. The momentum of a particle is determined by its mass and velocity and the angular momentum of a rigid body is determined by the distribution of mass and its angular velocity. In this chapter, we will introduce mass and its distribution.

13.2 Particles and Rigid Bodies

We will introduce and use the concepts of particles and rigid bodies in this chapter. Both are abstractions of real translating and rotating objects. Particles are points that have a location in Euclidean space which have a volumetrically infinitesimal mass. Rigid bodies are reference frames that have orientation which have an associated continuous distribution of mass. The distribution of mass can be thought of as a infinite collection of points distributed in a finite volumetric boundary. All of the points distributed in the volume are fixed to one another and translate together.

For example, an airplane can be modeled as a rigid body when one is concerned with both its translation and orientation. This could be useful when investigating its minimum turn radii and banking angle. But it could also be modeled as a particle when one is only concerned with its translation; for example when you observing the motion of the airplane from a location outside the Earth's atmosphere.

13.3 Mass

Given a set of ν particles with masses m_1, \dots, m_ν , the total mass, or *zeroth moment of mass*, of the set is defined as:

$$m := \sum_{i=1}^{\nu} m_i \quad (13.1)$$

Exercise

What is the mass of an object made up of two particles of mass m and a rigid body with mass $m/2$?

Solution

```
m = sm.symbols('m')
m_total = m + m + m/2
m_total
```

$$\frac{5m}{2} \quad (13.2)$$

For a rigid body consisting of a solid with a density ρ defined at each point within its volumetric V boundary, the total mass becomes an integral of the general form:

$$m := \int_{\text{solid}} \rho dV \quad (13.3)$$

Exercise

What is the mass of a cone with uniform density ρ , radius r , and height h ?

Solution

Using cylindrical coordinates to write $dV = r dz d\theta dr$, `integrate()` function can solve the triple integral:

$$\int_0^h \int_0^{2\pi} \int_0^{\frac{r}{h}z} \rho r dz d\theta dr \quad (13.4)$$

```
p, r, h, z, theta = sm.symbols('rho, r, h, z, theta')

sm.integrate(p*r, (r, 0, r/h*z), (theta, 0, 2*sm.pi), (z, 0, h))
```

$$\frac{\pi h r^2 \rho}{3} \quad (13.5)$$

13.4 Mass Center

If each particle in a set of S particles is located at positions $\bar{r}^{P_1/O}, \dots, \bar{r}^{P_\nu/O}$ the *first moment of mass* is defined as:

$$\sum_{i=1}^{\nu} m_i \bar{r}^{P_i/O}. \quad (13.6)$$

There is then a point S_o in which the first mass moment is equal to zero; fulfilling the following equation:

$$\sum_{i=1}^{\nu} m_i \bar{r}^{P_i/S_o} = 0. \quad (13.7)$$

This point S_o is referred to as the *mass center* (or *center of mass*) of the set of particles. The mass center's position can be found by dividing the first moment of mass by the zeroth moment of mass:

$$\bar{r}^{S_o/O} = \frac{\sum_{i=1}^{\nu} m_i \bar{r}^{P_i/O}}{\sum_{i=1}^{\nu} m_i}. \quad (13.8)$$

which is the first moment divided by the zeroth moment. For a solid body, this takes the integral form:

$$\bar{r}^{S_o/O} = \frac{\int_{\text{solid}} \rho \bar{r} dV}{\int_{\text{solid}} \rho dV} \quad (13.9)$$

The particle form (Eq. (13.8)) can be calculated using vectors and scalars in SymPy Mechanics. Here is an example of three particles each at an arbitrary location relative to O :

```

m1, m2, m3 = sm.symbols('m1, m2, m3')
x1, x2, x3 = me.dynamicsymbols('x1, x2, x3')
y1, y2, y3 = me.dynamicsymbols('y1, y2, y3')
z1, z2, z3 = me.dynamicsymbols('z1, z2, z3')

A = me.ReferenceFrame('A')

zeroth_moment = (m1 + m2 + m3)

first_moment = (m1*(x1*A.x + y1*A.y + z1*A.z) +
                 m2*(x2*A.x + y2*A.y + z2*A.z) +
                 m3*(x3*A.x + y3*A.y + z3*A.z))
first_moment

```

$$(m_1 x_1 + m_2 x_2 + m_3 x_3) \hat{a}_x + (m_1 y_1 + m_2 y_2 + m_3 y_3) \hat{a}_y + (m_1 z_1 + m_2 z_2 + m_3 z_3) \hat{a}_z \quad (13.10)$$

```

r_O_So = first_moment/zeroth_moment
r_O_So

```

$$\frac{m_1 x_1 + m_2 x_2 + m_3 x_3}{m_1 + m_2 + m_3} \hat{a}_x + \frac{m_1 y_1 + m_2 y_2 + m_3 y_3}{m_1 + m_2 + m_3} \hat{a}_y + \frac{m_1 z_1 + m_2 z_2 + m_3 z_3}{m_1 + m_2 + m_3} \hat{a}_z \quad (13.11)$$

Exercise

If $m_2 = 2m_1$ and $m_3 = 3m_1$ in the above example, find the mass center.

Solution

```

r_O_So.xreplace({m2: 2*m1, m3: 3*m1}).simplify()

```

$$\left(\frac{x_1}{6} + \frac{x_2}{3} + \frac{x_3}{2}\right) \hat{a}_x + \left(\frac{y_1}{6} + \frac{y_2}{3} + \frac{y_3}{2}\right) \hat{a}_y + \left(\frac{z_1}{6} + \frac{z_2}{3} + \frac{z_3}{2}\right) \hat{a}_z \quad (13.12)$$

13.5 Distribution of Mass

The inertia, or second moment of mass, describes the distribution of mass relative to a point about an axis. Inertia characterizes the resistance to angular acceleration in the same way that mass characterizes the resistance to linear acceleration. For a set of particles P_1, \dots, P_ν with positions $\bar{r}^{P_1/O}, \dots, \bar{r}^{P_\nu/O}$ all relative to a point O , the *inertia vector* about the unit vector \hat{n}_a is defined as ([Kane1985], pg. 61):

$$\bar{I}_a := \sum_{i=1}^{\nu} m_i \bar{r}^{P_i/O} \times \left(\hat{n}_a \times \bar{r}^{P_i/O} \right) \quad (13.13)$$

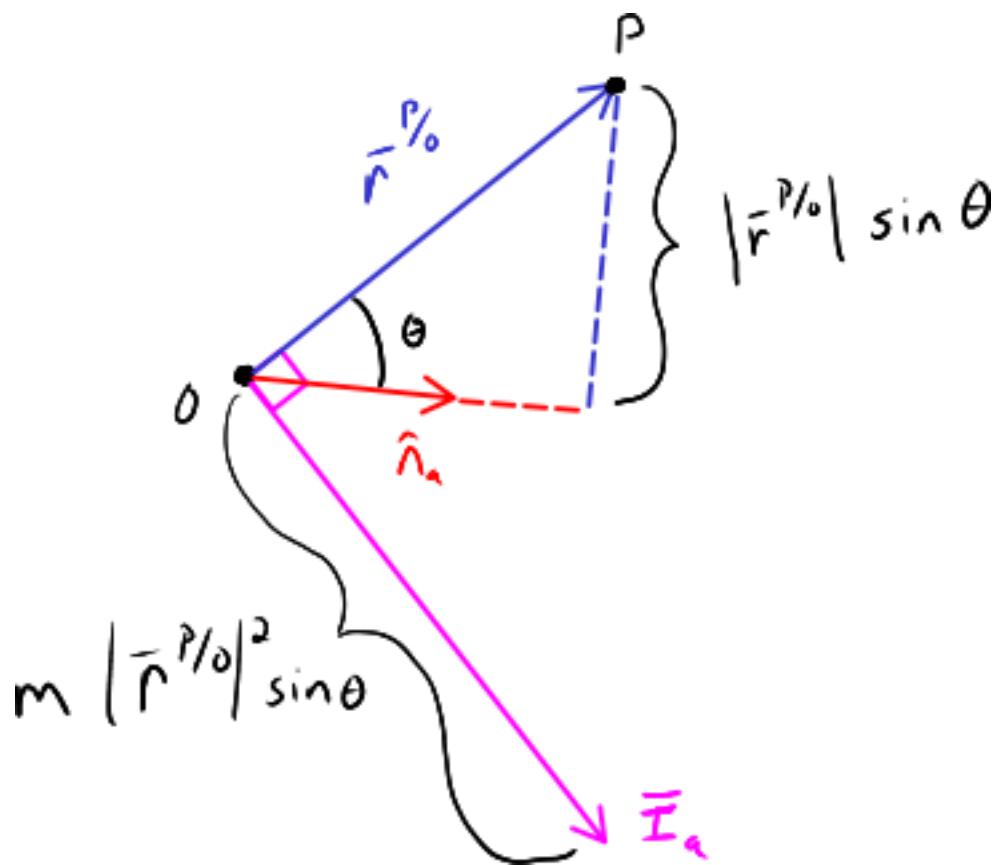


Fig. 13.1: Inertia vector for a single particle P of mass m and its relationship to \hat{n}_a .

This vector describes the sum of each mass's contribution to the mass distribution about a line that is parallel to \hat{n}_a and passes through O . Figure Fig. 13.1 shows a visual representation of this vector for a single particle P with mass m .

For this single particle, the magnitude of \bar{I}_a is:

$$|\bar{I}_a| = m |\bar{r}^{P/O}|^2 |\sin \theta| \quad (13.14)$$

where θ is angle between $\bar{r}^{P/O}$ and \hat{n}_a . We see that \bar{I}_a is always perpendicular to $\bar{r}^{P/O}$ and scales with m , $|\bar{r}^{P/O}|^2$, and $\sin \theta$. If \hat{n}_a happens to be parallel to $\bar{r}^{P/O}$ then the magnitude of \bar{I}_a is zero. If \hat{n}_a is perpendicular to $\bar{r}^{P/O}$ then the magnitude is:

$$|\bar{I}_a| = m |\bar{r}^{P/O}|^2 \quad (13.15)$$

The inertia vector fully describes the distribution of the particles with respect to O about \hat{n}_a .

A component of \bar{I}_a in the \hat{n}_b direction is called an *inertia scalar* and is defined as ([Kane1985], pg. 62):

$$I_{ab} := \bar{I}_a \cdot \hat{n}_b \quad (13.16)$$

The inertia scalar can be rewritten using Eq. (13.13) as:

$$I_{ab} = \sum_{i=1}^{\nu} m_i (\bar{r}^{P_i/O} \times \hat{n}_a) \cdot (\bar{r}^{P_i/O} \times \hat{n}_b). \quad (13.17)$$

This form implies that:

$$I_{ab} = I_{ba} \quad (13.18)$$

If $\hat{n}_a = \hat{n}_b$ then this inertia scalar is called a *moment of inertia* and if $\hat{n}_a \neq \hat{n}_b$ it is called a *product of inertia*. Moments of inertia describe the mass distribution about a single axis whereas products of inertia describe the mass distribution relative to two axes.

When $\hat{n}_a = \hat{n}_b$ Eq. (13.17) reduces to the moment of inertia:

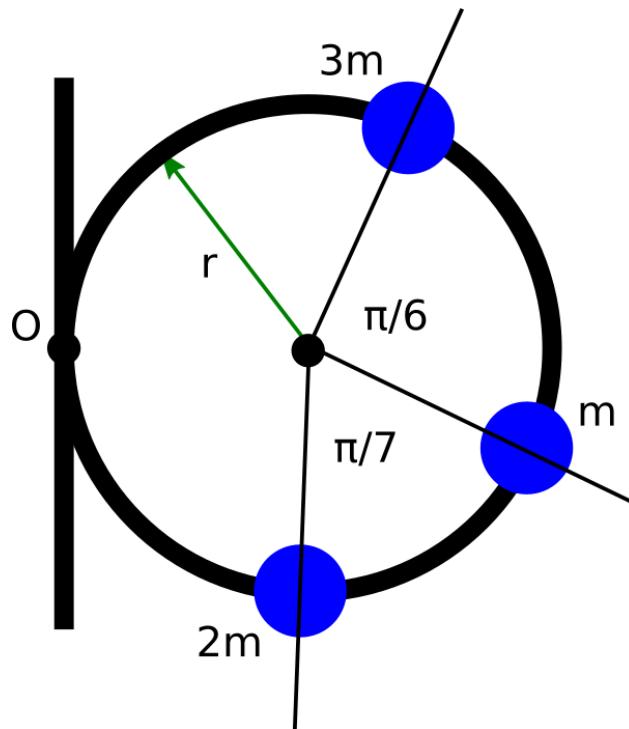
$$I_{aa} = \sum_{i=1}^{\nu} m_i (\bar{r}^{P_i/O} \times \hat{n}_a) \cdot (\bar{r}^{P_i/O} \times \hat{n}_a) \quad (13.19)$$

It is common to define the *radius of gyration* k_{aa} , which is the radius of a ring that has the same moment of inertia as the set of particles or rigid body. The radius of gyration about a line through O parallel to \hat{n}_a is defined as:

$$k_{aa} := \sqrt{\frac{I_{aa}}{m}} \quad (13.20)$$

Exercise

Three masses of m , $2m$, and $3m$ slide on a ring of radius r . Mass $3m$ always lies $\pi/6$ anitclockwise from m and mass $2m$ always lies $\pi/7$ clockwise from m . Find the acute angle from the line from the ring center to m to a line tangent to the ring at point O which minimizes the total radius of gyration of all three masses about the line tangent to the ring.



Solution

Define the necessary variables, including θ to locate mass m .

```
m, r, theta = sm.symbols('m, r, theta')
A = me.ReferenceFrame('A')
```

Create position vectors to each of the masses:

```
r_O_m = (r + r*sm.sin(theta))*A.x + r*sm.cos(theta)*A.y
r_O_2m = (r + r*sm.sin(theta + sm.pi/7))*A.x + r*sm.cos(theta + sm.pi/7)*A.y
r_O_3m = (r + r*sm.sin(theta - sm.pi/6))*A.x + r*sm.cos(theta - sm.pi/6)*A.y
```

Create the inertia scalar for a moment of inertia about the point O and \hat{a}_y .

```
Iyy = (m*me.dot(r_O_m.cross(A.y), r_O_m.cross(A.y)) +
2*m*me.dot(r_O_2m.cross(A.y), r_O_2m.cross(A.y)) +
3*m*me.dot(r_O_3m.cross(A.y), r_O_3m.cross(A.y)))
Iyy
```

$$m(r \sin(\theta) + r)^2 + 2m\left(r \sin\left(\theta + \frac{\pi}{7}\right) + r\right)^2 + 3m\left(-r \cos\left(\theta + \frac{\pi}{3}\right) + r\right)^2 \quad (13.21)$$

Recognizing that the radius of gyration is minimized when the moment of inertia is minimized, we can take the derivative of the moment of inertia with respect to θ and set that equal to zero.

```
dIyydtheta = sm.simplify(Iyy.diff(theta))
dIyydtheta
```

$$2mr^2 \left((\sin(\theta) + 1) \cos(\theta) + 2 \left(\sin\left(\theta + \frac{\pi}{7}\right) + 1 \right) \cos\left(\theta + \frac{\pi}{7}\right) - 3 \left(\cos\left(\theta + \frac{\pi}{3}\right) - 1 \right) \sin\left(\theta + \frac{\pi}{3}\right) \right) \quad (13.22)$$

We can divide through by mr^2 and solve numerically for θ since it is the only variable present in the expression.

```
theta_sol = sm.nsolve((dIyydtheta/m/r**2).evalf(), theta, 0)
theta_sol
```

$$-1.49935061382135 \quad (13.23)$$

In degrees that is:

```
import math

theta_sol*180/math.pi
```

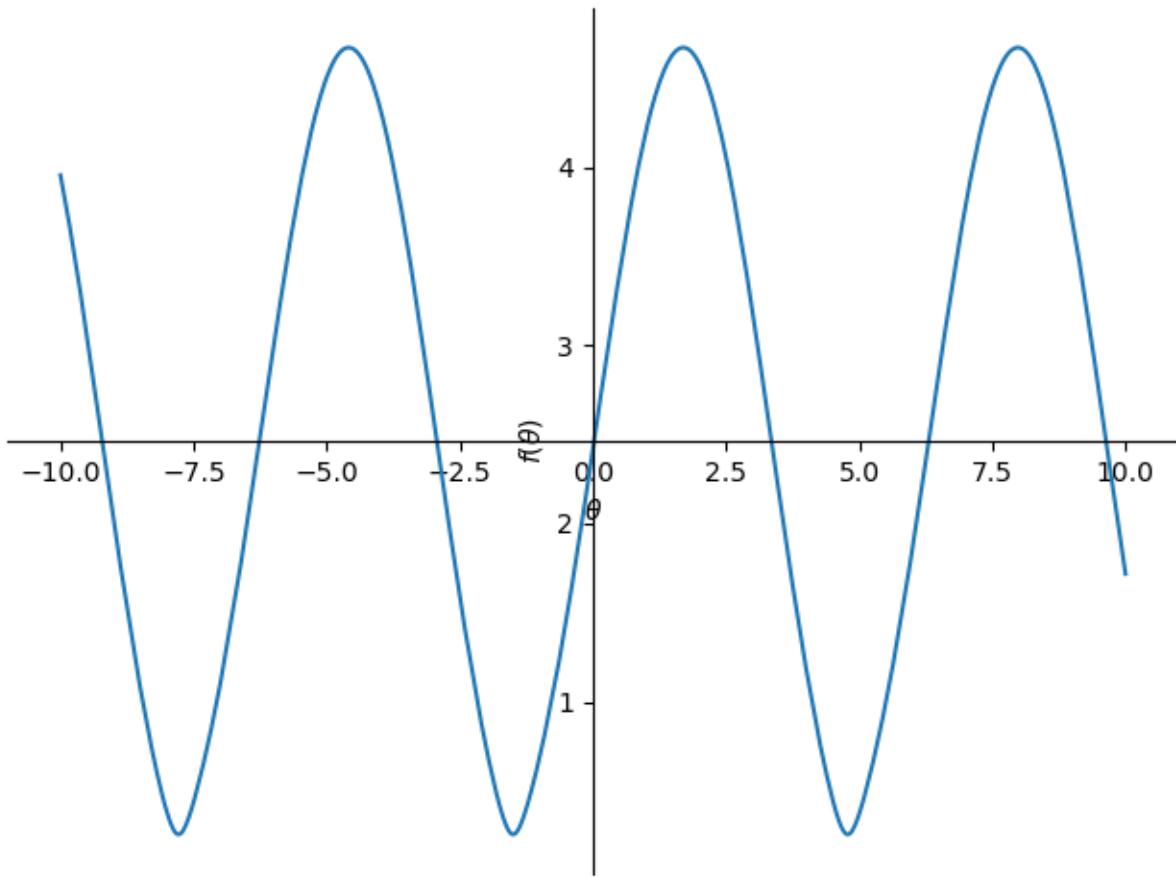
$$-85.9064621823125 \quad (13.24)$$

The `plot()` function can make quick plots of single variate functions. Here we see that rotating the set of masses around the ring will maximize and minimize the radius of gyration and that our solution is a minima. $m = r = 1$ was selected so we could plot only as a function of θ .

```
kyy = sm.sqrt(Iyy/m)
kyy
```

$$\sqrt{\frac{m(r \sin(\theta) + r)^2 + 2m(r \sin(\theta + \frac{\pi}{7}) + r)^2 + 3m(-r \cos(\theta + \frac{\pi}{3}) + r)^2}{m}} \quad (13.25)$$

```
sm.plot(kyy.xreplace({m: 1, r: 1}));
```



```
kyy.xreplace({m: 1, r: 1, theta: theta_sol}).evalf()
```

$$0.255558185585985 \quad (13.26)$$

13.6 Inertia Matrix

For mutually perpendicular unit vectors fixed in reference frame A , the moments of inertia with respect to O about each unit vector and the products of inertia among the pairs of perpendicular unit vectors can be computed using the inertia vector expressions in the prior section. This, in general, results in nine inertia scalars (6 unique scalars because of (13.18)) that describe the mass distribution of a set of particles or a rigid body in 3D space. These scalars are typically presented as a symmetric *inertia matrix* (also called an *inertia tensor*) that takes this form:

$$\begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix}_A \quad (13.27)$$

where the moments of inertia are on the diagonal and the products of inertia are the off diagonal entries. Eq. (13.18) holds for the products of inertia, i.e. $I_{xy} = I_{yx}$, $I_{xz} = I_{zx}$, and $I_{yz} = I_{zy}$, and the subscript A indicates that these scalars are relative to unit vectors $\hat{a}_x, \hat{a}_y, \hat{a}_z$.

This matrix (or second order tensor) is similar to the vectors (or first order tensors) we've already worked with:

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}_A \quad (13.28)$$

Recall that we have a notation for writing such a vector that allows us to combine components expressed in different reference frames:

$$v_1 \hat{a}_x + v_2 \hat{a}_y + v_3 \hat{a}_z \quad (13.29)$$

There also exists an analogous form for second order tensors that are associated with different reference frames called a dyadic.

13.7 Dyadics

If we introduce the `outer` product operator between two vectors we see that it generates a matrix akin to the inertia matrix above.

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}_A \otimes \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}_A = \begin{bmatrix} v_1 w_1 & v_1 w_2 & v_1 w_3 \\ v_2 w_1 & v_2 w_2 & v_2 w_3 \\ v_3 w_1 & v_3 w_2 & v_3 w_3 \end{bmatrix}_A \quad (13.30)$$

In SymPy Mechanics outer products can be taken between two vectors to create the dyadic \check{Q} using `outer()`:

```
v1, v2, v3 = sm.symbols('v1, v2, v3')
w1, w2, w3 = sm.symbols('w1, w2, w3')

A = me.ReferenceFrame('A')

v = v1*A.x + v2*A.y + v3*A.z
w = w1*A.x + w2*A.y + w3*A.z

Q = me.outer(v, w)
Q
```

$$v_1 w_1 \hat{a}_x \otimes \hat{a}_x + v_1 w_2 \hat{a}_x \otimes \hat{a}_y + v_1 w_3 \hat{a}_x \otimes \hat{a}_z + v_2 w_1 \hat{a}_y \otimes \hat{a}_x + v_2 w_2 \hat{a}_y \otimes \hat{a}_y + v_2 w_3 \hat{a}_y \otimes \hat{a}_z + v_3 w_1 \hat{a}_z \otimes \hat{a}_x + v_3 w_2 \hat{a}_z \otimes \hat{a}_y + v_3 w_3 \hat{a}_z \otimes \hat{a}_z \quad (13.31)$$

The result is not the matrix form shown in Eq. (13.30), but instead the result is a dyadic. The dyadic is the analogous form for second order tensors as what we've been using for first order tensors. If the matrix form is needed, it can be found with `to_matrix()`:

```
Q.to_matrix(A)
```

$$\begin{bmatrix} v_1 w_1 & v_1 w_2 & v_1 w_3 \\ v_2 w_1 & v_2 w_2 & v_2 w_3 \\ v_3 w_1 & v_3 w_2 & v_3 w_3 \end{bmatrix} \quad (13.32)$$

The dyadic is made up of scalars multiplied by unit dyads. Examples of unit dyads are:

```
me.outer(A.x, A.x)
```

$$\hat{a}_x \otimes \hat{a}_x \quad (13.33)$$

Unit dyads correspond to unit entries in the 3x3 matrix:

```
me.outer(A.x, A.x).to_matrix(A)
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (13.34)$$

Unit dyads are analogous to unit vectors. There are nine unit dyads in total associated with the three orthogonal unit vectors. Here is another example:

```
me.outer(A.y, A.z)
```

$$\hat{a}_y \otimes \hat{a}_z \quad (13.35)$$

```
me.outer(A.y, A.z).to_matrix(A)
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad (13.36)$$

These unit dyads can be formed from any unit vectors. This is convenient because we can create dyadics, just like vectors, which are made up of components in different reference frames. For example:

```
theta = sm.symbols("theta")
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
B.orient_axis(A, theta, A.x)
P = 2*me.outer(B.x, B.x) + 3*me.outer(A.x, B.y) + 4*me.outer(B.z, A.z)
P
```

$$2\hat{b}_x \otimes \hat{b}_x + 3\hat{a}_x \otimes \hat{b}_y + 4\hat{b}_z \otimes \hat{a}_z \quad (13.37)$$

The dyadic \check{P} can be expressed in unit dyads of A

```
P.express(A)
```

$$2\hat{a}_x \otimes \hat{a}_x + 3\cos(\theta)\hat{a}_x \otimes \hat{a}_y + 3\sin(\theta)\hat{a}_x \otimes \hat{a}_z - 4\sin(\theta)\hat{a}_y \otimes \hat{a}_z + 4\cos(\theta)\hat{a}_z \otimes \hat{a}_z \quad (13.38)$$

P.to_matrix(A)

$$\begin{bmatrix} 2 & 3\cos(\theta) & 3\sin(\theta) \\ 0 & 0 & -4\sin(\theta) \\ 0 & 0 & 4\cos(\theta) \end{bmatrix} \quad (13.39)$$

or B: :

P.express(B)

$$2\hat{b}_x \otimes \hat{b}_x + 3\hat{b}_x \otimes \hat{b}_y + 4\sin(\theta)\hat{b}_z \otimes \hat{b}_y + 4\cos(\theta)\hat{b}_z \otimes \hat{b}_z \quad (13.40)$$

P.to_matrix(B)

$$\begin{bmatrix} 2 & 3 & 0 \\ 0 & 0 & 0 \\ 0 & 4\sin(\theta) & 4\cos(\theta) \end{bmatrix} \quad (13.41)$$

The *unit dyadic* is defined as:

$$\check{U} := \hat{a}_x \otimes \hat{a}_x + \hat{a}_y \otimes \hat{a}_y + \hat{a}_z \otimes \hat{a}_z \quad (13.42)$$

The unit dyadic can be created with SymPy:

```
U = me.outer(A.x, A.x) + me.outer(A.y, A.y) + me.outer(A.z, A.z)
```

$$\hat{a}_x \otimes \hat{a}_x + \hat{a}_y \otimes \hat{a}_y + \hat{a}_z \otimes \hat{a}_z \quad (13.43)$$

and it represents the identity matrix in A:

U.to_matrix(A)

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (13.44)$$

Note that the unit dyadic is the same when expressed in any reference frame:

```
U.express(B).simplify()
```

$$\hat{b}_x \otimes \hat{b}_x + \hat{b}_y \otimes \hat{b}_y + \hat{b}_z \otimes \hat{b}_z \quad (13.45)$$

13.8 Properties of Dyadics

Dyadics have similar properties as vectors but are not necessarily commutative.

- Scalar multiplication: $\alpha(\bar{u} \otimes \bar{v}) = \alpha\bar{u} \otimes \bar{v} = \bar{u} \otimes \alpha\bar{v}$
- Distributive: $\bar{u} \otimes (\bar{v} + \bar{w}) = \bar{u} \otimes \bar{v} + \bar{u} \otimes \bar{w}$
- Left and right dot product with a vector (results in a vector):
 - $\bar{u} \cdot (\bar{v} \otimes \bar{w}) = (\bar{u} \cdot \bar{v})\bar{w}$
 - $(\bar{u} \otimes \bar{v}) \cdot \bar{w} = \bar{u}(\bar{v} \cdot \bar{w})$
- Left and right cross product with a vector (results in a dyadic):
 - $\bar{u} \times (\bar{v} \otimes \bar{w}) = (\bar{u} \times \bar{v}) \otimes \bar{w}$
 - $(\bar{u} \otimes \bar{v}) \times \bar{w} = \bar{u} \otimes (\bar{v} \times \bar{w})$
- Dot products between arbitrary vectors \bar{u} and arbitrary dyadics \check{V} are not commutative: $\check{V} \cdot \bar{u} \neq \bar{u} \cdot \check{V}$
- Dot products between arbitrary vectors and the unit dyadic are commutative and result in the vector itself: $\check{U} \cdot \bar{v} = \bar{v} \cdot \check{U} = \bar{v}$

13.9 Inertia Dyadic

Previously we defined the inertia vector as:

$$\bar{I}_a = \sum_{i=1}^{\nu} m_i \bar{r}^{P_i/O} \times \left(\hat{n}_a \times \bar{r}^{P_i/O} \right) \quad (13.46)$$

Using the [vector triple product](#) identity: $\bar{a} \times (\bar{b} \times \bar{c}) = \bar{b}(\bar{a} \cdot \bar{c}) - \bar{c}(\bar{a} \cdot \bar{b})$, the inertia vector can be written as ([Kane1985], pg. 68):

$$\bar{I}_a = \sum_{i=1}^{\nu} m_i \left[\hat{n}_a \left(\bar{r}^{P_i/O} \cdot \bar{r}^{P_i/O} \right) - \bar{r}^{P_i/O} \left(\bar{r}^{P_i/O} \cdot \hat{n}_a \right) \right] \quad (13.47)$$

Now by introducing a unit dyadic, we can write:

$$\bar{I}_a = \sum_{i=1}^{\nu} m_i \left[\left| \bar{r}^{P_i/O} \right|^2 \hat{n}_a \cdot \check{U} - \hat{n}_a \cdot \left(\bar{r}^{P_i/O} \otimes \bar{r}^{P_i/O} \right) \right] \quad (13.48)$$

\hat{n}_a can be pulled out of the summation:

$$\bar{I}_a = \hat{n}_a \cdot \sum_{i=1}^{\nu} m_i \left(\left| \bar{r}^{P_i/O} \right|^2 \check{U} - \bar{r}^{P_i/O} \otimes \bar{r}^{P_i/O} \right) \quad (13.49)$$

The *inertia dyadic* \check{I} of a set of S particles relative to O is now defined as:

$$\check{I}^{S/O} := \sum_{i=1}^{\nu} m_i \left(\left| \bar{r}^{P_i/O} \right|^2 \check{U} - \bar{r}^{P_i/O} \otimes \bar{r}^{P_i/O} \right) \quad (13.50)$$

where:

$$\bar{I}_a = \hat{n}_a \cdot \check{I}^{S/O} \quad (13.51)$$

Note that we have now described the inertia of the set of particles without needing to specify a vector \hat{n}_a . This inertia dyadic contains the complete description of inertia with respect to point O about any axis. The vectors and dyadics in Eq. (13.50) can be written in terms of any reference frame unit vectors or unit dyads, respectively.

If you have a solid body, an infinite set of points with a volumetric boundary, then you must solve the integral version of (13.50) where the position of any location in the particle is parameterize by τ which can represent a volume, line, or surface parameterization.

$$\check{I}^{S/O} := \int_{\text{solid}} \rho \left(\left| \bar{r}^{P(\tau)/O} \right|^2 \check{U} - \bar{r}^{P(\tau)/O} \otimes \bar{r}^{P(\tau)/O} \right) d\tau \quad (13.52)$$

In SymPy Mechanics, simple inertia dyadics in terms of the unit vectors of a single reference frame can quickly be generated with `inertia()`. For example:

```
Ixx, Iyy, Izz = sm.symbols('I_{xx}', 'I_{yy}', 'I_{zz}')
Ixy, Iyz, Ixz = sm.symbols('I_{xy}', 'I_{yz}', 'I_{xz}')
I = me.inertia(A, Ixx, Iyy, Izz, ixy=Ixy, iyz=Iyz, izx=Ixz)
```

$$I_{xx}\hat{a}_x \otimes \hat{a}_x + I_{xy}\hat{a}_x \otimes \hat{a}_y + I_{xz}\hat{a}_x \otimes \hat{a}_z + I_{xy}\hat{a}_y \otimes \hat{a}_x + I_{yy}\hat{a}_y \otimes \hat{a}_y + I_{yz}\hat{a}_y \otimes \hat{a}_z + I_{xz}\hat{a}_z \otimes \hat{a}_x + I_{yz}\hat{a}_z \otimes \hat{a}_y + I_{zz}\hat{a}_z \otimes \hat{a}_z \quad (13.53)$$

```
I.to_matrix(A)
```

$$\begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{bmatrix} \quad (13.54)$$

This inertia dyadic can easily be expressed relative to another reference frame if the orientation is defined (demonstrated above in [Dyadics](#)):

```
sm.trigsimp(I.to_matrix(B))
```

$$\begin{bmatrix} I_{xx} & I_{xy} \cos(\theta) + I_{xz} \sin(\theta) & -I_{xy} \sin(\theta) + I_{xz} \cos(\theta) \\ I_{xy} \cos(\theta) + I_{xz} \sin(\theta) & \frac{I_{yy} \cos(2\theta)}{2} + \frac{I_{yy}}{2} + I_{yz} \sin(2\theta) - \frac{I_{zz} \cos(2\theta)}{2} + \frac{I_{zz}}{2} & -\frac{I_{yy} \sin(2\theta)}{2} + I_{yz} \cos(2\theta) + \frac{I_{zz} \sin(2\theta)}{2} \\ -I_{xy} \sin(\theta) + I_{xz} \cos(\theta) & -\frac{I_{yy} \sin(2\theta)}{2} + I_{yz} \cos(2\theta) + \frac{I_{zz} \sin(2\theta)}{2} & -\frac{I_{yy} \cos(2\theta)}{2} + \frac{I_{yy}}{2} - I_{yz} \sin(2\theta) + \frac{I_{zz} \cos(2\theta)}{2} + \frac{I_{zz}}{2} \end{bmatrix} \quad (13.55)$$

This is equivalent to the matrix transform to express an inertia matrix in other reference frame (see some explanations on [stackexchange](#) about this transform):

$${}^B \mathbf{C}^A \mathbf{I} {}^A \mathbf{C}^B \quad (13.56)$$

```
sm.trigsimp(B.dcm(A)*I.to_matrix(A)*A.dcm(B))
```

$$\begin{bmatrix} I_{xx} & I_{xy} \cos(\theta) + I_{xz} \sin(\theta) & -I_{xy} \sin(\theta) + I_{xz} \cos(\theta) \\ I_{xy} \cos(\theta) + I_{xz} \sin(\theta) & \frac{I_{yy} \cos(2\theta)}{2} + \frac{I_{yy}}{2} + I_{yz} \sin(2\theta) - \frac{I_{zz} \cos(2\theta)}{2} + \frac{I_{zz}}{2} & -\frac{I_{yy} \sin(2\theta)}{2} + I_{yz} \cos(2\theta) + \frac{I_{zz} \sin(2\theta)}{2} \\ -I_{xy} \sin(\theta) + I_{xz} \cos(\theta) & -\frac{I_{yy} \sin(2\theta)}{2} + I_{yz} \cos(2\theta) + \frac{I_{zz} \sin(2\theta)}{2} & -\frac{I_{yy} \cos(2\theta)}{2} + \frac{I_{yy}}{2} - I_{yz} \sin(2\theta) + \frac{I_{zz} \cos(2\theta)}{2} + \frac{I_{zz}}{2} \end{bmatrix} \quad (13.57)$$

Exercise

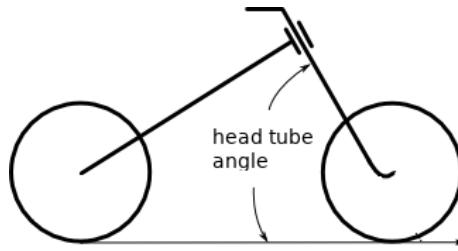


Fig. 13.2: Head tube angle of a bicycle.

Given the inertia dyadic of a bicycle's handlebar and fork assembly about its mass center where \hat{n}_x points from the center of the rear wheel to the center of the front wheel and \hat{n}_z points downward, normal to the ground, and the head tube angle is 68 degrees from the ground plane, find the moment of inertia about the tilted steer axis given the inertia dyadic:

```
N = me.ReferenceFrame('N')
I = (0.25*me.outer(N.x, N.x) +
     0.25*me.outer(N.y, N.y) +
     0.10*me.outer(N.z, N.z) -
     0.07*me.outer(N.x, N.z) -
     0.07*me.outer(N.z, N.x))
```

$$0.25\hat{n}_x \otimes \hat{n}_x + 0.25\hat{n}_y \otimes \hat{n}_y + 0.1\hat{n}_z \otimes \hat{n}_z - 0.07\hat{n}_x \otimes \hat{n}_z - 0.07\hat{n}_z \otimes \hat{n}_x \quad (13.58)$$

Solution

Create a new reference frame that is aligned with the steer axis.

```
H = me.ReferenceFrame('H')
H.orient_axis(N, 68.0*sm.pi/180, N.y)
```

Dot the inertia dyadic twice with \hat{n}_z (which is aligned with the steer axis) to get the moment of inertia about the steer axis:

```
I .dot (H .z) .dot (H .z) .evalf()
```

$$0.180324399093269 \quad (13.59)$$

Alternatively, you can use the matrix transformation.

```
I .to_matrix (N)
```

$$\begin{bmatrix} 0.25 & 0 & -0.07 \\ 0 & 0.25 & 0 \\ -0.07 & 0 & 0.1 \end{bmatrix} \quad (13.60)$$

```
I_H = (H .dcm (N) @ I .to_matrix (N) @ N .dcm (H)) .evalf()
I_H
```

$$\begin{bmatrix} 0.169675600906731 & 0 & 0.10245316380813 \\ 0 & 0.25 & 0 \\ 0.10245316380813 & 0 & 0.180324399093269 \end{bmatrix} \quad (13.61)$$

```
I_H [2, 2]
```

$$0.180324399093269 \quad (13.62)$$

13.10 Parallel Axis Theorem

If you know the central inertia dyadic of a rigid body B (or equivalently a set of particles) about its mass center B_o then it is possible to calculate the inertia dyadic about any other point O . To do so, you must account for the inertial contribution due to the distance between the points O and B_o . This is done with the [parallel axis theorem](#) ([Kane1985], pg. 70):

$$\check{I}^{B/O} = \check{I}^{B/B_o} + \check{I}^{B_o/O} \quad (13.63)$$

The last term is the inertia of a particle with mass m (total mass of the body or set of particles) located at the mass center about point O .

$$\check{I}^{B_o/O} = m \left(\left| \bar{r}^{B_o/O} \right|^2 \check{U} - \bar{r}^{B_o/O} \otimes \bar{r}^{B_o/O} \right) \quad (13.64)$$

When B_o is displaced from point O by three Cartesian distances d_x, d_y, d_z the general form of the last term in Eq. (13.63) can be found:

```

dx, dy, dz, m = sm.symbols('d_x, d_y, d_z, m')
N = me.ReferenceFrame('N')
r_O_Bo = dx*N.x + dy*N.y + dz*N.z
U = me.outer(N.x, N.x) + me.outer(N.y, N.y) + me.outer(N.z, N.z)
I_Bo_O = m*(me.dot(r_O_Bo, r_O_Bo)*U - me.outer(r_O_Bo, r_O_Bo))
I_Bo_O

```

$$m(d_y^2 + d_z^2) \hat{n}_x \otimes \hat{n}_x + m(d_x^2 + d_z^2) \hat{n}_y \otimes \hat{n}_y + m(d_x^2 + d_y^2) \hat{n}_z \otimes \hat{n}_z - d_x d_y m \hat{n}_x \otimes \hat{n}_y - d_x d_z m \hat{n}_x \otimes \hat{n}_z - d_x d_y m \hat{n}_y \otimes \hat{n}_x - d_y d_z m \hat{n}_y \otimes \hat{n}_z - d_x d_z m \hat{n}_z \otimes \hat{n}_x - d_y d_z m \hat{n}_z \otimes \hat{n}_y \quad (13.65)$$

The matrix form of this dyadic shows the typical presentation of the parallel axis addition term:

```
I_Bo_O.to_matrix(N)
```

$$\begin{bmatrix} m(d_y^2 + d_z^2) & -d_x d_y m & -d_x d_z m \\ -d_x d_y m & m(d_x^2 + d_z^2) & -d_y d_z m \\ -d_x d_z m & -d_y d_z m & m(d_x^2 + d_y^2) \end{bmatrix} \quad (13.66)$$

13.11 Principal Axes and Moments of Inertia

If the inertia vector \bar{I}_a with respect to point O is parallel to its unit vector \hat{n}_a then the line through O and parallel to \hat{n}_a is called a *principal axis* of the set of particles or rigid body. The plane that is normal to \hat{n}_a is called a *principal plane*. The moment of inertia about this principal axis is called a *principal moment of inertia*. The consequence of \bar{I}_a being parallel to \hat{n}_a is that the products of inertia are all zero. The *principal inertia dyadic* can then be written as so:

$$\check{I}^{B/O} = I_{11} \hat{b}_1 \otimes \hat{b}_1 + I_{22} \hat{b}_2 \otimes \hat{b}_2 + I_{33} \hat{b}_3 \otimes \hat{b}_3 \quad (13.67)$$

where $\hat{b}_1, \hat{b}_2, \hat{b}_3$ are mutually perpendicular unit vectors in B that are each parallel to a principal axis and I_{11}, I_{22}, I_{33} are all principal moments of inertia.

Geometrically symmetric objects with uniform mass density have principal planes that are perpendicular with the planes of symmetry of the geometry. But there also exist unique principal axes for non-symmetric and non-uniform density objects.

The principal axes and their associated principal moments of inertia can be found by solving the eigenvalue problem. The eigenvalues of an arbitrary inertia matrix are the principal moments of inertia and the eigenvectors are the unit vectors parallel to the mutually perpendicular principal axes. Recalling that the inertia matrix is a symmetric matrix of real numbers, we know then that it is Hermitian and therefore all its eigenvalues are real. Symmetric matrices are also diagonalizable and the eigenvectors will then be orthonormal.

Warning: Finding the eigenvalues of a 3x3 matrix require finding the roots of the [cubic equation](#). It is possible to find the symbolic solution, but it is not a simple result. Unless you really need the symbolic result, it is best to solve for principal axes and moments of inertia numerically.

Here is an example of finding the principal axes and associated moments of inertia with SymPy:

```
I = sm.Matrix([[1.0451, 0.0, -0.1123],
              [0.0, 2.403, 0.0],
              [-0.1123, 0.0, 1.8501]])
I
```

$$\begin{bmatrix} 1.0451 & 0 & -0.1123 \\ 0 & 2.403 & 0 \\ -0.1123 & 0 & 1.8501 \end{bmatrix} \quad (13.68)$$

The `eigenvals()` method on a SymPy matrix returns a list of tuples that each contain (eigenvalue, multiplicity, eigenvector):

```
ev1, ev2, ev3 = I.eigenvals()
```

The results are a bit confusing to parse but you can extract the relevant information as follows.

The first and largest eigenvalue (principal moment of inertia) and its associated eigenvector (principal axis direction) is:

```
ev1[0]
```

$$2.403 \quad (13.69)$$

```
ev1[2][0]
```

$$\begin{bmatrix} 0 \\ 1.0 \\ 0 \end{bmatrix} \quad (13.70)$$

This shows that the y axes was already the major principal axis. The second eigenvalue and its associated eigenvector is:

```
ev2[0]
```

$$1.02972736390139 \quad (13.71)$$

```
ev2[2][0]
```

$$\begin{bmatrix} -0.990760351805416 \\ 0 \\ -0.135624206137434 \end{bmatrix} \quad (13.72)$$

This is the smallest eigenvalue and thus the minor moment of inertia about the minor principal axis. The third eigenvalue and its associated eigenvector give the intermediate principal axis and the intermediate moment of inertia:

ev3 [0]

$$1.86547263609861 \quad (13.73)$$

ev3 [2] [0]

$$\begin{bmatrix} 0.135624206137434 \\ 0 \\ -0.990760351805416 \end{bmatrix} \quad (13.74)$$

13.12 Angular Momentum

The angular momentum vector of a rigid body B in reference frame A about point O is defined as:

$${}^A \bar{H}^{B/O} := \check{I}^{B/O} \cdot {}^A \bar{\omega}^B \quad (13.75)$$

The dyadic-vector dot product notation makes this definition succinct. If the point is instead the mass center of B , point B_o , then the inertia dyadic is the *central inertia dyadic* and the result is the *central angular momentum* in A is:

$${}^A \bar{H}^{B/B_o} = \check{I}^{B/B_o} \cdot {}^A \bar{\omega}^B \quad (13.76)$$

Here is an example of calculating the angular momentum expressed in the body fixed reference frame in SymPy Mechanics:

```
Ixx, Iyy, Izz = sm.symbols('I_{xx}', 'I_{yy}', 'I_{zz}')
Ix, Iy, Iz = sm.symbols('I_{xy}', 'I_{yz}', 'I_{xz}')
w1, w2, w3 = me.dynamicsymbols('omega1', 'omega2', 'omega3')

B = me.ReferenceFrame('B')
I = me.inertia(B, Ixx, Iyy, Izz, Ix, Iy, Iz)
A_w_B = w1*B.x + w2*B.y + w3*B.z
I .dot (A_w_B)
```

$$(I_{xx}\omega_1 + I_{xy}\omega_2 + I_{xz}\omega_3)\hat{b}_x + (I_{xy}\omega_1 + I_{yy}\omega_2 + I_{yz}\omega_3)\hat{b}_y + (I_{xz}\omega_1 + I_{yz}\omega_2 + I_{zz}\omega_3)\hat{b}_z \quad (13.77)$$

If the body fixed unit vectors happen to be aligned with the principal axes of the rigid body, then the central angular momentum simplifies:

```
I1, I2, I3 = sm.symbols('I_1', 'I_2', 'I_3')
w1, w2, w3 = me.dynamicsymbols('omega1', 'omega2', 'omega3')

B = me.ReferenceFrame('B')
```

(continues on next page)

(continued from previous page)

```
I = me.inertia(B, I1, I2, I3)  
A_w_B = w1*B.x + w2*B.y + w3*B.z  
I.dot(A_w_B)
```

$$I_1\omega_1\hat{b}_x + I_2\omega_2\hat{b}_y + I_3\omega_3\hat{b}_z \quad (13.78)$$

FORCE, MOMENT, AND TORQUE

Note: You can download this example as a Python script: `loads.py` or Jupyter Notebook: `loads.ipynb`.

```
import sympy as sm
import sympy.physics.mechanics as me
me.init_vprinting(use_latex='mathjax')

class ReferenceFrame(me.ReferenceFrame):
    def __init__(self, *args, **kwargs):
        kwargs.pop('latexs', None)
        lab = args[0].lower()
        tex = r'\hat{{}}_{{}}_{{}}'
        super(ReferenceFrame, self).__init__(*args,
                                             latexs=(tex.format(lab, 'x'),
                                                      tex.format(lab, 'y'),
                                                      tex.format(lab, 'z')),
                                             **kwargs)
me.ReferenceFrame = ReferenceFrame
```

14.1 Learning Objectives

After completing this chapter readers will be able to:

- Calculate the resultant force acting on a set of particles or bodies.
- Calculate the moment of a resultant about a point.
- Find the equivalent couple of a torque and resultant to simplify forces applied to bodies.
- Determine if a force is noncontributing or not.
- Define sign conventions for forces.
- Define forces for gravity, springs, friction, air drag, and collisions.

14.2 Force

A *force* is an abstraction we use to describe something that causes mass to move (e.g. accelerate from a stationary state). There are four **fundamental forces of nature** of which all other forces can be derived from. Moments and torques arise from forces and are useful in describing what causes distributed mass rotation. Forces, moments, and torques have magnitude and direction and thus we use vectors to describe them mathematically.

14.3 Bound and Free Vectors

Vectors may have a *line of action*. A line of action is parallel to the vector and passes through a particular point. If a vector has a line of action, it is said to be *bound* to its line of action. If a vector is not bound to a line of action it is said to be *free*.

Angular velocity is an example of a free vector. It has a direction and magnitude, but is not associated with any line of action. For example, if a disc can rotate around a fixed point, you can place a body anywhere on this disc and the body will always have the same angular velocity. A force vector, on the other hand, is bound. If a force is applied to a rigid body, we must know where on the body it is applied to resolve the force's effect. A force vector acting on rigid body B at point P has a line of action through P and parallel to the force vector.

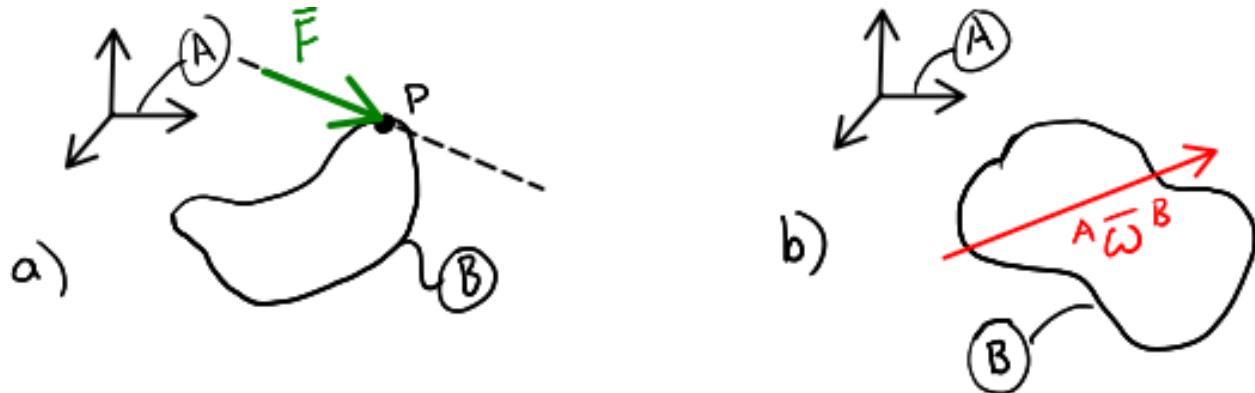


Fig. 14.1: Depiction of bound a) and free b) vectors.

14.4 Moment

If a vector is a bound vector, then we can define its *moment* about a point. The moment \bar{M} of bound vector \bar{v} about point P is a vector itself and is defined as ([Kane1985], pg. 90):

$$\bar{M} := \bar{r}^{L/P} \times \bar{v} \quad (14.1)$$

$\bar{r}^{L/P}$ is a position vector from P to any point L_i on the line of action L of \bar{v} . The cross product definition ensures that the the moment does not depend on the choice of the point on the line.

A moment can be the result of a set of vectors. The *resultant* of a set S of vectors $\bar{v}_1, \dots, \bar{v}_\nu$ is defined as:

$$\bar{R}^S := \sum_{i=1}^{\nu} \bar{v}_i \quad (14.2)$$

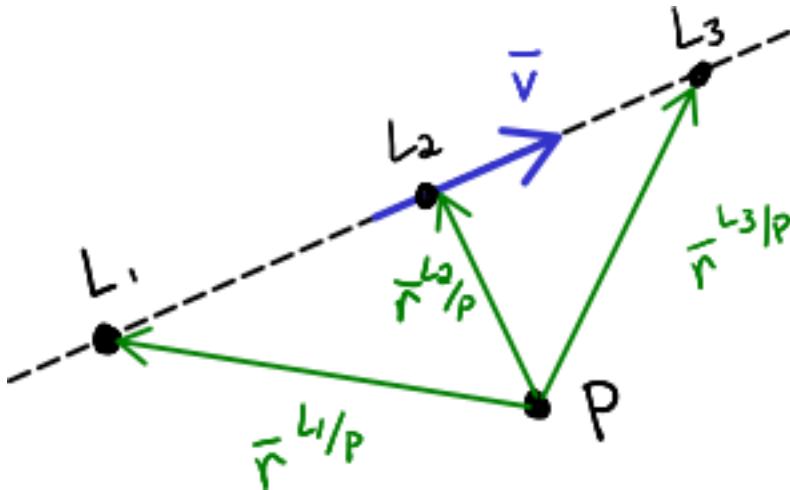


Fig. 14.2: \bar{v} is bound to a line L . The moment can be calculated based on a position vector from P to any point on the line, for example L_1, L_2 or L_3 as shown.

If each vector in the resultant is bound, the sum of the moments due to each vector about P is called the moment of \bar{R}^S about P . This summation can be written as:

$$\bar{M}^{S/P} = \sum_{i=1}^{\nu} \bar{r}^{L_i/P} \times \bar{v}_i \quad (14.3)$$

where L_i is a point on the line of action of the associated \bar{v}_i .

The moment of the set of bound vectors S about one point P is related to the moment of the set about another point Q by ([Kane1985], pg. 91):

$$\bar{M}^{S/P} = \bar{M}^{S/Q} + \bar{r}^{P/Q} \times \bar{R}^{S/Q} \quad (14.4)$$

where $\bar{R}^{S/Q}$ is the resultant of the set S bound to a line of action through point Q .

For example, take the set S of two bound vectors \bar{F}_1 and \bar{F}_2 bound to lines of action through points P_1 and P_2 , respectively. Below I've given the vectors some arbitrary direction and magnitude.

```

N = me.ReferenceFrame('N')
F1 = 2*N.x + 3*N.y
F2 = -4*N.x + 5*N.y
r_O_P1 = 2*N.x
r_O_P2 = 3*N.x

```

$\bar{M}^{S/P}$ can be calculated directly using Eq. (14.3):

```

r_O_P = -5*N.x
M_S_P = me.cross(r_O_P1 - r_O_P, F1) + me.cross(r_O_P2 - r_O_P, F2)
M_S_P

```

$$61\hat{n}_z \quad (14.5)$$

Or if $\bar{M}^{S/Q}$ is known, as well as $\bar{r}^{P/Q}$, then the Eq. (14.4) could be used:

```

r_O_Q = 5*N.y
M_S_Q = me.cross(r_O_P1 - r_O_Q, F1) + me.cross(r_O_P2 - r_O_Q, F2)
M_S_P = M_S_Q + me.cross(r_O_Q - r_O_P, F1 + F2)
M_S_P
    
```

$$61\hat{n}_z \quad (14.6)$$

14.5 Couple

A set S of bound vectors with a resultant equal to zero is called a *couple*. A couple can have as many vectors as desired or needed with a minimum number being two, such that $\bar{R}^S = 0$. A couple composed of two vectors is called a *simple couple*. Fig. 14.3 shows a few examples of couples.

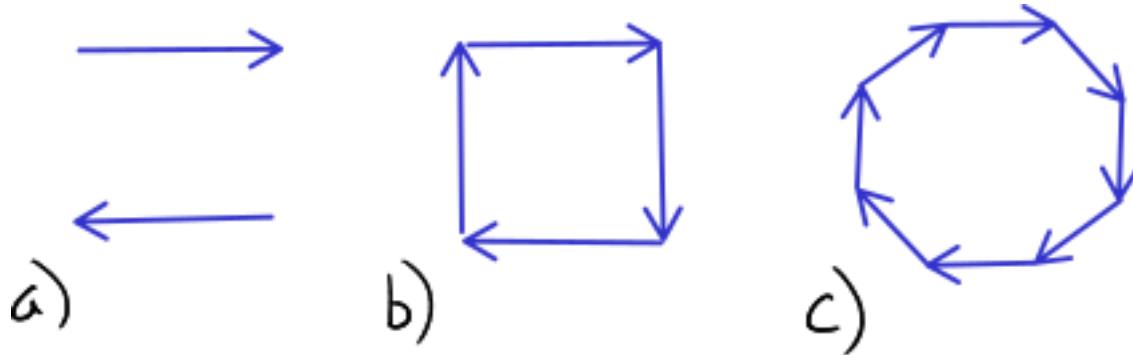


Fig. 14.3: Three couples: a) simple couple, b) & c) couples made up of multiple forces

The *torque* of a couple, \bar{T} , is the moment of the couple about a point. Because the resultant of a couple is zero it follows from (14.4), the torque of a couple is the same about all points. The torque, being a moment, is also a vector.

14.6 Equivalence & Replacement

Two sets of bound vectors are *equivalent* when they have these two properties:

1. equal resultants
2. equal moments about *any* point

If 1. and 2. are true, the sets are said to be *replacements* of each other. Couples that have equal torques are equivalent, because the resultants are zero and moments about any point are equal to the torque.

Given a set of bound vectors S and a set of bound vectors that consist of a torque of a couple \bar{T} and vector \bar{v} bound to an arbitrary point P it is a necessary and sufficient condition that the second set is a replacement of the first if ([Kane1985], pg. 95):

$$\begin{aligned} \bar{T} &= \bar{M}^{S/P} \\ \bar{v} &= \bar{R}^{S/P} \end{aligned} \quad (14.7)$$

This means that every set of bound vectors can be replaced by an equivalent torque of a couple and a single bound vector that is the resultant of the replaced set. This replacement simplifies the description of forces acting on bodies.

Take for example the birds eye view of a four wheeled car which has front steering and motors at each wheel allowing for precise control of the propulsion forces at each wheel. A diagram of the forces acting at each wheel is shown in Fig. 14.4.

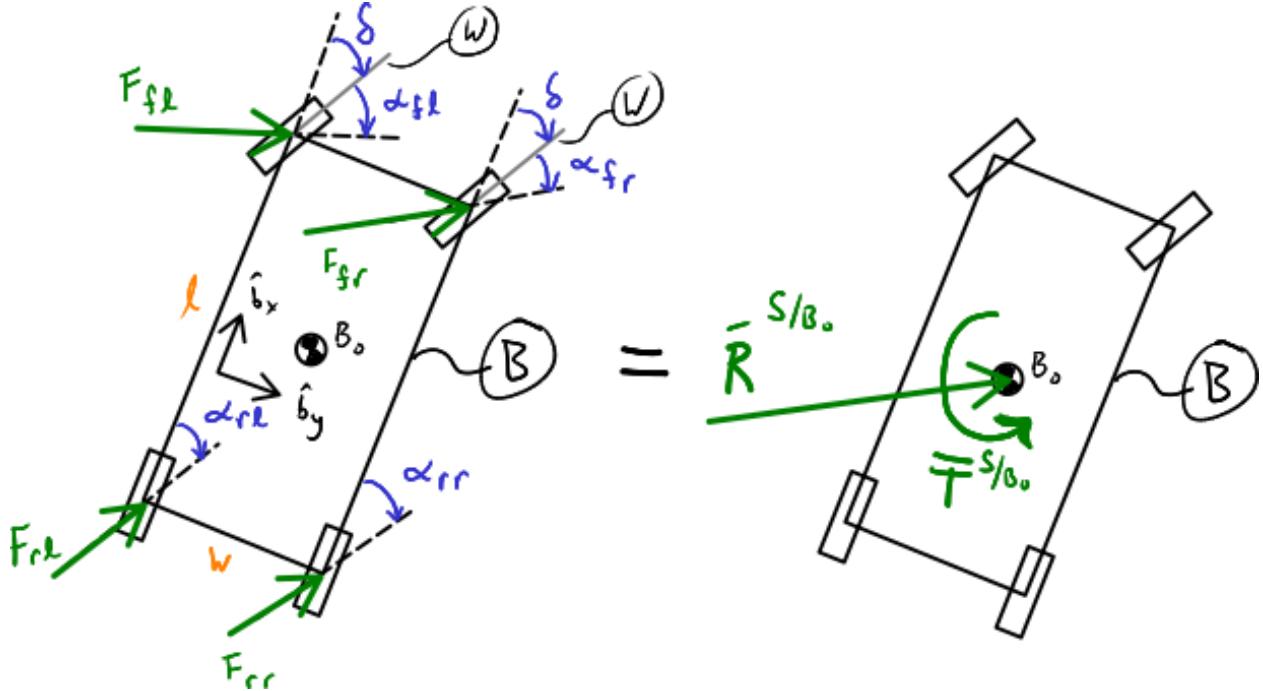


Fig. 14.4: Set S of forces acting at each tire can be replaced with a resultant and a torque at a specified point, in this case B_o .

In SymPy Mechanics, first define the symbols:

```
l, w = sm.symbols('l, w')
Ffl, Ffr, Frl, Frr = me.dynamicsymbols('F_{fl}, F_{fr}, F_{rl}, F_{rr}')
alphafl, alphafr = me.dynamicsymbols(r'\alpha_{fl}, \alpha_{fr}')
alpharl, alpharr = me.dynamicsymbols(r'\alpha_{rl}, \alpha_{rr}')
delta = me.dynamicsymbols('delta')
```

With the symbols defined, I use some auxiliary reference frames to establish the orientations with B behind the car body, W being the steered front wheels, and the others to establish the direction of the force at each wheel.

```
B = me.ReferenceFrame('B')
W = me.ReferenceFrame('W')
FR = me.ReferenceFrame('F_R')
FL = me.ReferenceFrame('F_L')
RR = me.ReferenceFrame('R_R')
RL = me.ReferenceFrame('R_L')

W.orient_axis(B, delta, B.z)
FR.orient_axis(W, alphafr, W.z)
FL.orient_axis(W, alphafl, W.z)
RR.orient_axis(B, alpharr, B.z)
RL.orient_axis(B, alpharl, B.z)
```

The resultant of the forces expressed in the B frame is then:

```
R = Ffl*FL.x + Ffr*FR.x + Frl*RL.x + Frr*RR.x
R.express(B).simplify()
```

$$(F_{fl} \cos(\alpha_{fl} + \delta) + F_{fr} \cos(\alpha_{fr} + \delta) + F_{rl} \cos(\alpha_{rl}) + F_{rr} \cos(\alpha_{rr}))\hat{b}_x + (F_{fl} \sin(\alpha_{fl} + \delta) + F_{fr} \sin(\alpha_{fr} + \delta) + F_{rl} \sin(\alpha_{rl}) + F_{rr} \sin(\alpha_{rr}))\hat{b}_y \quad (14.8)$$

This resultant is bound to a line of action through B_o . The associated couple is then calculated as the total moment about B_o :

```
T = (me.cross(l/2*B.x - w/2*B.y, Ffl*FL.x) +
      me.cross(l/2*B.x + w/2*B.y, Ffr*FR.x) +
      me.cross(-l/2*B.x - w/2*B.y, Frl*RL.x) +
      me.cross(-l/2*B.x + w/2*B.y, Frr*RR.x))
T = T.express(B).simplify()
T
```

$$\frac{(l \sin(\alpha_{fl} + \delta) + w \cos(\alpha_{fl} + \delta)) F_{fl}}{2} + \frac{(l \sin(\alpha_{fr} + \delta) - w \cos(\alpha_{fr} + \delta)) F_{fr}}{2} - \frac{(l \sin(\alpha_{rl}) - w \cos(\alpha_{rl})) F_{rl}}{2} - \frac{(l \sin(\alpha_{rr}) + w \cos(\alpha_{rr})) F_{rr}}{2} \quad (14.9)$$

Since we can always describe the forces acting on a rigid body as a resultant force and an associate torque of a couple, we will often take advantage of this simpler form for constructing models.

14.7 Specifying Forces and Torques

Forces are bound vectors, so we have to define their lines of action. This is best done by specifying the points on which each force acts, thus we will always use a vector and a point to fully describe the force. Methods and functions in SymPy Mechanics that make use of forces will typically require a tuple containing a point and a vector, for example the resultant force R^{S/B_o} acting on the mass center of the car would be specified like so:

```
Bo = me.Point('Bo')
force = (Bo, R)
force
```

```
(Bo, F_{fr}(t)*F_R.x + F_{fl}(t)*F_L.x + F_{rr}(t)*R_R.x + F_{rl}(t)*R_L.x)
```

Torques of a couple are free vectors (not bound to a line of action) but represent a couple acting on a rigid body, thus a reference frame associated with a rigid body and the vector representing the torque will be used to describe the torque in SymPy Mechanics. For example:

```
torque = (B, T)
torque
```

```
(B,
 ((l*sin(alpha_{fl}(t) + delta(t)) + w*cos(alpha_{fl}(t) + delta(t)))*F_{fl}(t)/2 +
  ↪ (l*sin(alpha_{fr}(t) + delta(t)) - w*cos(alpha_{fr}(t) + delta(t)))*F_{fr}(t)/2 -
  ↪ (l*sin(alpha_{rl}(t)) - w*cos(alpha_{rl}(t)))*F_{rl}(t)/2 - (l*sin(alpha_{rr}(t)) +
  ↪ (t)) + w*cos(alpha_{rr}(t)))*F_{rr}(t)/2)*B.z)
```

We will often refer to forces and torques collectively as *loads*.

Note: The two cells above do not render the math nicely due to this SymPy bug: <https://github.com/sympy/sympy/issues/24967>.

14.8 Equal & Opposite

Both forces and torques applied to a multibody system must obey [Newton's Third Law](#), i.e. that forces and torques act equal and opposite. Take for example a torque from a motor that causes a pinned lever B to rotate relative to the ground N shown in [Fig. 14.5](#). The motor torque can be modeled to occur between the stator and the rotor. We've arbitrarily selected the sign convention shown, i.e. a positive value of torque applies a positive torque to B and a negative torque to N if the torque is parallel to $\hat{n}_z = \hat{b}_z$.

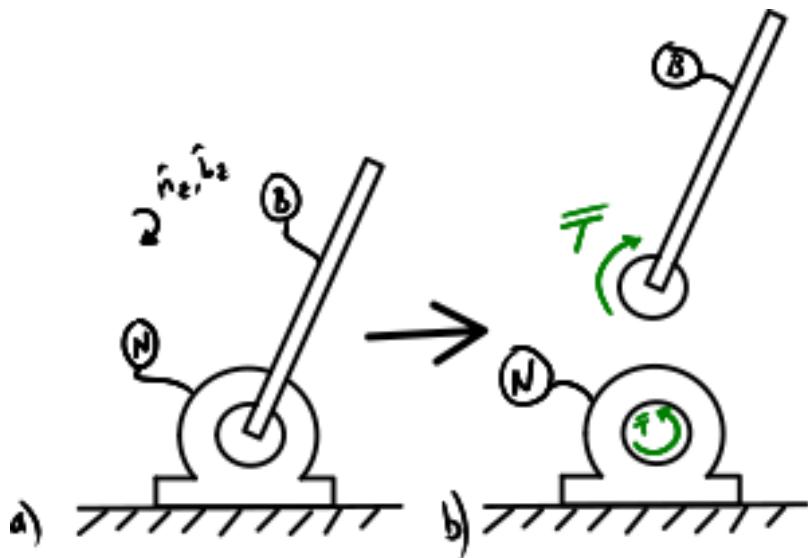


Fig. 14.5: A motor stator N fixed to ground with an arm fixed to the motor rotor B shown as one unit in a) and as separate bodies in b) with equal and opposite torque vectors applied to the pair of bodies representing the torque of a couple generated by the motor.

The motor torque can be specified as a time varying vector:

```
T, q = me.dynamicsymbols('T, q')
N = me.ReferenceFrame('N')
B = me.ReferenceFrame('B')
Tm = T*N.z
```

Then the equal and opposite torques are captured by these two tuples:

```
(B, Tm), (N, -Tm)
```

```
((B, T(t)*N.z), (N, -T(t)*N.z))
```

with equal and opposite torques applied to each body.

Warning: The sign conventions are really just a convention. It is also valid to choose $(B, -Tm)$, (N, Tm) or even (B, Tm) , (N, Tm) and $(B, -Tm)$, $(B, -Tm)$. But it is useful to choose a sign convention such that when the signs of angular velocity and torque are the same it corresponds to power into the system. So, for example, `B.orient_axis(N, q, N.z)` corresponds to $(T^*N.z, B)$ and power in. The key thing is that you know what your convention is so that you can interpret numerical results and signs correctly.

14.9 Contributing and Noncontributing Forces

Contributing forces are those that can do work on the multibody system. *Work* of a force \bar{F} acting over path S is defined as the following [line integral](#):

$$W = \int_S \bar{F} \cdot d\bar{s} \quad (14.10)$$

where $d\bar{s}$ is the differential vector tangent to the path at the point the force is applied.

For example, the gravitational force acting on a particle moving through a unidirectional constant gravitational field (i.e. where the gravitational force is equal in magnitude, doesn't change, and always the same direction) does work on the system.

Noncontributing forces never do work on the system. For example, when a force acts between two points that have no relative motion, no work is done. Examples of noncontributing forces:

1. contact forces on particles across smooth (frictionless) surfaces of rigid bodies
2. any internal contact and body (distance) forces between any two points in a rigid body
3. contact forces between bodies rolling without slipping on each other

In the next chapter, we will see how the use of generalized coordinates relieve us from having to specify any noncontributing forces.

14.10 Gravity

We will often be interested in a multibody system's motion when it is subject to gravitational forces. The simplest case is a constant unidirectional gravitational field, which is an appropriate model for objects moving on and near the Earth's surface. The gravitational forces can be applied solely to the mass centers of each rigid body as a resultant force. The gravitational torque on the bodies is zero because the force is equal in magnitude for each particle in the body. See [Kane1985] pg. 110 for the more general model of [Newton's Law of Universal Gravitation](#) where this is not the case. Studies of spacecraft dynamics often require considering both gravitational forces and moments.

In SymPy Mechanics, a gravitational force acting on a particle of mass m with acceleration due to gravity being g in the $-\hat{n}_y$ direction would take this form:

```
m, g = sm.symbols('m, g')
Fg = -m*g*N.y
```

$$-gm\hat{n}_y \quad (14.11)$$

14.11 Springs & Dampers

Idealized springs and dampers are useful models of elements that have distance and velocity dependent forces and torques. A spring with free length q_0 and where q_1, q_2 locate the ends of the spring along a line parallel to \hat{n}_x is shown in Fig. 14.6.

If we displace P in the positive \hat{n}_x direction the spring will apply a force in the negative \hat{n}_x direction on point P . So we chose a sign convention that the force on P from the spring is opposite the direction of the displacement.

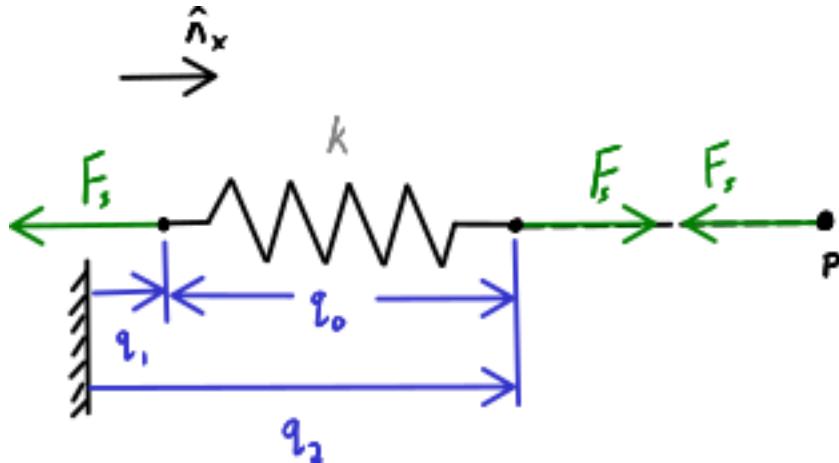


Fig. 14.6: Diagram of a spring with a sign convention that tension is positive. P is shown separated from the end of the spring to show the equal and opposite forces.

If the spring is linear with stiffness k the spring force vector is then:

```
q0, k = sm.symbols('q0, k')
q1, q2 = me.dynamicsymbols('q1, q2')

displacement = q2 - q1 - q0
displacement
```

$$-q_0 - q_1 + q_2 \quad (14.12)$$

Here a positive displacement represents the spring in tension and a negative displacement is compression.

```
Fs = -k*displacement*N.x
Fs
```

$$-k(-q_0 - q_1 + q_2) \hat{n}_x \quad (14.13)$$

Dampers are often used in parallel or series with springs to provide an energy dissipation via viscous-like friction. Springs combined with dampers allow for classical second order under-, over-, and critically-damped motion. A linear viscous damper with damping coefficient c that resists motion can be defined like so:

```
c = sm.symbols('c')
t = me.dynamicsymbols._t

Fc = -c*displacement.diff(t)*N.x
Fc
```

$$-c(-\dot{q}_1 + \dot{q}_2)\hat{n}_x \quad (14.14)$$

14.12 Friction

Coulomb's Law of Friction provides simple model of dry friction between two objects. When the two objects are in motion with respect to each other, there is a constant magnitude force that resists the motion. The force is independent of contact area and is proportional to the normal force between the objects. Coulomb's kinetic friction model takes the scalar form:

$$F_f = \begin{cases} \mu_k F_n & v < 0 \\ 0 & v = 0 \\ -\mu_k F_n & v > 0 \end{cases} \quad (14.15)$$

where F_N is the normal force between the two objects, v is the relative speed between the two objects, and μ_k is the coefficient of kinetic friction. At $v = 0$ kinetic friction is zero, but two objects in contact with a normal force can resist moving through static friction. When $v = 0$ any force perpendicular to the normal force can be generated up to a magnitude of $F_f = \mu_s F_n$ where μ_s is the coefficient of static friction and $\mu_s > \mu_k$. Eq. (14.15) leaves this static case ambiguous, but it can be extended to:

$$F_f = \begin{cases} \mu_k F_n & v < 0 \\ [-\mu_s F_n, \mu_s F_n] & v = 0 \\ -\mu_k F_n & v > 0 \end{cases} \quad (14.16)$$

SymPy's `Piecewise` is one way to create a symbolic representation of kinetic friction:

```
mu, m, g = sm.symbols('mu, m, g')
Fn = m*g
displacement = q2 - q1
Ff = sm.Piecewise((mu*Fn, displacement.diff(t) < 0),
                   (-mu*Fn, displacement.diff(t) > 0),
                   (0, True)) * N.x
Ff
```

$$\begin{cases} gm\mu & \text{for } \dot{q}_1 - \dot{q}_2 > 0 \\ -gm\mu & \text{for } \dot{q}_1 - \dot{q}_2 < 0 \hat{n}_x \\ 0 & \text{otherwise} \end{cases} \quad (14.17)$$

The `signum` function (`sign`) can also be used in a similar and simpler form:

```
Ff = -mu*Fn*sm.sign(displacement.diff(t)) * N.x
Ff
```

$$-gm\mu \text{sign}(-\dot{q}_1 + \dot{q}_2)\hat{n}_x \quad (14.18)$$

Eq. (14.16) is a sufficient model for many use cases, but it does not necessarily capture all observed effects. Fig. 14.7 shows a modification of Coulomb model that includes the `Stribeck effect` and viscous friction. Flores et. al have a nice summary of several other friction models that could be used [Flores2023].

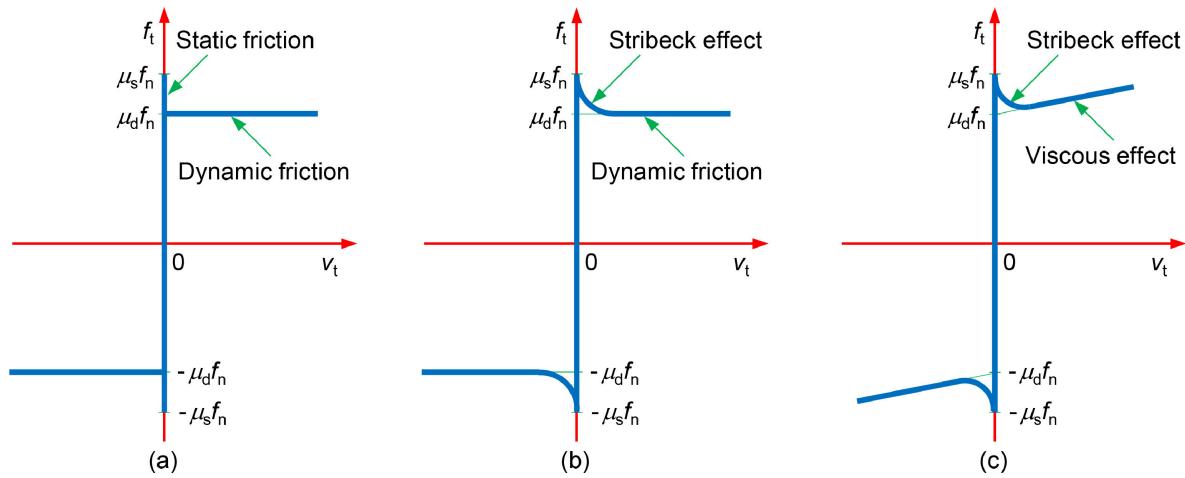


Fig. 14.7: Extensions to the (a) Coulomb Dry Friction model: (b) Stribeck effect and (c) Stribeck and viscous effects. Taken from [Flores2023] (Creative Commons BY-NC-ND 4.0).

14.13 Aerodynamic Drag

Aerodynamic [drag](#) of a blunt body at low Reynolds numbers is dominated by the frontal area drag and the magnitude of this drag force can be modeled with the following equation:

$$F_d = \frac{1}{2} \rho C_d A v^2 \quad (14.19)$$

where ρ is the density of the air, C_d is the drag coefficient, A is the frontal area, and v is the air speed relative to the body. If a body is moving in still air at an arbitrary velocity and point P is the aerodynamic center of the body then the aerodynamic drag force vector that opposes the motion can be found with such an equation:

```

A, Cd, rho = sm.symbols('A, C_d, rho')
ux, uy, uz = me.dynamicsymbols('u_x, u_y, u_z', real=True)

N_v_P = ux*N.x + uy*N.y + uz*N.z

Fd = -N_v_P.normalize() * Cd * A * rho / 2 * N_v_P.dot(N_v_P)
Fd

```

$$-\frac{AC_d \rho \sqrt{u_x^2 + u_y^2 + u_z^2} u_x}{2} \hat{n}_x - \frac{AC_d \rho \sqrt{u_x^2 + u_y^2 + u_z^2} u_y}{2} \hat{n}_y - \frac{AC_d \rho \sqrt{u_x^2 + u_y^2 + u_z^2} u_z}{2} \hat{n}_z \quad (14.20)$$

If the motion is only along the \hat{n}_x direction, for example, the equation for the drag force vector reduces to:

```
Fd.xreplace({uy: 0, uz: 0})
```

$$-\frac{AC_d \rho u_x |u_x|}{2} \hat{n}_x \quad (14.21)$$

Managing the correct direction of the force, so that it opposes motion and is applied at the aerodynamic center, is important. The drag coefficient and frontal area can also change dynamically depending on the shape of the object and the direction the air is flowing over it.

14.14 Collision

If two points, a point and a surface, or two surfaces collide the impact behavior depends on the material properties, mass, and kinematics of the colliding bodies. There are two general approaches to modeling collision. The first is the Newtonian method in which you consider the momentum change, impulse, before and after collision. For a particle impacting a surface, this takes the basic form:

$$mv^+ = -emv^- \quad (14.22)$$

where m is the particle's mass, v^- is the speed before impact, v^+ is the speed after impact, and e is the [coefficient of restitution](#). The momentum after impact will be opposite and equal to the momentum before impact for a purely elastic collision $e = 1$ and the magnitude of the momentum will be less if the collision is inelastic $0 < e < 1$. This approach can be extended to a multibody system; see [Flores2023] for an introduction to this approach.

The Newtonian model does not consider the explicit behavior of the force that generates the impulse at collision. Here we will take an alternative approach by modeling the force explicitly. Such contact force models can provide more accurate results, at the cost of longer computation times. Most impact force models build upon Hunt and Crossley's seminal model [Hunt1975] which is based on [Hertzian contact theory](#). Hunt and Crossley model the impact as a nonlinear function of penetration depth and its rate. The force is made up of a nonlinear stiffness and a damping term that take this form:

$$f_c = kz^n + cz^n\dot{z} \quad (14.23)$$

k is the nonlinear contact stiffness, n is the stiffness exponent, z the contact penetration, \dot{z} is the penetration velocity, and c is the hysteresis damping factor. The damping scales with the penetration depth. k and c can be determined from the material properties and the shape of the colliding objects and can be related to the coefficient of restitution. n is 3/2 based on the Hertzian contact theory.

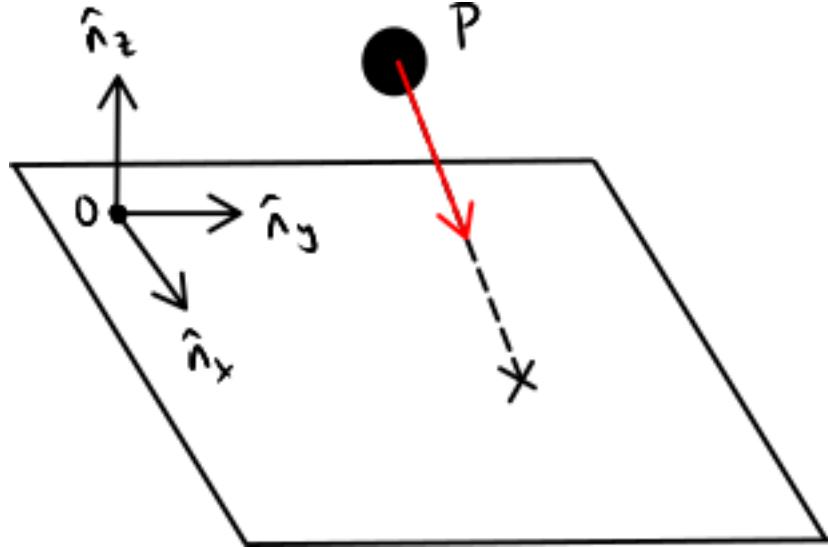


Fig. 14.8: Particle P colliding with a surface.

For example, if modeling a particle P that impacts a surface normal to \hat{n}_z that contains point O the penetration z_p of the particle into the surface (if positive z is out and negative z is inside the surface) can be described with:

$$z_p = \frac{|\bar{r}^{P/O} \cdot \hat{n}_z| - \bar{r}^{P/O} \cdot \hat{n}_z}{2} \quad (14.24)$$

This difference between the absolute value and the value itself is equivalent to this piecewise function:

$$z_p = \begin{cases} 0 & \bar{r}^{P/O} \cdot \hat{n}_z > 0 \\ \bar{r}^{P/O} \cdot \hat{n}_z & \bar{r}^{P/O} \cdot \hat{n}_z \leq 0 \end{cases} \quad (14.25)$$

In SymPy, this can be defined like so:

```
x, y, z, zd = me.dynamicsymbols('x, y, z, \dot{z}', real=True)

r_O_P = x*N.x + y*N.y + z*N.z

zh = r_O_P.dot(N.z)

zp = (sm.Abs(zh) - zh)/2
zp
```

$$-\frac{z}{2} + \frac{|z|}{2} \quad (14.26)$$

The force can now be formulated according to (14.23):

```
k, c = sm.symbols('k, c')

Fz = (k*zp**2 * (sm.S(3)/2) + c*zp**2 * (sm.S(3)/2) * zd) * N.z
Fz
```

$$(c \left(-\frac{z}{2} + \frac{|z|}{2} \right)^{\frac{3}{2}} \dot{z} + k \left(-\frac{z}{2} + \frac{|z|}{2} \right)^{\frac{3}{2}}) \hat{n}_z \quad (14.27)$$

We can check whether the force is correct for positive and negative z :

```
Fz.xreplace({z: sm.Symbol('z', positive=True)})
```

$$0 \quad (14.28)$$

```
Fz.xreplace({z: sm.Symbol('z', negative=True)})
```

$$(c(-z)^{\frac{3}{2}} \dot{z} + k(-z)^{\frac{3}{2}}) \hat{n}_z \quad (14.29)$$

More on the Hunt-Crossley model and alterations on the model are summarized in [Flores2023].

The impact force model is often combined with a friction model to generate a friction force for impacts that are not perfectly normal to the contacting surfaces. For example, Coulomb friction force can slow the particle's sliding on the surface if we know the tangential velocity components v_x and v_y at the contact location. This lets us write to tangential friction force components:

```
mu = sm.symbols('mu')

vx = r_O_P.dot(N.x).diff(t)
vy = r_O_P.dot(N.y).diff(t)

Fx = -sm.Abs(vx) / vx*mu*Fz.dot(N.z)*N.x
Fx
```

$$-\frac{\mu \left(c \left(-\frac{\dot{z}}{2} + \frac{|z|}{2} \right)^{\frac{3}{2}} \dot{z} + k \left(-\frac{\dot{z}}{2} + \frac{|z|}{2} \right)^{\frac{3}{2}} \right) |\dot{x}|}{\dot{x}} \hat{n}_x \quad (14.30)$$

```
Fy = -sm.Abs(vy)/vy*mu*Fz.dot(N.z)*N.y
Fy
```

$$-\frac{\mu \left(c \left(-\frac{\dot{z}}{2} + \frac{|z|}{2} \right)^{\frac{3}{2}} \dot{z} + k \left(-\frac{\dot{z}}{2} + \frac{|z|}{2} \right)^{\frac{3}{2}} \right) |\dot{y}|}{\dot{y}} \hat{n}_y \quad (14.31)$$

These measure numbers for the force vector then evaluate to zero when there is no penetration z_p and evaluates to a spring and damper and Coulomb friction when there is. For example, using so numerical values to set the penetration:

```
vz = me.dynamicsymbols('v_z', negative=True)
repl = {zd: vz, z: sm.Symbol('z', positive=True)}
Fx.xreplace(repl), Fy.xreplace(repl), Fz.xreplace(repl)
```

$$(0, 0, 0) \quad (14.32)$$

```
vz = me.dynamicsymbols('v_z', negative=True)
repl = {zd: vz, z: sm.Symbol('z', negative=True)}
Fx.xreplace(repl), Fy.xreplace(repl), Fz.xreplace(repl)
```

$$\left(-\frac{\mu \left(c(-z)^{\frac{3}{2}} v_z + k(-z)^{\frac{3}{2}} \right) |\dot{x}|}{\dot{x}} \hat{n}_x, -\frac{\mu \left(c(-z)^{\frac{3}{2}} v_z + k(-z)^{\frac{3}{2}} \right) |\dot{y}|}{\dot{y}} \hat{n}_y, (c(-z)^{\frac{3}{2}} v_z + k(-z)^{\frac{3}{2}}) \hat{n}_z \right) \quad (14.33)$$

Finally, the total force on the particle contacting the surface can be fully described:

```
Fx + Fy + Fz
```

$$-\frac{\mu \left(c \left(-\frac{\dot{z}}{2} + \frac{|z|}{2} \right)^{\frac{3}{2}} \dot{z} + k \left(-\frac{\dot{z}}{2} + \frac{|z|}{2} \right)^{\frac{3}{2}} \right) |\dot{x}|}{\dot{x}} \hat{n}_x - \frac{\mu \left(c \left(-\frac{\dot{z}}{2} + \frac{|z|}{2} \right)^{\frac{3}{2}} \dot{z} + k \left(-\frac{\dot{z}}{2} + \frac{|z|}{2} \right)^{\frac{3}{2}} \right) |\dot{y}|}{\dot{y}} \hat{n}_y + (c \left(-\frac{z}{2} + \frac{|z|}{2} \right)^{\frac{3}{2}} \dot{z} + k \left(-\frac{z}{2} + \frac{|z|}{2} \right)^{\frac{3}{2}}) \hat{n}_z \quad (14.34)$$

GENERALIZED FORCES

Note: You can download this example as a Python script: `generalized-forces.py` or Jupyter Notebook: `generalized-forces.ipynb`.

```
import sympy as sm
import sympy.physics.mechanics as me
me.init_vprinting(use_latex='mathjax')

class ReferenceFrame(me.ReferenceFrame):
    def __init__(self, *args, **kwargs):
        kwargs.pop('latexs', None)
        lab = args[0].lower()
        tex = r'\hat{{}}_{{}}'
        super(ReferenceFrame, self).__init__(*args,
                                             latexs=(tex.format(lab, 'x'),
                                                      tex.format(lab, 'y'),
                                                      tex.format(lab, 'z')),
                                             **kwargs)
me.ReferenceFrame = ReferenceFrame
```

15.1 Learning Objectives

After completing this chapter readers will be able to:

- Calculate partial velocities given generalized speeds
- Calculate generalized active forces for a system of particles and rigid bodies
- Calculate generalized inertia forces for a system of particles and rigid bodies

15.2 Introduction

At this point we have developed the three primary ingredients to formulate the equations of motion of a multibody system:

1. Angular and Translational Kinematics
2. Mass and Mass Distribution
3. Forces, Moments, and Torques

For a single rigid body B with mass m_B , mass center B_o , and central inertia dyadic \check{I}^{B/B_o} having a resultant force \bar{F} at B_o and moment \bar{M} about B_o the [Newton-Euler Equations of Motion](#) in the inertial reference frame N can be written as follows:

$$\begin{aligned}\bar{F} &= \frac{^N d \bar{p}}{dt} \quad \text{where } \bar{p} = m_B {}^N \bar{v}^{B_o} \\ \bar{M} &= \frac{^N d \bar{H}}{dt} \quad \text{where } \bar{H} = \check{I}^{B/B_o} \cdot {}^N \bar{\omega}^B\end{aligned}\tag{15.1}$$

The left hand side of the above equations describes the forces, moments, and torques (3.) acting on the rigid body and the right hand side describes the kinematics (1.) and the mass distribution (2.).

For a set of particles and rigid bodies that make up a multibody system defined with generalized coordinates, generalized speeds, and constraints, the generalized speeds characterize completely the motion of the system. The velocities and angular velocities of every particle and rigid body in the system are a function of these generalized speeds. The time rate of change of the generalized speeds $\frac{du}{dt}$ will then play a critical role in the formulation of the right hand side of the multibody system equations of motion.

Take for example the multibody system shown in [Fig. 15.1](#). A force \bar{F} applied at point Q may cause all three of the lower particles to move. The motion of the particles are described by the velocities, which are functions of the generalized speeds. Thus \bar{F} will, in general, cause all of the generalized speeds to change. But how much does each generalized speed change? The so called *partial velocities* of Q in N will provide the answer to this question.

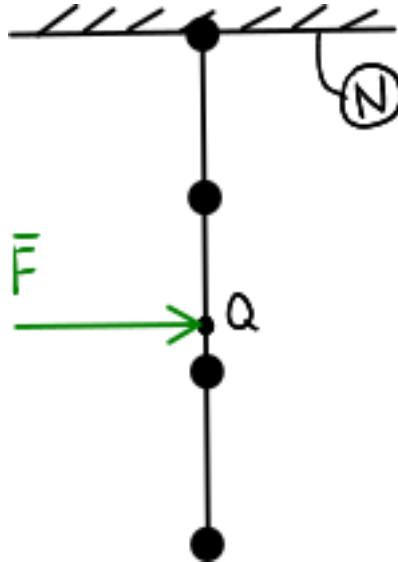


Fig. 15.1: Four particles attached by massless links making up a 3 link planar simple pendulum. The top particle is fixed in N . If the generalized coordinates q_1, q_2, q_3 represent the angles of the three pendulums then three generalized speeds could be defined as $u_i = \dot{q}_i$ for $i = 1, \dots, 3$.

15.3 Partial Velocities

Recall that all translational and angular velocities of a multibody system can be written in terms of the generalized speeds. By definition (Eq. (12.21)), these velocities can be expressed uniquely as linear functions of the generalized speeds. For a holonomic system with n degrees of freedom any translational velocity or angular velocity observed from a single reference frame can be written as ([Kane1985], pg. 45):

$$\bar{v} = \sum_{r=1}^n \bar{v}_r u_r + \bar{v}_t$$

$$\bar{\omega} = \sum_{r=1}^n \bar{\omega}_r u_r + \bar{\omega}_t$$
(15.2)

We call \bar{v}_r and $\bar{\omega}_r$ the r^{th} holonomic partial velocity and angular velocity in the single reference frame, respectively. \bar{v}_t and $\bar{\omega}_t$ are the remainder terms that are not linear in a generalized speed. Since the velocities are linear in the generalized speeds, the partial velocities are equal to the partial derivatives with respect to the generalized speeds:

$$\bar{v}_r = \frac{\partial \bar{v}}{\partial u_r}$$

$$\bar{\omega}_r = \frac{\partial \bar{\omega}}{\partial u_r}$$
(15.3)

Note: The reference frame these partials are taken with respect to must match that which the velocities are with respect to.

Given that the partial velocities are partial derivatives, means that we may interpret the partial velocities as the sensitivities of translational and angular velocities to changes in u_r . The partial velocities give an idea of how any given velocity or angular velocity will change if one of the generalized speeds changes. Figure Fig. 15.2 gives a graphical interpretation of how a velocity of P in N is made up of partial velocities and a remainder.

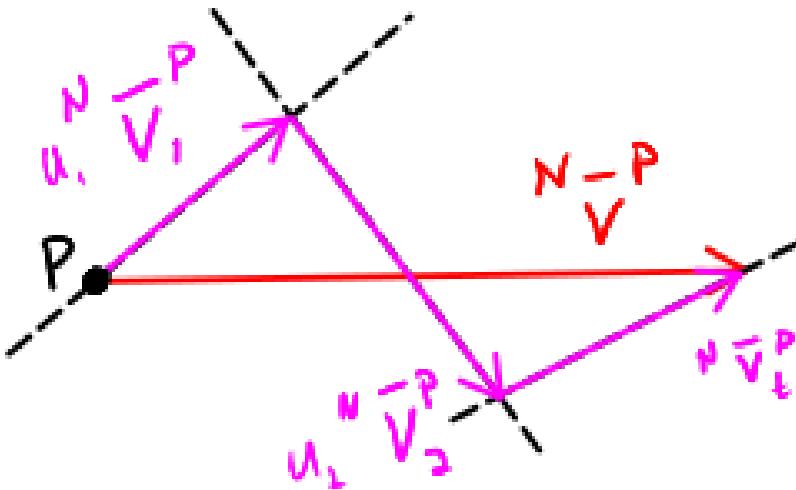


Fig. 15.2: Velocity vector ${}^N\bar{v}^P$ of point P shown expressed as a sum of linear combinations of generalized speeds and partial velocity vectors and a remainder vector. In this case there are two generalized speeds.

Partial velocities can be determined by inspection of velocity vector expressions or calculated by taking the appropriate partial derivatives. Take, for example, the single body system shown in Fig. 15.3. What are the partial velocities for ${}^N\bar{v}^A$, ${}^N\bar{v}^B$, and ${}^N\bar{\omega}^R$?

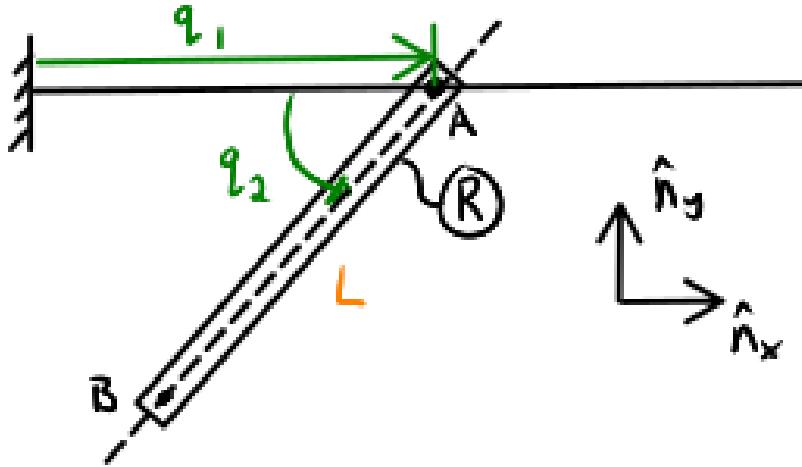


Fig. 15.3: A rod R pinned at A on the horizontal line. A 's horizontal translation is described by the generalized coordinate q_1 and the angle of the rod relative to the horizontal is described by the generalized coordinate q_2 .

First calculate the velocities and ensure they are only in terms of the generalized speeds and generalized coordinates. In this case, we have chosen $u_1 = \dot{q}_1$ and $u_2 = \dot{q}_2$.

```
L = sm.symbols('L')
q1, q2, u1, u2 = me.dynamicsymbols('q1, q2, u1, u2')

N = me.ReferenceFrame('N')
R = me.ReferenceFrame('R')

R.orient_axis(N, q2, N.z)
```

```
N_v_A = u1*N.x
N_v_A
```

$$u_1 \hat{n}_x \quad (15.4)$$

```
N_w_R = u2*N.z
N_w_R
```

$$u_2 \hat{n}_z \quad (15.5)$$

```
r_A_B = -L*R.x
N_v_B = N_v_A + me.cross(N_w_R, r_A_B)

N_v_B.express(N)
```

$$(Lu_2 \sin(q_2) + u_1)\hat{n}_x - Lu_2 \cos(q_2)\hat{n}_y \quad (15.6)$$

Now, take the partial derivatives with respect to the generalized speeds to find the six partial velocities. The sensitivity of point A 's linear motion is only a function of the first generalized speed, i.e. change in u_1 will cause accelerations in the \hat{n}_x direction.

```
v_A_1 = N_v_A.diff(u1, N)
v_A_2 = N_v_A.diff(u2, N)

v_A_1, v_A_2
```

$$(\hat{n}_x, 0) \quad (15.7)$$

The sensitivity of point B 's linear motion is a function of both generalized speeds, showing that acceleration in the \hat{n}_x direction is caused by change in both generalized speeds. In the \hat{n}_y direction motion change is only caused by change in u_2 .

```
v_B_1 = N_v_B.diff(u1, N)
v_B_2 = N_v_B.diff(u2, N)

v_B_1, v_B_2
```

$$(\hat{n}_x, -L\hat{r}_y) \quad (15.8)$$

Lastly, the sensitivity of the body R 's angular velocity to the two generalized speeds is only from u_2 in the \hat{n}_z direction.

```
w_R_1 = N_w_R.diff(u1, N)
w_R_2 = N_w_R.diff(u2, N)

w_R_1, w_R_2
```

$$(0, \hat{n}_z) \quad (15.9)$$

SymPy Mechanics provides a convenience function `partial_velocity()` to calculate a set of partial velocities for a set of generalized speeds:

```
me.partial_velocity((N_v_A, N_v_B, N_w_R), (u1, u2), N)
```

$$[[\hat{n}_x, 0], [\hat{n}_x, -L\hat{r}_y], [0, \hat{n}_z]] \quad (15.10)$$

15.4 Nonholonomic Partial Velocities

If a system is nonholonomic, it is also true that every translational and angular velocity can be expressed uniquely in terms of the p independent generalized speeds (see Eq. (12.58)). Thus, we can also define the *nonholonomic partial velocities*

\tilde{v}_r and nonholonomic partial angular velocities $\tilde{\omega}_r$ as per ([Kane1985], pg. 46):

$$\begin{aligned}\bar{v} &= \sum_{r=1}^p \tilde{v}_r u_r + \tilde{v}_t \\ \bar{\omega} &= \sum_{r=1}^p \tilde{\omega}_r u_r + \tilde{\omega}_t\end{aligned}\tag{15.11}$$

If you have found the n holonomic partial velocities, then you can use \mathbf{A}_n from (12.58) to find the nonholonomic partial velocities with:

$$\begin{aligned}\tilde{v}_r &= \bar{v}_r + [\bar{v}_{p+1} \dots \bar{v}_n] \mathbf{A}_n \hat{e}_r \\ \tilde{\omega}_r &= \bar{\omega}_r + [\bar{\omega}_{p+1} \dots \bar{\omega}_n] \mathbf{A}_n \hat{e}_r \quad \text{for } r = 1 \dots p\end{aligned}\tag{15.12}$$

where \hat{e}_r is a unit vector in the independent speed \bar{u}_s vector space, e.g. $\hat{e}_2 = [0, 1, 0, 0]^T$ if $p = 4$. See [Kane1985] pg. 48 for more explanation.

15.5 Generalized Active Forces

Suppose we have a holonomic multibody system made up of ν particles with n degrees of freedom in a reference frame A that are described by generalized speeds u_1, \dots, u_n . Each particle may have a resultant force \bar{R} applied to it. By projecting each of the forces onto the partial velocity of its associated particle and summing the projections, we arrive at the total scalar force contribution associated with changes in that generalized speed. We call these scalar values, one for each generalized speed, the *generalized active forces*. The r^{th} holonomic generalized active force for this system in A is defined as ([Kane1985], pg. 99):

$$F_r := \sum_{i=1}^{\nu} {}^A \bar{v}_r^{P_i} \cdot \bar{R}_i \tag{15.13}$$

where i represents the i^{th} particle.

Notice that the r^{th} generalized active force is:

1. a scalar value
2. has contributions from all particles except if ${}^A \bar{v}_r^{P_i} \perp \bar{R}_i$
3. associated with the r^{th} generalized speed

We will typically collect all of the generalized active forces in a column vector to allow for matrix operations with these values:

$$\bar{F}_r = \begin{bmatrix} \sum_{i=1}^{\nu} {}^A \bar{v}_1^{P_i} \cdot \bar{R}_i \\ \vdots \\ \sum_{i=1}^{\nu} {}^A \bar{v}_r^{P_i} \cdot \bar{R}_i \\ \vdots \\ \sum_{i=1}^{\nu} {}^A \bar{v}_n^{P_i} \cdot \bar{R}_i \end{bmatrix} \tag{15.14}$$

Eq. (15.13) shows that the partial velocities transform the forces applied to the multibody system from their Cartesian vector space to a new generalized speed vector space.

Now let us calculate the generalized active forces for a simple multibody system made up of only particles. Fig. 15.4 shows a double simple pendulum made up of two particles P_1 and P_2 with masses m_1 and m_2 respectively.

To calculate the generalized active forces we first find the velocities of each particle and write them in terms of the generalized speeds which we define as $u_1 = \dot{q}_1, u_2 = \dot{q}_2$.

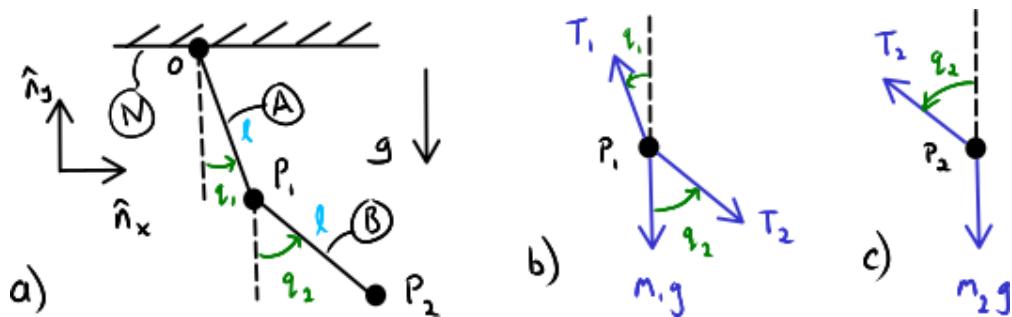


Fig. 15.4: Double simple pendulum a) kinematic schematic, b) free body diagram of P_1 , c) free body diagram of P_2 .

```

l = sm.symbols('l')
q1, q2, u1, u2 = me.dynamicsymbols('q1, q2, u1, u2')

N = me.ReferenceFrame('N')
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')

A.orient_axis(N, q1, N.z)
B.orient_axis(N, q2, N.z)

O = me.Point('O')
P1 = me.Point('P1')
P2 = me.Point('P2')

O.set_vel(N, 0)

P1.set_pos(O, -l*A.y)
P2.set_pos(P1, -l*B.y)

P1.v2pt_theory(O, N, A)
P2.v2pt_theory(P1, N, B)

P1.vel(N), P2.vel(N)

```

$$\left(l\dot{q}_1\hat{a}_x, l\dot{q}_1\hat{a}_x + l\dot{q}_2\hat{b}_x \right) \quad (15.15)$$

```

repl = {q1.diff(): u1, q2.diff(): u2}

N_v_P1 = P1.vel(N).xreplace(repl)
N_v_P2 = P2.vel(N).xreplace(repl)

N_v_P1, N_v_P2

```

$$\left(lu_1\hat{a}_x, lu_1\hat{a}_x + lu_2\hat{b}_x \right) \quad (15.16)$$

We will need the partial velocities of each particle with respect to the two generalized speeds, giving four partial velocities:

```
v_P1_1 = N_v_P1.diff(u1, N)
v_P1_2 = N_v_P1.diff(u2, N)
v_P2_1 = N_v_P2.diff(u1, N)
v_P2_2 = N_v_P2.diff(u2, N)
v_P1_1, v_P1_2, v_P2_1, v_P2_2
```

$$\left(l\hat{a}_x, 0, l\hat{a}_x, l\hat{b}_x \right) \quad (15.17)$$

To determine the resultant forces acting on each particle we isolate each particle from the system and draw a free body diagram with all of the forces acting on the particle. Each particle has a gravitational force as well as distance, or tension, forces that ensure the particle stays connected to the massless rod. The resultant forces on each particle are then:

```
T1, T2 = me.dynamicsymbols('T1, T2')
m1, m2, g = sm.symbols('m1, m2, g')

R1 = -m1*g*N.y + T1*A.y - T2*B.y
R1
```

$$-gm_1\hat{n}_y + T_1\hat{a}_y - T_2\hat{b}_y \quad (15.18)$$

```
R2 = -m2*g*N.y + T2*B.y
R2
```

$$-gm_2\hat{n}_y + T_2\hat{b}_y \quad (15.19)$$

With the resultants and the partial velocities defined, the two generalized active forces can then be found:

```
F1 = me.dot(v_P1_1, R1) + me.dot(v_P2_1, R2)
F1
```

$$-glm_1 \sin(q_1) - glm_2 \sin(q_1) \quad (15.20)$$

```
F2 = me.dot(v_P1_2, R1) + me.dot(v_P2_2, R2)
F2
```

$$-glm_2 \sin(q_2) \quad (15.21)$$

Notice that the distance forces T_1, T_2 are not present in the generalized active forces F_1 or F_2 . This is not by coincidence, but will always be true for noncontributing forces. They are in fact named “noncontributing” because they do not contribute to the generalized active forces (nor the full equations of motion we eventually arrive at). Noncontributing forces need not be considered in the resultants, in general, and we will not include them in further examples.

Notice also that these generalized forces have units of force \times length. This is because our generalized speeds are angular rates. If our generalized speeds were linear rates, the generalized forces would have units of force.

15.6 Generalized Active Forces on a Rigid Body

If a holonomic multibody system with n degrees of freedom in reference frame A includes a rigid body B then the loads acting on B can be described by a resultant force \bar{R} bound to line through an arbitrary point Q in B and a couple with torque \bar{T} . The generalized active force in A for a single rigid body in a multibody system is then defined as ([Kane1985], pg. 106):

$$(F_r)_B := {}^A\bar{v}_r^Q \cdot \bar{R} + {}^A\bar{\omega}_r^B \cdot \bar{T} \quad (15.22)$$

A generalized active force for each rigid body and particle in a system must be summed to obtain the total generalized active force.

To demonstrate finding the generalized active forces for a multibody system with two rigid bodies consider Fig. 15.5 which shows two thin rods of length l that are connected at points O and B_o .

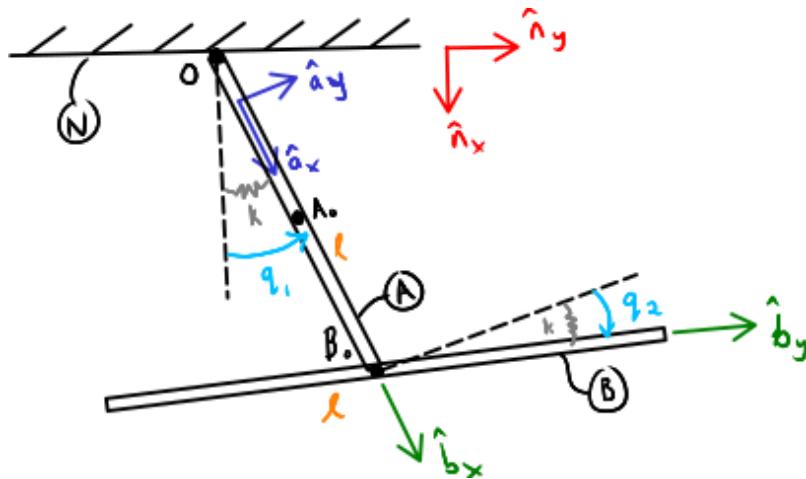


Fig. 15.5: A multibody system comprised of two uniformly dense thin rods of length l and mass m . Rod A is pinned at O and can rotate about \hat{n}_z through q_1 . Rod B is pinned to A and can rotate relative to A about \hat{n}_x through q_2 . Linear torsional springs of stiffness k with a free length of zero resists each relative rotation. Gravitational forces are in the \hat{n}_x direction.

The first step is to define the necessary velocities we'll need: translational velocities of the two mass centers and the angular velocities of each body. We use the simple definition of the generalized speeds $u_i = \dot{q}_i$.

```

m, g, k, l = sm.symbols('m, g, k, l')
q1, q2, u1, u2 = me.dynamicsymbols('q1, q2, u1, u2')

N = me.ReferenceFrame('N')
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')

A.orient_axis(N, q1, N.z)
B.orient_axis(A, q2, A.x)

A.set_ang_vel(N, u1*N.z)
B.set_ang_vel(A, u2*A.x)

O = me.Point('O')
Ao = me.Point('A_O')
Bo = me.Point('B_O')

```

(continues on next page)

(continued from previous page)

```

Ao.set_pos(0, 1/2*A.x)
Bo.set_pos(0, 1*A.x)

O.set_vel(N, 0)
Ao.v2pt_theory(O, N, A)
Bo.v2pt_theory(O, N, A)

Ao.vel(N), Bo.vel(N), A.ang_vel_in(N), B.ang_vel_in(N)

```

$$\left(\frac{lu_1}{2} \hat{a}_y, lu_1 \hat{a}_y, u_1 \hat{n}_z, u_2 \hat{a}_x + u_1 \hat{n}_z \right) \quad (15.23)$$

Now determine the holonomic partial velocities in N :

```

v_Ao_1 = Ao.vel(N).diff(u1, N)
v_Ao_2 = Ao.vel(N).diff(u2, N)
v_Bo_1 = Bo.vel(N).diff(u1, N)
v_Bo_2 = Bo.vel(N).diff(u2, N)

v_Ao_1, v_Ao_2, v_Bo_1, v_Bo_2

```

$$\left(\frac{l}{2} \hat{a}_y, 0, l \hat{a}_y, 0 \right) \quad (15.24)$$

and the holonomic partial angular velocities in N :

```

w_A_1 = A.ang_vel_in(N).diff(u1, N)
w_A_2 = A.ang_vel_in(N).diff(u2, N)
w_B_1 = B.ang_vel_in(N).diff(u1, N)
w_B_2 = B.ang_vel_in(N).diff(u2, N)

w_A_1, w_A_2, w_B_1, w_B_2

```

$$(\hat{n}_z, 0, \hat{n}_z, \hat{a}_x) \quad (15.25)$$

The resultant forces on the two bodies are simply the gravitational forces that act at each mass center (we ignore the noncontributing pin joint contact forces):

```

R_Ao = m*g*N.x
R_Bo = m*g*N.x

R_Ao, R_Bo

```

$$(gm\hat{n}_x, gm\hat{n}_x) \quad (15.26)$$

With linear torsion springs between frames A and N and frames A and B the torques acting on each body are:

$$T_A = -k * q1 * N.z + k * q2 * A.x$$

$$T_B = -k * q2 * A.x$$

$$T_A, T_B$$

$$(-kq_1 \hat{n}_z + kq_2 \hat{a}_x, -kq_2 \hat{a}_x) \quad (15.27)$$

Note that $kq_2 \hat{a}_x$ in \bar{T}_A is the reaction torque of body B on A via the torsional spring.

Now, a generalized active force component can be found for each body and each generalized speed using (15.22):

$$\begin{aligned} F1_A &= v_{Ao_1} \cdot \dot{R}_{Ao} + w_{A_1} \cdot \dot{T}_A \\ F1_B &= v_{Bo_1} \cdot \dot{R}_{Bo} + w_{B_1} \cdot \dot{T}_B \\ F2_A &= v_{Ao_2} \cdot \dot{R}_{Ao} + w_{A_2} \cdot \dot{T}_A \\ F2_B &= v_{Bo_2} \cdot \dot{R}_{Bo} + w_{B_2} \cdot \dot{T}_B \\ F1_A, F1_B, F2_A, F2_B \end{aligned}$$

$$\left(-\frac{glm \sin(q_1)}{2} - kq_1, -glm \sin(q_1), 0, -kq_2 \right) \quad (15.28)$$

Summing for each generalized speed and then stacking the two scalars in a column vector gives the generalized active forces for the system:

$$\begin{aligned} F1 &= F1_A + F1_B \\ F2 &= F2_A + F2_B \\ Fr &= sm.Matrix([F1, F2]) \\ Fr \end{aligned}$$

$$\begin{bmatrix} -\frac{3glm \sin(q_1)}{2} - kq_1 \\ -kq_2 \end{bmatrix} \quad (15.29)$$

15.7 Nonholonomic Generalized Active Forces

For a nonholonomic system with p degrees of freedom in reference frame A , the p generalized active forces can be formed instead. The nonholonomic generalized active force contributions from a particle P and rigid body B are:

$$\begin{aligned} (\tilde{F}_r)_P &= {}^A\tilde{v}^P \cdot \bar{R} \\ (\tilde{F}_r)_B &= {}^A\tilde{v}^Q \cdot \bar{R} + {}^A\tilde{\omega}^B \cdot \bar{T} \end{aligned} \quad (15.30)$$

As a corollary to (15.12), if the holonomic generalized active forces are known and nonholonomic constraints are introduced the nonholonomic generalized active forces can be found with

$$\tilde{F}_r = F_r + [F_{p+1} \dots F_n] \mathbf{A}_n \hat{e}_r \text{ for } r = 1 \dots p \quad (15.31)$$

where $[F_{p+1} \dots F_n]$ are the m holonomic generalized active forces associated with the dependent generalized speeds. See [Kane1985] pg. 99 for more information.

15.8 Generalized Inertia Forces

Analogous to the generalized active forces and their relationship to the left hand side of the Newton-Euler equations (Eq. (15.1)), *generalized inertia forces* map the right hand side of the Newton-Euler equations, time derivatives of linear and angular momentum, to the vector space of the generalized speeds for a multibody system. For a holonomic multibody system in A made up of a set of ν particles the i^{th} generalized inertia force is defined as ([Kane1985], pg. 124):

$$F_r^* := \sum_{i=1}^{\nu} {}^A \bar{v}_r^{P_i} \cdot \bar{R}_i^* \quad (15.32)$$

where the resultant *inertia force* on the i^{th} particle is:

$$\bar{R}_i^* := -m_i {}^A \bar{a}_i^{P_i} \quad (15.33)$$

The generalized inertia force for a single rigid body B with mass m_B , mass center B_o , and central inertia dyadic \check{I}^{B/B_o} is defined as:

$$(F_r^*)_B := {}^A \bar{v}_r^{B_o} \cdot \bar{R}^* + {}^A \bar{\omega}_r^B \cdot \bar{T}^* \quad (15.34)$$

where the inertia force on the body is:

$$\bar{R}^* := -m_B {}^A \bar{a}^{B_o} \quad (15.35)$$

and the *inertia torque* on the body are

$$\bar{T}^* := - \left({}^A \bar{\alpha}^B \cdot \check{I}^{B/B_o} + {}^A \bar{\omega}^B \times \check{I}^{B/B_o} \cdot {}^A \bar{\omega}^B \right) \quad (15.36)$$

Coming back to the system in Fig. 15.5 we can now calculate the generalized inertia forces for the two rigid body system. First, the velocities and partial velocities are found as before:

```

m, g, k, l = sm.symbols('m, g, k, l')
q1, q2, u1, u2 = me.dynamicsymbols('q1, q2, u1, u2')

N = me.ReferenceFrame('N')
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')

A.orient_axis(N, q1, N.z)
B.orient_axis(A, q2, A.x)

A.set_ang_vel(N, u1*N.z)
B.set_ang_vel(A, u2*A.x)

O = me.Point('O')
Ao = me.Point('A_O')
Bo = me.Point('B_O')

Ao.set_pos(O, l/2*A.x)
Bo.set_pos(O, l*A.x)

O.set_vel(N, 0)
Ao.v2pt_theory(O, N, A)
Bo.v2pt_theory(O, N, A)

v_Ao_1 = Ao.vel(N).diff(u1, N)

```

(continues on next page)

(continued from previous page)

```

v_Ao_2 = Ao.vel(N).diff(u2, N)
v_Bo_1 = Bo.vel(N).diff(u1, N)
v_Bo_2 = Bo.vel(N).diff(u2, N)

w_A_1 = A.ang_vel_in(N).diff(u1, N)
w_A_2 = A.ang_vel_in(N).diff(u2, N)
w_B_1 = B.ang_vel_in(N).diff(u1, N)
w_B_2 = B.ang_vel_in(N).diff(u2, N)
    
```

We will need the translational accelerations of the mass centers and the angular accelerations of each body expressed in terms of the generalized speeds, their derivatives, and the generalized coordinates:

```
Ao.acc(N), Bo.acc(N)
```

$$\left(-\frac{lu_1^2}{2}\hat{a}_x + \frac{l\dot{u}_1}{2}\hat{a}_y, -lu_1^2\hat{a}_x + l\dot{u}_1\hat{a}_y \right) \quad (15.37)$$

```
A.ang_acc_in(N), B.ang_acc_in(N)
```

$$(\dot{u}_1\hat{n}_z, \dot{u}_2\hat{a}_x + u_1u_2\hat{a}_y + \dot{u}_1\hat{n}_z) \quad (15.38)$$

The central moment of inertia of a thin uniformly dense rod of mass m and length L about any axis normal to its length is:

```
I = m*l**2/12
I
```

$$\frac{l^2m}{12} \quad (15.39)$$

This can be used to formulate the central inertia dyadics of each rod:

```

I_A_Ao = I*me.outer(A.y, A.y) + I*me.outer(A.z, A.z)
I_B_Bo = I*me.outer(B.x, B.x) + I*me.outer(B.z, B.z)
I_A_Ao, I_B_Bo
    
```

$$\left(\frac{l^2m}{12}\hat{a}_y \otimes \hat{a}_y + \frac{l^2m}{12}\hat{a}_z \otimes \hat{a}_z, \frac{l^2m}{12}\hat{b}_x \otimes \hat{b}_x + \frac{l^2m}{12}\hat{b}_z \otimes \hat{b}_z \right) \quad (15.40)$$

The resultant inertia forces acting at the mass center of each body are:

```

Rs_Ao = -m*Ao.acc(N)
Rs_Bo = -m*Bo.acc(N)

Rs_Ao, Rs_Bo
    
```

$$\left(\frac{lmu_1^2}{2}\hat{a}_x - \frac{lmu_1}{2}\hat{a}_y, lmu_1^2\hat{a}_x - lmu_1\hat{a}_y \right) \quad (15.41)$$

And the inertia torques acting on each body are:

```

Ts_A = -(A.ang_acc_in(N).dot(I_A_Ao) +
          me.cross(A.ang_vel_in(N), I_A_Ao).dot(A.ang_vel_in(N)))
Ts_A
    
```

$$-\frac{l^2 m \dot{u}_1}{12} \hat{a}_z \quad (15.42)$$

```

Ts_B = -(B.ang_acc_in(N).dot(I_B_Bo) +
          me.cross(B.ang_vel_in(N), I_B_Bo).dot(B.ang_vel_in(N)))
Ts_B
    
```

$$\left(-\frac{l^2 m u_1^2 \sin(q_2) \cos(q_2)}{12} - \frac{l^2 m \dot{u}_2}{12} \right) \hat{b}_x + \left(-\frac{l^2 m (-u_1 u_2 \sin(q_2) + \cos(q_2) \dot{u}_1)}{12} + \frac{l^2 m u_1 u_2 \sin(q_2)}{12} \right) \hat{b}_z \quad (15.43)$$

Now the generalized inertia forces can be formed by projecting the inertia force and inertia torque onto the partial velocities:

```

F1s_A = v_Ao_1.dot(Rs_Ao) + w_A_1.dot(Ts_A)
F1s_B = v_Bo_1.dot(Rs_Bo) + w_B_1.dot(Ts_B)
F2s_A = v_Ao_2.dot(Rs_Ao) + w_A_2.dot(Ts_A)
F2s_B = v_Bo_2.dot(Rs_Bo) + w_B_2.dot(Ts_B)
    
```

We then sum for each generalized speed and then stack them in a column vector \bar{F}_r^* :

```

F1s = F1s_A + F1s_B
F2s = F2s_A + F2s_B

Fr_s = sm.Matrix([F1s, F2s])
Fr_s
    
```

$$\begin{bmatrix} -\frac{4l^2 m \dot{u}_1}{3} + \left(-\frac{l^2 m (-u_1 u_2 \sin(q_2) + \cos(q_2) \dot{u}_1)}{12} + \frac{l^2 m u_1 u_2 \sin(q_2)}{12} \right) \cos(q_2) \\ -\frac{l^2 m u_1^2 \sin(q_2) \cos(q_2)}{12} - \frac{l^2 m \dot{u}_2}{12} \end{bmatrix} \quad (15.44)$$

15.9 Nonholonomic Generalized Inertia Forces

For a nonholonomic system with p degrees of freedom in reference frame A , the p generalized active forces can be formed instead. The nonholonomic generalized active force contributions from a particle P and rigid body B are:

$$\begin{aligned} (\tilde{F}_r^*)_P &= {}^A \tilde{v}^P \cdot \bar{R}^* \\ (\tilde{F}_r^*)_B &= {}^A \tilde{v}^Q \cdot \bar{R}^* + {}^A \tilde{\omega}^B \cdot \bar{T}^* \end{aligned} \quad (15.45)$$

Similar to Eq. (15.31), the nonholonomic generalized inertia forces can be calculated from the holonomic generalized inertia forces and \mathbf{A}_n :

$$\tilde{F}_r^* = F_r^* + [F_{p+1}^* \dots F_n^*] \mathbf{A}_n \hat{e}_r \text{ for } r = 1 \dots p \quad (15.46)$$

More information about the relation between the nonholonomic and holonomic generalized inertia forces is give in [Kane1985] pg. 124.

UNCONSTRAINED EQUATIONS OF MOTION

Note: You can download this example as a Python script: eom.py or Jupyter Notebook: eom.ipynb.

```
import sympy as sm
import sympy.physics.mechanics as me
me.init_vprinting(use_latex='mathjax')

class ReferenceFrame(me.ReferenceFrame):
    def __init__(self, *args, **kwargs):
        kwargs.pop('latexs', None)
        lab = args[0].lower()
        tex = r'\hat{{}}_{{}}_{{}}'
        super(ReferenceFrame, self).__init__(*args,
                                             latexs=(tex.format(lab, 'x'),
                                                      tex.format(lab, 'y'),
                                                      tex.format(lab, 'z')),
                                             **kwargs)
me.ReferenceFrame = ReferenceFrame
```

16.1 Learning Objectives

After completing this chapter readers will be able to:

- Form the dynamical differential equations for a multibody system where $n = p$.
- Calculate the dynamical differential equations of motion for a single rigid body.
- Form the equations of motion for a multibody system where $n = p$.
- Write the equations of motion in implicit and explicit forms.

16.2 Dynamical Differential Equations

In the previous chapter, we introduced the generalized active forces \bar{F}_r and the generalized inertia forces \bar{F}_r^* . Together, these two forces give us the *dynamical differential equations*. The dynamical differential equations for a holonomic system with $p = n$ degrees of freedom in an inertial reference frame. They are a function of the generalized coordinates, the generalized speeds, the time derivatives of the generalized speeds, and time. The dynamical differential equations take this form:

$$\bar{F}_r + \bar{F}_r^* = \bar{f}_d(\dot{\bar{u}}, \bar{u}, \bar{q}, t) = 0 \quad (16.1)$$

These are the Newton-Euler equations for a multibody system in the form presented in [Kane1985] pg. 158, thus we also call these equations *Kane's Equations*. The dynamical differential equations can only be formed when motion is viewed from an *inertial reference frame*, because an inertial reference frame is one where Newton's First Law holds, i.e. objects at rest stay at rest unless an external force acts on them. An inertial reference frame is one that is not accelerating, or can be assumed not to be with respect to the motion of the bodies of interest.

\bar{F}_r^* is always linear in the time derivatives of the generalized speeds and contains velocity dependent terms such as the centripetal and Coriolis forces and the rotational velocity coupling terms. These forces are sometimes called *fictitious forces*. \bar{F}_r are contributing forces due to body and environment interactions. Texts about dynamics will often present the dynamical differential equations in this form:

$$-\bar{F}_r^* = \bar{F}_r \rightarrow \mathbf{M}(\bar{q}, t)\dot{\bar{u}} + \bar{C}(\bar{u}, \bar{q}, t) = \bar{F}(\bar{u}, \bar{q}, t) \quad (16.2)$$

where \mathbf{M} is called the *mass matrix*, \bar{C} are the forces due to the various velocity effects, and \bar{F} are the contributing externally applied forces.

16.3 Body Fixed Newton-Euler Equations

To show that Kane's Equations are equivalent to the Newton-Euler equations you may have seen before, we can find the dynamical differential equations for a single rigid body using Kane's method and then show the results in the canonical form. For a rigid body B moving in an inertial reference frame A with its velocity and angular velocity expressed in body fixed coordinates and acted upon by a resultant force \bar{F} at the mass center B_o and a moment about the mass center \bar{M} we need these variables, reference frames, and points:

```
m, Ixx, Iyy, Izz = sm.symbols('m, I_{xx}, I_{yy}, I_{zz}')
Ixxy, Ixzy, Ixzx = sm.symbols('I_{xy}, I_{yz}, I_{xz}')
Fx, Fy, Fz, Mx, My, Mz = me.dynamicsymbols('F_x, F_y, F_z, M_x, M_y, M_z')
u1, u2, u3, u4, u5, u6 = me.dynamicsymbols('u1, u2, u3, u4, u5, u6')

A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')

Bo = me.Point('Bo')
```

Now define the angular velocity of the body and the velocity of the mass center in terms of six generalized coordinates expressed in body fixed coordinates.

```
A_w_B = u4*B.x + u5*B.y + u6*B.z
B.set_ang_vel(A, A_w_B)

A_v_Bo = u1*B.x + u2*B.y + u3*B.z
Bo.set_vel(A, A_v_Bo)
```

Now we can find the six partial velocities and partial angular velocities. Note that we use the `var_in_dcm=False` keyword argument. We do this because the generalized speeds are not present in the unspecified direction cosine matrix relating A and B . This allows the derivative in A to be formed without use of a direction cosine matrix. Generalized speeds will never be present in a direction cosine matrix.

```
v_Bo_1 = A_v_Bo.diff(u1, A, var_in_dcm=False)
v_Bo_2 = A_v_Bo.diff(u2, A, var_in_dcm=False)
v_Bo_3 = A_v_Bo.diff(u3, A, var_in_dcm=False)
v_Bo_4 = A_v_Bo.diff(u4, A, var_in_dcm=False)
v_Bo_5 = A_v_Bo.diff(u5, A, var_in_dcm=False)
v_Bo_6 = A_v_Bo.diff(u6, A, var_in_dcm=False)

v_Bo_1, v_Bo_2, v_Bo_3, v_Bo_4, v_Bo_5, v_Bo_6
```

$$\left(\hat{b}_x, \hat{b}_y, \hat{b}_z, 0, 0, 0 \right) \quad (16.3)$$

```
w_B_1 = A_w_B.diff(u1, A, var_in_dcm=False)
w_B_2 = A_w_B.diff(u2, A, var_in_dcm=False)
w_B_3 = A_w_B.diff(u3, A, var_in_dcm=False)
w_B_4 = A_w_B.diff(u4, A, var_in_dcm=False)
w_B_5 = A_w_B.diff(u5, A, var_in_dcm=False)
w_B_6 = A_w_B.diff(u6, A, var_in_dcm=False)

w_B_1, w_B_2, w_B_3, w_B_4, w_B_5, w_B_6
```

$$\left(0, 0, 0, \hat{b}_x, \hat{b}_y, \hat{b}_z \right) \quad (16.4)$$

The `partial_velocity()` function does this same thing. Notice that due to our velocity definitions, we get a very simple set of partial velocities.

```
par_vels = me.partial_velocity([A_v_Bo, A_w_B], [u1, u2, u3, u4, u5, u6], A)

par_vels
```

$$\left[\left[\hat{b}_x, \hat{b}_y, \hat{b}_z, 0, 0, 0 \right], \left[0, 0, 0, \hat{b}_x, \hat{b}_y, \hat{b}_z \right] \right] \quad (16.5)$$

Now form the generalized active forces:

```
T = Mx*B.x + My*B.y + Mz*B.z
R = Fx*B.x + Fy*B.y + Fz*B.z

F1 = v_Bo_1.dot(R) + w_B_1.dot(T)
F2 = v_Bo_2.dot(R) + w_B_2.dot(T)
F3 = v_Bo_3.dot(R) + w_B_3.dot(T)
F4 = v_Bo_4.dot(R) + w_B_4.dot(T)
F5 = v_Bo_5.dot(R) + w_B_5.dot(T)
F6 = v_Bo_6.dot(R) + w_B_6.dot(T)

Fr = sm.Matrix([F1, F2, F3, F4, F5, F6])
Fr
```

$$\begin{bmatrix} F_x \\ F_y \\ F_z \\ M_x \\ M_x \\ M_z \end{bmatrix} \quad (16.6)$$

and the generalized inertia forces:

```

I = me.inertia(B, Ixx, Iyy, Izz, Ixy, Iyz, Ixz)

Rs = -m*Bo.acc(A)
Ts = -(B.ang_acc_in(A).dot(I) + me.cross(A_w_B, I).dot(A_w_B))

F1s = v_Bo_1.dot(Rs) + w_B_1.dot(Ts)
F2s = v_Bo_2.dot(Rs) + w_B_2.dot(Ts)
F3s = v_Bo_3.dot(Rs) + w_B_3.dot(Ts)
F4s = v_Bo_4.dot(Rs) + w_B_4.dot(Ts)
F5s = v_Bo_5.dot(Rs) + w_B_5.dot(Ts)
F6s = v_Bo_6.dot(Rs) + w_B_6.dot(Ts)

Fr_s = sm.Matrix([F1s, F2s, F3s, F4s, F5s, F6s])
Fr_s

```

$$\begin{bmatrix} -m(-u_2u_6 + u_3u_5 + \dot{u}_1) \\ -m(u_1u_6 - u_3u_4 + \dot{u}_2) \\ -m(-u_1u_5 + u_2u_4 + \dot{u}_3) \\ -I_{xx}\dot{u}_4 - I_{xy}\dot{u}_5 - I_{xz}\dot{u}_6 - (-I_{xy}u_6 + I_{xz}u_5)u_4 - (-I_{yy}u_6 + I_{yz}u_5)u_5 - (-I_{yz}u_6 + I_{zz}u_5)u_6 \\ -I_{xy}\dot{u}_4 - I_{yy}\dot{u}_5 - I_{yz}\dot{u}_6 - (I_{xx}u_6 - I_{xz}u_4)u_4 - (I_{xy}u_6 - I_{yz}u_4)u_5 - (I_{xz}u_6 - I_{zz}u_4)u_6 \\ -I_{xz}\dot{u}_4 - I_{yz}\dot{u}_5 - I_{zz}\dot{u}_6 - (-I_{xx}u_5 + I_{xy}u_4)u_4 - (-I_{xy}u_5 + I_{yy}u_4)u_5 - (-I_{xz}u_5 + I_{yz}u_4)u_6 \end{bmatrix} \quad (16.7)$$

and finally Kane's Equations:

```
Fr + Fr_s
```

$$\begin{bmatrix} -m(-u_2u_6 + u_3u_5 + \dot{u}_1) + F_x \\ -m(u_1u_6 - u_3u_4 + \dot{u}_2) + F_y \\ -m(-u_1u_5 + u_2u_4 + \dot{u}_3) + F_z \\ -I_{xx}\dot{u}_4 - I_{xy}\dot{u}_5 - I_{xz}\dot{u}_6 - (-I_{xy}u_6 + I_{xz}u_5)u_4 - (-I_{yy}u_6 + I_{yz}u_5)u_5 - (-I_{yz}u_6 + I_{zz}u_5)u_6 + M_x \\ -I_{xy}\dot{u}_4 - I_{yy}\dot{u}_5 - I_{yz}\dot{u}_6 - (I_{xx}u_6 - I_{xz}u_4)u_4 - (I_{xy}u_6 - I_{yz}u_4)u_5 - (I_{xz}u_6 - I_{zz}u_4)u_6 + M_x \\ -I_{xz}\dot{u}_4 - I_{yz}\dot{u}_5 - I_{zz}\dot{u}_6 - (-I_{xx}u_5 + I_{xy}u_4)u_4 - (-I_{xy}u_5 + I_{yy}u_4)u_5 - (-I_{xz}u_5 + I_{yz}u_4)u_6 + M_z \end{bmatrix} \quad (16.8)$$

We can put Kane's Equations in canonical form (Eq. (16.2)) by extracting the mass matrix, which is the linear coefficient matrix of $\dot{\bar{u}}$:

```

u = sm.Matrix([u1, u2, u3, u4, u5, u6])
t = me.dynamicsymbols._t
ud = u.diff(t)

```

The mass matrix is:

```
M = -Frs.jacobian(u)
M
```

$$\begin{bmatrix} m & 0 & 0 & 0 & 0 & 0 \\ 0 & m & 0 & 0 & 0 & 0 \\ 0 & 0 & m & 0 & 0 & 0 \\ 0 & 0 & 0 & I_{xx} & I_{xy} & I_{xz} \\ 0 & 0 & 0 & I_{xy} & I_{yy} & I_{yz} \\ 0 & 0 & 0 & I_{xz} & I_{yz} & I_{zz} \end{bmatrix} \quad (16.9)$$

The velocity forces vector is:

```
C = -Frs.xreplace({udi: 0 for udi in u})
C
```

$$\begin{bmatrix} m(-u_2u_6 + u_3u_5) \\ m(u_1u_6 - u_3u_4) \\ m(-u_1u_5 + u_2u_4) \\ (-I_{xy}u_6 + I_{xz}u_5)u_4 + (-I_{yy}u_6 + I_{yz}u_5)u_5 + (-I_{yz}u_6 + I_{zz}u_5)u_6 \\ (I_{xx}u_6 - I_{xz}u_4)u_4 + (I_{xy}u_6 - I_{yz}u_4)u_5 + (I_{xz}u_6 - I_{zz}u_4)u_6 \\ (-I_{xx}u_5 + I_{xy}u_4)u_4 + (-I_{xy}u_5 + I_{yy}u_4)u_5 + (-I_{xz}u_5 + I_{yz}u_4)u_6 \end{bmatrix} \quad (16.10)$$

And the forcing vector is:

```
F = Fr
F
```

$$\begin{bmatrix} F_x \\ F_y \\ F_z \\ M_x \\ M_x \\ M_z \end{bmatrix} \quad (16.11)$$

This example may seem overly complicated when using Kane's method, but it is a systematic method that works for any number of rigid bodies and particles in a system.

16.4 Equations of Motion

The kinematical and dynamical differential equations constitute the *equations of motion* for a holonomic multibody system. These equations are ordinary differential equations in the generalized speeds and generalized coordinates.

$$\begin{aligned} \bar{f}_d(\dot{\bar{u}}, \bar{u}, \bar{q}, t) &= 0 \\ \bar{f}_k(\dot{\bar{q}}, \bar{u}, \bar{q}, t) &= 0 \end{aligned} \quad (16.12)$$

and since they are both linear in $\dot{\bar{u}}$ and $\dot{\bar{q}}$, respectively, they can be written in a combined form:

$$\begin{bmatrix} \mathbf{M}_k & 0 \\ 0 & \mathbf{M}_d \end{bmatrix} \begin{bmatrix} \dot{\bar{q}} \\ \dot{\bar{u}} \end{bmatrix} + \begin{bmatrix} \bar{g}_k(\bar{u}, \bar{q}, t) \\ \bar{g}_d(\bar{u}, \bar{q}, t) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad (16.13)$$

which we write as:

$$\mathbf{M}_m \dot{\bar{x}} + \bar{g}_m = \bar{0} \quad (16.14)$$

where $\bar{x} = [\bar{q} \quad \bar{u}]^T$ is called the *state* of the system and is comprised of the generalized coordinates and generalized speeds.

16.5 Example of Kane's Equations

Returning to the example from the previous chapter, I will add an additional particle of mass $m/4$ at point Q that can slide along the rod B and is attached to point B_o via a linear translational spring with stiffness k_l and located by generalized coordinate q_3 . The torsional spring stiffness has been renamed to k_t . See Fig. 16.1 for a visual description.

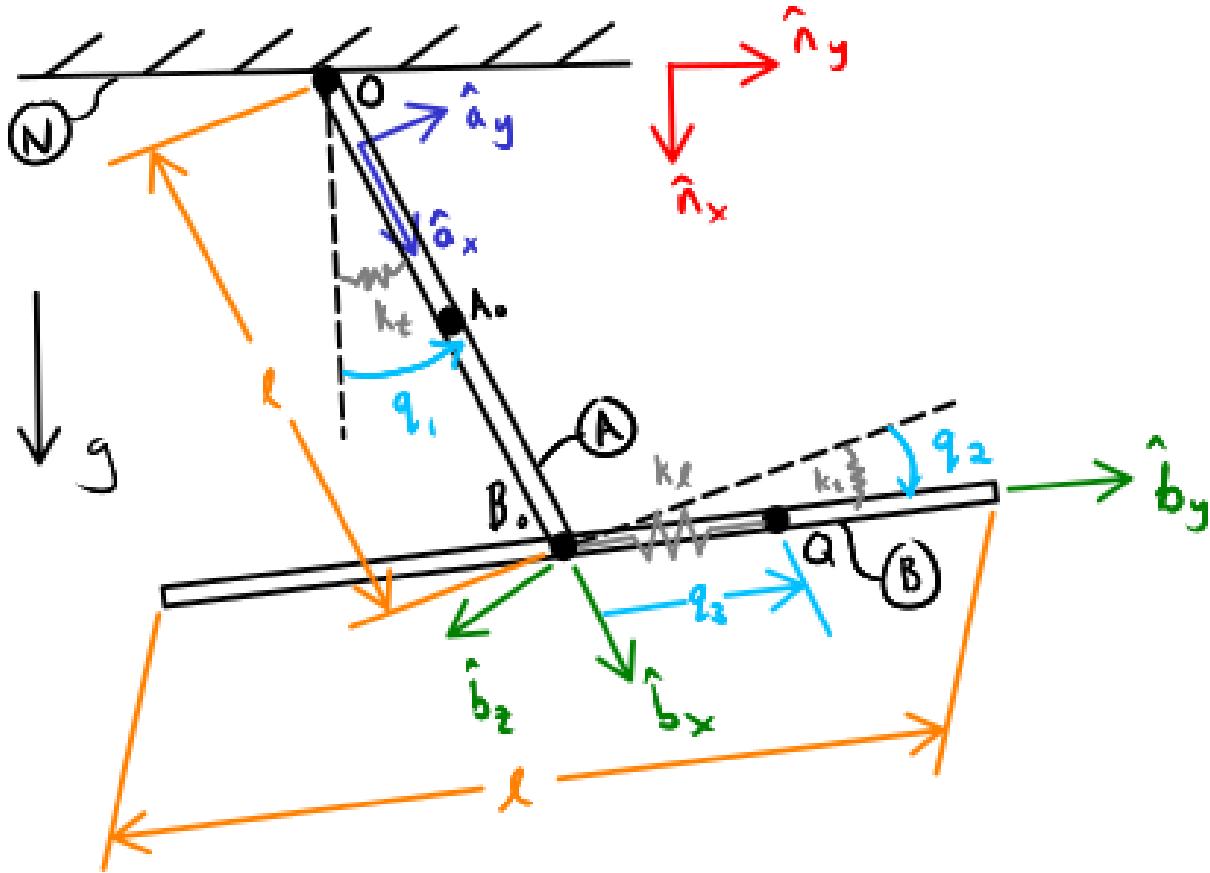


Fig. 16.1: Three dimensional pendulum made up of two pinned rods and a sliding mass on rod B . Each degree of freedom is resisted by a linear spring. When the generalized coordinates are all zero, the two rods are perpendicular to each other.

The following code is reproduced from the prior chapter and gives the velocities and angular velocities of A_o , B_o , A , and B in the inertial reference frame N .

```

m, g, kt, kl, l = sm.symbols('m, g, k_t, k_l, l')
q1, q2, q3 = me.dynamicsymbols('q1, q2, q3')
u1, u2, u3 = me.dynamicsymbols('u1, u2, u3')

N = me.ReferenceFrame('N')
    
```

(continues on next page)

(continued from previous page)

```

A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')

A.orient_axis(N, q1, N.z)
B.orient_axis(A, q2, A.x)

A.set_ang_vel(N, u1*N.z)
B.set_ang_vel(A, u2*A.x)

O = me.Point('O')
Ao = me.Point('A_O')
Bo = me.Point('B_O')

Ao.set_pos(O, 1/2*A.x)
Bo.set_pos(O, 1*A.x)

O.set_vel(N, 0)
Ao.v2pt_theory(O, N, A)
Bo.v2pt_theory(O, N, A)

Ao.vel(N), Bo.vel(N), A.ang_vel_in(N), B.ang_vel_in(N)

```

$$\left(\frac{lu_1}{2} \hat{a}_y, lu_1 \hat{a}_y, u_1 \hat{n}_z, u_2 \hat{a}_x + u_1 \hat{n}_z \right) \quad (16.15)$$

We now have the particle at Q so we need its velocity for its contribution to F_r and F_r^* . Q is moving in B so the one point velocity theorem can be used.

```

Q = me.Point('Q')
Q.set_pos(Bo, q3*B.y)
Q.set_vel(B, u3*B.y)
Q.v1pt_theory(Bo, N, B)

Q.vel(N)

```

$$-q_3 u_1 \cos(q_2) \hat{b}_x + u_3 \hat{b}_y + q_3 u_2 \hat{b}_z + lu_1 \hat{a}_y \quad (16.16)$$

We will also need the accelerations of the points and frames for the generalized inertia forces. For points A_o , B_o and frames A and B these are nicely expressed in terms of \dot{u} , \ddot{u} , \ddot{q} :

```

Ao.acc(N), Bo.acc(N), A.ang_acc_in(N), B.ang_acc_in(N)

```

$$\left(-\frac{lu_1^2}{2} \hat{a}_x + \frac{l\dot{u}_1}{2} \hat{a}_y, -lu_1^2 \hat{a}_x + l\dot{u}_1 \hat{a}_y, \dot{u}_1 \hat{n}_z, \dot{u}_2 \hat{a}_x + u_1 u_2 \hat{a}_y + \dot{u}_1 \hat{n}_z \right) \quad (16.17)$$

but the acceleration of point Q contains $\dot{\ddot{q}}$ terms, so we need to eliminate those with the kinematical differential equations:

```

Q.acc(N)

```

$$(q_3 u_1 u_2 \sin(q_2) + q_3 u_1 \sin(q_2) \dot{q}_2 - q_3 \cos(q_2) \dot{u}_1 - u_1 u_3 \cos(q_2) - u_1 \cos(q_2) \dot{q}_3) \hat{b}_x + (-q_3 u_1^2 \cos^2(q_2) - q_3 u_2^2 + \dot{u}_3) \hat{b}_y + (q_3 u_1^2 \sin(q_2) + q_3 u_1 \cos(q_2) \dot{q}_2 - q_3 \sin(q_2) \dot{u}_1 - u_1 u_3 \sin(q_2) - u_1 \sin(q_2) \dot{q}_3) \hat{b}_z \quad (16.18)$$

```
t = me.dynamicsymbols._t

qdot_repl = {q1.diff(t): u1,
             q2.diff(t): u2,
             q3.diff(t): u3}

Q.set_acc(N, Q.acc(N).xreplace(qdot_repl))
Q.acc(N)
```

$$(2q_3u_1u_2 \sin(q_2) - q_3 \cos(q_2)\dot{u}_1 - 2u_1u_3 \cos(q_2))\hat{b}_x + (-q_3u_1^2 \cos^2(q_2) - q_3u_2^2 + \dot{u}_3)\hat{b}_y + (q_3u_1^2 \sin(q_2) \cos(q_2) + q_3\dot{u}_2 + 2u_2u_3)\hat{b}_z \quad (16.19)$$

Warning: Be careful when making substitutions when expressions contain derivatives and double derivatives. The order in which you make the substitutions matter and the printer that SymPy is using may not show you what you think you are looking at. Take this expression:

```
expr = m*q1.diff(t, 2) + kt*q1.diff(t) + kl*q1
expr
```

$$k_l q_1 + k_t \dot{q}_1 + m \ddot{q}_1 \quad (16.20)$$

Let's say you need to make these substitutions: $q_1 = \frac{q_2}{q_1}$, $\dot{q}_1 = u_1$, $\ddot{q}_1 = \dot{u}_1$. It may seem obvious that the \ddot{q}_1 substitution should be done before q_1 , but care may be needed to help the computer realize this. If the highest derivatives are substituted first with successive calls to `.xreplace()` then you get:

```
expr1 = expr.xreplace({q1.diff(t, 2): u1.diff(t)}).xreplace({q1.diff(t): u1})
expr1
```

$$\frac{k_l q_2}{q_1} + k_t u_1 + m \dot{u}_1 \quad (16.21)$$

But if you substitute in the opposite order you get:

```
expr2 = expr.xreplace({q1: q2/q1}).xreplace({q1.diff(t): u1}).xreplace({q1.diff(t, 2): u1.diff(t)})
expr2
```

$$\frac{k_l q_2}{q_1} + k_t \left(\frac{\dot{q}_2}{q_1} - \frac{q_2 \dot{q}_1}{q_1^2} \right) + m \left(\frac{-\frac{\left(\ddot{q}_1 - \frac{2\dot{q}_1^2}{q_1} \right) q_2}{q_1} + \ddot{q}_2 - \frac{2\dot{q}_1 \dot{q}_2}{q_1}}{q_1} \right) \quad (16.22)$$

which is a very different answer.

Checking the `str()` or `srepr()` versions of the expressions can help diagnose what is going on. The string representation of the first expression is as expected:

```
print(expr1)
```

```
k_l*q2(t)/q1(t) + k_t*u1(t) + m*Derivative(u1(t), t)
```

The string representation of the second expression shows that the q_1 symbol was substituted into each derivative term.

```
print(expr2)
```

```
k_1*q2(t)/q1(t) + k_t*Derivative(q2(t)/q1(t), t) + m*Derivative(q2(t)/q1(t), (t, 2))
```

`expr2` shows different results depending on how you print it! The typeset math evaluates the derivatives and the string representation does not.

If you put all of the substitutions in the same dictionary, SymPy should substitute the terms in the expected order:

```
expr.xreplace({q1: q2/q1, q1.diff(t): u1, q1.diff(t, 2): u1.diff(t)})
```

$$\frac{k_l q_2}{q_1} + k_t u_1 + m \dot{u}_1 \quad (16.23)$$

```
expr.xreplace({q1.diff(t, 2): u1.diff(t), q1.diff(t): u1, q1: q2/q1})
```

$$\frac{k_l q_2}{q_1} + k_t u_1 + m \dot{u}_1 \quad (16.24)$$

Now we formulate the resultant forces and torques on each relevant point and frame:

```
R_Ao = m*g*N.x
R_Bo = m*g*N.x + k1*q3*B.y
R_Q = m/4*g*N.x - k1*q3*B.y
T_A = -kt*q1*N.z + kt*q2*A.x
T_B = -kt*q2*A.x
```

Note the equal and opposite spring forces that act on the pairs of points and pairs of reference frames. We ignored the reaction torque on N from A because N is our inertial reference frame.

The inertia dyadics of the two rods are:

```
I = m*1**2/12
I_A_Ao = I*me.outer(A.y, A.y) + I*me.outer(A.z, A.z)
I_B_Bo = I*me.outer(B.x, B.x) + I*me.outer(B.z, B.z)
```

To form the equations of motion, start by finding all of the partial velocities of the two mass centers A_o, B_o , one particle Q , and two bodies A, B :

```
v_Ao_1 = Ao.vel(N).diff(u1, N)
v_Bo_1 = Bo.vel(N).diff(u1, N)
v_Q_1 = Q.vel(N).diff(u1, N)

v_Ao_2 = Ao.vel(N).diff(u2, N)
v_Bo_2 = Bo.vel(N).diff(u2, N)
v_Q_2 = Q.vel(N).diff(u2, N)

v_Ao_3 = Ao.vel(N).diff(u3, N)
v_Bo_3 = Bo.vel(N).diff(u3, N)
v_Q_3 = Q.vel(N).diff(u3, N)
```

(continues on next page)

(continued from previous page)

```
w_A_1 = A.ang_vel_in(N).diff(u1, N)
w_B_1 = B.ang_vel_in(N).diff(u1, N)

w_A_2 = A.ang_vel_in(N).diff(u2, N)
w_B_2 = B.ang_vel_in(N).diff(u2, N)

w_A_3 = A.ang_vel_in(N).diff(u3, N)
w_B_3 = B.ang_vel_in(N).diff(u3, N)
```

The three generalized active forces are then formed by dotting the partial velocities with the associated load:

```
F1 = v_Ao_1.dot(R_Ao) + v_Bo_1.dot(R_Bo) + v_Q_1.dot(R_Q) + w_A_1.dot(T_A) + w_B_1.
    ↪dot(T_B)
F2 = v_Ao_2.dot(R_Ao) + v_Bo_2.dot(R_Bo) + v_Q_2.dot(R_Q) + w_A_2.dot(T_A) + w_B_2.
    ↪dot(T_B)
F3 = v_Ao_3.dot(R_Ao) + v_Bo_3.dot(R_Bo) + v_Q_3.dot(R_Q) + w_A_3.dot(T_A) + w_B_3.
    ↪dot(T_B)
```

The generalized force vector \bar{F}_r is then:

```
Fr = sm.Matrix([F1, F2, F3])
Fr
```

$$\begin{bmatrix} -\frac{7glm \sin(q_1)}{4} - \frac{g m q_3 \cos(q_1) \cos(q_2)}{4} - k_t q_1 \\ \frac{g m q_3 \sin(q_1) \sin(q_2)}{4} - k_t q_2 \\ -\frac{g m \sin(q_1) \cos(q_2)}{4} - k_l q_3 \end{bmatrix} \quad (16.25)$$

The three generalized inertia forces are similarly formed but with the resultant inertial forces:

```
TAs = -(A.ang_acc_in(N).dot(I_A_Ao) + me.cross(A.ang_vel_in(N), I_A_Ao).dot(A.ang_vel_
    ↪in(N)))
TBS = -(B.ang_acc_in(N).dot(I_B_Bo) + me.cross(B.ang_vel_in(N), I_B_Bo).dot(B.ang_vel_
    ↪in(N)))

F1s = v_Ao_1.dot(-m*Ao.acc(N)) + v_Bo_1.dot(-m*Bo.acc(N)) + v_Q_1.dot(-m/4*Q.acc(N))
F1s += w_A_1.dot(TAs) + w_B_1.dot(TBS)

F2s = v_Ao_2.dot(-m*Ao.acc(N)) + v_Bo_2.dot(-m*Bo.acc(N)) + v_Q_2.dot(-m/4*Q.acc(N))
F2s += w_A_2.dot(TAs) + w_B_2.dot(TBS)

F3s = v_Ao_3.dot(-m*Ao.acc(N)) + v_Bo_3.dot(-m*Bo.acc(N)) + v_Q_3.dot(-m/4*Q.acc(N))
F3s += w_A_3.dot(TAs) + w_B_3.dot(TBS)
```

Finally the generalized inertia force vector is:

```
Frs = sm.Matrix([F1s, F2s, F3s])
Frs
```

$$\begin{bmatrix} -\frac{19l^2mu_1}{12} - \frac{l m q_3 u_1^2 \cos(q_2)}{4} + l \left(-\frac{m(-q_3 u_1^2 \cos^2(q_2) - q_3 u_2^2 + u_3)}{4} \cos(q_2) + \frac{m(q_3 u_1^2 \sin(q_2) \cos(q_2) + q_3 \dot{u}_2 + 2u_2 u_3) \sin(q_2)}{4} \right) + \frac{m(2q_3 u_1 u_2 \sin(q_2) - q_3 u_1^2 \sin^2(q_2) - q_3 u_2^2 \sin(q_2) + u_3 \sin(q_2))}{4} \\ -\frac{l^2 m u_1^2 \sin(q_2) \cos(q_2)}{12} - \frac{l^2 m \dot{u}_2}{12} + \frac{l m q_3 \sin(q_2) \dot{u}_1}{4} - \frac{m(q_3 u_1^2 \sin(q_2))}{4} \\ -\frac{l m \cos(q_2) \dot{u}_1}{4} - \frac{m(-q_3 u_1^2 \cos^2(q_2) - q_3 u_2^2 + u_3)}{4} \end{bmatrix} \quad (16.26)$$

Notice that the dynamical differential equations are only functions of the time varying variables $\dot{\bar{u}}, \bar{u}, \bar{q}$:

```
me.find_dynamicsymbols(Fr)
```

$$\{q_1, q_2, q_3\} \quad (16.27)$$

```
me.find_dynamicsymbols(Frs)
```

$$\{q_2, q_3, u_1, u_2, u_3, \dot{u}_1, \dot{u}_2, \dot{u}_3\} \quad (16.28)$$

16.6 Implicit and Explicit Form

Eq. (16.14) is written in an *implicit form*, meaning that the derivatives are not explicitly solved for. The *explicit form* is found by inverting \mathbf{M}_m :

$$\dot{\bar{x}} = -\mathbf{M}_m^{-1}\bar{g}_m = \bar{f}_m(\bar{x}, t) \quad (16.29)$$

To determine how the state changes over time, these explicit differential equations can be solved by integrating them with respect to time:

$$\bar{x}(t) = \int_{t_0}^{t_f} \bar{f}_m(\bar{x}, t) dt \quad (16.30)$$

\bar{f}_m is, in general, nonlinear in time, thus analytical solutions are impossible to find. To solve this integral we must numerically integrate \bar{f}_m . To do so, it will be useful to extract the symbolic forms of \mathbf{M}_k , \bar{g}_k , \mathbf{M}_d , and \bar{g}_d .

Our example problem has a simple definition of the kinematical differential equations:

$$\begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \quad (16.31)$$

so \mathbf{M}_k is the identity matrix and need not be formed:

$$\mathbf{M}_k \dot{\bar{q}} + \bar{g}_k = 0 \rightarrow - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} + \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad (16.32)$$

But we will need \mathbf{M}_d to solve explicitly for $\dot{\bar{u}}$. Recall that we can use the Jacobian to extract the linear coefficients of $\dot{\bar{u}}$ and then find the terms that aren't functions of $\dot{\bar{u}}$ by substitution (See Sec. [Solving Linear Systems](#)).

Form the column vector $\dot{\bar{u}}$:

```
u = sm.Matrix([u1, u2, u3])
ud = u.diff(t)
ud
```

$$\begin{bmatrix} \dot{u}_1 \\ \dot{u}_2 \\ \dot{u}_3 \end{bmatrix} \quad (16.33)$$

Extract the coefficients of \dot{u} :

```
Md = Frs.jacobian(ud)
Md
```

$$\begin{bmatrix} -\frac{l^2 m \cos^2(q_2)}{12} - \frac{19l^2 m}{12} - \frac{m q_3^2 \cos^2(q_2)}{4} & \frac{l m q_3 \sin(q_2)}{4} & -\frac{l m \cos(q_2)}{4} \\ \frac{l m q_3 \sin(q_2)}{4} & -\frac{l^2 m}{12} - \frac{m q_3^2}{4} & 0 \\ -\frac{l m \cos(q_2)}{4} & 0 & -\frac{m}{4} \end{bmatrix} \quad (16.34)$$

Make a substitution dictionary to set $\dot{u} = \bar{0}$:

```
ud_zerod = {udr: 0 for udr in ud}
ud_zerod
```

$$\{\dot{u}_1 : 0, \dot{u}_2 : 0, \dot{u}_3 : 0\} \quad (16.35)$$

Find \bar{g}_d with $\bar{g}_d = \bar{F}_r^*|_{\dot{u}=\bar{0}} + \bar{F}_r$:

```
gd = Frs.xreplace(ud_zerod) + Fr
gd
```

$$\begin{bmatrix} -\frac{7 g l m \sin(q_1)}{4} - \frac{g m q_3 \cos(q_1) \cos(q_2)}{4} - k_t q_1 + \frac{l^2 m u_1 u_2 \sin(q_2) \cos(q_2)}{6} - \frac{l m q_3 u_1^2 \cos(q_2)}{4} + l \left(-\frac{m(-q_3 u_1^2 \cos^2(q_2) - q_3 u_2^2)}{4} \cos(q_2) + \frac{m(q_3 u_1^2 \sin(q_2) + 2 u_1 u_2 \cos(q_2))}{4} \right) \\ \frac{g m q_3 \sin(q_1) \sin(q_2)}{4} - k_t q_2 - \frac{l^2 m u_1^2 \sin(q_2) \cos(q_2)}{12} - \frac{m(q_3 u_1^2 \sin(q_2) \cos(q_2) + 2 u_1 u_2 \cos(q_2))}{4} \\ -\frac{g m \sin(q_1) \cos(q_2)}{4} - k_l q_3 - \frac{m(-q_3 u_1^2 \cos^2(q_2) - q_3 u_2^2)}{4} \end{bmatrix} \quad (16.36)$$

Check that neither are functions of \dot{u} :

```
me.find_dynamicsymbols(Md)
```

$$\{q_2, q_3\} \quad (16.37)$$

```
me.find_dynamicsymbols(gd)
```

$$\{q_1, q_2, q_3, u_1, u_2, u_3\} \quad (16.38)$$

SIMULATION AND VISUALIZATION

Note: You can download this example as a Python script: `simulation.py` or Jupyter Notebook: `simulation.ipynb`.

```
import sympy as sm
import sympy.physics.mechanics as me
me.init_vprinting(use_latex='mathjax')

class ReferenceFrame(me.ReferenceFrame):
    def __init__(self, *args, **kwargs):
        kwargs.pop('latexs', None)
        lab = args[0].lower()
        tex = r'\hat{{\{}{\}}}_{{\{}{\}}}'
        super(ReferenceFrame, self).__init__(*args,
                                             latexs=(tex.format(lab, 'x'),
                                                      tex.format(lab, 'y'),
                                                      tex.format(lab, 'z')),
                                             **kwargs)
me.ReferenceFrame = ReferenceFrame
```

17.1 Learning Objectives

After completing this chapter readers will be able to:

- evaluate equations of motion numerically
- numerically integrate the ordinary differential equations of a multibody system
- plot the system's state trajectories versus time
- compare integration methods to observe integration error
- create a simple animation of the motion of the multibody system

17.2 Numerical Integration

As mentioned at the end of the prior chapter, we will need to numerically integrate the equations of motion. If they are in explicit form, this integral describes how we can arrive at trajectories in time for the state variables by integrating with respect to time from an initial time t_0 to a final time t_f . Recall that the time derivative of the state \bar{x} is:

$$\dot{\bar{x}}(t) = \bar{f}_m(\bar{x}, t) = -\mathbf{M}_m^{-1}\bar{g}_m \quad (17.1)$$

We can then find \bar{x} by integration with respect to time:

$$\bar{x}(t) = \int_{t_0}^{t_f} \bar{f}_m(\bar{x}, t) dt \quad (17.2)$$

It is possible to form $-\mathbf{M}_m^{-1}\bar{g}_m$ symbolically and it may be suitable or preferable for a given problem, but there are some possible drawbacks. For example, if the degrees of freedom are quite large, the resulting symbolic equations become exponentially more complex. Thus, it is generally best to move from symbolics to numerics before formulating the explicit ordinary differential equations.

17.3 Numerical Evaluation

The `NumPy` library is the de facto base library for numeric computing with Python. NumPy allows us to do `array` programming with Python by providing floating point array data types and vectorized operators to enable repeat operations across arrays of values. In Sec. [Evaluating Symbolic Expressions](#) we introduced the SymPy function `lambdify()`. `lambdify()` will be our way to bridge the symbolic world of SymPy with the numeric world of NumPy.

We will import NumPy like so, by convention:

```
import numpy as np
```

Warning: Beware that mixing SymPy and NumPy data types will rarely, if at all, provide you with functioning code. Be careful because sometimes it may look like the two libraries mix. For example, you can do this:

```
a, b, c, d = sm.symbols('a, b, c, d')
mat = np.array([[a, b], [c, d]])
mat
```



```
array([[a, b],
       [c, d]], dtype=object)
```

which gives a NumPy array containing SymPy symbols. But this will almost certainly cause you problems as you move forward. The process you should always follow for the purposes of this text is:

```
sym_mat = sm.Matrix([[a, b], [c, d]])
eval_sym_mat = sm.lambdify((a, b, c, d), sym_mat)
num_mat = eval_sym_mat(1.0, 2.0, 3.0, 4.0)
num_mat
```



```
array([[1., 2.],
       [3., 4.]])
```

Also, be careful because NumPy and SymPy have many functions that are named the same and you likely don't want to mix them up:

```
np.cos(5) + sm.cos(5)
```

$$0.283662185463226 + \cos(5) \quad (17.3)$$

We import NumPy as `np` and SymPy as `sm` to ensure functions with the same names can coexist.

Returning to the example of the two rods and the sliding mass from the previous chapter, we regenerate the symbolic equations of motion and stop when we have \bar{q} , \bar{u} , \mathbf{M}_k , \bar{g}_k , \mathbf{M}_d , and \bar{g}_d . The following drop down has the SymPy code to generate these symbolic vectors and matrices take from the prior chapter.

Symbolic Setup Code

```

m, g, kt, kl, l = sm.symbols('m, g, k_t, k_l, l')
q1, q2, q3 = me.dynamicsymbols('q1, q2, q3')
u1, u2, u3 = me.dynamicsymbols('u1, u2, u3')

N = me.ReferenceFrame('N')
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')

A.orient_axis(N, q1, N.z)
B.orient_axis(A, q2, A.x)

A.set_ang_vel(N, u1*N.z)
B.set_ang_vel(A, u2*A.x)

O = me.Point('O')
Ao = me.Point('A_O')
Bo = me.Point('B_O')
Q = me.Point('Q')

Ao.set_pos(O, 1/2*A.x)
Bo.set_pos(O, l*A.x)
Q.set_pos(Bo, q3*B.y)

O.set_vel(N, 0)
Ao.v2pt_theory(O, N, A)
Bo.v2pt_theory(O, N, A)
Q.set_vel(B, u3*B.y)
Q.v1pt_theory(Bo, N, B)

t = me.dynamicsymbols._t

qdot_repl = {q1.diff(t): u1,
             q2.diff(t): u2,
             q3.diff(t): u3}

Q.set_acc(N, Q.acc(N).xreplace(qdot_repl))

R_Ao = m*g*N.x
R_Bo = m*g*N.x + kl*q3*B.y
R_Q = m/4*g*N.x - kl*q3*B.y
T_A = -kt*q1*N.z + kt*q2*A.x
T_B = -kt*q2*A.x

I = m*l**2/12

```

(continues on next page)

(continued from previous page)

```

I_A_Ao = I*me.outer(A.y, A.y) + I*me.outer(A.z, A.z)
I_B_Bo = I*me.outer(B.x, B.x) + I*me.outer(B.z, B.z)

points = [Ao, Bo, Q]
forces = [R_Ao, R_Bo, R_Q]
masses = [m, m, m/4]

frames = [A, B]
torques = [T_A, T_B]
inertias = [I_A_Ao, I_B_Bo]

Fr_bar = []
Frs_bar = []

for ur in [u1, u2, u3]:

    Fr = 0
    Frs = 0

    for Pi, Ri, mi in zip(points, forces, masses):
        vr = Pi.vel(N).diff(ur, N)
        Fr += vr.dot(Ri)
        Rs = -mi*Pi.acc(N)
        Frs += vr.dot(Rs)

    for Bi, Ti, Ii in zip(frames, torques, inertias):
        wr = Bi.ang_vel_in(N).diff(ur, N)
        Fr += wr.dot(Ti)
        Ts = -(Bi.ang_acc_in(N).dot(Ii) +
               me.cross(Bi.ang_vel_in(N), Ii).dot(Bi.ang_vel_in(N)))
        Frs += wr.dot(Ts)

    Fr_bar.append(Fr)
    Frs_bar.append(Frs)

Fr = sm.Matrix(Fr_bar)
Frs = sm.Matrix(Frs_bar)

q = sm.Matrix([q1, q2, q3])
u = sm.Matrix([u1, u2, u3])

qd = q.diff(t)
ud = u.diff(t)

ud_zerod = {udr: 0 for udr in ud}

Mk = -sm.eye(3)
gk = u

Md = Frs.jacobian(ud)
gd = Frs.xreplace(ud_zerod) + Fr

```

q, u, qd, ud

$$\left(\begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix}, \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}, \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix}, \begin{bmatrix} \dot{u}_1 \\ \dot{u}_2 \\ \dot{u}_3 \end{bmatrix} \right) \quad (17.4)$$

Mk, gk

$$\left(\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \right) \quad (17.5)$$

Md, gd

$$\left(\begin{bmatrix} -\frac{l^2 m \cos^2(q_2)}{12} - \frac{19l^2 m}{12} - \frac{mq_3^2 \cos^2(q_2)}{4} & \frac{lmq_3 \sin(q_2)}{4} & -\frac{lm \cos(q_2)}{4} \\ \frac{lmq_3 \sin(q_2)}{4} & -\frac{l^2 m}{12} - \frac{mq_3^2}{4} & 0 \\ -\frac{lm \cos(q_2)}{4} & 0 & -\frac{m}{4} \end{bmatrix}, \begin{bmatrix} -\frac{7glm \sin(q_1)}{4} - \frac{gmq_3 \cos(q_1) \cos(q_2)}{4} - k_t q_1 + \frac{l^2 m u_1 u_2 \sin(q_2)}{6} \\ 0 \\ 0 \end{bmatrix} \right), \quad (17.6)$$

Additionally, we will define a column vector \bar{p} that contains all of the constant parameters in the equations of motion. We should know these from our problem definition but they can also be found using `free_symbols()`:

Mk.free_symbols | gk.free_symbols | Md.free_symbols | gd.free_symbols

$$\{g, k_l, k_t, l, m, t\} \quad (17.7)$$

The `|` operator does the union of Python sets, which is the date type that `free_symbols` returns. `t` is not a constant parameter, but the rest are. We can then define the symbolic `p` as:

```
p = sm.Matrix([g, k_l, k_t, l, m])
p
```

$$\begin{bmatrix} g \\ k_l \\ k_t \\ l \\ m \end{bmatrix} \quad (17.8)$$

Now we will create a function to evaluate \mathbf{M}_k , \bar{g}_k , \mathbf{M}_d , and \bar{g}_d . given \bar{q} , \bar{u} and \bar{p} .

```
eval_eom = sm.lambdify((q, u, p), [Mk, gk, Md, gd])
```

To test out the function `eval_eom()` we need some NumPy 1D arrays for \bar{q} , \bar{u} and \bar{p} .

Warning: Make sure to use consistent units when you introduce numbers! I recommend always using force = mass \times acceleration $\rightarrow N = kg \cdot m \cdot s^{-2}$ and torque = inertia \times angular acceleration $\rightarrow N \cdot m = kg \cdot m^2 \cdot rad \cdot s^{-2}$.

The `deg2rad()` and `rad2deg()` are helpful for angle conversions. All SymPy and NumPy trigonometric functions operate on radians, so you'll have to convert if you prefer thinking in degrees. My recommendation is to only use degrees when displaying the outputs, so keep any calls to these two functions at the input and output of your whole computation pipeline.

Here I introduce `q_vals`, `u_vals`, and `p_vals`, each a 1D NumPy array. Make sure to use a different variable name than your symbols so you can distinguish the symbolic and numeric matrices and arrays.

```
q_vals = np.array([
    np.deg2rad(25.0), # q1, rad
    np.deg2rad(5.0), # q2, rad
    0.1, # q3, m
])
q_vals, type(q_vals), q_vals.shape
```

```
(array([0.43633231, 0.08726646, 0.1]), numpy.ndarray, (3,))
```

```
u_vals = np.array([
    0.1, # u1, rad/s
    2.2, # u2, rad/s
    0.3, # u3, m/s
])
u_vals, type(u_vals), u_vals.shape
```

```
(array([0.1, 2.2, 0.3]), numpy.ndarray, (3,))
```

```
p_vals = np.array([
    9.81, # g, m/s**2
    2.0, # kl, N/m
    0.01, # kt, Nm/rad
    0.6, # l, m
    1.0, # m, kg
])
p_vals, type(p_vals), p_vals.shape
```

```
(array([9.81, 2.0, 0.01, 0.6, 1.0]), numpy.ndarray, (5,))
```

Now we can call `eval_eom` with the numeric inputs to get the numerical values of all of the equation of motion matrices and vectors:

```
Mk_vals, gk_vals, Md_vals, gd_vals = eval_eom(q_vals, u_vals, p_vals)
Mk_vals, gk_vals, Md_vals, gd_vals
```

```
(array([[-1, 0, 0],
       [0, -1, 0],
       [0, 0, -1]]),
array([[0.1],
       [2.2],
       [0.3]]),
array([[[-0.60225313, 0.00130734, -0.1494292],
       [0.00130734, -0.0325, 0.0]]])
```

(continues on next page)

(continued from previous page)

```

[-0.1494292 ,  0.          , -0.25      ]]),  

array([[[-4.48963535],  

[-0.02486744],  

[-1.1112791 ]]))

```

Now we can solve for the state derivatives, $\dot{\bar{q}}$ and $\dot{\bar{u}}$, numerically using NumPy's `solve()` function (not the same as SymPy's `solve()`!) for linear systems of equations ($\mathbf{A}\bar{x} = \bar{b}$ type systems).

We first numerically solve the kinematical differential equations for $\dot{\bar{q}}$:

```

qd_vals = np.linalg.solve(-Mk_vals, np.squeeze(gk_vals))  

qd_vals  
  

array([0.1, 2.2, 0.3])

```

In this case, $\dot{\bar{q}} = \bar{u}$ but for nontrivial generalized speed definitions that will not be so. This next linear system solve gives the accelerations $\dot{\bar{u}}$:

```

ud_vals = np.linalg.solve(-Md_vals, np.squeeze(gd_vals))  

ud_vals  
  

array([-7.46056427, -1.06525862,  0.01418834])

```

Note: Note the use of `squeeze()`. This forces `gk_vals` and `gd_vals` to be a 1D array with shape(3,) instead of a 2D array of shape(3, 1). This then causes `qd_vals` and `ud_vals` to be 1D arrays instead of 2D.

```

np.linalg.solve(-Mk_vals, gk_vals)  
  

array([[0.1],  

[2.2],  

[0.3]])

```

17.4 Simulation

To simulate the system forward in time, we solve the [initial value problem](#) of the ordinary differential equations by numerically integrating $\bar{f}_m(t, \bar{x}, \bar{p})$. A simple way to do so, is to use Euler's Method:

$$\bar{x}_{i+1} = \bar{x}_i + \Delta t \bar{f}_m(t_i, \bar{x}_i, \bar{p}) \quad (17.9)$$

Starting with $t_i = t_0$ and some initial values of the states $\bar{x}_i = \bar{x}_0$, the state at Δt in the future is computed. We repeat this until $t_i = t_f$ to find the trajectories of \bar{x} with respect to time.

The following function implements Euler's Method:

```

def euler_integrate(rhs_func, tspan, x0_vals, p_vals, delt=0.03):  

    """Returns state trajectory and corresponding values of time resulting  

    from integrating the ordinary differential equations with Euler's  

    Method.  
  

    Parameters

```

(continues on next page)

(continued from previous page)

```
=====
rhs_func : function
    Python function that evaluates the derivative of the state and takes
    this form ``dxdt = f(t, x, p)``.
tspan : 2-tuple of floats
    The initial time and final time values: (t0, tf).
x0_vals : array_like, shape(2*n,)
    Values of the state x at t0.
p_vals : array_like, shape(o,)
    Values of constant parameters.
delt : float
    Integration time step in seconds/step.

Returns
=====
ts : ndarray(m, )
    Monotonically increasing values of time.
xs : ndarray(m, 2*n)
    State values at each time in ts.

"""
# generate monotonically increasing values of time.
duration = tspan[1] - tspan[0]
num_samples = round(duration/delt) + 1
ts = np.arange(tspan[0], tspan[0] + delt*num_samples, delt)

# create an empty array to hold the state values.
x = np.empty((len(ts), len(x0_vals)))

# set the initial conditions to the first element.
x[0, :] = x0_vals

# use a for loop to sequentially calculate each new x.
for i, ti in enumerate(ts[:-1]):
    x[i + 1, :] = x[i, :] + delt*rhs_func(ti, x[i, :], p_vals)

return ts, x
```

I used `linspace()` to generate equally spaced values between t_0 and t_f . Now we need a Python function that represents $\bar{f}_m(t_i, \bar{x}_i, \bar{p})$. This function evaluates the right hand side of the explicitly ordinary differential equations which calculates the time derivatives of the state.

```
def eval_rhs(t, x, p):
    """Return the right hand side of the explicit ordinary differential
    equations which evaluates the time derivative of the state ``x`` at time
    ``t``.

    Parameters
    ======
    t : float
        Time in seconds.
    x : array_like, shape(6,)
        State at time t: [q1, q2, q3, u1, u2, u3]
    p : array_like, shape(5,)
        Constant parameters: [g, k1, kt, l, m]
```

(continues on next page)

(continued from previous page)

```

Returns
=====
xd : ndarray, shape(6,)
    Derivative of the state with respect to time at time ``t``.

"""

# unpack the q and u vectors from x
q = x[:3]
u = x[3:]

# evaluate the equations of motion matrices with the values of q, u, p
Mk, gk, Md, gd = eval_eom(q, u, p)

# solve for q' and u'
qd = np.linalg.solve(-Mk, np.squeeze(gk))
ud = np.linalg.solve(-Md, np.squeeze(gd))

# pack dq/dt and du/dt into a new state time derivative vector dx/dt
xd = np.empty_like(x)
xd[:3] = qd
xd[3:] = ud

return xd

```

With the function evaluated and numerical values already defined above we can check to see if it works. First combine \bar{q} and \bar{u} into a single column vector of the initial conditions x_0 and pick an arbitrary value for time.

```

x0 = np.empty(6)
x0[:3] = q_vals
x0[3:] = u_vals

t0 = 0.1

```

Now execute the function:

```
eval_rhs(t0, x0, p_vals)
```

```
array([ 0.1        ,  2.2        ,  0.3        , -7.46056427, -1.06525862,
       0.01418834])
```

It seems to work, giving a result for the time derivative of the state vector, matching the results we had above. Now we can try out the `euler_integrate()` function to integration from t_0 to t_f :

```
tf = 2.0
```

```
ts, xs = euler_integrate(eval_rhs, (t0, tf), x0, p_vals)
```

Our `euler_integrate()` function returns the state trajectory and the corresponding time. They look like:

```
ts
```

```
array([0.1 , 0.13, 0.16, 0.19, 0.22, 0.25, 0.28, 0.31, 0.34, 0.37, 0.4 ,
       0.43, 0.46, 0.49, 0.52, 0.55, 0.58, 0.61, 0.64, 0.67, 0.7 , 0.73,
       0.76, 0.79, 0.82, 0.85, 0.88, 0.91, 0.94, 0.97, 1. , 1.03, 1.06,
```

(continues on next page)

(continued from previous page)

```
1.09, 1.12, 1.15, 1.18, 1.21, 1.24, 1.27, 1.3, 1.33, 1.36, 1.39,
1.42, 1.45, 1.48, 1.51, 1.54, 1.57, 1.6, 1.63, 1.66, 1.69, 1.72,
1.75, 1.78, 1.81, 1.84, 1.87, 1.9, 1.93, 1.96, 1.99])
```

```
type(ts), ts.shape
```

```
(numpy.ndarray, (64,))
```

```
xs
```

```
array([[ 4.36332313e-01,  8.72664626e-02,  1.00000000e-01,
       1.00000000e-01,  2.20000000e+00,  3.00000000e-01],
       [ 4.39332313e-01,  1.53266463e-01,  1.09000000e-01,
       -1.23816928e-01,  2.16804224e+00,  3.00425650e-01],
       [ 4.35617805e-01,  2.18307730e-01,  1.18012770e-01,
       -3.48867554e-01,  2.13303503e+00,  2.99423633e-01],
       [ 4.25151779e-01,  2.82298781e-01,  1.26995479e-01,
       -5.72475739e-01,  2.09463526e+00,  2.97361542e-01],
       [ 4.07977506e-01,  3.45137839e-01,  1.35916325e-01,
       -7.91929289e-01,  2.05197871e+00,  2.94628416e-01],
       [ 3.84219628e-01,  4.06697200e-01,  1.44755177e-01,
       -1.00444789e+00,  2.00382451e+00,  2.91554565e-01],
       [ 3.54086191e-01,  4.66811935e-01,  1.53501814e-01,
       -1.20715867e+00,  1.94877129e+00,  2.88335792e-01],
       [ 3.17871431e-01,  5.25275074e-01,  1.62151888e-01,
       -1.39708735e+00,  1.88551903e+00,  2.84974403e-01],
       [ 2.75958810e-01,  5.81840645e-01,  1.70701120e-01,
       -1.57117194e+00,  1.81314156e+00,  2.81247424e-01],
       [ 2.28823652e-01,  6.36234892e-01,  1.79138543e-01,
       -1.72630399e+00,  1.73132870e+00,  2.76708292e-01],
       [ 1.77034532e-01,  6.88174753e-01,  1.87439792e-01,
       -1.85939863e+00,  1.64055711e+00,  2.70722509e-01],
       [ 1.21252574e-01,  7.37391466e-01,  1.95561467e-01,
       -1.96749077e+00,  1.54215738e+00,  2.62531540e-01],
       [ 6.22278505e-02,  7.83656188e-01,  2.03437413e-01,
       -2.04785035e+00,  1.43826241e+00,  2.51334050e-01],
       [ 7.92340069e-04,  8.26804060e-01,  2.10977435e-01,
       -2.09810628e+00,  1.33164459e+00,  2.36370890e-01],
       [-6.21508484e-02,  8.66753398e-01,  2.18068561e-01,
       -2.11636717e+00,  1.22547077e+00,  2.17000797e-01],
       [-1.25641864e-01,  9.03517521e-01,  2.24578585e-01,
       -2.10132625e+00,  1.12301845e+00,  1.92757262e-01],
       [-1.88681651e-01,  9.37208074e-01,  2.30361303e-01,
       -2.05234005e+00,  1.02739894e+00,  1.63382255e-01],
       [-2.50251853e-01,  9.68030042e-01,  2.35262771e-01,
       -1.96947297e+00,  9.41324864e-01,  1.28837671e-01],
       [-3.09336042e-01,  9.96269788e-01,  2.39127901e-01,
       -1.85350427e+00,  8.66944026e-01,  8.92990821e-02],
       [-3.64941170e-01,  1.02227811e+00,  2.41806873e-01,
       -1.70589822e+00,  8.05745012e-01,  4.51378231e-02],
       [-4.16118116e-01,  1.04645046e+00,  2.43161008e-01,
       -1.52874283e+00,  7.58527111e-01, -3.10326579e-03],
       [-4.61980401e-01,  1.06920627e+00,  2.43067910e-01,
       -1.32466575e+00,  7.25419993e-01, -5.47370444e-02],
       [-5.01720374e-01,  1.09096887e+00,  2.41425799e-01],
```

(continues on next page)

(continued from previous page)

-1.09673819e+00,	7.05937072e-01,	-1.08961098e-01],
[-5.34622520e-01,	1.11214698e+00,	2.38156966e-01,
-8.48377776e-01,	6.99048783e-01,	-1.64889059e-01],
[-5.60073853e-01,	1.13311845e+00,	2.33210294e-01,
-5.83259685e-01,	7.03266085e-01,	-2.21585204e-01],
[-5.77571643e-01,	1.15421643e+00,	2.26562738e-01,
-3.05242994e-01,	7.16728855e-01,	-2.78102687e-01],
[-5.86728933e-01,	1.17571830e+00,	2.18219657e-01,
-1.83153063e-02,	7.37297421e-01,	-3.33524242e-01],
[-5.87278392e-01,	1.19783722e+00,	2.08213930e-01,
2.73444426e-01,	7.62647870e-01,	-3.87002450e-01],
[-5.79075060e-01,	1.22071666e+00,	1.96603856e-01,
5.65888142e-01,	7.90372680e-01,	-4.37795193e-01],
[-5.62098415e-01,	1.24442784e+00,	1.83470001e-01,
8.54811324e-01,	8.18087698e-01,	-4.85291412e-01],
[-5.36454076e-01,	1.26897047e+00,	1.68911258e-01,
1.13596271e+00,	8.43544507e-01,	-5.29023016e-01],
[-5.02375194e-01,	1.29427680e+00,	1.53040568e-01,
1.40505807e+00,	8.64744480e-01,	-5.68660838e-01],
[-4.60223452e-01,	1.32021914e+00,	1.35980743e-01,
1.65780471e+00,	8.80047808e-01,	-6.03995480e-01],
[-4.10489311e-01,	1.34662057e+00,	1.17860878e-01,
1.88994245e+00,	8.88268730e-01,	-6.34907053e-01],
[-3.53791037e-01,	1.37326863e+00,	9.88136667e-02,
2.09730514e+00,	8.88747420e-01,	-6.61330213e-01],
[-2.90871883e-01,	1.39993105e+00,	7.89737603e-02,
2.27590461e+00,	8.81389870e-01,	-6.83221797e-01],
[-2.22594745e-01,	1.42637275e+00,	5.84771064e-02,
2.42203585e+00,	8.66668771e-01,	-7.00537408e-01],
[-1.49933669e-01,	1.45237281e+00,	3.74609842e-02,
2.53239896e+00,	8.45580400e-01,	-7.13220652e-01],
[-7.39617005e-02,	1.47774023e+00,	1.60643646e-02,
2.60422940e+00,	8.19554682e-01,	-7.21205288e-01],
[4.16518154e-03,	1.50232687e+00,	-5.57179403e-03,
2.63542555e+00,	7.90319326e-01,	-7.24427319e-01],
[8.32279481e-02,	1.52603645e+00,	-2.73046136e-02,
2.62466048e+00,	7.59725678e-01,	-7.22841890e-01],
[1.61967762e-01,	1.54882822e+00,	-4.89898703e-02,
2.57146487e+00,	7.29553925e-01,	-7.16439450e-01],
[2.39111708e-01,	1.57071483e+00,	-7.04830538e-02,
2.47627011e+00,	7.01325433e-01,	-7.05256667e-01],
[3.13399812e-01,	1.59175460e+00,	-9.16407538e-02,
2.34040397e+00,	6.76154998e-01,	-6.89379799e-01],
[3.83611931e-01,	1.61203925e+00,	-1.12322148e-01,
2.16603688e+00,	6.54671168e-01,	-6.68940608e-01],
[4.48593037e-01,	1.63167938e+00,	-1.32390366e-01,
1.95608296e+00,	6.37018712e-01,	-6.44106831e-01],
[5.07275526e-01,	1.65078994e+00,	-1.51713571e-01,
1.71406620e+00,	6.22938813e-01,	-6.15070314e-01],
[5.58697512e-01,	1.66947811e+00,	-1.70165680e-01,
1.44396681e+00,	6.11906749e-01,	-5.82036029e-01],
[6.02016516e-01,	1.68783531e+00,	-1.87626761e-01,
1.15006549e+00,	6.03298605e-01,	-5.45214730e-01],
[6.36518481e-01,	1.70593427e+00,	-2.03983203e-01,
8.36802201e-01,	5.96558484e-01,	-5.04821104e-01],
[6.61622547e-01,	1.72383102e+00,	-2.19127836e-01,
5.08663225e-01,	5.91343485e-01,	-4.61078217e-01],

(continues on next page)

(continued from previous page)

```
[ 6.76882444e-01,  1.74157133e+00, -2.32960183e-01,
  1.70104973e-01,  5.87631757e-01, -4.14227908e-01],
[ 6.81985593e-01,  1.75920028e+00, -2.45387020e-01,
 -1.74482332e-01,  5.85786780e-01, -3.64545604e-01],
[ 6.76751123e-01,  1.77677388e+00, -2.56323388e-01,
 -5.20773373e-01,  5.86577261e-01, -3.12357072e-01],
[ 6.61127922e-01,  1.79437120e+00, -2.65694100e-01,
 -8.64475606e-01,  5.91156371e-01, -2.58054046e-01],
[ 6.35193654e-01,  1.81210589e+00, -2.73435722e-01,
 -1.20127673e+00,  6.01006553e-01, -2.02105739e-01],
[ 5.99155352e-01,  1.83013609e+00, -2.79498894e-01,
 -1.52678495e+00,  6.17857094e-01, -1.45063860e-01],
[ 5.53351803e-01,  1.84867180e+00, -2.83850810e-01,
 -1.83647709e+00,  6.43581213e-01, -8.75598764e-02],
[ 4.98257491e-01,  1.86797924e+00, -2.86477606e-01,
 -2.12567032e+00,  6.80078039e-01, -3.02944186e-02],
[ 4.34487381e-01,  1.88838158e+00, -2.87386438e-01,
 -2.38953235e+00,  7.29142906e-01, -2.59803906e-02],
[ 3.62801410e-01,  1.91025587e+00, -2.86607027e-01,
 -2.62314224e+00,  7.92327527e-01, -8.04846656e-02],
[ 2.84107143e-01,  1.93402569e+00, -2.84192487e-01,
 -2.82160885e+00,  8.70790639e-01, -1.32439931e-01],
[ 1.99458878e-01,  1.96014941e+00, -2.80219289e-01,
 -2.98024641e+00,  9.65140541e-01, -1.81104004e-01],
[ 1.10051485e-01,  1.98910363e+00, -2.74786169e-01,
 -3.09479777e+00,  1.07527459e+00, -2.25812638e-01]])
```

```
type(xs), xs.shape
```

```
(numpy.ndarray, (64, 6))
```

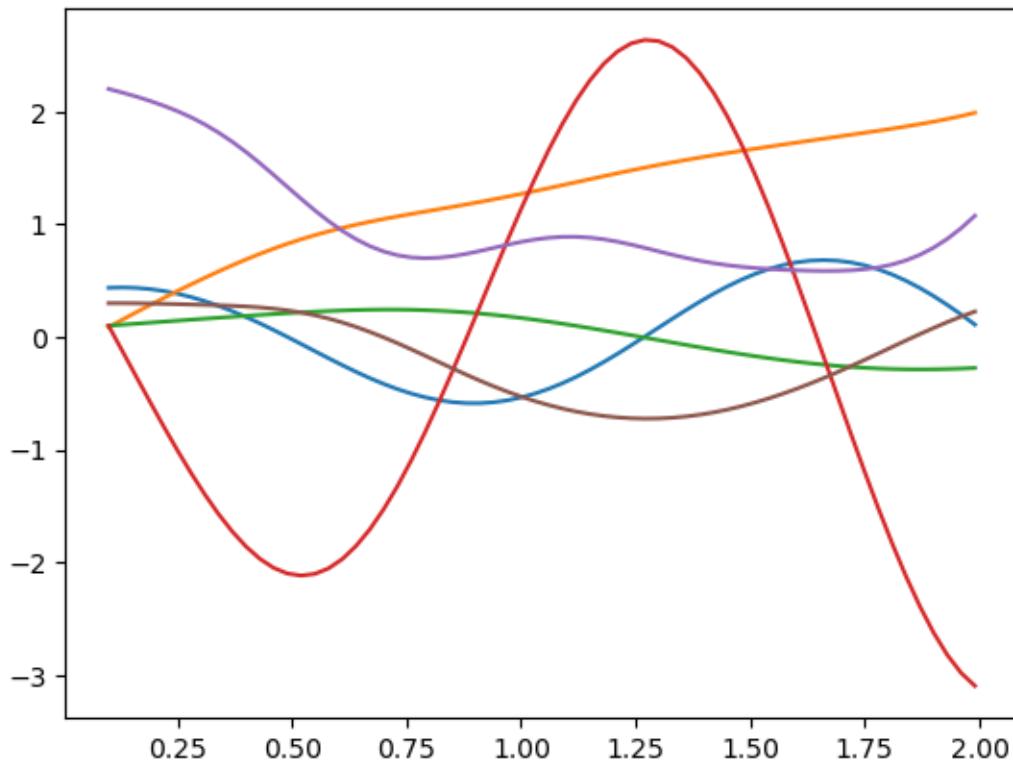
17.5 Plotting Simulation Trajectories

Matplotlib is the most widely used Python library for making plots. Browse [their example gallery](#) to get an idea of the library's capabilities. We will use matplotlib to visualize the state trajectories and animate our system. The convention for importing the main functionality of matplotlib is:

```
import matplotlib.pyplot as plt
```

The `plot()` function offers the simplest way to plot a chart of x values versus y values. I designed the output of `euler_integrate()` to work well with this plotting function. To make a basic plot use:

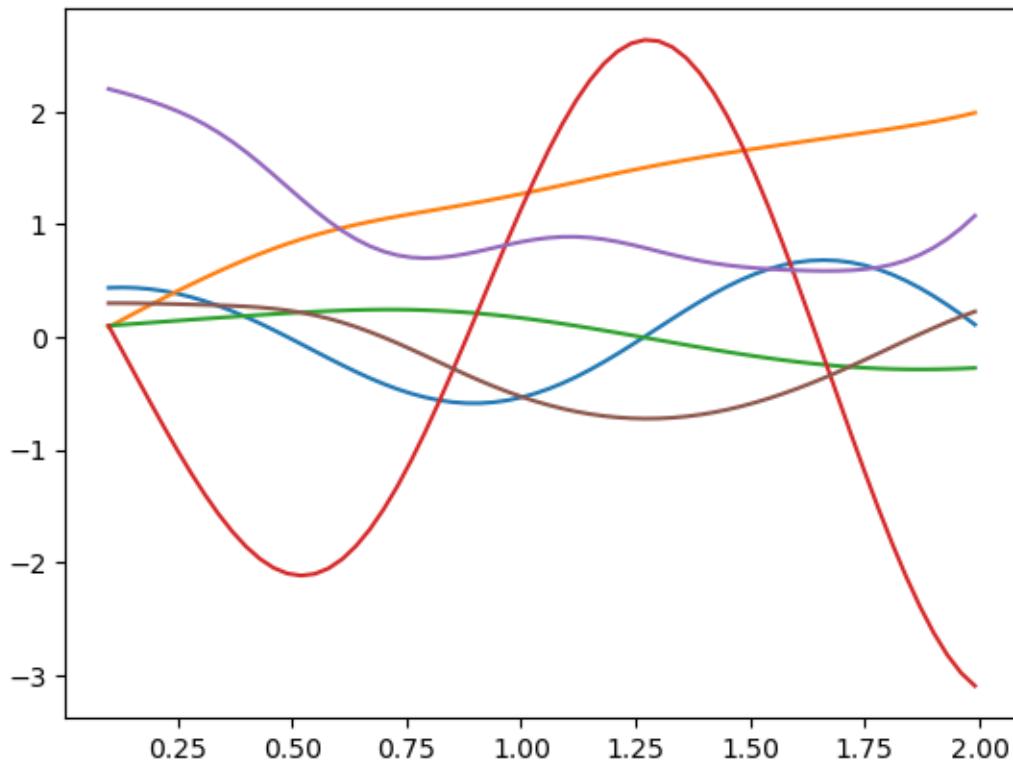
```
plt.plot(ts, xs);
```



Note: The closing semicolon at the end of the statement suppresses the display of the returned objects in Jupyter. See the difference here:

```
plt.plot(ts, xs)
```

```
[<matplotlib.lines.Line2D at 0x7f35d5f31720>,
 <matplotlib.lines.Line2D at 0x7f35d5f31600>,
 <matplotlib.lines.Line2D at 0x7f35d5f31630>,
 <matplotlib.lines.Line2D at 0x7f35d6110310>,
 <matplotlib.lines.Line2D at 0x7f35d6110280>,
 <matplotlib.lines.Line2D at 0x7f35d5fc1900>]
```



This plot shows that the state trajectory changes with respect to time, but without some more information it is hard to interpret. The following function uses `subplots()` to make a figure with panels for the different state variables. I use `vlatex()` to include the symbolic symbol names in the legends. The other matplotlib functions and methods I use are:

- `set_size_inches()`
- `plot()`
- `legend()`
- `set_ylabel()`
- `set_xlabel()`
- `tight_layout()`

I also make use of array slicing notation to select which rows and columns I want from each array. See the NumPy documentation [Indexing on ndarrays](#) for information on how this works.

```
def plot_results(ts, xs):  
    """Returns the array of axes of a 4 panel plot of the state trajectory  
    versus time.  
  
    Parameters  
    ======  
    ts : array_like, shape(m, )  
        Values of time.  
    xs : array_like, shape(m, 6)  
        Values of the state trajectories corresponding to ``ts`` in order  
        [q1, q2, q3, u1, u2, u3].
```

(continues on next page)

(continued from previous page)

```

Returns
=====
axes : ndarray, shape(4,)
    Matplotlib axes for each panel.

"""

fig, axes = plt.subplots(4, 1, sharex=True)
fig.set_size_inches((10.0, 6.0))

axes[0].plot(ts, np.rad2deg(xs[:, :2]))
axes[1].plot(ts, xs[:, 2])
axes[2].plot(ts, np.rad2deg(xs[:, 3:5]))
axes[3].plot(ts, xs[:, 5])

axes[0].legend([me.vlatex(q[0], mode='inline'),
                 me.vlatex(q[1], mode='inline')])
axes[1].legend([me.vlatex(q[2], mode='inline')])
axes[2].legend([me.vlatex(u[0], mode='inline'),
                 me.vlatex(u[1], mode='inline')])
axes[3].legend([me.vlatex(u[2], mode='inline')])

axes[0].set_ylabel('Angle [deg]')
axes[1].set_ylabel('Distance [m]')
axes[2].set_ylabel('Angular Rate [deg/s]')
axes[3].set_ylabel('Speed [m/s]')

axes[3].set_xlabel('Time [s]')

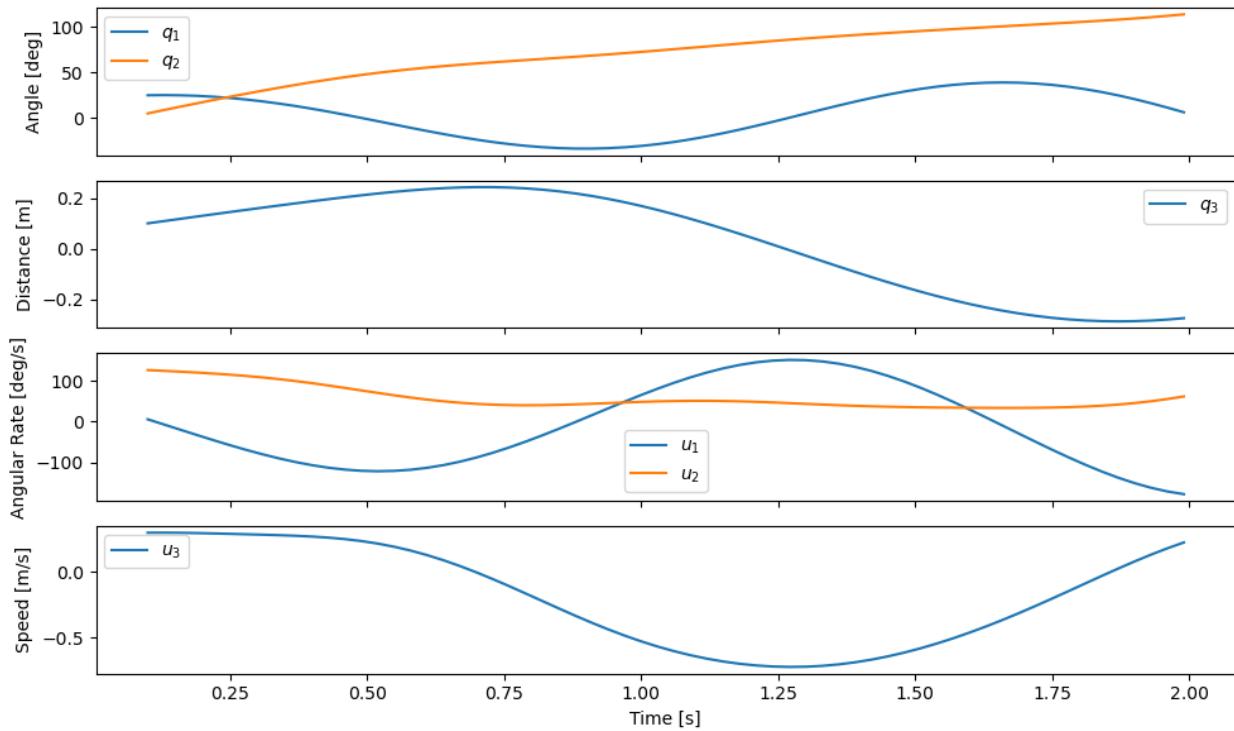
fig.tight_layout()

return axes

```

Our function now gives an interpretable view of the results:

```
plot_results(ts, xs);
```



We now see that q_1 oscillates between $\pm 40\text{deg}$ with a single period. q_2 grows to around $\pm 100\text{deg}$, and q_3 has half an oscillation between -0.2 and 0.2 meters. For the initial conditions and constants we choose, this seems physically feasible.

17.6 Integration with SciPy

Our `euler_integrate()` function seems to do the trick, but all numerical integrators suffer from numerical errors. Careful attention to `truncation error` is needed to keep the error in the resulting trajectories within some acceptable tolerance for your problem's needs. Euler's Method has poor truncation error unless very small time steps are chosen. But more time steps results in longer computation time. There are a large number of other numerical integration methods that provide better results with fewer time steps, but at the cost of more complexity in the integration algorithm.

SciPy is built on top of NumPy and provides a large assortment of battle tested numerical methods for NumPy arrays, including numerical methods for integration. We are solving the initial value problem of ordinary differential equations and SciPy includes the function `solve_ivp()` for this purpose. `solve_ivp()` provides access to a several different integration methods that are suitable for different problems. The default method used is a `Runge-Kutta` method that works well for non-stiff problems.

We will only be using `solve_ivp()` from SciPy so we can import it directly with:

```
from scipy.integrate import solve_ivp
```

We can use `solve_ivp()` in much the same way as our `euler_integrate()` function (in fact I designed `euler_integrate()` to mimic `solve_ivp()`). The difference is that `solve_ivp()` takes a function that evaluates the right hand side of the ordinary differential equations that is of the form $f(t, x)$ (no p !). Our parameter vector p must be passed to the `args=` optional keyword argument in `solve_ivp()` to get things to work. If we only have one extra argument, as we do $f(t, x, p)$, then we must make a 1-tuple `(p_vals,)`. Other than that, the inputs are the same as `euler_integrate()`. `solve_ivp()` returns a solution object that contains quite a bit of information (other than the trajectories). See the documentation for `solve_ivp()` for all the details and more examples.

Here is how we use the integrator with our previously defined system:

```
result = solve_ivp(eval_rhs, (t0, tf), x0, args=(p_vals,))
```

The time values are in the `result.t` attribute:

```
result.t
```

```
array([0.1          , 0.13120248, 0.29762659, 0.54422251, 0.80410512,
       1.06256808, 1.31993114, 1.57487353, 1.83715709, 2.          ])
```

and the state trajectory is in the `result.y` attribute:

```
result.y
```

```
array([[ 0.43633231,   0.4358189 ,   0.31525018,  -0.11015295, -0.46375645,
       -0.34485627,   0.12132567,   0.43601266,   0.27143017, -0.01911091],
       [ 0.08726646,   0.15537688,   0.49605864,   0.88149593,  1.13596246,
       1.39359322,   1.67416762,   1.92225131,   2.13926064,  2.29230085],
       [ 0.1          ,   0.10935888,   0.15799993,   0.21797738,  0.21721945,
       0.11520445,  -0.03955252,  -0.17887644,  -0.25034975, -0.24594355],
       [ 0.1          ,  -0.13284291,  -1.26834506,  -1.88255608, -0.55453818,
       1.39013663,   1.86357587,   0.37219434,  -1.5003001 , -1.93004498],
       [ 2.2          ,   2.16517843,   1.90078084,   1.19931529,  0.90164316,
       1.09025742,   1.04941978,   0.88566504,   0.83634928,  1.07386933],
       [ 0.3          ,   0.299615 ,   0.28218305,   0.16978927, -0.2076191 ,
       -0.54349971,  -0.61398805,  -0.44313122,  -0.08440758,  0.13024152]])
```

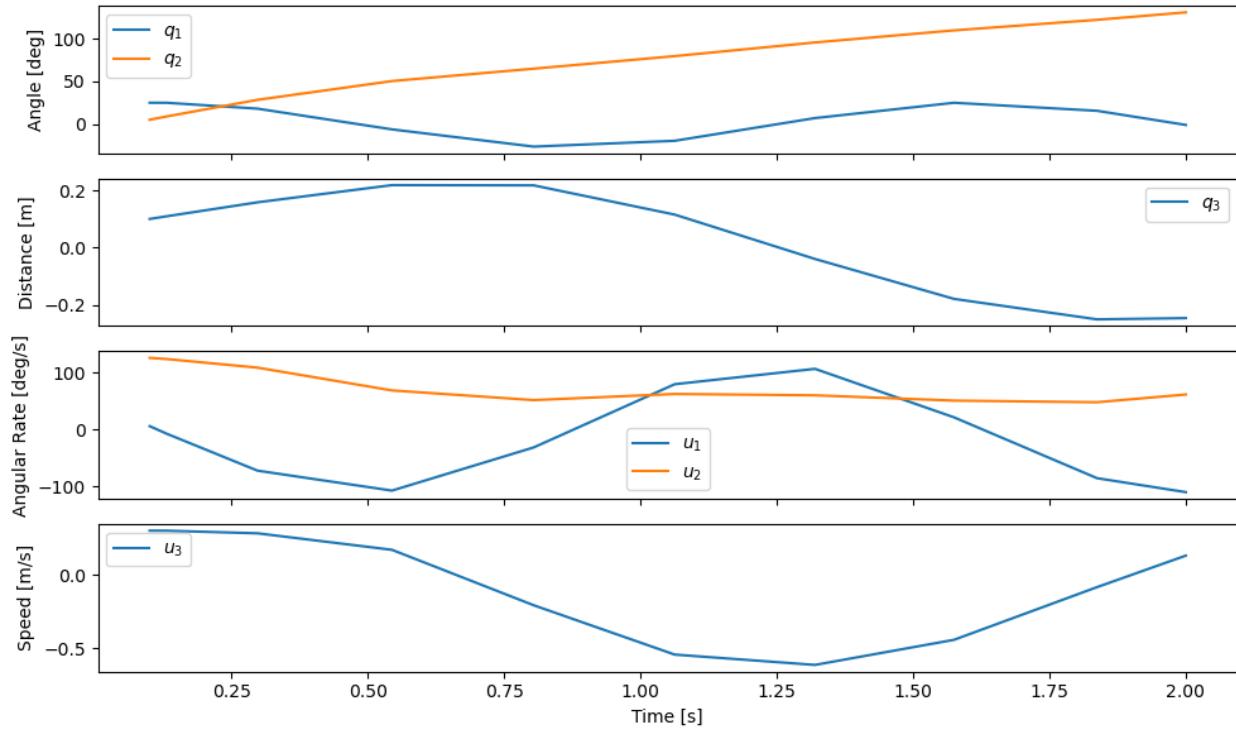
Note the shape of the trajectory array:

```
np.shape(result.y)
```

(6, 10) (17.10)

It is the transpose of our `xs` computed above. Knowing that we can use our `plot_results()` function to view the results. I use `transpose()` to transpose the array before passing it into the plot function.

```
plot_results(result.t, np.transpose(result.y));
```

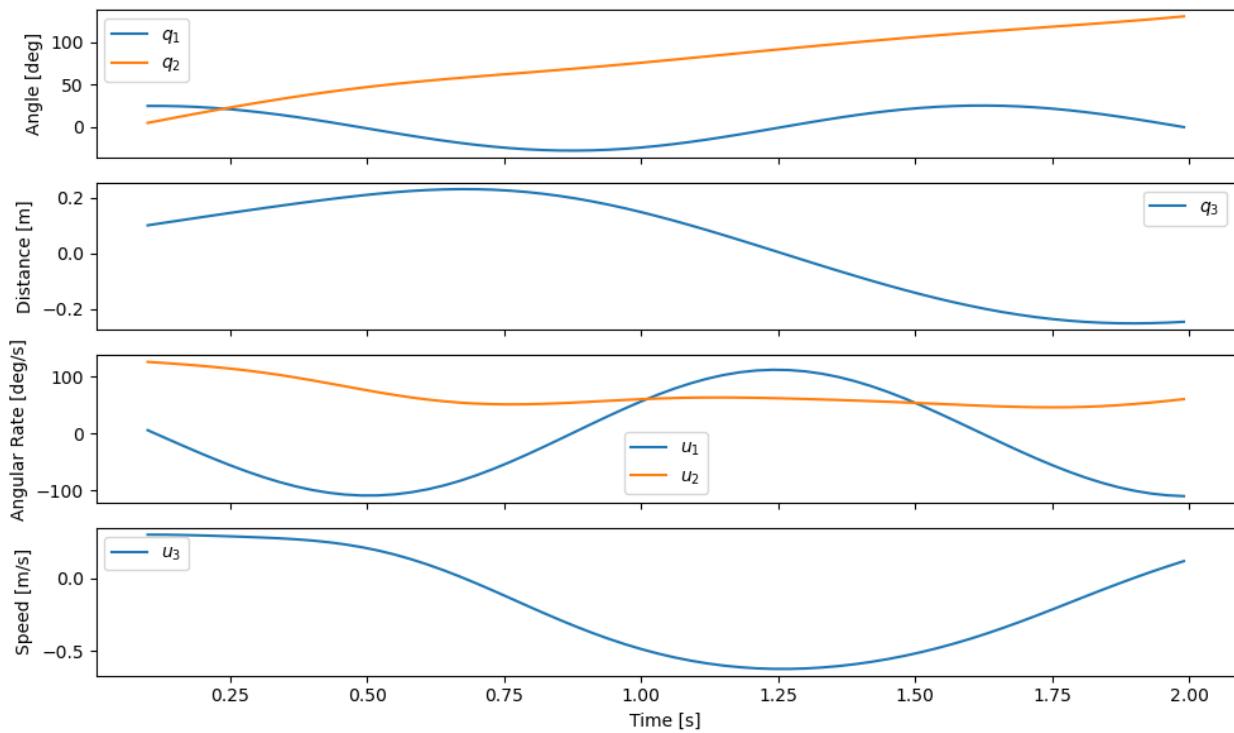


The default result is very coarse in time (only 10 steps!). This is because the underlying integration algorithm adaptively selects the necessary time steps to stay within the desired maximum truncation error. The Runge-Kutta method gives good accuracy with fewer integration steps in this case.

If you want to specify which time values you'd like the result presented at you can do so by interpolating the results by providing the time values with the keyword argument `t_eval`=.

```
result = solve_ivp(eval_rhs, (t0, tf), x0, args=(p_vals,), t_eval=ts)
```

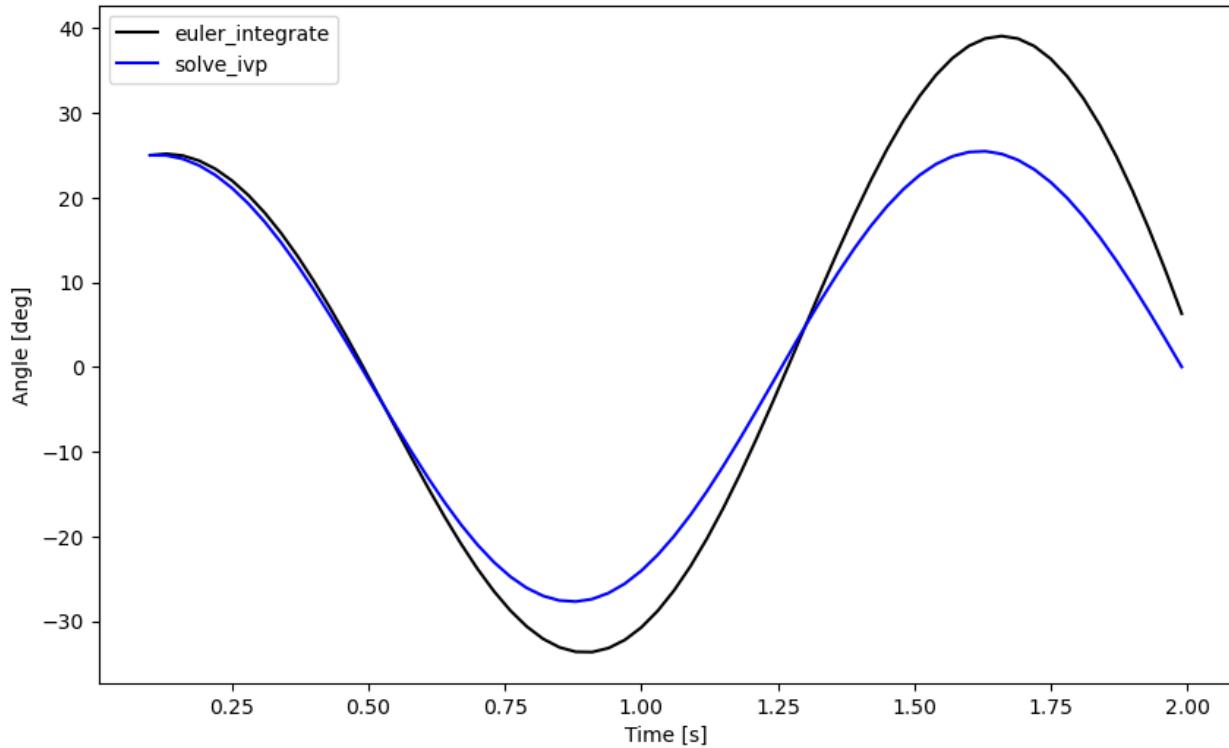
```
plot_results(result.t, np.transpose(result.y));
```



Lastly, let's compare the results from `euler_integrate()` with `solve_ivp()`, the latter of which uses a Runge-Kutta method that has lower truncation error. We'll plot only q_1 for this comparison.

```
fig, ax = plt.subplots()
fig.set_size_inches((10.0, 6.0))

ax.plot(ts, np.rad2deg(xs[:, 0]), 'k',
        result.t, np.rad2deg(np.transpose(result.y)[:, 0]), 'b');
ax.legend(['euler_integrate', 'solve_ivp'])
ax.set_xlabel('Time [s]')
ax.set_ylabel('Angle [deg]');
```



You can clearly see that the Euler Method deviates from the more accurate Runge-Kutta method. You'll need to learn more about truncation error and the various integration methods to ensure you are getting the results you desire. For now, be aware that truncation error and [floating point arithmetic error](#) can give you inaccurate results.

Now set `xs` equal to the `solve_ivp()` result for use in the next section:

```
xs = np.transpose(result.y)
```

17.7 Animation with Matplotlib

Matplotlib also provides tools to make animations by iterating over data and updating the plot. I'll create a very simple set of plots that give 4 views of interesting points in our system.

Matplotlib's plot axes default to displaying the abscissa (x) horizontal and positive towards the right and the ordinate (y) vertical and positive upwards. The coordinate system in Fig. 16.1 has \hat{n}_x positive downwards and \hat{n}_y positive to the right. We can create a viewing reference frame M that matches matplotlib's axes like so:

```
M = me.ReferenceFrame('M')
M.orient_axis(N, sm.pi/2, N.z)
```

Now \hat{n}_x is positive to the right, \hat{n}_y is positive upwards, and \hat{n}_z points out of the screen.

I'll also introduce a couple of points on each end of the rod B , just for visualization purposes:

```
B1 = me.Point('B_1')
Br = me.Point('B_r')
B1.set_pos(Bo, -1/2*B.y)
Br.set_pos(Bo, 1/2*B.y)
```

Now, we can project the four points B_o, Q, B_l, B_r onto the unit vectors of M using `lambdify()` to get the Cartesian coordinates of each point relative to point O . I use `row_join()` to stack the matrices together to build a single matrix with all points' coordinates.

```
coordinates = O.pos_from(O).to_matrix(M)
for point in [Bo, Q, Bl, Br]:
    coordinates = coordinates.row_join(point.pos_from(O).to_matrix(M))

eval_point_coords = sm.lambdify((q, p), coordinates)
eval_point_coords(q_vals, p_vals)
```

```
array([[ 0.          ,  0.25357096,  0.34385686, -0.01728675,  0.52442866],
       [ 0.          , -0.54378467, -0.50168367, -0.67008769, -0.41748165],
       [ 0.          ,  0.          ,  0.00871557, -0.02614672,  0.02614672]])
```

The first row are the x coordinates of each point, the second row has the y coordinates, and the last the z coordinates.

Now create the desired 4 panel figure with three 2D views of the system and one with a 3D view using the initial conditions and constant parameters shown. I make use of `add_subplot()` to control if the panel is 2D or 3D. `set_aspect()` ensures that the abscissa and ordinate dimensions display in a 1:1 ratio.

```
# initial configuration of the points
x, y, z = eval_point_coords(q_vals, p_vals)

# create a figure
fig = plt.figure()
fig.set_size_inches((10.0, 10.0))

# setup the subplots
ax_top = fig.add_subplot(2, 2, 1)
ax_3d = fig.add_subplot(2, 2, 2, projection='3d')
ax_front = fig.add_subplot(2, 2, 3)
ax_right = fig.add_subplot(2, 2, 4)

# common line and marker properties for each panel
line_prop = {
    'color': 'black',
    'marker': 'o',
    'markerfacecolor': 'blue',
    'markersize': 10,
}

# top view
lines_top, = ax_top.plot(x, z, **line_prop)
ax_top.set_xlim((-0.5, 0.5))
ax_top.set_ylim((0.5, -0.5))
ax_top.set_title('Top View')
ax_top.set_xlabel('x')
ax_top.set_ylabel('z')
ax_top.set_aspect('equal')

# 3d view
lines_3d, = ax_3d.plot(x, z, y, **line_prop)
ax_3d.set_xlim((-0.5, 0.5))
ax_3d.set_ylim((0.5, -0.5))
ax_3d.set_zlim((-0.8, 0.2))
ax_3d.set_xlabel('x')
ax_3d.set_ylabel('z')
```

(continues on next page)

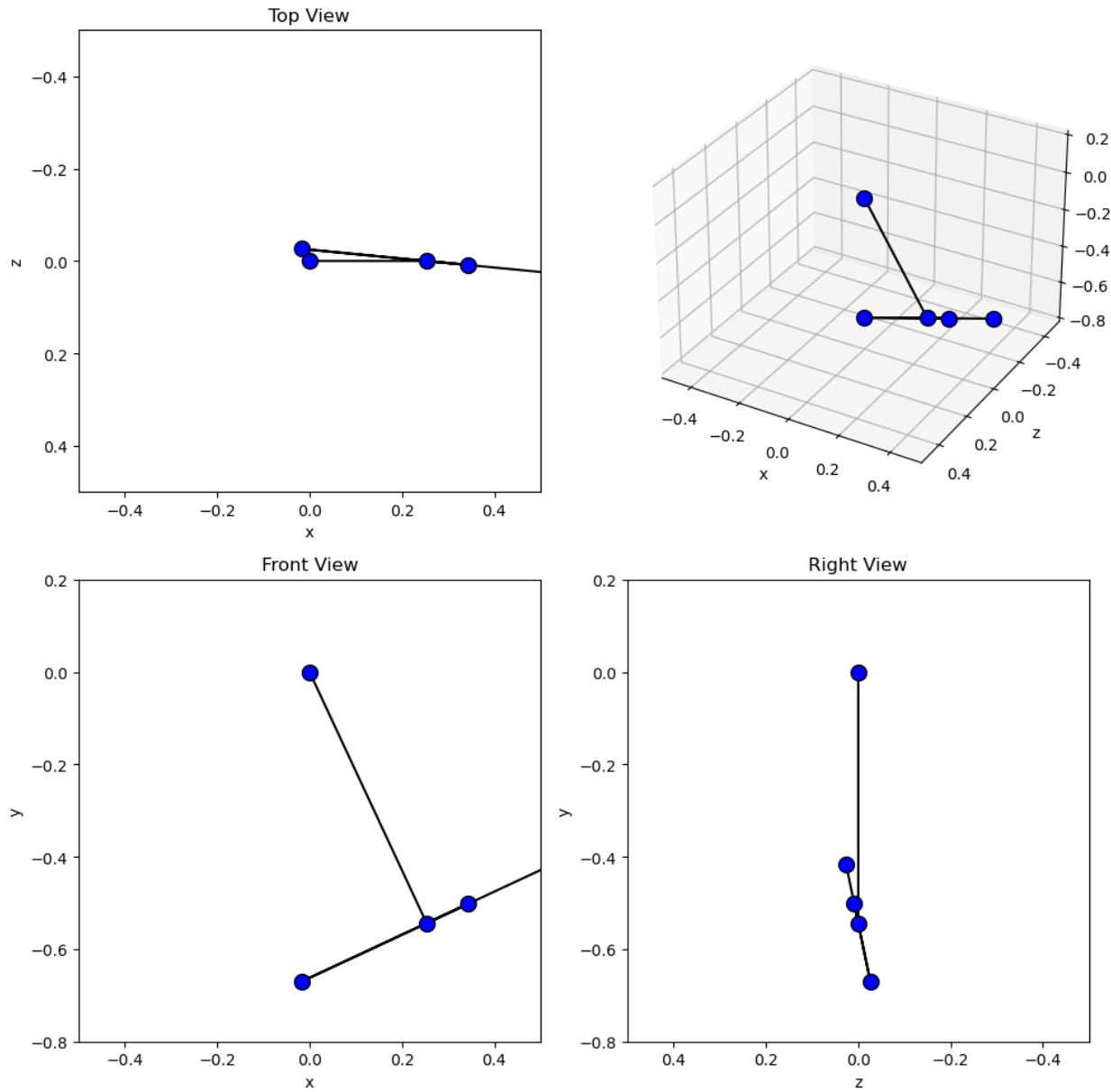
(continued from previous page)

```
ax_3d.set_zlabel('y')

# front view
lines_front, = ax_front.plot(x, y, **line_prop)
ax_front.set_xlim((-0.5, 0.5))
ax_front.set_ylim((-0.8, 0.2))
ax_front.set_title('Front View')
ax_front.set_xlabel('x')
ax_front.set_ylabel('y')
ax_front.set_aspect('equal')

# right view
lines_right, = ax_right.plot(z, y, **line_prop)
ax_right.set_xlim((0.5, -0.5))
ax_right.set_ylim((-0.8, 0.2))
ax_right.set_title('Right View')
ax_right.set_xlabel('z')
ax_right.set_ylabel('y')
ax_right.set_aspect('equal')

fig.tight_layout()
```



Now we will use `FuncAnimation` to generate an animation. See the [animation examples](#) for more information on creating animations with matplotlib.

First import `FuncAnimation()`:

```
from matplotlib.animation import FuncAnimation
```

Now create a function that takes an frame index i , calculates the configuration of the points for the i^{th} state in `xs`, and updates the data for the lines we have already plotted with `set_data()` and `set_data_3d()`.

```
def animate(i):
    x, y, z = eval_point_coords(xs[i, :3], p_vals)
    lines_top.set_data(x, z)
    lines_3d.set_data_3d(x, z, y)
    lines_front.set_data(x, y)
    lines_right.set_data(z, y)
```

Now provide the figure, the animation update function, and the number of frames to `FuncAnimation`:

```
ani = FuncAnimation(fig, animate, len(ts))
```

`FuncAnimation` can create an interactive animation, movie files, and other types of outputs. Here I take advantage of IPython's `HTML` display function and the `to_jshtml()` method to create a web browser friendly visualization of the animation.

```
from IPython.display import HTML
HTML(ani.to_jshtml(fps=30))
```

```
<IPython.core.display.HTML object>
```

If we've setup our animation correctly and our equations of motion are correct, we should see physically believable motion of our system. In this case, it looks like we've successfully simulated and visualized our first multibody system!

THREE DIMENSIONAL VISUALIZATION

Note: You can download this example as a Python script: `visualization.py` or Jupyter Notebook: `visualization.ipynb`.

```
from scipy.integrate import solve_ivp
import numpy as np
import sympy as sm
import sympy.physics.mechanics as me

class ReferenceFrame(me.ReferenceFrame):
    def __init__(self, *args, **kwargs):
        kwargs.pop('latexs', None)
        lab = args[0].lower()
        tex = r'\hat{{}}_{{}}_{{}}'
        super(ReferenceFrame, self).__init__(*args,
                                             latexs=(tex.format(lab, 'x'),
                                                      tex.format(lab, 'y'),
                                                      tex.format(lab, 'z')),
                                             **kwargs)
me.ReferenceFrame = ReferenceFrame
```

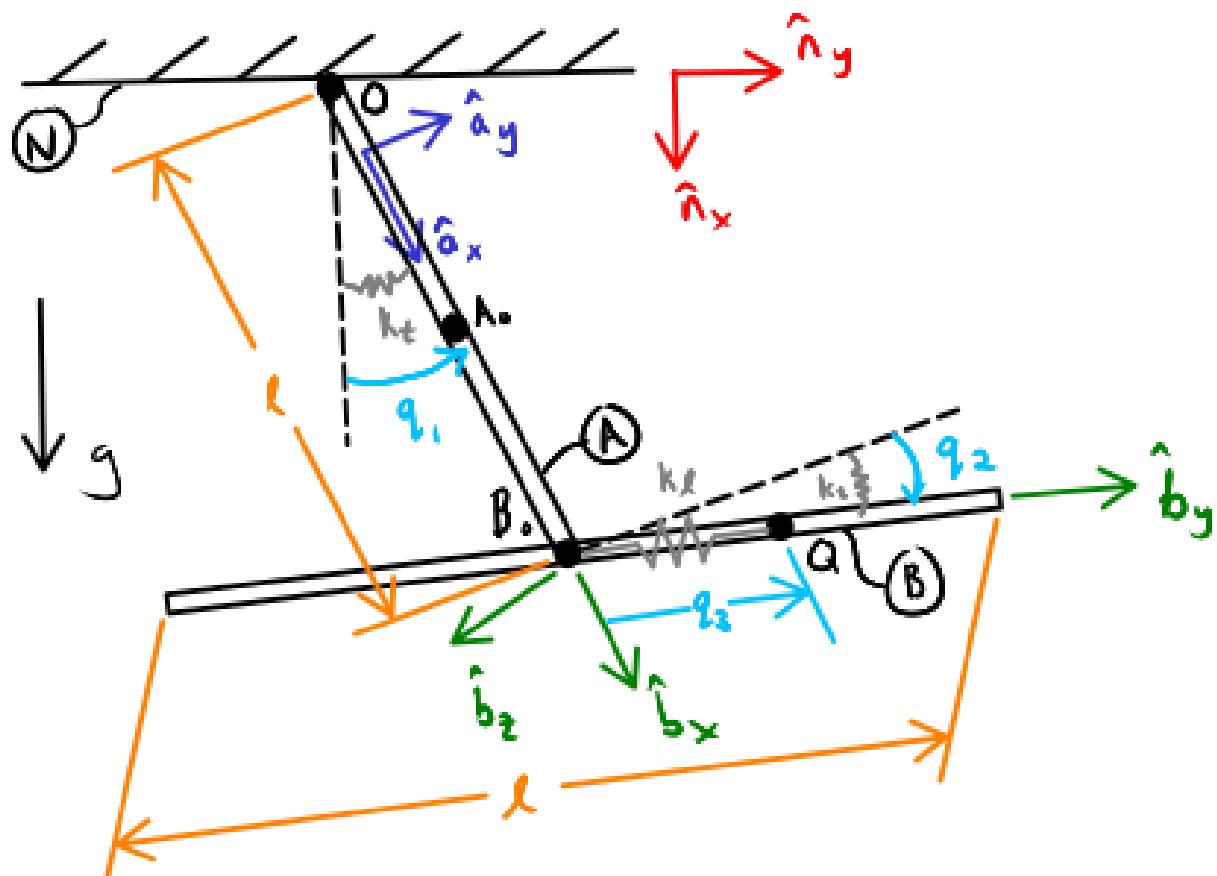
In this chapter, I will give a basic introduction to creating three dimensional graphics to visualize the motion of your multibody system. There are many software tools for generating interactive three dimensional graphics from classic lower level tools like [OpenGL](#) to graphical user interfaces for drawing and animating 3D models like [Blender](#)¹. We will use [pythreejs](#) which is a Python wrapper to the [three.js](#) Javascript library that is built on [WebGL](#) which is a low level graphics library similar to OpenGL but made to execute through your web browser. Check out the [demos](#) on three.js's website to get an idea of how powerful the tool is for 3D visualizations in the web browser.

I will again use the example in [Fig. 16.1](#). Here is the figure for that system:

The following dropdown has all of the code to construct the model and simulate it with the time values `ts` and the state trajectories `xs` as the final output.

Modeling and Simulation Code

¹ Blender was birthed in the Netherlands!



```

m, g, kt, kl, l = sm.symbols('m, g, k_t, k_l, l')
q1, q2, q3 = me.dynamicsymbols('q1, q2, q3')
u1, u2, u3 = me.dynamicsymbols('u1, u2, u3')

N = me.ReferenceFrame('N')
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')

A.orient_axis(N, q1, N.z)
B.orient_axis(A, q2, A.x)

A.set_ang_vel(N, u1*N.z)
B.set_ang_vel(A, u2*A.x)

O = me.Point('O')
Ao = me.Point('A_O')
Bo = me.Point('B_O')
Q = me.Point('Q')

Ao.set_pos(O, l/2*A.x)
Bo.set_pos(O, l*A.x)
Q.set_pos(Bo, q3*B.y)

O.set_vel(N, 0)
Ao.v2pt_theory(O, N, A)
Bo.v2pt_theory(O, N, A)
Q.set_vel(B, u3*B.y)
Q.v1pt_theory(Bo, N, B)

t = me.dynamicsymbols._t

qdot_repl = {q1.diff(t): u1,
              q2.diff(t): u2,
              q3.diff(t): u3}

Q.set_acc(N, Q.acc(N).xreplace(qdot_repl))

R_Ao = m*g*N.x
R_Bo = m*g*N.x + kl*q3*B.y
R_Q = m/4*g*N.x - kl*q3*B.y
T_A = -kt*q1*N.z + kt*q2*A.x
T_B = -kt*q2*A.x

I = m*l**2/12
I_A_Ao = I*me.outer(A.y, A.y) + I*me.outer(A.z, A.z)
I_B_Bo = I*me.outer(B.x, B.x) + I*me.outer(B.z, B.z)

points = [Ao, Bo, Q]
forces = [R_Ao, R_Bo, R_Q]
masses = [m, m, m/4]

frames = [A, B]
torques = [T_A, T_B]
inertias = [I_A_Ao, I_B_Bo]

Fr_bar = []
Frs_bar = []

```

(continues on next page)

(continued from previous page)

```

for ur in [u1, u2, u3]:
    Fr = 0
    Frs = 0

    for Pi, Ri, mi in zip(points, forces, masses):
        vr = Pi.vel(N).diff(ur, N)
        Fr += vr.dot(Ri)
        Rs = -mi*Pi.acc(N)
        Frs += vr.dot(Rs)

    for Bi, Ti, Ii in zip(frames, torques, inertias):
        wr = Bi.ang_vel_in(N).diff(ur, N)
        Fr += wr.dot(Ti)
        Ts = -(Bi.ang_acc_in(N).dot(Ii) +
               me.cross(Bi.ang_vel_in(N), Ii).dot(Bi.ang_vel_in(N)))
        Frs += wr.dot(Ts)

    Fr_bar.append(Fr)
    Frs_bar.append(Frs)

Fr = sm.Matrix(Fr_bar)
Frs = sm.Matrix(Frs_bar)

q = sm.Matrix([q1, q2, q3])
u = sm.Matrix([u1, u2, u3])
p = sm.Matrix([g, kl, kt, l, m])

qd = q.diff(t)
ud = u.diff(t)

ud_zerod = {udr: 0 for udr in ud}

Mk = -sm.eye(3)
gk = u

Md = Frs.jacobian(ud)
gd = Frs.xreplace(ud_zerod) + Fr

eval_eom = sm.lambdify((q, u, p), [Mk, gk, Md, gd])

def eval_rhs(t, x, p):
    """Return the right hand side of the explicit ordinary differential
    equations which evaluates the time derivative of the state ``x`` at time
    ``t``.

    Parameters
    ==========
    t : float
        Time in seconds.
    x : array_like, shape(6,)
        State at time t: [q1, q2, q3, u1, u2, u3]
    p : array_like, shape(5,)
        Constant parameters: [g, kl, kt, l, m]

    Returns
    """

```

(continues on next page)

(continued from previous page)

```

=====
xd : ndarray, shape(6,)
    Derivative of the state with respect to time at time ``t``.

"""

# unpack the q and u vectors from x
q = x[:3]
u = x[3:]

# evaluate the equations of motion matrices with the values of q, u, p
Mk, gk, Md, gd = eval_eom(q, u, p)

# solve for q' and u'
qd = np.linalg.solve(-Mk, np.squeeze(gk))
ud = np.linalg.solve(-Md, np.squeeze(gd))

# pack dq/dt and du/dt into a new state time derivative vector dx/dt
xd = np.empty_like(x)
xd[:3] = qd
xd[3:] = ud

return xd

q_vals = np.array([
    np.deg2rad(25.0),  # q1, rad
    np.deg2rad(5.0),  # q2, rad
    0.1,  # q3, m
])

u_vals = np.array([
    0.1,  # u1, rad/s
    2.2,  # u2, rad/s
    0.3,  # u3, m/s
])

p_vals = np.array([
    9.81,  # g, m/s**2
    3.0,  # kl, N/m
    0.01,  # kt, Nm/rad
    0.6,  # l, m
    1.0,  # m, kg
])

x0 = np.empty(6)
x0[:3] = q_vals
x0[3:] = u_vals

fps = 20
t0, tf = 0.0, 10.0
ts = np.linspace(t0, tf, num=int(fps*(tf - t0)))
result = solve_ivp(eval_rhs, (t0, tf), x0, args=(p_vals,), t_eval=ts)
xs = result.y.T

```

```
ts.shape, xs.shape
```

```
((200,), (200, 6))
```

18.1 pythreejs

pythreejs allows you to use three.js via Python. The functions and objects that pythreejs makes available are found in its documentation, but since these have a 1:1 mapping to the three.js code, you'll also find more comprehensive information in the ThreeJS documentation. We will import pythreejs like so:

```
import pythreejs as p3js
```

pythreejs has many primitive geometric shapes, for example `CylinderGeometry` can be used to create cylinders and cones:

```
cyl_geom = p3js.CylinderGeometry(radiusTop=2.0, radiusBottom=10.0, height=50.0)  
cyl_geom
```

```
CylinderGeometry(height=50.0, radiusBottom=10.0, radiusTop=2.0)
```

The image above is interactive; you can use your mouse or trackpad to click, hold, and move the object.

If you want to apply a material to the surface of the geometry you create a `Mesh` which associates a `Material` with the geometry. For example, you can color the above cylinder like so:

```
red_material = p3js.MeshStandardMaterial(color='red')  
  
cyl_mesh = p3js.Mesh(geometry=cyl_geom, material=red_material)  
  
cyl_mesh
```

```
Mesh(geometry=CylinderGeometry(height=50.0, radiusBottom=10.0, radiusTop=2.0),  
  material=MeshStandardMaterial(a...)
```

18.2 Creating a Scene

Here I create a new orange cylinder that is displaced from the origin of the scene and that has its own coordinate axes. `AxesHelper` creates simple X (red), Y (green), and Z (blue) affixed to the mesh. `position` is overridden to set the position.

```
cyl_geom = p3js.CylinderGeometry(radiusTop=0.1, radiusBottom=0.5, height=2.0)  
cyl_material = p3js.MeshStandardMaterial(color='orange', wireframe=True)  
cyl_mesh = p3js.Mesh(geometry=cyl_geom, material=cyl_material)  
axes = p3js.AxesHelper()  
cyl_mesh.add(axes)  
cyl_mesh.position = (3.0, 3.0, 3.0)
```

Now we will create a `Scene` which can contain multiple meshes and other objects like lights, cameras, and axes. There is a fair amount of boiler plate code for creating the static scene. All of the objects should be added to the `children`= keyword argument of `Scene`. The last line creates a `WebGLBufferRenderer` that links the camera view to the scene and enables `OrbitControls` to allow zooming, panning, and rotating with a mouse or trackpad.

```

view_width = 600
view_height = 400

camera = p3js.PerspectiveCamera(position=[10.0, 10.0, 10.0],
                                  aspect=view_width/view_height)
dir_light = p3js.DirectionalLight(position=[0.0, 10.0, 10.0])
ambient_light = p3js.AmbientLight()

axes = p3js.AxesHelper()
scene = p3js.Scene(children=[cyl_mesh, axes, camera, dir_light, ambient_light])
controller = p3js.OrbitControls(controlling=camera)
renderer = p3js.Renderer(camera=camera,
                        scene=scene,
                        controls=[controller],
                        width=view_width,
                        height=view_height)

```

Now display the scene by calling the renderer:

```
renderer
```

```
Renderer(camera=PerspectiveCamera(aspect=1.5, position=(10.0, 10.0, 10.0),  
projectionMatrix=(1.0, 0.0, 0.0, 0....
```

18.3 Transformation Matrices

The location and orientation of any given mesh is stored in its [transformation matrix](#). A transformation matrix is commonly used in graphics applications because it can describe the position, orientation, scaling, and skewing of a mesh of points. A transformation matrix that only describes rotation and position takes this form:

$$\mathbf{T} = \begin{bmatrix} {}^N\mathbf{C}^B & \bar{0} \\ \bar{r}^{P/O} & 1 \end{bmatrix} \quad \mathbf{T} \in \mathbb{R}^{4 \times 4} \quad (18.1)$$

Here the direction cosine matrix of a mesh B with respect to the scene's global reference frame N is stored in the first three rows and columns, the position vector to a reference point P fixed in the mesh relative to the scene's origin point O is stored in the first three columns of the bottom row. If there is no rotation or translation, the transformation matrix becomes the identity matrix. This matrix is stored in the `matrix` attribute of the mesh:

```
cyl_mesh.matrix
```

```
(1.0,
 0.0,
 0.0,
 0.0,
 0.0,
 1.0,
 0.0,
 0.0,
 0.0,
 0.0,
 1.0,
 0.0,
 0.0,
 0.0,
```

(continues on next page)

(continued from previous page)

```
0.0,
1.0)
```

Notice that the 4x4 matrix is stored “flattened” in a single list of 16 values.

```
len(cyl_mesh.matrix)
```

```
16
```

If you change this list to a NumPy array you can `reshape()` it and `flatten()` it to see the connection.

```
np.array(cyl_mesh.matrix).reshape(4, 4)
```

```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

```
np.array(cyl_mesh.matrix).reshape(4, 4).flatten()
```

```
array([1., 0., 0., 0., 0., 1., 0., 0., 0., 0., 1., 0., 0., 0., 0., 1.])
```

Each mesh/geometry has its own local coordinate system and origin. For the cylinder, the origin is at the geometric center and the axis of the cylinder is aligned with its local Y axis. For our body A , we need the cylinder’s axis to align with our \hat{a}_x vector. To solve this, we need to create a new reference frame in which its Y unit vector is aligned with the \hat{a}_x . I introduce reference frame A_c for this purpose:

```
Ac = me.ReferenceFrame('Ac')
Ac.orient_axis(A, sm.pi/2, A.z)
```

Now we can create a transformation matrix for A_c and A_o . A_o aligns with the cylinder mesh’s origin and A_c aligns with its coordinate system.

```
TA = sm.eye(4)
TA[:3, :3] = Ac.dcm(N)
TA[3, :3] = sm.transpose(Ao.pos_from(O).to_matrix(N))
TA
```

$$\begin{bmatrix} -\sin(q_1(t)) & \cos(q_1(t)) & 0 & 0 \\ -\cos(q_1(t)) & -\sin(q_1(t)) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{l \cos(q_1(t))}{2} & \frac{l \sin(q_1(t))}{2} & 0 & 1 \end{bmatrix} \quad (18.2)$$

The B rod is already correctly aligned with the cylinder geometry’s local coordinate system so we do not need to introduce a new reference frame for its transformation matrix.

```
TB = sm.eye(4)
TB[:3, :3] = B.dcm(N)
TB[3, :3] = sm.transpose(Bo.pos_from(O).to_matrix(N))
TB
```

$$\begin{bmatrix} \cos(q_1(t)) & \sin(q_1(t)) & 0 & 0 \\ -\sin(q_1(t))\cos(q_2(t)) & \cos(q_1(t))\cos(q_2(t)) & \sin(q_2(t)) & 0 \\ \sin(q_1(t))\sin(q_2(t)) & -\sin(q_2(t))\cos(q_1(t)) & \cos(q_2(t)) & 0 \\ l\cos(q_1(t)) & l\sin(q_1(t)) & 0 & 1 \end{bmatrix} \quad (18.3)$$

Lastly, we will introduce a sphere mesh to show the location of point Q . We can choose any reference frame because a sphere looks the same from all directions, but I choose to use the B frame here since we describe the point as sliding along the rod B . This choice will play a role in making the local coordinate axes visualize a bit better in the final animations.

```
TQ = sm.eye(4)
TQ[:3, :3] = B.dcm(N)
TQ[3, :3] = sm.transpose(Q.pos_from(O).to_matrix(N))
TQ
```

$$\begin{bmatrix} \cos(q_1(t)) & \sin(q_1(t)) & 0 & 0 \\ -\sin(q_1(t))\cos(q_2(t)) & \cos(q_1(t))\cos(q_2(t)) & \sin(q_2(t)) & 0 \\ \sin(q_1(t))\sin(q_2(t)) & -\sin(q_2(t))\cos(q_1(t)) & \cos(q_2(t)) & 0 \\ l\cos(q_1(t)) - q_3(t)\sin(q_1(t))\cos(q_2(t)) & l\sin(q_1(t)) + q_3(t)\cos(q_1(t))\cos(q_2(t)) & q_3(t)\sin(q_2(t)) & 1 \end{bmatrix} \quad (18.4)$$

Now that we have symbolic transformation matrices, let's flatten them all to be in the form that three.js needs:

```
TA = TA.reshape(16, 1)
TB = TB.reshape(16, 1)
TQ = TQ.reshape(16, 1)
```

```
TA
```

$$\begin{bmatrix} -\sin(q_1(t)) \\ \cos(q_1(t)) \\ 0 \\ 0 \\ -\cos(q_1(t)) \\ -\sin(q_1(t)) \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ l\cos(q_1(t)) \\ \frac{l\sin(q_1(t))}{2} \\ \frac{l\sin(q_1(t))}{2} \\ 0 \\ 1 \end{bmatrix} \quad (18.5)$$

Now create a function to numerically evaluate the transformation matrices given the generalized coordinates and constants of the system:

```
eval_transform = sm.lambdify((q, p), (TA, TB, TQ))
eval_transform(q_vals, p_vals)
```

```
(array([[ -0.42261826],
       [ 0.90630779],
       [ 0.          ],
       [ 0.          ],
       [-0.90630779],
       [-0.42261826],
       [ 0.          ],
       [ 0.          ],
       [ 0.          ],
       [ 1.          ],
       [ 0.          ],
       [ 0.27189234],
       [ 0.12678548],
       [ 0.          ],
       [ 1.          ]]),
 array([[ 0.90630779],
       [ 0.42261826],
       [ 0.          ],
       [ 0.          ],
       [-0.42101007],
       [ 0.90285901],
       [ 0.08715574],
       [ 0.          ],
       [ 0.03683361],
       [-0.07898993],
       [ 0.9961947 ],
       [ 0.          ],
       [ 0.54378467],
       [ 0.25357096],
       [ 0.          ],
       [ 1.          ]]),
 array([[ 0.90630779],
       [ 0.42261826],
       [ 0.          ],
       [ 0.          ],
       [-0.42101007],
       [ 0.90285901],
       [ 0.08715574],
       [ 0.          ],
       [ 0.03683361],
       [-0.07898993],
       [ 0.9961947 ],
       [ 0.          ],
       [ 0.50168367],
       [ 0.34385686],
       [ 0.00871557],
       [ 1.          ]]))
```

Finally, create a list of lists for the transformation matrices at each time in t_s , as this is the form needed for the animation data below:

```
TAs = []
TBS = []
TQS = []

for xi in xs:
```

(continues on next page)

(continued from previous page)

```
TAi, TBi, TQi = eval_transform(xi[:3], p_vals)
TAs.append(TAi.squeeze().tolist())
TBs.append(TBi.squeeze().tolist())
TQs.append(TQi.squeeze().tolist())
```

Here are the first two numerical transformation matrices to see what we have created:

```
TAs[:2]
```

```
[[ -0.42261826174069944,
  0.9063077870366499,
  0.0,
  0.0,
 -0.9063077870366499,
 -0.42261826174069944,
  0.0,
  0.0,
  0.0,
  0.0,
  1.0,
  0.0,
  0.27189233611099495,
  0.12678547852220984,
  0.0,
  1.0],
 [-0.4187739215332694,
  0.9080905255775148,
  0.0,
  0.0,
 -0.9080905255775148,
 -0.4187739215332694,
  0.0,
  0.0,
  0.0,
  0.0,
  1.0,
  0.0,
  0.2724271576732544,
  0.12563217645998082,
  0.0,
  1.0]]
```

18.4 Geometry and Mesh Definitions

Create two cylinders for rods *A* and *B* and a sphere for particle *Q*:

```
rod_radius = p_vals[3]/20  # 1/20
sphere_radius = p_vals[3]/16  # 1/16

geom_A = p3js.CylinderGeometry(
    radiusTop=rod_radius,
    radiusBottom=rod_radius,
    height=p_vals[3],  # 1
)
```

(continues on next page)

(continued from previous page)

```
geom_B = p3js.CylinderGeometry(
    radiusTop=rod_radius,
    radiusBottom=rod_radius,
    height=p_vals[3],  # 1
)

geom_Q = p3js.SphereGeometry(radius=sphere_radius)
```

Now create meshes for each body and add a material of a different color for each mesh. Each mesh will need a unique name so that we can associate the animation information with the correct object. After the creation of the mesh set `matrixAutoUpdate` to `false` so that we can manually specify the transformation matrix during the animation. Lastly, add local coordinate axes to each mesh and set the transformation matrix to the initial configuration.

```
arrow_length = 0.2

mesh_A = p3js.Mesh(
    geometry=geom_A,
    material=p3js.MeshStandardMaterial(color='red'),
    name='mesh_A',
)
mesh_A.matrixAutoUpdate = False
mesh_A.add(p3js.AxesHelper(arrow_length))
mesh_A.matrix = TAs[0]

mesh_B = p3js.Mesh(
    geometry=geom_B,
    material=p3js.MeshStandardMaterial(color='blue'),
    name='mesh_B',
)
mesh_B.matrixAutoUpdate = False
mesh_B.add(p3js.AxesHelper(arrow_length))
mesh_B.matrix = TBs[0]

mesh_Q = p3js.Mesh(
    geometry=geom_Q,
    material=p3js.MeshStandardMaterial(color='green'),
    name='mesh_Q',
)
mesh_Q.matrixAutoUpdate = False
mesh_Q.add(p3js.AxesHelper(arrow_length))
mesh_Q.matrix = TQs[0]
```

18.5 Scene Setup

Now create a scene and renderer similar to as we did earlier. Include the camera, lighting, coordinate axes, and all of the meshes.

```
view_width = 600
view_height = 400

camera = p3js.PerspectiveCamera(position=[1.5, 0.6, 1],
                                 up=[-1.0, 0.0, 0.0],
                                 aspect=view_width/view_height)
```

(continues on next page)

(continued from previous page)

```

key_light = p3js.DirectionalLight(position=[0, 10, 10])
ambient_light = p3js.AmbientLight()

axes = p3js.AxesHelper()

children = [mesh_A, mesh_B, mesh_Q, axes, camera, key_light, ambient_light]

scene = p3js.Scene(children=children)

controller = p3js.OrbitControls(controlling=camera)
renderer = p3js.Renderer(camera=camera, scene=scene, controls=[controller],
    width=view_width, height=view_height)

```

18.6 Animation Setup

three.js uses the concept of a “track” to track the data that changes over time for an animation. A `VectorKeyframeTrack` can be used to associate time varying transformation matrices with a specific mesh. Create a track for each mesh. Make sure that the `name` keyword argument matches the name of the mesh with this syntax: `scene/<mesh name>.matrix`. The `times` and `values` keyword arguments hold the simulation time values and the list of transformation matrices at each time, respectively.

```

track_A = p3js.VectorKeyframeTrack(
    name="scene/mesh_A.matrix",
    times=ts,
    values=TAs
)

track_B = p3js.VectorKeyframeTrack(
    name="scene/mesh_B.matrix",
    times=ts,
    values=TBs
)

track_Q = p3js.VectorKeyframeTrack(
    name="scene/mesh_Q.matrix",
    times=ts,
    values=TQs
)

```

Now create an `AnimationAction` that links the tracks to a play/pause button and associates this with the scene.

```

tracks = [track_B, track_A, track_Q]
duration = ts[-1] - ts[0]
clip = p3js.AnimationClip(tracks=tracks, duration=duration)
action = p3js.AnimationAction(p3js.AnimationMixer(scene), clip, scene)

```

You can find more about setting up animations with pythreejs in their documentation:

<https://pythreejs.readthedocs.io/en/stable/examples/Animation.html>

18.7 Animated Interactive 3D Visualization

With the scene and animation now defined the renderer and the animation controls can be displayed with:

```
renderer
```

```
Renderer(camera=PerspectiveCamera(aspect=1.5, position=(1.5, 0.6, 1.0),  
projectionMatrix=(1.0, 0.0, 0.0, 0.0, ...
```

```
action
```

```
AnimationAction(clip=AnimationClip(duration=10.0, tracks=(VectorKeyframeTrack(name=  
'scene/mesh_B.matrix', time...
```

The axes attached to the inertial reference frame and each mesh are the local coordinate system for that object. The X axis is red, the Y axis is green, the Z axis is blue.

The animation can be used to confirm realistic motion of the multibody system and to visually explore the various motions that can occur.

EQUATIONS OF MOTION WITH NONHOLONOMIC CONSTRAINTS

Note: You can download this example as a Python script: `nonholonomic-eom.py` or Jupyter Notebook: `nonholonomic-eom.ipynb`.

```
from IPython.display import HTML
from matplotlib.animation import FuncAnimation
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
import numpy as np
import sympy as sm
import sympy.physics.mechanics as me

me.init_vprinting(use_latex='mathjax')
```

```
class ReferenceFrame(me.ReferenceFrame):
    def __init__(self, *args, **kwargs):
        kwargs.pop('latexs', None)
        lab = args[0].lower()
        tex = r'\hat{{\{}{\}}}_{{}}'
        super(ReferenceFrame, self).__init__(*args,
                                             latexs=(tex.format(lab, 'x'),
                                                      tex.format(lab, 'y'),
                                                      tex.format(lab, 'z')),
                                             **kwargs)
    me.ReferenceFrame = ReferenceFrame
```

19.1 Learning Objectives

After completing this chapter readers will be able to:

- formulate the p dynamical differential equations for a nonholonomic system
- simulate a nonholonomic multibody system
- calculate trajectories of dependent speeds

19.2 Introduction

In chapters, *Holonomic Constraints* and *Nonholonomic Constraints*, I introduced two types of constraints: holonomic (configuration) constraints and nonholonomic (motion) constraints. Holonomic constraints are nonlinear constraints in the coordinates¹. Nonholonomic constraints are linear in the generalized speeds, by definition. We will address the nonholonomic equations of motion first, as they are slightly easier to deal with.

Nonholonomic constraint equations are linear in both the independent and dependent generalized speeds (see Sec. *Snakeboard*). We have shown that you can explicitly solve for the dependent generalized speeds \bar{u}_r as a function of the independent generalized speeds \bar{u}_s . This means that number of dynamical differential equations can be reduced to p from n with m nonholonomic constraints. Recall that the nonholonomic constraints take this form:

$$\bar{f}_n(\bar{u}_s, \bar{u}_r, \bar{q}, t) = \mathbf{M}_n \bar{u}_r + \bar{g}_n = 0 \in \mathbb{R}^m \quad (19.1)$$

and u_r can be solved for as so:

$$\bar{u}_r = -\mathbf{M}_n(\bar{q}, t)^{-1} \bar{g}_n(\bar{u}_s, \bar{q}, t) \quad (19.2)$$

which is the same as Eq. (12.58) we originally developed:

$$\bar{u}_r = \mathbf{A}_n \bar{u}_s + \bar{b}_n \quad (19.3)$$

Using Eq. (19.2) equation we can now write our equations of motion as n kinematical differential equations and p dynamical differential equations.

$$\begin{aligned} \bar{f}_k(\bar{u}_s, \dot{\bar{q}}, \bar{q}, t) &= \mathbf{M}_k \dot{\bar{q}} + \bar{g}_k = 0 \in \mathbb{R}^n \\ \bar{f}_d(\dot{\bar{u}}_s, \bar{u}_s, \bar{q}, t) &= \mathbf{M}_d \dot{\bar{u}}_s + \bar{g}_d = 0 \in \mathbb{R}^p \end{aligned} \quad (19.4)$$

and these can be written in explicit form:

$$\begin{aligned} \dot{\bar{q}} &= -\mathbf{M}_k(\bar{q}, t)^{-1} \bar{g}_k(\bar{u}_s, \bar{q}, t) \\ \dot{\bar{u}}_s &= -\mathbf{M}_d(\bar{q}, t)^{-1} \bar{g}_d(\bar{u}_s, \bar{q}, t) \end{aligned} \quad (19.5)$$

This leaves us with $n + p$ equations of motion, instead of $2n$ equations seen in a holonomic system. Nonholonomic constraints reduce the number of degrees of freedom and thus fewer dynamical differential equations are necessary to fully describe the motion.

19.3 Snakeboard Equations of Motion

Let's revisit the snakeboard example (see Sec. *Snakeboard*) and develop the equations of motion for that nonholonomic system. This system only has nonholonomic constraints and we selected u_1 and u_2 as the dependent speeds. For simplicity, we will assume that the mass and moments of inertia of the three bodies are the same.

19.3.1 1. Declare all the variables

First introduce the necessary variables; adding I for the central moment of inertia of each body and m as the mass of each body. Then create column matrices for the various sets of variables.

¹ They can be linear in the coordinates, but then there is little reason not to solve for the dependent coordinates and eliminate them.

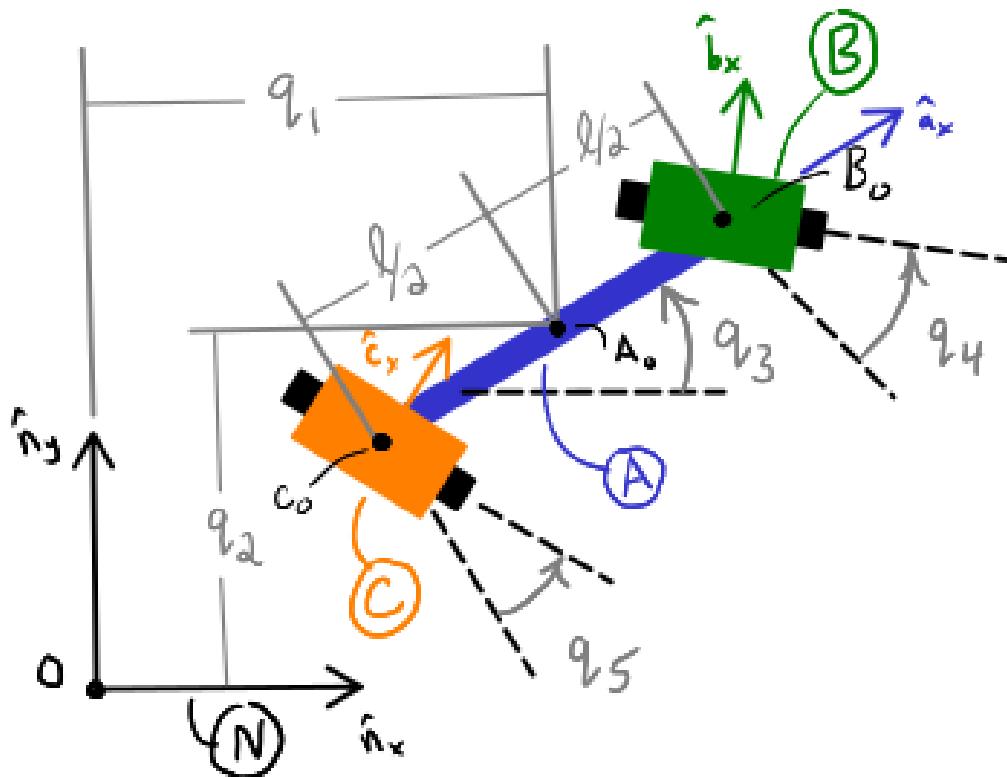


Fig. 19.1: Configuration diagram of a planar Snakeboard model.

```

q1, q2, q3, q4, q5 = me.dynamicsymbols('q1, q2, q3, q4, q5')
u1, u2, u3, u4, u5 = me.dynamicsymbols('u1, u2, u3, u4, u5')
l, I, m = sm.symbols('l, I, m')
t = me.dynamicsymbols._t

p = sm.Matrix([l, I, m])
q = sm.Matrix([q1, q2, q3, q4, q5])
us = sm.Matrix([u3, u4, u5])
ur = sm.Matrix([u1, u2])
u = ur.col_join(us)

q, ur, us, u, p

```

$$\left(\begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \end{bmatrix}, \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}, \begin{bmatrix} u_3 \\ u_4 \\ u_5 \end{bmatrix}, \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix}, \begin{bmatrix} l \\ I \\ m \end{bmatrix} \right) \quad (19.6)$$

We will also need column matrices for the time derivatives of each set of variables and some dictionaries to zero out any of these variables in various expressions we create.

```

qd = q.diff()
urd = ur.diff(t)
usd = us.diff(t)

```

(continues on next page)

(continued from previous page)

```
ud = u.diff(t)

qd, urd, usd, ud
```

$$\left(\begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \\ \dot{q}_5 \end{bmatrix}, \begin{bmatrix} \dot{u}_1 \\ \dot{u}_2 \end{bmatrix}, \begin{bmatrix} \dot{u}_3 \\ \dot{u}_4 \\ \dot{u}_5 \end{bmatrix}, \begin{bmatrix} \dot{u}_1 \\ \dot{u}_2 \\ \dot{u}_3 \\ \dot{u}_4 \\ \dot{u}_5 \end{bmatrix} \right) \quad (19.7)$$

```
qd_zero = {qdi: 0 for qdi in qd}
ur_zero = {ui: 0 for ui in ur}
us_zero = {ui: 0 for ui in us}
urd_zero = {udi: 0 for udi in urd}
usd_zero = {udi: 0 for udi in usd}

qd_zero, ur_zero, us_zero
```

$$(\{\dot{q}_1 : 0, \dot{q}_2 : 0, \dot{q}_3 : 0, \dot{q}_4 : 0, \dot{q}_5 : 0\}, \{u_1 : 0, u_2 : 0\}, \{u_3 : 0, u_4 : 0, u_5 : 0\}) \quad (19.8)$$

```
urd_zero, usd_zero
```

$$(\{\dot{u}_1 : 0, \dot{u}_2 : 0\}, \{\dot{u}_3 : 0, \dot{u}_4 : 0, \dot{u}_5 : 0\}) \quad (19.9)$$

19.3.2 2. Establish the kinematics

The following code sets up the orientations, positions, and velocities exactly as done in the original example. All of the velocities are in terms of \bar{q} and $\dot{\bar{q}}$.

```
N = me.ReferenceFrame('N')
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
C = me.ReferenceFrame('C')

A.orient_axis(N, q3, N.z)
B.orient_axis(A, q4, A.z)
C.orient_axis(A, q5, A.z)

A.ang_vel_in(N)
B.ang_vel_in(N)
C.ang_vel_in(N)

O = me.Point('O')
```

(continues on next page)

(continued from previous page)

```

Ao = me.Point('A_o')
Bo = me.Point('B_o')
Co = me.Point('C_o')

Ao.set_pos(0, q1*N.x + q2*N.y)
Bo.set_pos(Ao, 1/2*A.x)
Co.set_pos(Ao, -1/2*A.x)

O.set_vel(N, 0)
Bo.v2pt_theory(Ao, N, A)
Co.v2pt_theory(Ao, N, A);

```

19.3.3 3. Specify the kinematical differential equations

Now create the $n = 5$ kinematical differential equations \bar{f}_k :

```

fk = sm.Matrix([
    u1 - q1.diff(t),
    u2 - q2.diff(t),
    u3 - 1*q3.diff(t)/2,
    u4 - q4.diff(t),
    u5 - q5.diff(t),
])

```

It is a good idea to use `find_dynamicsymbols()` to check which functions of time are present in the various equations. This function is invaluable when the equations begin to become very large.

```
me.find_dynamicsymbols(fk)
```

$$\{q_1, q_2, q_3, q_4, q_5, u_1, u_2, u_3, u_4, u_5, \dot{q}_1, \dot{q}_2, \dot{q}_3, \dot{q}_4, \dot{q}_5\} \quad (19.10)$$

Symbolically solve these equations for \dot{q} and setup a dictionary we can use for substitutions:

```

Mk = fk.jacobian(qd)
gk = fk.xreplace(qd_zero)
qd_sol = -Mk.LUsolve(gk)
qd_repl = dict(zip(qd, qd_sol))
qd_repl

```

$$\left\{ \dot{q}_1 : u_1, \dot{q}_2 : u_2, \dot{q}_3 : \frac{2u_3}{l}, \dot{q}_4 : u_4, \dot{q}_5 : u_5 \right\} \quad (19.11)$$

19.3.4 4. Establish the nonholonomic constraints

Create the $m = 2$ nonholonomic constraints:

```
fn = sm.Matrix([Bo.vel(N).dot(B.y), Co.vel(N).dot(C.y)])
fn
```

$$\begin{bmatrix} \frac{l \cos(q_4) \dot{q}_3}{2} + (-\sin(q_3) \sin(q_4) + \cos(q_3) \cos(q_4)) \dot{q}_2 + (-\sin(q_3) \cos(q_4) - \sin(q_4) \cos(q_3)) \dot{q}_1 \\ -\frac{l \cos(q_5) \dot{q}_3}{2} + (-\sin(q_3) \sin(q_5) + \cos(q_3) \cos(q_5)) \dot{q}_2 + (-\sin(q_3) \cos(q_5) - \sin(q_5) \cos(q_3)) \dot{q}_1 \end{bmatrix} \quad (19.12)$$

and rewrite them in terms of the generalized speeds:

```
fn = fn.xreplace(qd_repl)
fn
```

$$\begin{bmatrix} (-\sin(q_3) \sin(q_4) + \cos(q_3) \cos(q_4)) u_2 + (-\sin(q_3) \cos(q_4) - \sin(q_4) \cos(q_3)) u_1 + u_3 \cos(q_4) \\ (-\sin(q_3) \sin(q_5) + \cos(q_3) \cos(q_5)) u_2 + (-\sin(q_3) \cos(q_5) - \sin(q_5) \cos(q_3)) u_1 - u_3 \cos(q_5) \end{bmatrix} \quad (19.13)$$

```
me.find_dynamicsymbols(fn)
```

$$\{q_3, q_4, q_5, u_1, u_2, u_3\} \quad (19.14)$$

With the nonholonomic constraint equations we choose $\bar{u}_r = [u_1 \ u_2]^T$ and symbolically for these dependent speeds.

```
Mn = fn.jacobian(ur)
gn = fn.xreplace(ur_zero)
ur_sol = Mn.LUsolve(-gn)
ur_repl = dict(zip(ur, ur_sol))
```

In our case, the dependent generalized speeds are only a function of one independent generalized speed, u_3 .

```
me.find_dynamicsymbols(ur_sol)
```

$$\{q_3, q_4, q_5, u_3\} \quad (19.15)$$

Exercise

Why does u_1 and u_2 not depend on q_1, q_2, u_4 and u_5 ?

Our kinematical differential equations can now be rewritten in terms of the independent generalized speeds. We only need to rewrite \bar{g}_k for later use in our numerical functions.

```
gk = gk.xreplace(ur_repl)
me.find_dynamicsymbols(gk)
```

$$\{q_3, q_4, q_5, u_3, u_4, u_5\} \quad (19.16)$$

19.3.5 5. Rewrite velocities in terms of independent speeds

The snakeboard model, as described, has no generalized active forces because there are no contributing external forces acting on the system, so we only need to generate the nonholonomic generalized inertia forces \bar{F}_r^* . We now then calculate the velocities we will need to form \bar{F}_r^* and make sure they are written only in terms of the independent generalized speeds.

```

N_w_A = A.ang_vel_in(N).xreplace(qd_repl).xreplace(ur_repl)
N_w_B = B.ang_vel_in(N).xreplace(qd_repl).xreplace(ur_repl)
N_w_C = C.ang_vel_in(N).xreplace(qd_repl).xreplace(ur_repl)
N_v_Ao = Ao.vel(N).xreplace(qd_repl).xreplace(ur_repl)
N_v_Bo = Bo.vel(N).xreplace(qd_repl).xreplace(ur_repl)
N_v_Co = Co.vel(N).xreplace(qd_repl).xreplace(ur_repl)

vels = (N_w_A, N_w_B, N_w_C, N_v_Ao, N_v_Bo, N_v_Co)

for vel in vels:
    print(me.find_dynamicsymbols(vel, reference_frame=N))

```

```

{u3(t)}
{u4(t), u3(t)}
{u3(t), u5(t)}
{q4(t), q5(t), q3(t), u3(t)}
{q4(t), q5(t), q3(t), u3(t)}
{q4(t), q5(t), q3(t), u3(t)}

```

19.3.6 6. Compute the partial velocities

With the velocities only in terms of the independent generalized speeds, we can calculate the p nonholonomic partial velocities:

```
w_A, w_B, w_C, v_Ao, v_Bo, v_Co = me.partial_velocity(vels, us, N)
```

19.3.7 7. Rewrite the accelerations in terms of the independent generalized speeds

We can also write the accelerations in terms of only the independent generalized speeds, their time derivatives, and the generalized coordinates. To do so, we need to differentiate the nonholonomic constraints so that we can eliminate the dependent *generalized accelerations*, $\dot{\bar{u}}_r$. Differentiating the constraints with respect to time and then substituting for the dependent generalized speeds gives us equations for the dependent generalized accelerations.

$$\begin{aligned} \dot{\bar{f}}_n(\dot{\bar{u}}_r, \dot{\bar{u}}_s, \bar{u}_s, \bar{u}_r, \bar{q}, t) &= \mathbf{M}_{nd}\dot{\bar{u}}_r + \bar{g}_{nd} = 0 \in \mathbb{R}^m \\ \dot{\bar{u}}_r &= -\mathbf{M}_{nd}(\bar{q}, t)^{-1}\bar{g}_{nd}(\dot{\bar{u}}_s, \bar{u}_s, \bar{q}, t) \end{aligned} \quad (19.17)$$

First, time differentiate the nonholonomic constraints and eliminate the time derivatives of the generalized coordinates.

```

fnd = fn.diff(t).xreplace(qd_repl)

me.find_dynamicsymbols(fnd)

```

$$\{q_3, q_4, q_5, u_1, u_2, u_3, u_4, u_5, \dot{u}_1, \dot{u}_2, \dot{u}_3\} \quad (19.18)$$

Now solve for the dependent generalized accelerations. Note that I replace the dependent generalized speeds in \bar{g}_{nd} instead of \dot{f}_n earlier. This is to avoid replacing the u_1 and u_2 terms in the `Derivative(u1, t)` and `Derivative(u2, t)` terms.

```
Mnd = fnd.jacobian(urd)
gnd = fnd.xreplace(urd_zero).xreplace(ur_repl)
urd_sol = Mnd.LUsolve(-gnd)
urd_repl = dict(zip(urd, urd_sol))

me.find_dynamicsymbols(urd_sol)
```

$$\{q_3, q_4, q_5, u_3, u_4, u_5, \dot{u}_3\} \quad (19.19)$$

19.3.8 8. Create the generalized forces

Now we can form the inertia forces and inertia torques. First check what derivatives appear in the accelerations.

```
Rs_Ao = -m*Ao.acc(N)
Rs_Bo = -m*Bo.acc(N)
Rs_Co = -m*Co.acc(N)

(me.find_dynamicsymbols(Rs_Ao, reference_frame=N) |
 me.find_dynamicsymbols(Rs_Bo, reference_frame=N) |
 me.find_dynamicsymbols(Rs_Co, reference_frame=N))
```

$$\{q_1, q_2, q_3, \ddot{q}_1, \ddot{q}_2, \dot{q}_3, \ddot{q}_3\} \quad (19.20)$$

We'll need to replace the \ddot{q} first and then the \dot{q} . Create the first replacement by differentiating the expressions for \dot{q} .

Warning: If you use chained replacements, e.g. `.xreplace().xreplace().xreplace()` you have to be careful about the order of replacements so that you don't substitute symbols inside a derivative, e.g. `Derivative(u, t)`. If you have `expr = Derivative(u, t) + u` then you need to replace the entire derivative first: `expr.xreplace({u.diff(): 1}).xreplace({u: 2})`.

```
qdd_repl = {k.diff(t): v.diff(t).xreplace(urd_repl) for k, v in qd_repl.items()}
```

```
Rs_Ao = -m*Ao.acc(N).xreplace(qdd_repl).xreplace(qd_repl)
Rs_Bo = -m*Bo.acc(N).xreplace(qdd_repl).xreplace(qd_repl)
Rs_Co = -m*Co.acc(N).xreplace(qdd_repl).xreplace(qd_repl)

(me.find_dynamicsymbols(Rs_Ao, reference_frame=N) |
 me.find_dynamicsymbols(Rs_Bo, reference_frame=N) |
 me.find_dynamicsymbols(Rs_Co, reference_frame=N))
```

$$\{q_3, q_4, q_5, u_3, u_4, u_5, \dot{u}_3\} \quad (19.21)$$

The motion is planar so the generalized inertia torques are simply angular accelerations dotted with the central inertia dyadics.

```
I_A_Ao = I*me.outer(A.z, A.z)
I_B_Bo = I*me.outer(B.z, B.z)
I_C_Co = I*me.outer(C.z, C.z)
```

Now have a look at which functions are present in the inertia torques:

```
Ts_A = -A.ang_acc_in(N).dot(I_A_Ao)
Ts_B = -B.ang_acc_in(N).dot(I_B_Bo)
Ts_C = -C.ang_acc_in(N).dot(I_C_Co)

(me.find_dynamicssymbols(Ts_A, reference_frame=N) |
 me.find_dynamicssymbols(Ts_B, reference_frame=N) |
 me.find_dynamicssymbols(Ts_C, reference_frame=N))
```

$$\{q_3, q_4, q_5, \ddot{q}_3, \ddot{q}_4, \ddot{q}_5\} \quad (19.22)$$

and eliminate the dependent generalized accelerations:

```
Ts_A = -A.ang_acc_in(N).dot(I_A_Ao).xreplace(qdd_repl)
Ts_B = -B.ang_acc_in(N).dot(I_B_Bo).xreplace(qdd_repl)
Ts_C = -C.ang_acc_in(N).dot(I_C_Co).xreplace(qdd_repl)

(me.find_dynamicssymbols(Ts_A, reference_frame=N) |
 me.find_dynamicssymbols(Ts_B, reference_frame=N) |
 me.find_dynamicssymbols(Ts_C, reference_frame=N))
```

$$\{u_3, u_4, u_5, \dot{u}_3, \dot{u}_4, \dot{u}_5\} \quad (19.23)$$

19.3.9 9. Formulate the dynamical differential equations

All of the components are present to formulate the nonholonomic generalized inertia forces. After we form them, make sure they are only a function of the independent generalized speeds, their time derivatives, and the generalized coordinates.

```
Frs = []
for i in range(len(us)):
    Frs.append(v_Ao[i].dot(Rs_Ao) + v_Bo[i].dot(Rs_Bo) + v_Co[i].dot(Rs_Co) +
               w_A[i].dot(Ts_A) + w_B[i].dot(Ts_B) + w_C[i].dot(Ts_C))
Frs = sm.Matrix(Frs)

me.find_dynamicssymbols(Frs)
```

$$\{q_3, q_4, q_5, u_3, u_4, u_5, \dot{u}_3, \dot{u}_4, \dot{u}_5\} \quad (19.24)$$

At this point you may have noticed that q_1 and q_2 have not appeared in any equations. This means that the dynamics do not depend on the planar location of the snakeboard. q_1 and q_2 are called *ignorable coordinates* if they do not appear in the equations of motion. It is only coincidence that the time derivatives of these ignorable coordinates are equal to the two dependent generalized speeds.

Lastly, extract the linear coefficients and the remainder for the dynamical differential equations.

```
Md = Frs.jacobian(usd)
gd = Frs.xreplace(usd_zero)
```

And one last time, check that \mathbf{M}_d and \mathbf{g}_d are only functions of the independent generalized speeds and the generalized coordinates.

```
me.find_dynamicsymbols(Md)
```

$$\{q_3, q_4, q_5\} \quad (19.25)$$

```
me.find_dynamicsymbols(gd)
```

$$\{q_3, q_4, q_5, u_3, u_4, u_5\} \quad (19.26)$$

We now have \mathbf{M}_k , \bar{g}_k , \mathbf{M}_d and \bar{g}_d and can proceed to numerical evaluation.

19.4 Simulate the Snakeboard

We now move to numerical evaluation for the simulation. First, create a function that evaluates the matrices of the equations of motion.

```
eval_kd = sm.lambdify((q, us, p), (Mk, gk, Md, gd), cse=True)
```

Now create a function that evaluates the right hand side of the explicit ordinary differential equations for use with `solve_ivp()`.

```
def eval_rhs(t, x, p):
    """Returns the time derivative of the states.

    Parameters
    ======
    t : float
    x : array_like, shape(8,)
        x = [q1, q2, q3, q4, q5, u3, u4, u5]
    p : array_like, shape(3,)
        p = [l, I, m]

    Returns
    ======
    xd : ndarray, shape(8,)
        xd = [q1d, q2d, q3d, q4d, q5d, u3d, u4d, u5d]
```

(continues on next page)

(continued from previous page)

```
"""
q, us = x[:5], x[5:]

Mk, gk, Md, gd = eval_kd(q, us, p)

qd = -np.linalg.solve(Mk, gk.squeeze())
usd = -np.linalg.solve(Md, gd.squeeze())

return np.hstack((qd, usd))

```

Now introduce some numeric values for the constant parameters and the initial condition of the state. I've selected some values here that will put the snakeboard in an initial state of motion.

```
p_vals = np.array([
    0.7,    # l [m]
    0.1,    # I [kg*m^2]
    1.0,    # m [kg]
])

q0 = np.array([
    0.0,    # q1 [m]
    0.0,    # q2 [m]
    0.0,    # q3 [rad]
    np.deg2rad(5.0),    # q4 [rad]
    -np.deg2rad(5.0),    # q5 [rad]
])

us0 = np.array([
    0.1,    # u3 [m/s]
    0.01,   # u4 [rad/s]
    -0.01,   # u5 [rad/s]
])

x0 = np.hstack((q0, us0))
p_vals, x0
```

```
(array([0.7, 0.1, 1.]),
 array([ 0.          ,  0.          ,  0.          ,  0.08726646, -0.08726646,
        0.1         ,  0.01         , -0.01        ]))
```

Check whether `eval_rhs()` works with these arrays:

```
eval_rhs(1.0, x0, p_vals)
```

```
array([ 1.14300523,  0.          ,  0.28571429,  0.01          ,
       0.01143537, -0.03267249, -0.03267249])
```

We can now integrate the equations of motion to find the state trajectories. I setup the time array for the solution to correspond to 30 frames per second for later use in the animation of the motion.

```
t0, tf = 0.0, 8.0
```

```
fps = 20
ts = np.linspace(t0, tf, num=int(fps*(tf - t0)))
```

(continues on next page)

(continued from previous page)

```
sol = solve_ivp(eval_rhs, (t0, tf), x0, args=(p_vals,), t_eval=ts)

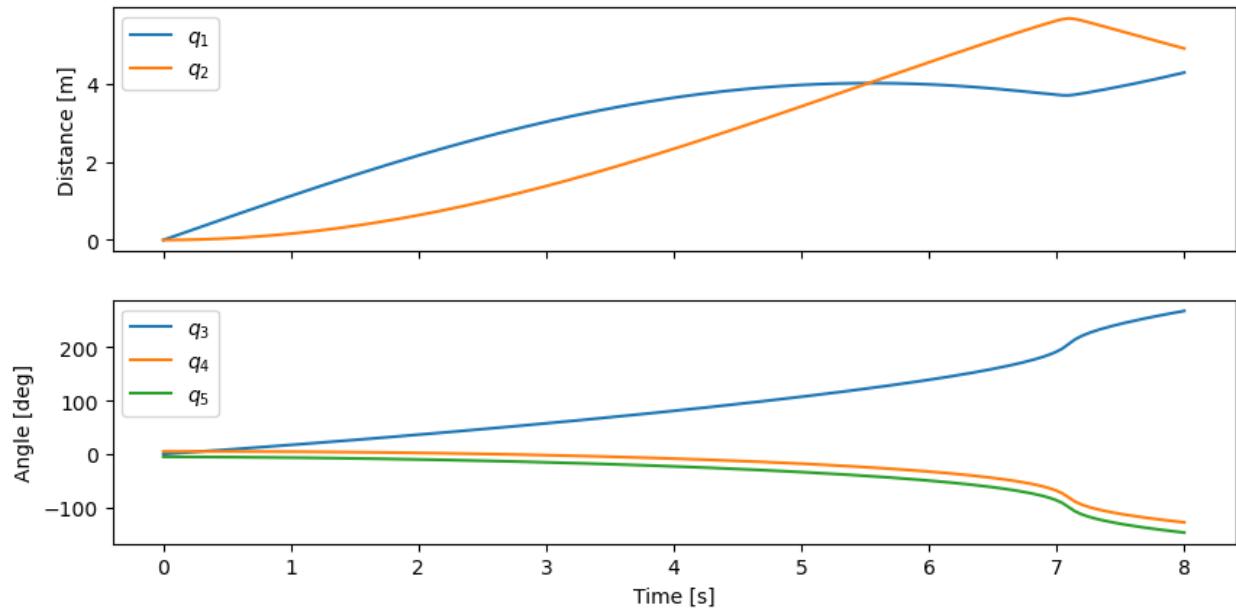
xs = np.transpose(sol.y)
```

Now we can plot the state trajectories to see if there is realistic motion.

```
fig, axes = plt.subplots(2, 1, sharex=True)
fig.set_figwidth(10.0)

axes[0].plot(ts, xs[:, :2])
axes[0].legend(['$q_1$', '$q_2$'])
axes[0].set_ylabel('Distance [m]')

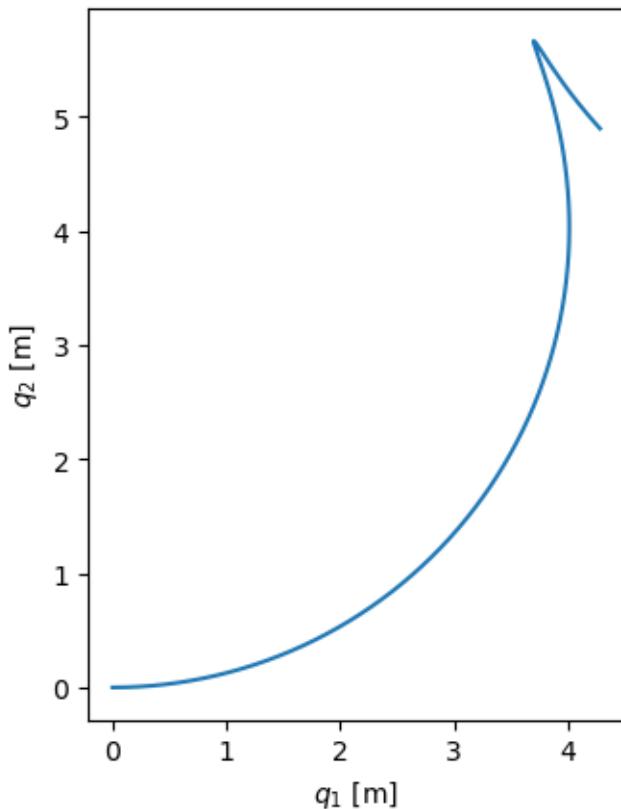
axes[1].plot(ts, np.rad2deg(xs[:, 2:5]))
axes[1].legend(['$q_3$', '$q_4$', '$q_5$'])
axes[1].set_ylabel('Angle [deg]')
axes[1].set_xlabel('Time [s]');
```



We see that the x and y positions vary over several meters and that there is a sharp transition around about 7 seconds. $q_3(t)$ shows that the primary angle of the snakeboard grows with time and does almost a full rotation. Plotting the path on the ground plane of A_o gives a bit more insight to the motion.

```
fig, ax = plt.subplots()
fig.set_figwidth(10.0)

ax.plot(xs[:, 0], xs[:, 1])
ax.set_aspect('equal')
ax.set_xlabel('$q_1$ [m]')
ax.set_ylabel('$q_2$ [m]');
```



We see that the snakeboard curves to the left but eventually makes a very sharp trajectory change. An animation will provide an even more clear idea of the motion of this nonholonomic system.

19.5 Animate the Snakeboard

We will animate the snakeboard as a collection of lines and points and animate the 2D motion with matplotlib. First, create some new points that represent the location of the left and right wheels on bodies B and C .

```

Bl = me.Point('B_l')
Br = me.Point('B_r')
Cr = me.Point('C_r')
Cl = me.Point('C_l')

Bl.set_pos(Bo, -1/4*B.y)
Br.set_pos(Bo, 1/4*B.y)
Cl.set_pos(Co, -1/4*C.y)
Cr.set_pos(Co, 1/4*C.y)

```

Create a function that numerically evaluates the Cartesian coordinates of all the points we want to plot given the generalized coordinates.

```

coordinates = Cl.pos_from(O).to_matrix(N)
for point in [Co, Cr, Co, Ao, Bo, Bl, Br]:
    coordinates = coordinates.row_join(point.pos_from(O).to_matrix(N))

```

(continues on next page)

(continued from previous page)

```
eval_point_coords = sm.lambdify((q, p), coordinates, cse=True)
eval_point_coords(q0, p_vals)
```

```
array([[-0.36525225, -0.35, -0.33474775, -0.35, 0.,
       0.35, 0.36525225, 0.33474775],
      [-0.17433407, -0., 0.17433407, -0., 0.,
       0., -0.17433407, 0.17433407],
      [0., 0., 0., 0., 0.,
       0., 0., 0.]]))
```

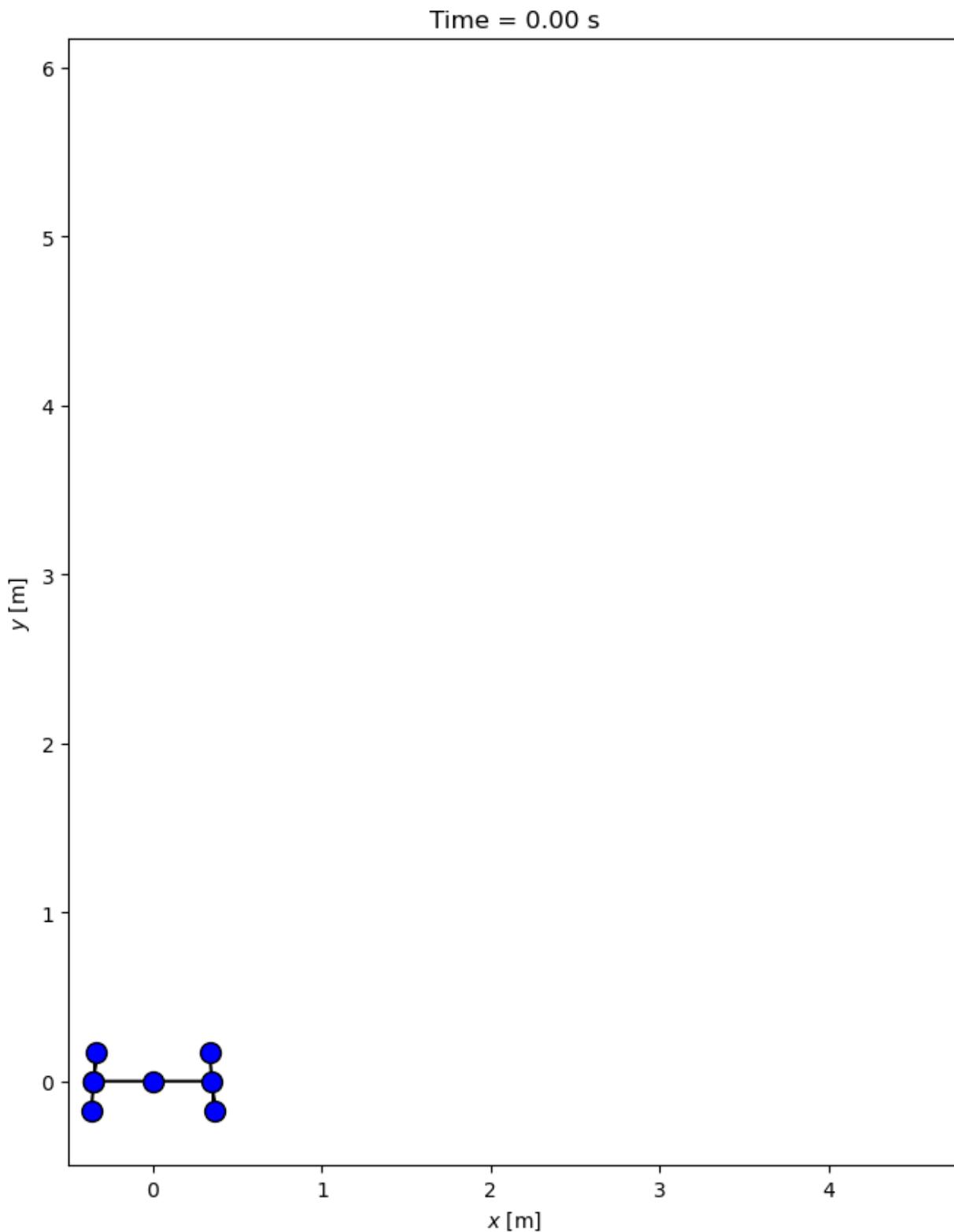
Now create a plot of the initial configuration:

```
x, y, z = eval_point_coords(q0, p_vals)

fig, ax = plt.subplots()
fig.set_size_inches((10.0, 10.0))
ax.set_aspect('equal')

lines, = ax.plot(x, y, color='black',
                  marker='o', markerfacecolor='blue', markersize=10)
# some empty lines to use for the wheel paths
bl_path, = ax.plot([], [])
br_path, = ax.plot([], [])
cl_path, = ax.plot([], [])
cr_path, = ax.plot([], [])

title_template = 'Time = {:.2f} s'
title_text = ax.set_title(title_template.format(t0))
ax.set_xlim((np.min(xs[:, 0]) - 0.5, np.max(xs[:, 0]) + 0.5))
ax.set_ylim((np.min(xs[:, 1]) - 0.5, np.max(xs[:, 1]) + 0.5))
ax.set_xlabel('$x$ [m]')
ax.set_ylabel('$y$ [m]');
```



And, finally, animate the motion:

```

coords = []
for xi in xs:
    coords.append(eval_point_coords(xi[:5], p_vals))
coords = np.array(coords) # shape(600, 3, 8)

def animate(i):
    title_text.set_text(title_template.format(sol.t[i]))
    lines.set_data(coords[i, 0, :], coords[i, 1, :])
    cl_path.set_data(coords[:i, 0, 0], coords[:i, 1, 0])
    cr_path.set_data(coords[:i, 0, 2], coords[:i, 1, 2])
    bl_path.set_data(coords[:i, 0, 6], coords[:i, 1, 6])
    br_path.set_data(coords[:i, 0, 7], coords[:i, 1, 7])

ani = FuncAnimation(fig, animate, len(sol.t))

HTML(ani.to_jshtml(fps=fps))
  
```

```

<IPython.core.display.HTML object>
  
```

19.6 Calculating Dependent Speeds

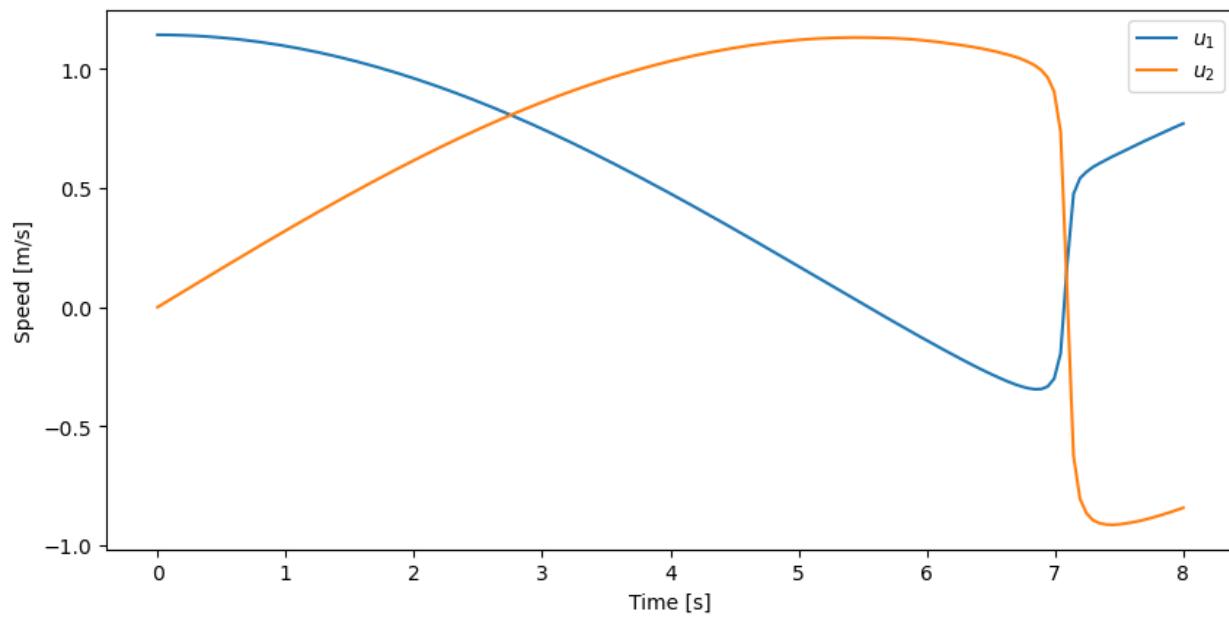
Since we have eliminated the dependent generalized speeds (u_1 and u_2) from the equations of motion, these are not computed from `solve_ivp()`. If these are needed, it is possible to calculate them using the constraint equations. Here I loop through time to calculate \bar{u}_r at each time step and then plot the results.

```

x = sm.Matrix([q1, q2, q3, q4, q5, u3, u4, u5])
eval_ur = sm.lambdify((x, p), ur_sol, cse=True)

ur_vals = []
for xi in xs:
    ur_vals.append(eval_ur(xi, p_vals))
ur_vals = np.array(ur_vals).squeeze()

fig, ax = plt.subplots()
fig.set_figwidth(10.0)
ax.plot(ts, ur_vals)
ax.set_ylabel('Speed [m/s]')
ax.set_xlabel('Time [s]')
ax.legend(['$u_1$', '$u_2$']);
  
```



EQUATIONS OF MOTION WITH HOLONOMIC CONSTRAINTS

Note: You can download this example as a Python script: `holonomic-eom.py` or Jupyter Notebook: `holonomic-eom.ipynb`.

```
from IPython.display import HTML
from matplotlib.animation import FuncAnimation
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
import numpy as np
import sympy as sm
import sympy.physics.mechanics as me

me.init_vprinting(use_latex='mathjax')
```

```
class ReferenceFrame(me.ReferenceFrame):
    def __init__(self, *args, **kwargs):
        kwargs.pop('latexs', None)
        lab = args[0].lower()
        tex = r'\hat{{}}_{{}}_{{}}'
        super(ReferenceFrame, self).__init__(*args,
                                             latexs=(tex.format(lab, 'x'),
                                                      tex.format(lab, 'y'),
                                                      tex.format(lab, 'z')),
                                             **kwargs)
    me.ReferenceFrame = ReferenceFrame
```

20.1 Learning Objectives

After completing this chapter readers will be able to:

- formulate the differential algebraic equations of motion for a multibody system that includes additional holonomic constraints
- simulate a system with additional constraints using a differential algebraic equation integrator
- compare simulation results that do and do not manage constraint drift

20.2 Introduction

When there are holonomic constraints present the equations of motion are comprised of the kinematical differential equations $\bar{f}_k = 0$, dynamical differential equations $\bar{f}_d = 0$, and the holonomic constraint equations $\bar{f}_h = 0$. This set of equations are called [differential algebraic equations](#) and the algebraic equations cannot be solved for explicitly, as we did with the nonholonomic algebraic constraint equations.

In a system such as this, there are $N = n + M$ total coordinates, with n generalized coordinates \bar{q} and M additional dependent coordinates \bar{q}_r . The holonomic constraints take this form:

$$\bar{f}_h(\bar{q}, \bar{q}_r, t) = 0 \in \mathbb{R}^M \quad (20.1)$$

n generalized speeds \bar{u} and M dependent speeds \bar{u}_r can be introduced using N kinematical differential equations.

$$\bar{f}_h(\dot{\bar{q}}, \dot{\bar{q}}_r, \bar{u}, \bar{u}_r, \bar{q}, \bar{q}_r, t) = 0 \in \mathbb{R}^N \quad (20.2)$$

We can formulate the equations of motion by transforming the holonomic constraints into a function of generalized speeds. These equations are then treated just like nonholonomic constraints described in the previous Chp. [Equations of Motion with Nonholonomic Constraints](#).

$$\dot{\bar{f}}_h(\bar{u}, \bar{u}_r, \bar{q}, \bar{q}_r, t) = \mathbf{M}_{hd}\bar{u}_r + \bar{g}_{hd} = 0 \in \mathbb{R}^M \quad (20.3)$$

We can solve for M dependent generalized speeds:

$$\bar{u}_r = -\mathbf{M}_{hd}^{-1}\bar{g}_{hd} \in \mathbb{R}^M \quad (20.4)$$

and then rewrite the kinematical and dynamical differential equations in terms of the generalized speeds, their time derivatives, the generalized coordinates, and the dependent coordinates.

$$\begin{aligned} \bar{f}_k(\dot{\bar{q}}, \dot{\bar{q}}_r, \bar{u}, \bar{q}, \bar{q}_r, t) &= 0 \in \mathbb{R}^N \\ \bar{f}_d(\dot{\bar{u}}, \bar{u}, \bar{q}, \bar{q}_r, t) &= 0 \in \mathbb{R}^n \end{aligned} \quad (20.5)$$

This final set of equations has $N + n$ state variables and can be integrated as a set of ordinary differential equations or the $N + n + M$ equations can be integrated as a set of differential algebraic equations. We will demonstrate the differences in the results for the two approaches.

20.3 Four-bar Linkage Equations of Motion

To demonstrate the formulation of the equations of motion of a system with an explicit holonomic constraints, let's revisit the four-bar linkage from Sec. [Four-Bar Linkage](#). We will now make P_2 and P_3 particles, each with mass m and include the effects of gravity in the $-\hat{n}_y$ direction.

As before, we setup the system by disconnecting the kinematic loop at point P_4 and then use this open loop to derive equations for the holonomic constraints that close the loop.

20.3.1 1. Declare all of the variables

We have three coordinates, only one of which is a generalized coordinate. I use q to hold the single generalized coordinate, qr for the two dependent coordinates, and qN to hold all the coordinates; similarly for the generalized speeds.

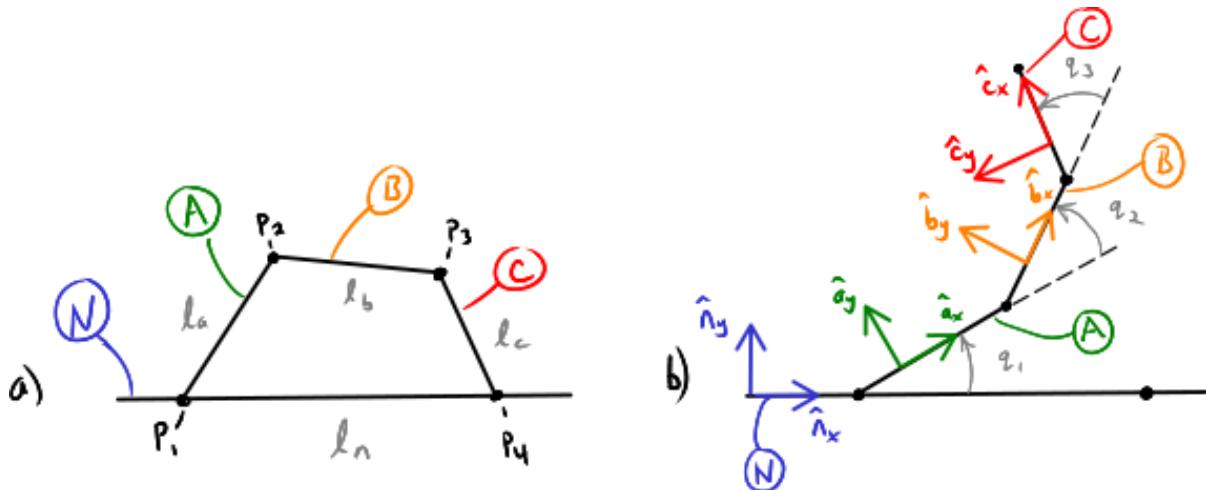


Fig. 20.1: a) Shows four links in a plane A, B, C , and N with respective lengths l_a, l_b, l_c, l_n connected in a closed loop at points P_1, P_2, P_3, P_4 . b) Shows the same linkage that has been separated at point P_4 to make it an open chain of links.

```

q1, q2, q3 = me.dynamicsymbols('q1, q2, q3')
u1, u2, u3 = me.dynamicsymbols('u1, u2, u3')
la, lb, lc, ln = sm.symbols('l_a, l_b, l_c, l_n')
m, g = sm.symbols('m, g')
t = me.dynamicsymbols._t

p = sm.Matrix([la, lb, lc, ln, m, g])

q = sm.Matrix([q1])
qr = sm.Matrix([q2, q3])
qN = q.col_join(qr)

u = sm.Matrix([u1])
ur = sm.Matrix([u2, u3])
uN = u.col_join(ur)

qdN = qN.diff(t)
ud = u.diff(t)

p, q, qr, qN, u, ur, uN, qdN, ud

```

$$\left(\begin{bmatrix} l_a \\ l_b \\ l_c \\ l_n \\ m \\ g \end{bmatrix}, \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix}, \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}, \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix}, \begin{bmatrix} \dot{u}_1 \\ \dot{u}_2 \\ \dot{u}_3 \end{bmatrix} \right) \quad (20.6)$$

```

ur_zero = {ui: 0 for ui in ur}
uN_zero = {ui: 0 for ui in uN}
qdN_zero = {qdi: 0 for qdi in qdN}
ud_zero = {udi: 0 for udi in ud}

```

20.3.2 2. Setup the open loop kinematics and holonomic constraints

Start by defining the orientation of the reference frames and positions of the points in terms of the $N = 3$ coordinates, leaving P_4 unconstrained.

```
N = me.ReferenceFrame('N')
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')
C = me.ReferenceFrame('C')

A.orient_axis(N, q1, N.z)
B.orient_axis(A, q2, A.z)
C.orient_axis(B, q3, B.z)

P1 = me.Point('P1')
P2 = me.Point('P2')
P3 = me.Point('P3')
P4 = me.Point('P4')

P2.set_pos(P1, la*A.x)
P3.set_pos(P2, lb*B.x)
P4.set_pos(P3, lc*C.x)
```

20.3.3 3. Create the holonomic constraints

Now $M = 2$ holonomic constraints can be found by closing the loop.

```
loop = P4.pos_from(P1) - ln*N.x

fh = sm.Matrix([loop.dot(N.x), loop.dot(N.y)])
fh = sm.trigsimp(fh)
fh
```

$$\begin{bmatrix} l_a \cos(q_1) + l_b \cos(q_1 + q_2) + l_c \cos(q_1 + q_2 + q_3) - l_n \\ l_a \sin(q_1) + l_b \sin(q_1 + q_2) + l_c \sin(q_1 + q_2 + q_3) \end{bmatrix} \quad (20.7)$$

Warning: Be careful about using `trigsimp()` on larger problems, as it can really slow down the calculations. It is not necessary to use, but I do so here so that the resulting equations are human readable in this context.

Note that these constraints are only a function of the N coordinates, not their time derivatives.

```
me.find_dynamicsymbols(fh)
```

$$\{q_1, q_2, q_3\} \quad (20.8)$$

20.3.4 4. Specify the kinematical differential equations

Use simple definitions for the generalized speed u_1 and the dependent speeds u_2 and u_3 . We create $N = 3$ generalized speeds even though the degrees of freedom are $n = 1$.

```
fk = sm.Matrix([
    q1.diff(t) - u1,
    q2.diff(t) - u2,
    q3.diff(t) - u3,
])
Mk = fk.jacobian(qdN)
gk = fk.xreplace(qdN_zero)
qdN_sol = -Mk.LUsolve(gk)
qd_repl = dict(zip(qdN, qdN_sol))
qd_repl
```

$$\{\dot{q}_1 : u_1, \dot{q}_2 : u_2, \dot{q}_3 : u_3\} \quad (20.9)$$

20.3.5 5. Solve for the dependent speeds

Differentiate the holonomic constraints with respect to time to arrive at a motion constraint. This is equivalent to setting ${}^N\bar{v}^{P_4} = 0$.

```
fhd = fh.diff(t).xreplace(qd_repl)
fhd = sm.trigsimp(fhd)
fhd
```

$$\left[-l_a u_1 \sin(q_1) - l_b (u_1 + u_2) \sin(q_1 + q_2) - l_c (u_1 + u_2 + u_3) \sin(q_1 + q_2 + q_3) \right] \quad (20.10)$$

These holonomic motion constraints are functions of the coordinates and speeds.

```
me.find_dynamicsymbols(fhd)
```

$$\{q_1, q_2, q_3, u_1, u_2, u_3\} \quad (20.11)$$

Choose u_2 and u_3 as the dependent speeds and solve the linear equations for these dependent speeds.

```
Mhd = fhd.jacobian(ur)
ghd = fhd.xreplace(ur_zero)
ur_sol = sm.trigsimp(-Mhd.LUsolve(ghd))
ur_repl = dict(zip(ur, ur_sol))
ur_repl[u2]
```

$$\frac{-l_a u_1 \sin(q_1) - \frac{l_a (l_b \sin(q_2) + l_c \sin(q_2 + q_3)) u_1 \sin(q_1 + q_2 + q_3)}{l_b \sin(q_3)} - l_b u_1 \sin(q_1 + q_2) - l_c u_1 \sin(q_1 + q_2 + q_3)}{l_b \sin(q_1 + q_2) + l_c \sin(q_1 + q_2 + q_3)} \quad (20.12)$$

```
ur_repl[u3]
```

$$\frac{l_a (l_b \sin(q_2) + l_c \sin(q_2 + q_3)) u_1}{l_b l_c \sin(q_3)} \quad (20.13)$$

20.3.6 6. Write velocities in terms of the generalized speeds

We have three simple rotations and we can write the three angular velocities only in terms of u_1 by using the expressions for the independent speeds from the previous step.

```
A.set_ang_vel(N, u1*N.z)
B.set_ang_vel(A, ur_repl[u2]*A.z)
C.set_ang_vel(B, ur_repl[u3]*B.z)
```

Now, by using the two point velocity theorem the velocities of each point will also only be in terms of u_1 .

```
P1.set_vel(N, 0)
P2.v2pt_theory(P1, N, A)
P3.v2pt_theory(P2, N, B)
P4.v2pt_theory(P3, N, C)

(me.find_dynamicsymbols(P2.vel(N), reference_frame=N) |
 me.find_dynamicsymbols(P3.vel(N), reference_frame=N) |
 me.find_dynamicsymbols(P4.vel(N), reference_frame=N))
```

$$\{q_1, q_2, q_3, u_1\} \quad (20.14)$$

We'll also need the kinematical differential equations only in terms of the one generalized speed u_1 , so replace the dependent speeds in \bar{g}_k .

```
gk = gk.xreplace(ur_repl)
```

20.3.7 7. Form the generalized active forces

We have a holonomic system so the number of degrees of freedom is $n = 1$. There are two particles that move and gravity acts on each of them, as a contributing force. The resultant contributing forces on each of the particles are:

```
R_P2 = -m*g*N.y
R_P3 = -m*g*N.y
```

The partial velocities of each particle are easily found for the single generalized speed and \bar{F}_r is:

```
Fr = sm.Matrix([
    P2.vel(N).diff(u1, N).dot(R_P2) + P3.vel(N).diff(u1, N).dot(R_P3)
])
Fr
```

$$\left[-2gl_a m \cos(q_1) - gl_b m \left(1 + \frac{-l_a \sin(q_1) - \frac{l_a(l_b \sin(q_2) + l_c \sin(q_2+q_3)) \sin(q_1+q_2+q_3)}{l_b \sin(q_3)} - l_b \sin(q_1+q_2) - l_c \sin(q_1+q_2+q_3)}{l_b \sin(q_1+q_2) + l_c \sin(q_1+q_2+q_3)} \right) (-\sin(q_1) \sin(q_2) + \cos(q_1) \cos(q_2) \sin(q_3)) \right] \dot{q}_3 \quad (20.15)$$

Check to make sure our generalized active forces do not contain dependent speeds.

```
me.find_dynamicsymbols(Fr)
```

$$\{q_1, q_2, q_3\} \quad (20.16)$$

20.3.8 8. Form the generalized inertia forces

To calculate the generalized inertia forces we need the acceleration of each particle. These should be only functions of \dot{u}_1 , u_1 , and the three coordinates. For P_2 , that is already true:

```
me.find_dynamicsymbols(P2.acc(N), reference_frame=N)
```

$$\{q_1, u_1, \dot{u}_1\} \quad (20.17)$$

but for P_3 we need to make some substitutions:

```
me.find_dynamicsymbols(P3.acc(N), reference_frame=N)
```

$$\{q_1, q_2, q_3, u_1, \dot{q}_1, \dot{q}_2, \dot{q}_3, \dot{u}_1\} \quad (20.18)$$

Knowing that, the inertia resultants can be written as:

```
Rs_P2 = -m*P2.acc(N)
Rs_P3 = -m*P3.acc(N).xreplace(qd_repl).xreplace(ur_repl)
```

and the generalized inertia forces can be formed and we can make sure they are not functions of the dependent speeds.

```
Frs = sm.Matrix([
    P2.vel(N).diff(u1, N).dot(Rs_P2) + P3.vel(N).diff(u1, N).dot(Rs_P3)
])
me.find_dynamicsymbols(Frs)
```

$$\{q_1, q_2, q_3, u_1, \dot{u}_1\} \quad (20.19)$$

20.3.9 9. Equations of motion

Finally, the matrix form of dynamical differential equations is found as we have done before.

```
Md = Frs.jacobian(ud)
gd = Frs.xreplace(ud_zero) + Fr
```

And we can check to make sure the dependent speeds have been eliminated.

```
me.find_dynamicsymbols(Mk), me.find_dynamicsymbols(gk)
```

$$(\{\}, \{q_1, q_2, q_3, u_1\}) \quad (20.20)$$

```
me.find_dynamicsymbols(Md), me.find_dynamicsymbols(gd)
```

$$(\{q_1, q_2, q_3\}, \{q_1, q_2, q_3, u_1\}) \quad (20.21)$$

20.4 Simulate without constraint enforcement

The equations of motion are functions of all three coordinates, yet two of them are dependent on the other. For the evaluation of the right hand side of the equations to be valid, the coordinates must satisfy the holonomic constraints. As presented, Eqs. (20.5) only contain the constraints that the velocity and acceleration of point P_4 must be zero, but the position constraint is not explicitly present. Neglecting the position constraint will cause numerical issues during integration, as we will see.

Create an `eval_rhs(t, x, p)` as we have done before, noting that $\bar{f}_d \in \mathbb{R}^1$.

```
eval_k = sm.lambdify((qN, u, p), (Mk, gk))
eval_d = sm.lambdify((qN, u, p), (Md, gd))

def eval_rhs(t, x, p):
    """Return the derivative of the state at time t.

    Parameters
    ======
    t : float
    x : array_like, shape(4,)
        x = [q1, q2, q3, u1]
    p : array_like, shape(6,)
        p = [la, lb, lc, ln, m, g]

    Returns
    ======
    xd : ndarray, shape(4,)
        xd = [q1d, q2d, q3d, u1d]
```

(continues on next page)

(continued from previous page)

```
"""
qN = x[:3]  # shape(3, )
u = x[3:]  # shape(1, )

Mk, gk = eval_k(qN, u, p)
qNd = -np.linalg.solve(Mk, np.squeeze(gk))

# Md, gd, and ud are each shape(1, 1)
Md, gd = eval_d(qN, u, p)
ud = -np.linalg.solve(Md, gd)[0]

return np.hstack((qNd, ud))

```

Here I select some feasible bar lengths. See the section on the [Grashof condition](#) to learn more about selecting lengths in four-bar linkages.

```
p_vals = np.array([
    0.8,  # la [m]
    2.0,  # lb [m]
    1.0,  # lc [m]
    2.0,  # ln [m]
    1.0,  # m [kg]
    9.81, # g [m/s^2]
])
```

Now we need to generate coordinates that are consistent with the constraints. \bar{f}_h is nonlinear in all of the coordinates. We can solve these equations for the dependent coordinates using numerical [root finding methods](#). SciPy's `fsolve()` function is capable of finding the roots for sets of nonlinear equations, given a good guess.

We'll import `fsolve` directly like so:

```
from scipy.optimize import fsolve
```

`fsolve()` requires a function that evaluates expressions that equal to zero and a guess for the roots of that function, at a minimum. Nonlinear functions will most certainly have multiple solutions for its roots and `fsolve()` will converge to one of the solutions. The better the provided the guess the more likely it will converge on the desired solution. Our function should evaluate the holonomic constraints given the dependent coordinates. We can use `lambdify()` to create this function. I make the first argument \bar{q}_r because these are the values we want to solve for using `fsolve()`.

```
eval_fh = sm.lambdify((qr, q1, p), fh)
```

Now select a desired value for the generalized coordinate q_1 and guesses for q_2 and q_3 .

```
q1_val = np.deg2rad(10.0)
qr_guess = np.deg2rad([10.0, -150.0])
```

`eval_fh()` returns a 2x1 array so a lambda function is used to squeeze the output. q_2 and q_3 that satisfy the constraints are then found with:

```
q2_val, q3_val = fsolve(
    lambda qr, q1, p: np.squeeze(eval_fh(qr, q1, p)),  # squeeze to a 1d array
    qr_guess,  # initial guess for q2 and q3
    args=(q1_val, p_vals)) # known values in fh
```

Now we have values of the coordinates that satisfy the constraints.

```
qN_vals = np.array([q1_val, q2_val, q3_val])
np.rad2deg(qN_vals)
```

```
array([ 10.          ,  6.57526576, -151.3836336 ])
```

We can check that they return zero (or better stated as within `fsolve()`'s tolerance):

```
eval_fh(qN_vals[1:], qN_vals[0], p_vals)
```

```
array([[-2.44249065e-14],
       [ 5.88751270e-13]])
```

Exercise

There are most often multiple solutions for the dependent coordinates for a given value of the dependent coordinates. What are the other possible solutions for these parameter values?

Now that we have consistent coordinates, the initial state vector can be created. We will start at an initial state of rest with $u_1(t_0) = 0$.

```
u1_val = 0.0
x0 = np.hstack((qN_vals, u1_val))
x0
```

```
array([ 0.17453293,  0.11476004, -2.64214284,  0.          ])
```

We will integrate over 30 seconds to show how the constraints hold up over a longer period of time.

```
t0, tf, fps = 0.0, 30.0, 20
```

With consistent coordinates the initial conditions can be set and `eval_rhs()` tested.

```
eval_rhs(t0, x0, p_vals)
```

```
array([ 0.          , -0.          , -0.          , -9.4688079])
```

At every time step in the simulation the holonomic constraints should be satisfied. To check this we will need to evaluate the constraints \bar{f}_h at each time step. The following function does this and returns the *constraint residuals* at each time step.

```
def eval_constraints(xs, p):
    """Returns the value of the left hand side of the holonomic constraints
    at each time instance.

    Parameters
    ======
    xs : ndarray, shape(n, 4)
        States at each of n time steps.
    p : ndarray, shape(6,)
        Constant parameters.

    Returns
    ======
    con : ndarray, shape(n, 2)
```

(continues on next page)

(continued from previous page)

```

    fh evaluated at each xi in xs.

"""
con = []
for xi in xs: # xs is shape(n, 4)
    con.append(eval_fh(xi[1:3], xi[0], p).squeeze())
return np.array(con)

```

The dependent initial conditions need to be solved before each simulation and the constraints evaluated, so it will be helpful to package this process into a reusable function. The following function takes the simulation parameters and returns the simulation results. I have set the integration tolerances explicitly as `rtol=1e-3` and `atol=1e-6`. These happen to be the default tolerances for `solve_ivp()` and we will use three different approaches and we want to make sure the tolerances are set the same for each integration so we can fairly compare the results.

```

def simulate(eval_rhs, t0, tf, fps, q1_0, u1_0, q2_0g, q3_0g, p):
    """Returns the simulation results.

    Parameters
    ======
    eval_rhs : function
        Function that returns the derivatives of the states in the form:
        ``eval_rhs(t, x, p)``.
    t0 : float
        Initial time in seconds.
    tf : float
        Final time in seconds.
    fps : integer
        Number of "frames" per second to output.
    q1_0 : float
        Initial q1 angle in radians.
    u1_0 : float
        Initial u1 rate in radians/s.
    q2_0g : float
        Guess for the initial q2 angle in radians.
    q3_0g : float
        Guess for the initial q3 angle in radians.
    p : array_like, shape(6,)
        Constant parameters p = [la, lb, lc, ln, m, g].

    Returns
    ======
    ts : ndarray, shape(n,)
        Time values.
    xs : ndarray, shape(n, 4)
        State values at each time.
    con : ndarray, shape(n, 2)
        Constraint violations at each time in meters.

"""
# generate the time steps
ts = np.linspace(t0, tf, num=int(fps*(tf - t0)))

# solve for the dependent coordinates
q2_val, q3_val = fsolve(
    lambda qr, q1, p: np.squeeze(eval_fh(qr, q1, p)),

```

(continues on next page)

(continued from previous page)

```

[q2_0g, q3_0g],
args=(q1_0, p))

# establish the initial conditions
x0 = np.array([q1_val, q2_val, q3_val, u1_0])

# integrate the equations of motion
sol = solve_ivp(eval_rhs, (ts[0], ts[-1]), x0, args=(p,), t_eval=ts,
                rtol=1e-3, atol=1e-6)
xs = np.transpose(sol.y)
ts = sol.t

# evaluate the constraints
con = eval_constraints(xs, p)

return ts, xs, con

```

Similarly, create a function that can be reused for plotting the state trajectories and the constraint residuals.

```

def plot_results(ts, xs, con):
    """Returns the array of axes of a 4 panel plot of the state trajectory
    versus time.

    Parameters
    =====
    ts : array_like, shape(n, )
        Values of time.
    xs : array_like, shape(n, 4)
        Values of the state trajectories corresponding to ``ts`` in order
        [q1, q2, q3, u1].
    con : array_like, shape(n, 2)
        x and y constraint residuals of P4 at each time in ``ts``.

    Returns
    =====
    axes : ndarray, shape(3,)
        Matplotlib axes for each panel.

    """
    fig, axes = plt.subplots(3, 1, sharex=True)

    fig.set_size_inches((10.0, 6.0))

    axes[0].plot(ts, np.rad2deg(xs[:, :3])) # q1(t), q2(t), q3(t)
    axes[1].plot(ts, np.rad2deg(xs[:, 3])) # u1(t)
    axes[2].plot(ts, np.squeeze(con)) # fh(t)

    axes[0].legend(['$q_1$', '$q_2$', '$q_3$'])
    axes[1].legend(['$u_1$'])
    axes[2].legend(['$r' '\cdot \hat{x}$', '$r' '\cdot \hat{y}$'])

    axes[0].set_ylabel('Angle [deg]')
    axes[1].set_ylabel('Angular Rate [deg/s]')
    axes[2].set_ylabel('Distance [m]')
    axes[2].set_xlabel('Time [s]')

    fig.tight_layout()

```

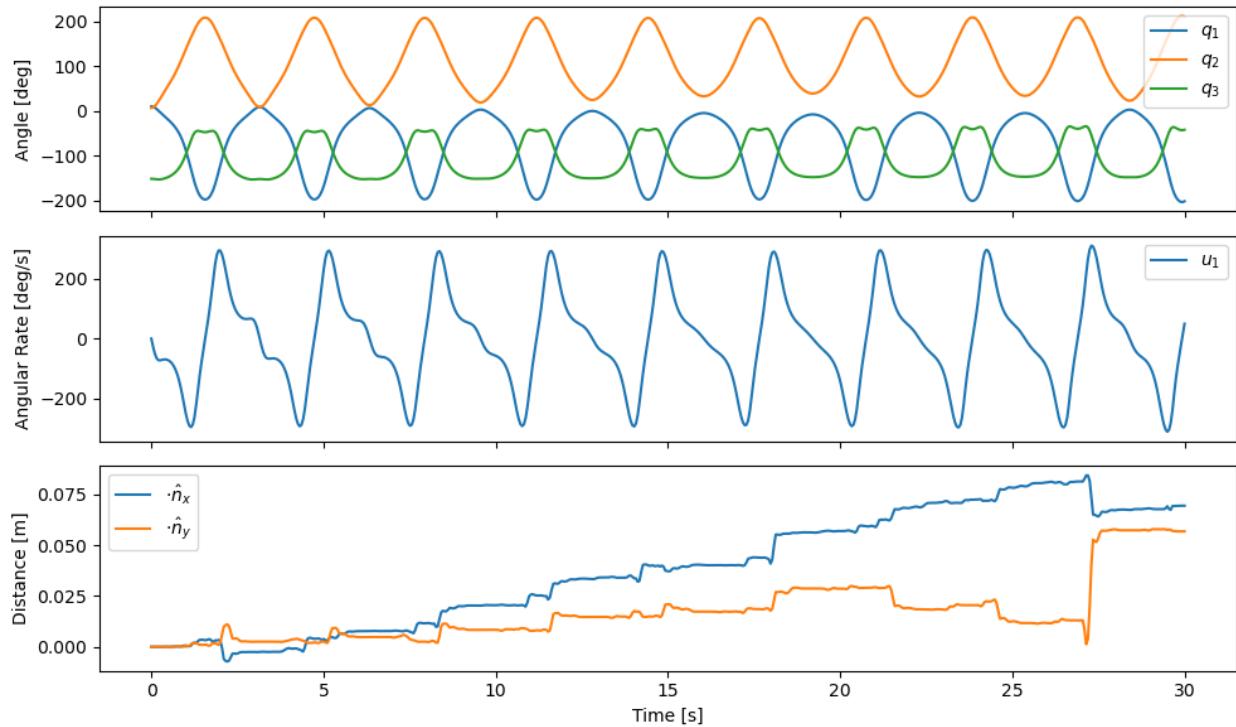
(continues on next page)

(continued from previous page)

```
    return axes
```

With the functions in place we can simulate the system and plot the results.

```
ts, xs, con = simulate(
    eval_rhs,
    t0=t0,
    tf=tf,
    fps=fps,
    q1_0=np.deg2rad(10.0),
    u1_0=0.0,
    q2_0g=np.deg2rad(10.0),
    q3_0g=np.deg2rad(-150.0),
    p=p_vals,
)
plot_results(ts, xs, con);
```



At first glance, the linkage seems to simulate fine with realistic angle values and angular rates. The motion is periodic but looking closely, for example at $u_1(t)$, you can see that the angular rate changes in each successive period. The last graph shows the holonomic constraint residuals across time. This graph shows that the constraints are satisfied at the beginning of the simulation but that the residuals grow over time. This accumulation of error grows as large as 8 cm near the end of the simulation. The drifting constraint residuals are the cause of the variations of motion among the oscillation periods. Tighter integration tolerances can reduce the drifting constraint residuals, but that will come at an unnecessary computational cost and not fully solve the issue.

The effect of the constraints not staying satisfied throughout the simulation can also be seen if the system is animated.

20.5 Animate the Motion

We'll animate the four bar linkage multiple times so it is useful to create some functions to for the repeated use. Start by creating a function that evaluates the point locations, as we have done before.

```
coordinates = P2.pos_from(P1).to_matrix(N)
for point in [P3, P4, P1, P2]:
    coordinates = coordinates.row_join(point.pos_from(P1).to_matrix(N))
eval_point_coords = sm.lambdify((qN, p), coordinates)
```

Now create a function that plots the initial configuration of the linkage and returns any objects we may need in the animation code.

```
def setup_animation_plot(ts, xs, p):
    """Returns objects needed for the animation.

    Parameters
    ==========
    ts : array_like, shape(n, )
        Values of time.
    xs : array_like, shape(n, 4)
        Values of the state trajectories corresponding to ``ts`` in order
        [q1, q2, q3, u1].
    p : array_like, shape(6, )

    """
    x, y, z = eval_point_coords(xs[0, :3], p)

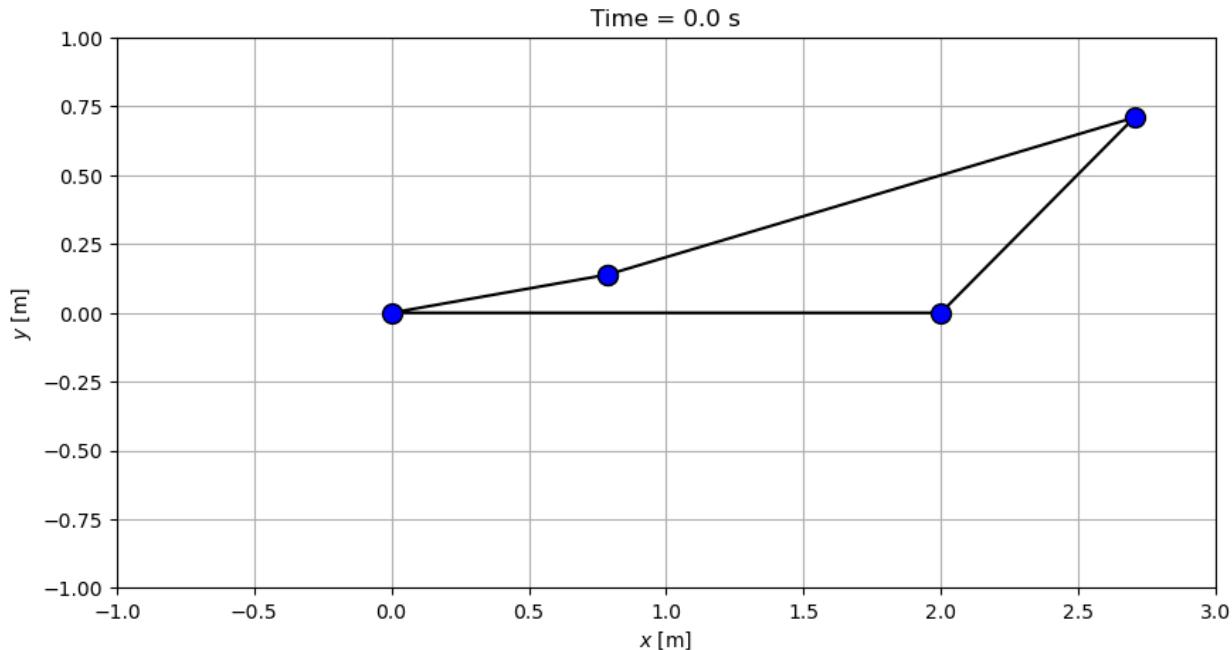
    fig, ax = plt.subplots()
    fig.set_size_inches((10.0, 10.0))
    ax.set_aspect('equal')
    ax.grid()

    lines, = ax.plot(x, y, color='black',
                      marker='o', markerfacecolor='blue', markersize=10)

    title_text = ax.set_title('Time = {:.1f} s'.format(ts[0]))
    ax.set_xlim((-1.0, 3.0))
    ax.set_ylim((-1.0, 1.0))
    ax.set_xlabel('$x$ [m]')
    ax.set_ylabel('$y$ [m]')

    return fig, ax, title_text, lines

setup_animation_plot(ts, xs, p_vals);
```



Now we can create a function that initializes the plot, runs the animation and displays the results in Jupyter.

```
def animate_linkage(ts, xs, p):
    """Returns an animation object.

    Parameters
    ======
    ts : array_like, shape(n, )
    xs : array_like, shape(n, 4)
        x = [q1, q2, q3, u1]
    p : array_like, shape(6, )
        p = [la, lb, lc, ln, m, g]

    """
    # setup the initial figure and axes
    fig, ax, title_text, lines = setup_animation_plot(ts, xs, p)

    # precalculate all of the point coordinates
    coords = []
    for xi in xs:
        coords.append(eval_point_coords(xi[:3], p))
    coords = np.array(coords)

    # define the animation update function
    def update(i):
        title_text.set_text('Time = {:.1f} s'.format(ts[i]))
        lines.set_data(coords[i, 0, :], coords[i, 1, :])

    # close figure to prevent premature display
    plt.close()

    # create and return the animation
    return FuncAnimation(fig, update, len(ts))
```

Now, keep an eye on P_4 during the animation of the simulation.

```
HTML(animate_linkage(ts, xs, p_vals).to_jshtml(fps=fps))
```

```
<IPython.core.display.HTML object>
```

20.6 Correct Dependent Coordinates

Above we are relying on the integration of the differential equations to generate the coordinates. Because there is accumulated integration error in each state and nothing is enforcing the constraint among the coordinates, the constraint residuals grow with time and the point P_4 drifts from its actual location. One possible way to address this is to correct the dependent coordinates at each evaluation of the state derivatives. We can use `fsolve()` to do so, in the same way we solved for the initial conditions. Below, I force the dependent coordinates to satisfy the constraints to the default tolerance of `fsolve()` as the first step in `eval_rhs()`.

```
def eval_rhs_fsolve(t, x, p):
    """Return the derivative of the state at time t.

    Parameters
    =====
    t : float
    x : array_like, shape(4,)
        x = [q1, q2, q3, u1]
    p : array_like, shape(6,)
        p = [la, lb, lc, ln, m, g]

    Returns
    =====
    xd : ndarray, shape(4,)
        xd = [q1d, q2d, q3d, u1d]

    Notes
    =====

    Includes a holonomic constraint correction.

    """
    qN = x[:3]
    u = x[3:]

    # correct the dependent coordinates
    qN[1:] = fsolve(lambda qr, q1, p: np.squeeze(eval_fh(qr, q1, p)),
                    qN[1:], # guess with current solution for q2 and q3
                    args=(qN[0], p_vals))

    Mk, gk = eval_k(qN, u, p)
    qNd = -np.linalg.solve(Mk, np.squeeze(gk))

    Md, gd = eval_d(qN, u, p)
    ud = -np.linalg.solve(Md, gd)[0]

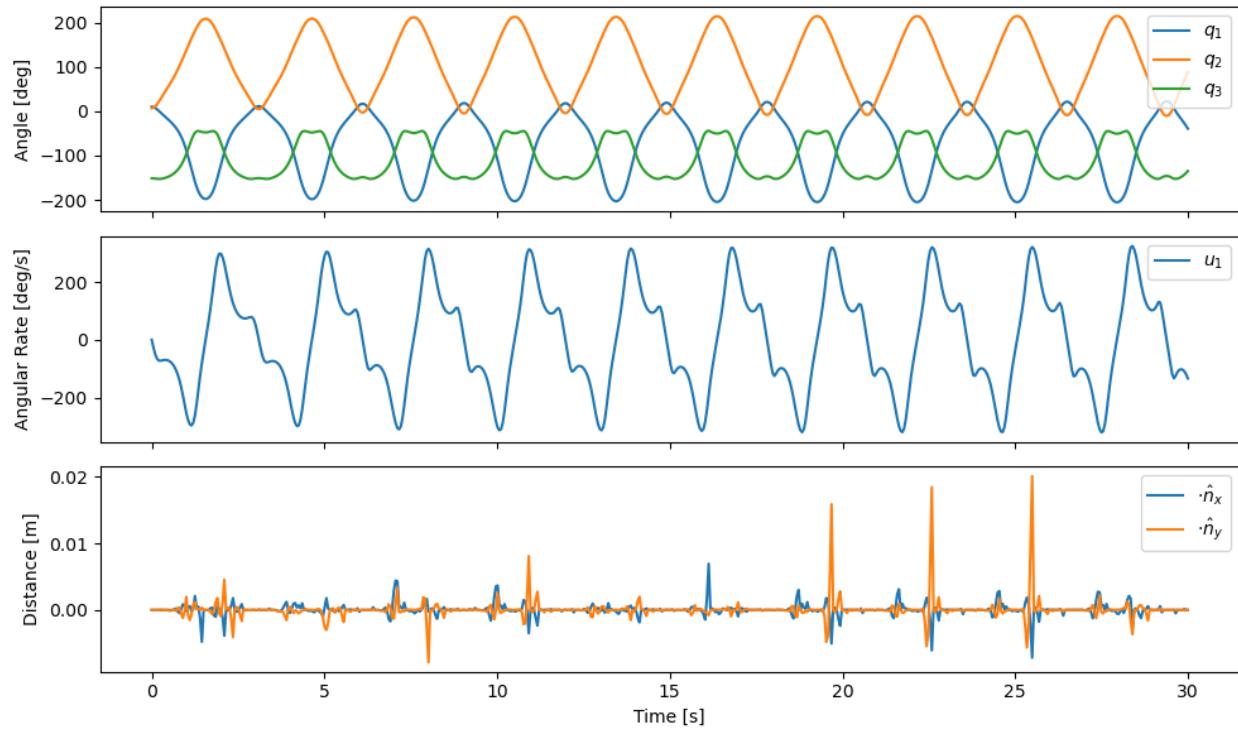
    return np.hstack((qNd, ud))
```

Now we can simulate with the same integrator tolerances and see if it improves the results.

```

ts_fsolve, xs_fsolve, con_fsolve = simulate(
    eval_rhs_fsolve,
    t0=t0,
    tf=tf,
    fps=fps,
    q1_0=np.deg2rad(10.0),
    u1_0=0.0,
    q2_0g=np.deg2rad(20.0),
    q3_0g=np.deg2rad(-150.0),
    p=p_vals,
)
plot_results(ts_fsolve, xs_fsolve, con_fsolve);

```



```
HTML(animate_linkage(ts_fsolve, xs_fsolve, p_vals).to_jshtml(fps=fps))
```

```
<IPython.core.display.HTML object>
```

This result is much improved. The motion is more consistently periodic and the constraint residuals do not grow over time. The constraint violations do reach large values at some times but tighter integration tolerances can bring those down in magnitude. Looking closely at the trajectory of q_2 , you see that the solution drifts to increasingly negative minima, so this solution still has weaknesses. Another potential downside of this approach is that `fsolve()` can be a computationally costly function to run depending on the complexity of the constraints and the desired solver tolerances. Fortunately, there are dedicated differential algebraic equation solvers that apply more efficient and accurate numerical methods to maintain the constraints in the initial value problem.

20.7 Simulate Using a DAE Solver

In the prior simulation, we numerically solved for q_2 and q_3 at each time step to provide a correction to those two variables. This can be effective with tight integration tolerances, but is still a computationally naive approach. There are more robust and efficient numerical methods for correcting the state variables at each time step. For example, the [SUNDIALS](#) library includes the [IDA](#) solver for solving the initial value problem of a set of differential algebraic equations. IDA uses a variation of an implicit backward differentiation method (similar to those offered in `solve_ivp()`) but efficiently handles the algebraic constraints. IDA is written in C and [scikits.odes](#) provides a Python interface to many SUNDIALS solvers, including IDA.

To use [scikits.odes](#)'s differential algebraic solver, we need to write the equations of motion in implicit form. We now can write the equations of motion of a holonomic system with M holonomic constraints and n degrees of freedom as this minimal set of equations:

$$\begin{aligned}\bar{f}_k(\dot{\bar{q}}, \bar{u}, \bar{q}, \bar{q}_r, t) &= 0 \in \mathbb{R}^n \\ \bar{f}_d(\dot{\bar{u}}, \bar{u}, \bar{q}, \bar{q}_r, t) &= 0 \in \mathbb{R}^n \\ \bar{f}_h(\bar{q}, \bar{q}_r, t) &= 0 \in \mathbb{R}^M\end{aligned}\tag{20.22}$$

Note the reduced kinematical differential equation from our prior implementations, i.e. we will not find \bar{q}_r from integration alone. This gives $2n + M$ equations in $2n + M$ state variables $\bar{u}, \bar{q}, \bar{q}_r$.

The [scikits.odes](#) `dae()` function is similar to `solve_ivp()` but has various other options and a different solution output. `dae()` works with the explicit form of the equations, exactly as shown in Eq. (20.22). We need to build a function that returns the left hand side of the equations and we will call the output of those equations the “residual”, which should equate to zero at all times.

We will import the `dae` function directly, as that is all we need from [scikits.odes](#).

```
from scikits.odes import dae
```

We now need to design a function that evaluates the left hand side of Eq. (20.22) and it needs to have a specific function signature. In addition to the arguments in `eval_rhs()` above, this function needs the time derivative of the states and a vector to store the result in.

Note: `eval_eom()` does not return a value. It only sets the individual values in the `residual` array. So if you run `eval_eom()` and check `residual` you will see it has changed.

```
def eval_eom(t, x, xd, residual, p):
    """Returns the residual vector of the equations of motion.

    Parameters
    ==========
    t : float
        Time at evaluation.
    x : ndarray, shape(4,)
        State vector at time t: x = [q1, q2, q3, u1].
    xd : ndarray, shape(4,)
        Time derivative of the state vector at time t: xd = [q1d, q2d, q3d, u1d].
    residual : ndarray, shape(4,)
        Vector to store the residuals in: residuals = [fk, fd, fh1, fh2].
    p : ndarray, shape(6,)
        Constant parameters: p = [la, lb, lc, ln, m, g]

    """

```

(continues on next page)

(continued from previous page)

```

q1, q2, q3, u1 = x
q1d, __, __, u1d = xd  # ignore the q2d and q3d values

Md, gd = eval_d([q1, q2, q3], [u1], p)

residual[0] = -q1d + u1  # fk, float
residual[1] = Md[0]*u1d + gd[0]  # fd, float
residual[2:] = eval_fh([q2, q3], [q1], p).squeeze()  # fh, shape(2,)
```

We already have the initial state defined `x0`, but we need to initialize the time derivatives of the states. These must be consistent with the equations of motion, including the constraints. In our case, $u_1 = 0$ so \dot{q}_1, \dot{q}_2 and \dot{q}_3 will also be zero. But we do need to solve \bar{f}_d for the initial \dot{u}_1 .

```

Md_vals, gd_vals = eval_d(x0[:3], x0[3:], p_vals)

xd0 = np.array([
    0.0,  # q1d [rad/s]
    0.0,  # q2d [rad/s]
    0.0,  # q3d [rad/s]
    -np.linalg.solve(Md_vals, gd_vals)[0][0],  # u1d [rad/s^2]
])
xd0
```

```
array([ 0.        ,  0.        ,  0.        , -9.4688079])
```

Now I'll create an empty array to store the residual results in using `empty()`.

```
residual = np.empty(4)
residual
```

```
array([ 0.        ,  0.        ,  0.        , -9.4688079])
```

With all of the arguments for `eval_eom()` prepared, we can see if it updates the residual properly. We should get a residual of approximately zero if we've set consistent initial conditions.

```
eval_eom(t0, x0, xd0, residual, p_vals)
residual
```

```
array([ 0.00000000e+00,  0.00000000e+00, -2.44249065e-14,  5.88751270e-13])
```

It looks like our functions works! Now we can integrate the differential algebraic equations with the IDA integrator. We first initialize a solver with the desired integrator parameters. I've set `rtol` and `atol` to be the same size as our prior integrations. The `algebraic_vars_idx` argument is used to indicate which indices of `residual` correspond to the holonomic constraints. Lastly, `old_api` is set to false to use the newest solution outputs from `scikits.odes`.

```

solver = dae('ida',
             eval_eom,
             rtol=1e-3,
             atol=1e-6,
             algebraic_vars_idx=[2, 3],
             user_data=p_vals,
             old_api=False)
```

To find a solution, the desired time array and the initial conditions are provided to `.solve()`. The time and state values are stored in `.values.t` and `.values.y`.

```

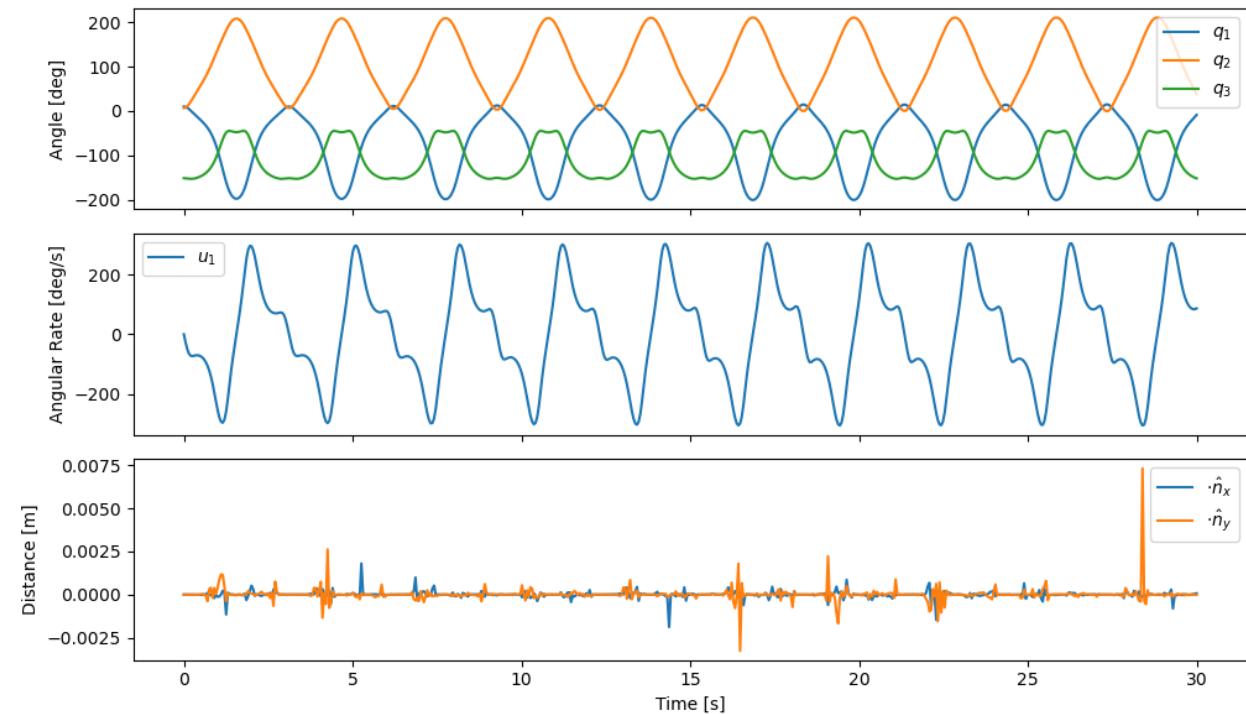
solution = solver.solve(ts, x0, xd0)

ts_dae = solution.values.t
xs_dae = solution.values.y
con_dae = eval_constraints(xs_dae, p_vals)

```

Now we can have a look at the results. The constraints are held to the order we specified in the integrator options.

```
plot_results(ts_dae, xs_dae, con_dae);
```



```
HTML(animate_linkage(ts_dae, xs_dae, p_vals).to_jshtml(fps=fps))
```

```
<IPython.core.display.HTML object>
```

With the same integration tolerances as we used in the two prior simulations, IDA keeps the constraint residuals under 8 mm for the duration of the simulation. This is an order of magnitude better than our prior approach.

Knowing that the IDA solution is better than the prior two solutions, we can compare them directly. Below I plot the trajectory of u_1 from each of the integration methods. This clearly shows the relative error in the solutions which both become quite large over time.

```

fig, ax = plt.subplots()
fig.set_size_inches(10.0, 6.0)

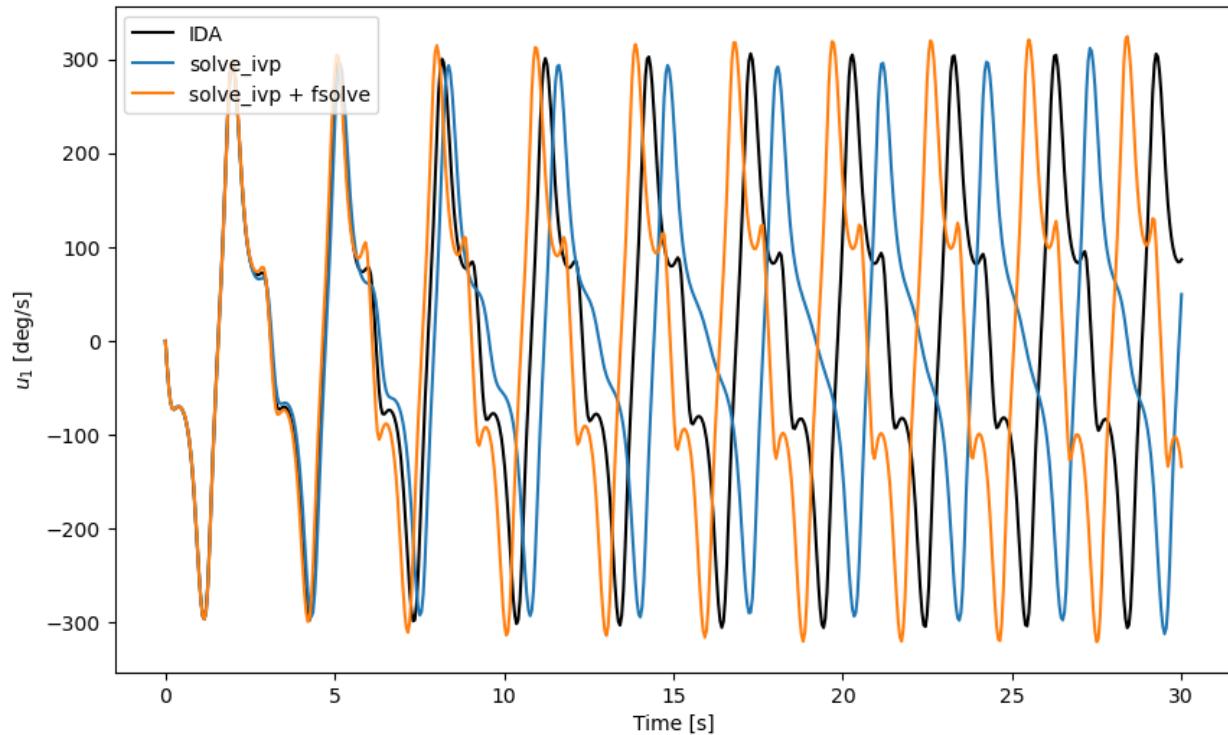
ax.plot(
    ts_dae, np.rad2deg(xs_dae[:, -1]), 'black',
    ts, np.rad2deg(xs[:, -1]), 'C0',
    ts_fsolve, np.rad2deg(xs_fsolve[:, -1]), 'C1',
)
ax.set_xlabel('Time [s]')
ax.set_ylabel('$u_1$ [deg/s]')

```

(continues on next page)

(continued from previous page)

```
ax.legend(['IDA', 'solve_ivp', 'solve_ivp + fsolve']);
```



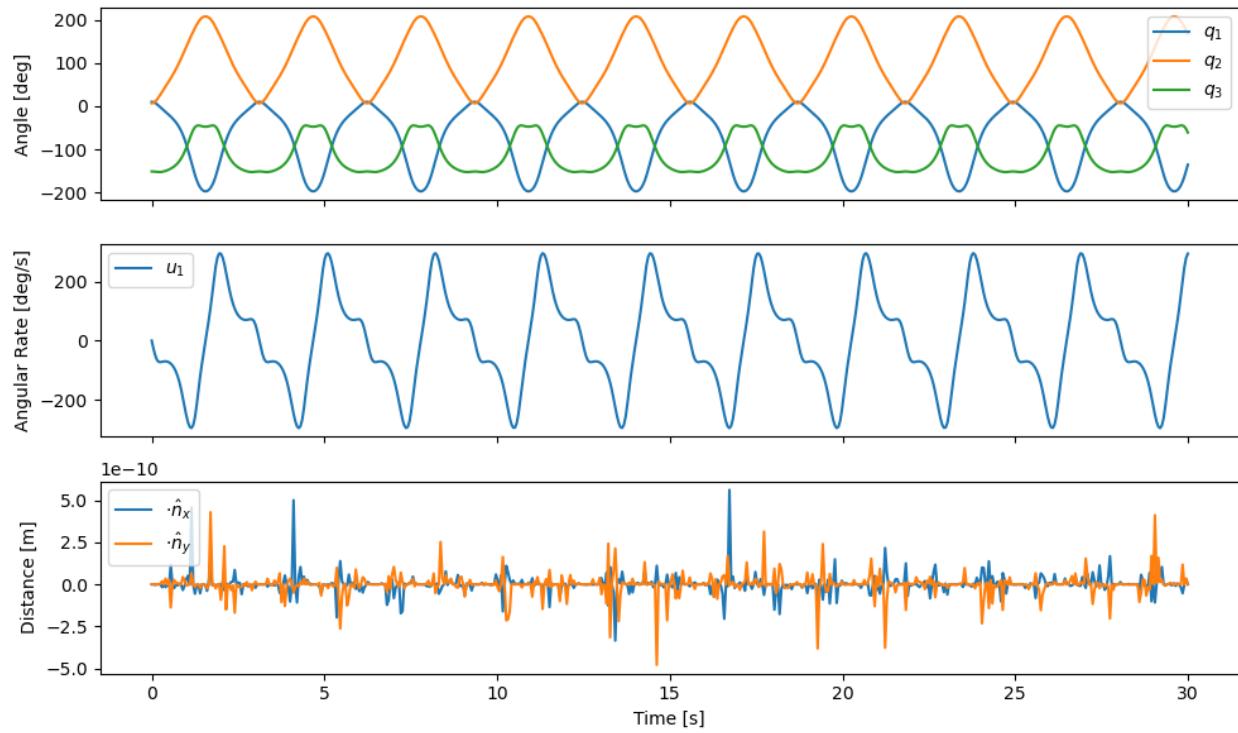
The constraints and integration error can be enforced to tighter tolerances. With `rtol` and `atol` set to `1e-10` the constraint residuals stay below `5e-10` meters for this simulation and a consistent periodic solution is realized.

```
solver = dae('ida',
              eval_eom,
              rtol=1e-10,
              atol=1e-10,
              algebraic_vars_idx=[2, 3],
              user_data=p_vals,
              old_api=False)

solution = solver.solve(ts, x0, xd0)

ts_dae = solution.values.t
xs_dae = solution.values.y
con_dae = eval_constraints(xs_dae, p_vals)

plot_results(ts_dae, xs_dae, con_dae);
```



EXPOSING NONCONTRIBUTING FORCES

Note: You can download this example as a Python script: `noncontributing.py` or Jupyter Notebook: `noncontributing.ipynb`.

```
import numpy as np
import sympy as sm
import sympy.physics.mechanics as me
me.init_vprinting(use_latex='mathjax')
```

```
class ReferenceFrame(me.ReferenceFrame):
    def __init__(self, *args, **kwargs):
        kwargs.pop('latexs', None)
        lab = args[0].lower()
        tex = r'\hat{{}}_{{}}_{{}}'
        super(ReferenceFrame, self).__init__(*args,
                                             latexs=(tex.format(lab, 'x'),
                                                      tex.format(lab, 'y'),
                                                      tex.format(lab, 'z')),
                                             **kwargs)
me.ReferenceFrame = ReferenceFrame
```

21.1 Learning Objectives

After completing this chapter readers will be able to:

- apply Newton's Second Law to write equations of motion with maximal coordinates, which naturally expose non-contributing forces
- use the auxiliary generalized speed method to expose noncontributing forces in Kane's minimal coordinate formulation

21.2 Introduction

Kane's formulation relieves us from having to consider noncontributing forces (See Sec. [Contributing and Noncontributing Forces](#)), but often we are interested in one or more of these noncontributing forces. In this chapter, I will show how you can find the equation for a noncontributing force by introducing *auxiliary generalized speeds*. But first, let's solve the equations of motion for a system by directly applying Newton's Second Law of motion, which requires us to explicitly define all contributing and noncontributing forces.

21.3 Double Pendulum Example

Fig. 21.1 shows a schematic of a simple planar double pendulum described by two generalized coordinates q_1 and q_2 . The particles P_1 and P_2 have masses m_1 and m_2 , respectively. The lengths of the first and second pendulum arms are l_1 and l_2 , respectively. On the right, the free body diagrams depict the two tension forces T_1 and T_2 that act on each particle to keep them at their respective radial locations.

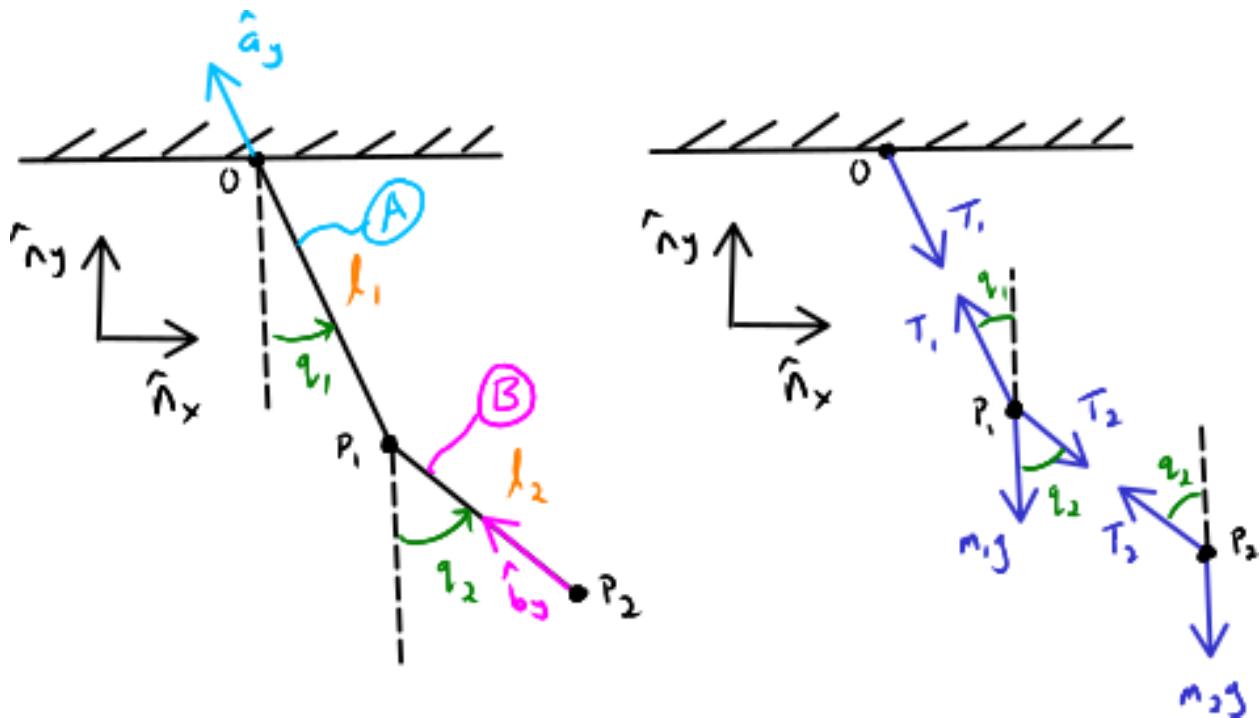


Fig. 21.1: On the left, a kinematic diagram of a simple double planar pendulum with two links A of length l_1 and B of length l_2 . On the right are free body diagrams of each particle showing all of the contributing and noncontributing forces acting on them. Gravity acts in the $-\hat{n}_y$ direction.

Start by creating all of the necessary variables. The tension forces are time varying quantities.

```

m1, m2, l1, l2, g = sm.symbols('m1, m2, l1, l2, g')
q1, q2, u1, u2, T1, T2 = me.dynamicsymbols('q1, q2, u1, u2, T1, T2')
t = me.dynamicsymbols._t

p = sm.Matrix([m1, m2, l1, l2, g])
q = sm.Matrix([q1, q2])
u = sm.Matrix([u1, u2])

```

(continues on next page)

(continued from previous page)

```
r = sm.Matrix([T1, T2])
ud = u.diff(t)
p, q, u, r, ud
```

$$\left(\begin{bmatrix} m_1 \\ m_2 \\ l_1 \\ l_2 \\ g \end{bmatrix}, \begin{bmatrix} q_1 \\ q_2 \end{bmatrix}, \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}, \begin{bmatrix} T_1 \\ T_2 \end{bmatrix}, \begin{bmatrix} \dot{u}_1 \\ \dot{u}_2 \end{bmatrix} \right) \quad (21.1)$$

Both pendulums' configuration are described by angles relative to the vertical direction. We will choose the generalized speeds to be $\bar{u} = \dot{q}$ and set the angular velocities to be in terms of them.

```
N = me.ReferenceFrame('N')
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')

A.orient_axis(N, q1, N.z)
B.orient_axis(N, q2, N.z)

A.set_ang_vel(N, u1*N.z)
B.set_ang_vel(N, u2*N.z)
```

Now the positions, velocities, and accelerations of each particle can be formed.

```
O = me.Point('O')
P1 = O.locatenew('P1', -l1*A.y)
P2 = P1.locatenew('P2', -l2*B.y)

O.set_vel(N, 0)
P1.v2pt_theory(O, N, A)
```

$$l_1 u_1 \hat{a}_x \quad (21.2)$$

```
P2.v2pt_theory(P1, N, B)
```

$$l_1 u_1 \hat{a}_x + l_2 u_2 \hat{b}_x \quad (21.3)$$

```
P1.a2pt_theory(O, N, A)
```

$$l_1 \dot{u}_1 \hat{a}_x + l_1 u_1^2 \hat{a}_y \quad (21.4)$$

P2.a2pt_theory(P1, N, B)

$$l_1\dot{u}_1\hat{a}_x + l_1u_1^2\hat{a}_y + l_2\dot{u}_2\hat{b}_x + l_2u_2^2\hat{b}_y \quad (21.5)$$

All of the kinematics are strictly in terms of the generalized coordinates and the generalized speeds.

21.4 Apply Newton's Second Law Directly

Direction application of Newton's Second Law can be done if *all* of the forces (noncontributing and contributing) are described for each of the two particles. Vector equations representing the law for each particle are:

$$\begin{aligned} \sum \bar{F}^{P_1} &= m_1^N \bar{a}^{P_1} \\ \sum \bar{F}^{P_2} &= m_2^N \bar{a}^{P_2} \end{aligned} \quad (21.6)$$

From the free body diagram (Fig. 21.1) we see that all of the forces acting on P_1 are:

F_P1 = T1*A.y - T2*B.y - m1*g*N.y
F_P1.express(N)

$$(-T_1 \sin(q_1) + T_2 \sin(q_2))\hat{n}_x + (-gm_1 + T_1 \cos(q_1) - T_2 \cos(q_2))\hat{n}_y \quad (21.7)$$

and all of the forces acting on P_2 are:

F_P2 = T2*B.y - m2*g*N.y
F_P2.express(N)

$$-T_2 \sin(q_2)\hat{n}_x + (-gm_2 + T_2 \cos(q_2))\hat{n}_y \quad (21.8)$$

Now we can form the two vector expressions of Newton's Second Law for each particle. Moving everything to the right hand side gives:

$$\begin{aligned} \bar{0} &= \sum \bar{F}^{P_1} - m_1^N \bar{a}^{P_1} \\ \bar{0} &= \sum \bar{F}^{P_2} - m_2^N \bar{a}^{P_2} \end{aligned} \quad (21.9)$$

zero_P1 = F_P1 - m1*p1.acc(N)
zero_P2 = F_P2 - m2*p2.acc(N)

These two planar vector equations can then be written as four scalar equations by extracting the \hat{n}_x and \hat{n}_y measure numbers.

```
fd = sm.Matrix([
    zero_P1.dot(N.x),
    zero_P1.dot(N.y),
    zero_P2.dot(N.x),
    zero_P2.dot(N.y),
])
fd
```

$$\begin{bmatrix} -l_1 m_1 \cos(q_1) \dot{u}_1 - (-l_1 m_1 u_1^2 + T_1) \sin(q_1) + T_2 \sin(q_2) \\ -g m_1 - l_1 m_1 \sin(q_1) \dot{u}_1 + (-l_1 m_1 u_1^2 + T_1) \cos(q_1) - T_2 \cos(q_2) \\ l_1 m_2 u_1^2 \sin(q_1) - l_1 m_2 \cos(q_1) \dot{u}_1 - l_2 m_2 \cos(q_2) \dot{u}_2 - (-l_2 m_2 u_2^2 + T_2) \sin(q_2) \\ -g m_2 - l_1 m_2 u_1^2 \cos(q_1) - l_1 m_2 \sin(q_1) \dot{u}_1 - l_2 m_2 \sin(q_2) \dot{u}_2 + (-l_2 m_2 u_2^2 + T_2) \cos(q_2) \end{bmatrix} \quad (21.10)$$

It is important to note that these scalar equations are linear in both the time derivatives of the generalized speeds \dot{u}_1, \dot{u}_2 as well as the two noncontributing force magnitudes T_1, T_2 and that all four equations are coupled in these four variables.

```
(me.find_dynamicsymbols(fd[0]), me.find_dynamicsymbols(fd[1]),
 me.find_dynamicsymbols(fd[2]), me.find_dynamicsymbols(fd[3]))
```

$$(\{T_1, T_2, q_1, q_2, u_1, \dot{u}_1\}, \{T_1, T_2, q_1, q_2, u_1, \dot{u}_1\}, \{T_2, q_1, q_2, u_1, u_2, \dot{u}_1, \dot{u}_2\}, \{T_2, q_1, q_2, u_1, u_2, \dot{u}_1, \dot{u}_2\}) \quad (21.11)$$

That means we can write the equations as:

$$\bar{f}_d(\dot{\bar{u}}, \bar{q}, \bar{r}, t) = \mathbf{M}_d \begin{bmatrix} \dot{\bar{u}} \\ \bar{r} \end{bmatrix} + \bar{g}_d \quad (21.12)$$

where $\bar{r} = [T_1 \ T_2]^T$. The linear coefficient matrix and the remainder can be extracted as usual:

```
ud, r
```

$$\left(\begin{bmatrix} \dot{u}_1 \\ \dot{u}_2 \end{bmatrix}, \begin{bmatrix} T_1 \\ T_2 \end{bmatrix} \right) \quad (21.13)$$

```
udr = ud.col_join(r)
udr_zero = {v: 0 for v in udr}

Md = fd.jacobian(udr)
gd = fd.xreplace(udr_zero)

Md, udr, gd
```

$$\begin{pmatrix} -l_1 m_1 \cos(q_1) & 0 & -\sin(q_1) & \sin(q_2) \\ -l_1 m_1 \sin(q_1) & 0 & \cos(q_1) & -\cos(q_2) \\ -l_1 m_2 \cos(q_1) & -l_2 m_2 \cos(q_2) & 0 & -\sin(q_2) \\ -l_1 m_2 \sin(q_1) & -l_2 m_2 \sin(q_2) & 0 & \cos(q_2) \end{pmatrix}, \begin{bmatrix} \dot{u}_1 \\ \dot{u}_2 \\ T_1 \\ T_2 \end{bmatrix}, \begin{bmatrix} l_1 m_1 u_1^2 \sin(q_1) \\ -g m_1 - l_1 m_1 u_1^2 \cos(q_1) \\ l_1 m_2 \dot{u}_1^2 \sin(q_1) + l_2 m_2 u_2^2 \sin(q_2) \\ -g m_2 - l_1 m_2 u_1^2 \cos(q_1) - l_2 m_2 u_2^2 \cos(q_2) \end{bmatrix} \quad (21.14)$$

The four equations are fully coupled, so we must solve for the four variables simultaneously. When applying Newton's Second Law directly, additional coupled equations for each noncontributing force are necessary to solve the dynamical differential equations. When formulating the equations with Kane's method, similar equations for the noncontributing forces can be generated, but the noncontributing forces will remain absent from the dynamical differential equations.

21.5 Auxiliary Generalized Speeds

When we form Kane's equations, noncontributing forces will not be present in the equations of motion as they are above in the classical Newton formulation, but it is possible to expose select noncontributing forces by taking advantage of the role of the partial velocities. Forces and torques that are not normal to the partial velocity will contribute to the equations of motion. It is then possible to introduce fictitious partial velocities via an auxiliary generalized speed, along with a force or torque that acts in the same direction of the fictitious motion to generate extra equations for the noncontributing forces or torques. See [Kane1985] pg. 114 for more explanation of this idea.

As an example, here I introduce two fictitious generalized speeds, u_3 and u_4 that lets each particle have motion relative to its fixed location on the pendulum arm in the direction of the two noncontributing forces that we desire to know. Fig. 21.2 shows the two additional speeds and the associated forces. We introduce these speeds without introducing any related generalized coordinates.

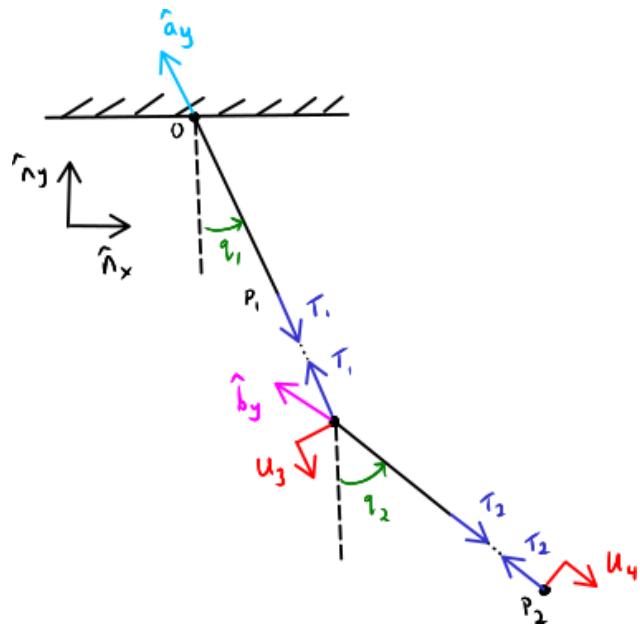


Fig. 21.2: Kinematic diagram of the double pendulum showing the fictitious auxiliary generalized speeds u_3 and u_4 and the associated contributing forces.

First find the velocity of P_1 with the additional velocity component and store this separately in N_v_P1a to indicate it is affected by this auxiliary generalized speed.

```
u3, u4 = me.dynamicsymbols('u3, u4')
N_v_P1a = P1.vel(N) - u3*A.y
N_v_P1a
```

$$l_1 u_1 \hat{a}_x - u_3 \hat{a}_y \quad (21.15)$$

Similarly, write the velocity of P_2 using the velocity two point theorem and adding the auxiliary component. Note that the pendulum arm does not change in length because we have not added any generalized coordinates, so the two auxiliary velocities can be simply added in each step.

```
N_v_P2a = N_v_P1a + me.cross(B.ang_vel_in(N), P2.pos_from(P1)) - u4*B.y
N_v_P2a
```

$$l_1 u_1 \hat{a}_x - u_3 \hat{a}_y + l_2 u_2 \hat{b}_x - u_4 \hat{b}_y \quad (21.16)$$

These two velocities will be used to generate the partial velocities for two additional generalized active forces and generalized inertia forces, one for each of the auxiliary generalized speeds u_3 and u_4 .

21.6 Auxiliary Generalized Active Forces

We now have four generalized speeds, two of which are auxiliary generalized speeds. With these speeds we will formulate four generalized active forces. The generalized active forces associated with u_1 and u_2 are no different than if we were not exposing the noncontributing forces, so we follow the usual procedure.

```
R_P1 = -m1*g*N.y
R_P2 = -m2*g*N.y
```

```
F1 = P1.vel(N).diff(u1, N).dot(R_P1) + P2.vel(N).diff(u1, N).dot(R_P2)
F1
```

$$-gl_1 m_1 \sin(q_1) - gl_1 m_2 \sin(q_1) \quad (21.17)$$

```
F2 = P1.vel(N).diff(u2, N).dot(R_P1) + P2.vel(N).diff(u2, N).dot(R_P2)
F2
```

$$-gl_2 m_2 \sin(q_2) \quad (21.18)$$

For F_3 and F_4 , the contributing forces we wish to know that are associated with the auxiliary generalized speeds are added to the resultant acting on the two particles.

```
R_P1_aux = R_P1 + T1*A.y - T2*B.y
R_P2_aux = R_P2 + T2*B.y
```

Now the velocities of the particles that include the auxiliary generalized speeds are used to calculate the partial velocities and the auxiliary generalized active forces are formed.

```
F3 = N_v_P1a.diff(u3, N).dot(R_P1_aux) + N_v_P2a.diff(u3, N).dot(R_P2_aux)
F3
```

$$gm_1 \cos(q_1) + gm_2 \cos(q_1) - T_1 \quad (21.19)$$

```
F4 = N_v_P1a.diff(u4, N).dot(R_P1_aux) + N_v_P2a.diff(u4, N).dot(R_P2_aux)
F4
```

$$gm_2 \cos(q_2) - T_2 \quad (21.20)$$

Finally, we form \bar{F}_r that consists of the two normal generalized active forces and the two auxiliary generalized active forces, the later two containing the unknown force magnitudes T_1 and T_2 .

```
Fr = sm.Matrix([F1, F2, F3, F4])
Fr
```

$$\begin{bmatrix} -gl_1 m_1 \sin(q_1) - gl_1 m_2 \sin(q_1) \\ -gl_2 m_2 \sin(q_2) \\ gm_1 \cos(q_1) + gm_2 \cos(q_1) - T_1 \\ gm_2 \cos(q_2) - T_2 \end{bmatrix} \quad (21.21)$$

21.7 Auxiliary Generalized Inertia Forces

Similar to the generalized active forces, the generalized inertia forces for u_1 and u_2 are computed as usual. See [Kane1985] pg. 169 and pg. 217 for more explanation.

```
Rs_P1 = -m1*P1.acc(N)
Rs_P2 = -m2*P2.acc(N)
```

```
F1s = P1.vel(N).diff(u1, N).dot(Rs_P1) + P2.vel(N).diff(u1, N).dot(Rs_P2)
F1s
```

$$-l_1^2 m_1 \dot{u}_1 - l_1^2 m_2 \dot{u}_1 + l_1 (-l_2 m_2 (\sin(q_1) \sin(q_2) + \cos(q_1) \cos(q_2)) \dot{u}_2 - l_2 m_2 (\sin(q_1) \cos(q_2) - \sin(q_2) \cos(q_1)) u_2^2) \quad (21.22)$$

```
F2s = P1.vel(N).diff(u2, N).dot(Rs_P1) + P2.vel(N).diff(u2, N).dot(Rs_P2)
F2s
```

$$-l_2^2 m_2 \dot{u}_2 + l_2 (-l_1 m_2 (\sin(q_1) \sin(q_2) + \cos(q_1) \cos(q_2)) \dot{u}_1 - l_1 m_2 (-\sin(q_1) \cos(q_2) + \sin(q_2) \cos(q_1)) u_1^2) \quad (21.23)$$

The auxiliary generalized inertia forces are found using the velocities where u_3 and u_4 are present, but the acceleration of the particles need not include u_3 and u_4 , because they are equal to zero because u_3 and u_4 are actually equal to zero.

```
F3s = N_v_P1a.diff(u3, N).dot(Rs_P1) + N_v_P2a.diff(u3, N).dot(Rs_P2)
F3s
```

$$l_1 m_1 u_1^2 + l_1 m_2 u_1^2 + l_2 m_2 (\sin(q_1) \sin(q_2) + \cos(q_1) \cos(q_2)) u_2^2 + l_2 m_2 (-\sin(q_1) \cos(q_2) + \sin(q_2) \cos(q_1)) \dot{u}_2 \quad (21.24)$$

```
F4s = N_v_P1a.diff(u4, N).dot(Rs_P1) + N_v_P2a.diff(u4, N).dot(Rs_P2)
F4s
```

$$l_1 m_2 (\sin(q_1) \sin(q_2) + \cos(q_1) \cos(q_2)) u_1^2 + l_1 m_2 (\sin(q_1) \cos(q_2) - \sin(q_2) \cos(q_1)) \dot{u}_1 + l_2 m_2 u_2^2 \quad (21.25)$$

And finally, \bar{F}_r^* is formed for all four generalized speeds:

```
Frs = sm.Matrix([F1s, F2s, F3s, F4s])
Frs = sm.trigsimp(Frs)
Frs
```

$$\begin{bmatrix} -l_1 (l_1 m_1 \dot{u}_1 + l_1 m_2 \dot{u}_1 + l_2 m_2 u_2^2 \sin(q_1 - q_2) + l_2 m_2 \cos(q_1 - q_2) \dot{u}_2) \\ l_2 m_2 (l_1 u_1^2 \sin(q_1 - q_2) - l_1 \cos(q_1 - q_2) \dot{u}_1 - l_2 \dot{u}_2) \\ l_1 m_1 u_1^2 + l_1 m_2 u_1^2 + l_2 m_2 u_2^2 \cos(q_1 - q_2) - l_2 m_2 \sin(q_1 - q_2) \dot{u}_2 \\ m_2 (l_1 u_1^2 \cos(q_1 - q_2) + l_1 \sin(q_1 - q_2) \dot{u}_1 + l_2 u_2^2) \end{bmatrix} \quad (21.26)$$

Warning: In this example, $u_3, u_4, \dot{u}_3, \dot{u}_4$ are not present in the auxiliary generalized inertia forces but you may end up with auxiliary speeds and their derivatives in your auxiliary generalized inertia forces. If you do, you need to set them all to zero to arrive at the desired equations.

21.8 Augmented Dynamical Differential Equations

We can now form Kane's dynamical differential equations which I will name \bar{f}_a to indicate they include the auxiliary equations. These equations are linear in $\dot{u}_1, \dot{u}_2, T_1$ and T_2 .

```
fa = Frs + Fr
me.find_dynamicsymbols(fa)
```

$$\{T_1, T_2, q_1, q_2, u_1, u_2, \dot{u}_1, \dot{u}_2\} \quad (21.27)$$

Now when we extract the linear coefficients, we see that the dynamical differential equations (the first two rows) are independent of the unknown force magnitudes, allowing us to use the equations for \dot{u} independently.

```
Ma = fa.jacobian(udr)
ga = fa.xreplace(udr_zero)

Ma, udr, ga
```

$$\begin{pmatrix} -l_1 (l_1 m_1 + l_1 m_2) & -l_1 l_2 m_2 \cos(q_1 - q_2) & 0 & 0 \\ -l_1 l_2 m_2 \cos(q_1 - q_2) & -l_2^2 m_2 & 0 & 0 \\ 0 & -l_2 m_2 \sin(q_1 - q_2) & -1 & 0 \\ l_1 m_2 \sin(q_1 - q_2) & 0 & 0 & -1 \end{pmatrix}, \begin{bmatrix} \dot{u}_1 \\ \dot{u}_2 \\ T_1 \\ T_2 \end{bmatrix}, \begin{bmatrix} -gl_1 m_1 \sin(q_1) - gl_1 m_2 \sin(q_1) - l_1 l_2 m_2 u_2^2 \sin(q_1 - q_2) \\ -gl_2 m_2 \sin(q_2) + l_1 l_2 m_2 u_1^2 \sin(q_1 - q_2) \\ gm_1 \cos(q_1) + gm_2 \cos(q_1) + l_1 m_1 u_1^2 + l_1 m_2 u_1^2 + l_2 m_2 u_2^2 \\ gm_2 \cos(q_2) + m_2 (l_1 u_1^2 \cos(q_1 - q_2) + l_2 u_2^2) \end{bmatrix} \quad (21.28)$$

We can solve the system to find functions for T_1 and T_2 , if desired.

```
udr_sol = -Ma.LUsolve(ga)
```

```
T1_sol = sm.trigsimp(udr_sol[2])
T1_sol
```

$$gm_1 \cos(q_1) + gm_2 \cos(q_1) + l_1 m_1 u_1^2 + l_1 m_2 u_1^2 + l_2 m_2 u_2^2 \cos(q_1 - q_2) + \frac{l_2 m_2 \left(-gl_2 m_2 \sin(q_2) + l_1 l_2 m_2 u_1^2 \sin(q_1 - q_2) + \frac{l_1 l_2 m_2 (gm_1 \sin(q_1) + gm_2 \sin(q_1) + l_2 m_2 u_2^2 \sin(q_1 - q_2)) \cos(q_1 - q_2)}{l_1 m_1 + l_1 m_2} \right) \sin(q_2)}{l_1 l_2 m_2 \cos(q_2)} \quad (21.29)$$

```
T2_sol = sm.trigsimp(udr_sol[3])
T2_sol
```

$$gm_2 \cos(q_2) + \frac{l_1 l_2 m_2^2 \left(-gl_2 m_2 \sin(q_2) + l_1 l_2 m_2 u_1^2 \sin(q_1 - q_2) + \frac{l_1 l_2 m_2 (gm_1 \sin(q_1) + gm_2 \sin(q_1) + l_2 m_2 u_2^2 \sin(q_1 - q_2)) \cos(q_1 - q_2)}{l_1 m_1 + l_1 m_2} \right) \sin(q_1)}{(l_1 m_1 + l_1 m_2) \left(\frac{l_1 l_2 m_2^2 \cos^2(q_1 - q_2)}{l_1 m_1 + l_1 m_2} - l_2^2 m_2 \right)} \quad (21.30)$$

21.9 Compare Newton and Kane Results

To ensure that the Newton approach and the Kane approach do produce equivalent results, we can numerically evaluate the equations with the same inputs and see if the results are the same. Here are some arbitrary numerical values for the states and constants.

```
q0 = np.array([
    np.deg2rad(15.0),  # q1 [rad]
    np.deg2rad(25.0),  # q2 [rad]
])

u0 = np.array([
    np.deg2rad(123.0),  # u1 [rad/s]
    np.deg2rad(-41.0),  # u2 [rad/s]
])

p_vals = np.array([
    1.2,  # m1 [kg]
    5.6,  # m2 [kg]
    1.34,  # l1 [m]
    6.7,  # l2 [m]
    9.81,  # g [m/s^2]
])
```

Create numeric functions to evaluate the two sets of matrices and execute both functions with the same numerical inputs from above.

```

eval_d = sm.lambdify((q, u, p), (Md, gd))
eval_a = sm.lambdify((q, u, p), (Ma, ga))

Md_vals, gd_vals = eval_d(q0, u0, p_vals)
Ma_vals, ga_vals = eval_a(q0, u0, p_vals)

```

Now compare the solutions for $[\dot{\bar{u}} \quad \dot{\bar{r}}]$.

```
-np.linalg.solve(Md_vals, np.squeeze(gd_vals))
```

```
array([ 8.09538007, -2.37332094, 109.88598116, 92.50997719])
```

```
-np.linalg.solve(Ma_vals, np.squeeze(ga_vals))
```

```
array([ 8.09538007, -2.37332094, 109.88598116, 92.50997719])
```

For this set of inputs, the outputs are the same showing that using the auxiliary speed approach gives the same results, with the slight advantage that the dynamical differential equations are not coupled to the equations for the noncontributing forces in Kane's method.

The forces can also be evaluated directly from the symbolic solutions, which is useful for post simulation application.

```

eval_forces = sm.lambdify((q, u, p), (T1_sol, T2_sol))
eval_forces(q0, u0, p_vals)

```

(109.885981161619, 92.5099771909879) (21.31)

CHAPTER
TWENTYTWO

ENERGY AND POWER

Note: You can download this example as a Python script: `energy.py` or Jupyter Notebook: `energy.ipynb`.

```
from IPython.display import HTML
from matplotlib.animation import FuncAnimation
from scikits.odes import dae
from scipy.optimize import fsolve
import matplotlib.pyplot as plt
import numpy as np
import sympy as sm
import sympy.physics.mechanics as me
me.init_vprinting(use_latex='mathjax')
```

```
class ReferenceFrame(me.ReferenceFrame):
    def __init__(self, *args, **kwargs):
        kwargs.pop('latexs', None)
        lab = args[0].lower()
        tex = r'\hat{{}}_{{}}_{{}}'
        super(ReferenceFrame, self).__init__(*args,
                                             latexs=(tex.format(lab, 'x'),
                                                      tex.format(lab, 'y'),
                                                      tex.format(lab, 'z')),
                                             **kwargs)
me.ReferenceFrame = ReferenceFrame
```

22.1 Learning Objectives

After completing this chapter readers will be able to:

- calculate the kinetic and potential energy of a multibody system
- evaluate a simulation for energy gains and losses

22.2 Introduction

So far we have investigated multibody systems from the perspective of forces and their relationship to motion. It is also useful to understand these systems from a power and energy perspective. Power P is the time rate of change in work W done where work is the energy gained, dissipated, or exchanged in a system.

$$P = \frac{dW}{dt} \quad (22.1)$$

Conversely, work is the integral of power:

$$W(t) = \int_{t_0}^{t_f} P(t) dt \quad (22.2)$$

The work done by a force \bar{F} acting on a point located by position vector $\bar{r}(t)$ is calculated as:

$$W = \int_{\bar{r}(t_0)}^{\bar{r}(t_1)} \bar{F} \cdot d\bar{r} = \int_{t_0}^{t_1} \bar{F} \cdot \dot{\bar{r}} dt \quad (22.3)$$

From which we also see $P = \bar{F} \cdot \dot{\bar{r}}$.

Energy in a multibody system comes in many forms and can be classified as kinetic, potential (conservative), or non-conservative. Any energy that enters or leaves the system is non-conservative.

22.3 Kinetic Energy

Kinetic energy K is an instantaneous measure of the energy due to motion of all of the particles and rigid bodies in a system. A rigid body will, in general, have a translational and a rotational component of kinetic energy. A particle cannot rotate so it only has translational kinetic energy. Kinetic energy can be thought of as the work done by the generalized inertia forces \bar{F}_r^* with going from the current state to rest.

Translational kinetic energy of a particle Q of mass m in reference frame N is:

$$K_Q := \frac{1}{2} m |{}^N \bar{v}^Q|^2 = \frac{1}{2} m {}^N \bar{v}^Q \cdot {}^N \bar{v}^Q \quad (22.4)$$

If Q is the mass center of a rigid body, the equation represents the translational kinetic energy of the rigid body. The rotational kinetic energy of a rigid body B with mass center B_o in N is added to its translational kinetic energy and the total kinetic energy of B is defined as:

$$K_B := \frac{1}{2} m {}^N \bar{v}^{B_o} \cdot {}^N \bar{v}^{B_o} + \frac{1}{2} {}^N \bar{\omega}^B \cdot {}^N \bar{I}^{B/B_o} \cdot {}^N \bar{\omega}^B \quad (22.5)$$

The total kinetic energy in a multibody system is the sum of the kinetic energies for all particles and rigid bodies.

22.4 Potential Energy

Some of the generalized active force contributions in inertial reference frame N can be written as

$$F_r = - \frac{\partial V}{\partial q_r} \quad (22.6)$$

when $\bar{u} = \dot{\bar{q}}$ and where V is strictly a function of the generalized coordinates and time, i.e. $V(\bar{q}, t)$. These functions V are potential energies in N . The associated generalized active force contributions are from [conservative forces](#). They are

forces for which the work done by the force for any path $\bar{r}(t)$ starting and ending at the same position equals zero. The most common conservative forces seen in multibody systems are gravitational forces and ideal spring forces, but there are conservative forces related to electrostatic forces, magnetic forces, etc.

For small objects at Earth's surface we model gravity as a uniform field and the potential energy of a particle or rigid body is:

$$V = mgh \quad (22.7)$$

where m is the body or particle's mass, g is the acceleration due to gravity at the Earth's surface, and $h(\bar{q}, t)$ is the distance parallel to the gravitational field direction of the particle or body with respect to an arbitrary reference point.

A linear spring generates a conservative force $F = kx$ between two points P and Q and its potential energy is:

$$V_s = \frac{1}{2}k \left| \bar{r}^{P/Q} \right|^2 = \frac{1}{2}k \bar{r}^{P/Q} \cdot \bar{r}^{P/Q} \quad (22.8)$$

The sum of all potential energies in a system give the total potential energy of the system.

22.5 Total Energy

The total energy of the system is:

$$E := K + V \quad (22.9)$$

If \bar{F}_r is only made up of conservative forces, then the system is conservative and will not lose energy as it moves, it simply exchanges kinetic for potential and vice versa, i.e. E is constant for conservative systems.

22.6 Energetics of Jumping

Let's create a simple multibody model of a person doing a vertical jump like shown in the video below so that we can calculate the kinetic and potential energy.

We can model the jumper in a single plane with two rigid bodies representing the thigh B and the calf A of the legs lumping the left and right leg segments together. The mass centers of the leg segments lie on the line connecting the segment end points but at some distance from the ends d_a, d_b . To avoid having to stabilize the jumper, we can assume that particles representing the foot P_f and the upper body P_u can only move vertically and are always aligned vertically over one another. The foot P_f , knee P_k , and hip P_u are all modeled as pin joints. The mass of the foot m_f and the mass of the upper body are modeled as particles at P_f and P_u , respectively. We will model a collision force F_f from the ground N acting on the foot P_f using the Hunt-Crossley formulation described in [Collision](#). We will actuate the jumper using only a torque acting between the thigh and the calf T_k that represents the combine forces of the muscles attached between the two leg segments. [Fig. 22.1](#) shows a free body diagram of the model.

22.6.1 Equations of Motion

We first define all of the necessary symbols:

```
g = sm.symbols('g')
mu, ma, mb, mf = sm.symbols('m_u, m_a, m_b, m_f')
Ia, Ib = sm.symbols('I_a, I_b')
kf, cf, kk, ck = sm.symbols('k_f, c_f, k_k, c_k')
la, lb, da, db = sm.symbols('l_a, l_b, d_a, d_b')
```

(continues on next page)

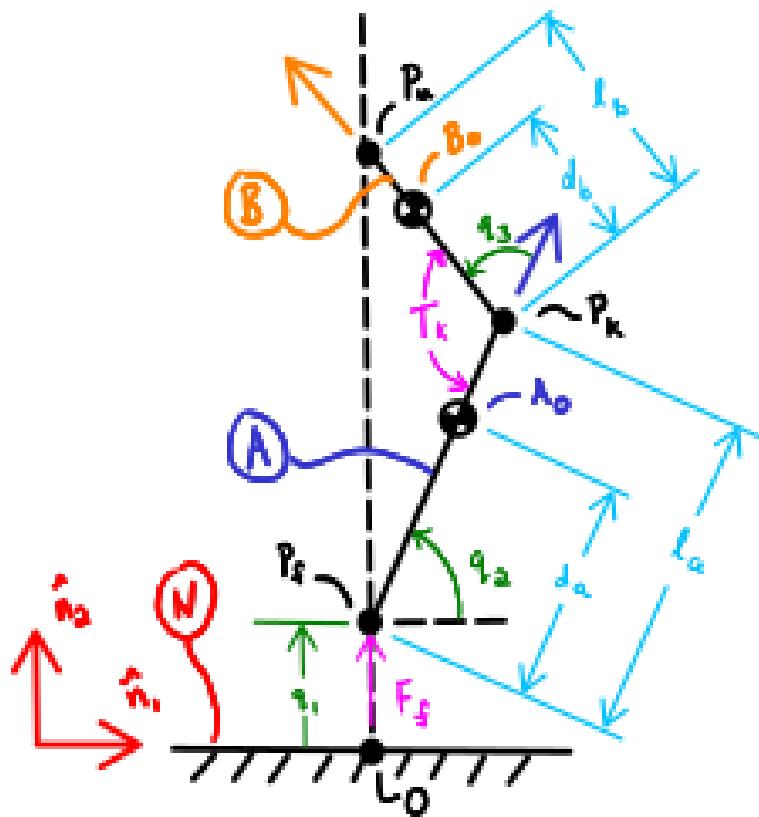


Fig. 22.1: Free body diagram of a simple model of a human jumper.

(continued from previous page)

```

q1, q2, q3 = me.dynamicsymbols('q1, q2, q3', real=True)
u1, u2, u3 = me.dynamicsymbols('u1, u2, u3', real=True)
Tk = me.dynamicsymbols('T_k')

t = me.dynamicsymbols._t

q = sm.Matrix([q1, q2, q3])
u = sm.Matrix([u1, u2, u3])
ud = u.diff(t)
us = sm.Matrix([u1, u3])
usd = us.diff(t)
p = sm.Matrix([
    Ia,
    Ib,
    cf,
    ck,
    da,
    db,
    g,
    kf,
    kk,
    la,
    lb,
    ma,
    mb,
    mf,
    mu,
])
r = sm.Matrix([Tk])

```

Then we set up the kinematics:

```

N = me.ReferenceFrame('N')
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')

A.orient_axis(N, q2, N.z)
B.orient_axis(A, q3, N.z)

A.set_ang_vel(N, u2*N.z)
B.set_ang_vel(A, u3*N.z)

O = me.Point('O')
Ao, Bo = me.Point('A_o'), me.Point('B_o')
Pu, Pk, Pf = me.Point('P_u'), me.Point('P_k'), me.Point('P_f')

Pf.set_pos(O, q1*N.y)
Ao.set_pos(Pf, da*A.x)
Pk.set_pos(Pf, la*A.x)
Bo.set_pos(Pk, db*B.x)
Pu.set_pos(Pk, lb*B.x)

O.set_vel(N, 0)
Pf.set_vel(N, u1*N.y)
Pk.v2pt_theory(Pf, N, A)
Pu.v2pt_theory(Pk, N, B)

```

(continues on next page)

(continued from previous page)

```

qd_repl = {q1.diff(t): u1, q2.diff(t): u2, q3.diff(t): u3}
qdd_repl = {q1.diff(t, 2): u1.diff(t), q2.diff(t, 2): u2.diff(t), q3.diff(t, 2): u3.
            .diff(t) }

holonomic = Pu.pos_from(0).dot(N.x)
vel_con = holonomic.diff(t).xreplace(qd_repl)
acc_con = vel_con.diff(t).xreplace(qdd_repl).xreplace(qd_repl)

# q2 is dependent

u2_repl = {u2: sm.solve(vel_con, u2)[0]}
u2d_repl = {u2.diff(t): sm.solve(acc_con, u2.diff(t))[0].xreplace(u2_repl)}

```

Gravity acts on all the masses and mass centers and we have a single force acting on the foot from the ground that includes the collision stiffness and damping terms with coefficients k_f and c_f respectively.

```

R_Pu = -mu*g*N.y
R_Ao = -ma*g*N.y
R_Bo = -mb*g*N.y

zp = (sm.Abs(q1) - q1)/2
damping = sm.Piecewise((-cf*u1, q1<0), (0.0, True))
Ff = (kf*zp**2*(sm.S(3)/2) + damping)*N.y

R_Pf = -mf*g*N.y + Ff
R_Pf

```

$$(-gm_f + k_f \left(-\frac{q_1}{2} + \frac{|q_1|}{2} \right)^{\frac{3}{2}} + \begin{cases} -c_f u_1 & \text{for } q_1 < 0 \\ 0.0 & \text{otherwise} \end{cases}) \hat{n}_y \quad (22.10)$$

The torques on the thigh and calf will include a passive stiffness and damping to represent muscle tendons and tissue effects with coefficients k_k and c_k respectively as well as the muscle actuation torque T_k .

```

T_A = (kk*(q3 - sm.pi/2) + ck*u3 + Tk)*N.z
T_B = -T_A
T_A

```

$$(c_k u_3 + k_k \left(q_3 - \frac{\pi}{2} \right) + T_k) \hat{n}_z \quad (22.11)$$

Define the inertia dyadics for the legs:

```

I_A_Ao = Ia*me.outer(N.z, N.z)
I_B_Bo = Ib*me.outer(N.z, N.z)

```

Finally, formulate Kane's equations:

```

points = [Pu, Ao, Bo, Pf]
forces = [R_Pu, R_Ao, R_Bo, R_Pf]
masses = [mu, ma, mb, mf]

frames = [A, B]
torques = [T_A, T_B]

```

(continues on next page)

(continued from previous page)

```

inertias = [I_A_Ao, I_B_Bo]

Fr_bar = []
Frs_bar = []

for ur in [u1, u3]:
    Fr = 0
    Frs = 0

    for Pi, Ri, mi in zip(points, forces, masses):
        N_v_Pi = Pi.vel(N).xreplace(u2_repl)
        vr = N_v_Pi.diff(ur, N)
        Fr += vr.dot(Ri)
        N_a_Pi = Pi.acc(N).xreplace(u2d_repl).xreplace(u2_repl)
        Rs = -mi*N_a_Pi
        Frs += vr.dot(Rs)

    for Bi, Ti, Ii in zip(frames, torques, inertias):
        N_w_Bi = Bi.ang_vel_in(N).xreplace(u2_repl)
        N_alp_Bi = Bi.ang_acc_in(N).xreplace(u2d_repl).xreplace(u2_repl)
        wr = N_w_Bi.diff(ur, N)
        Fr += wr.dot(Ti)
        Ts = -(N_alp_Bi.dot(Ii) + me.cross(N_w_Bi, Ii).dot(N_w_Bi))
        Frs += wr.dot(Ts)

    Fr_bar.append(Fr)
    Frs_bar.append(Frs)

Fr = sm.Matrix(Fr_bar)
Frs = sm.Matrix(Frs_bar)
kane_eq = Fr + Frs

```

22.6.2 Energy

The total potential energy is derived based on the height of all the particles and rigid body mass centers above a reference point O on the ground and the two springs: passive knee stiffness and the ground-foot stiffness. The work done by these two springs can be found using `integrate()`:

```

Vf = -sm.integrate(kf*zp** (sm.S(3)/2), q1)
Vf

```

$$-\begin{cases} 0 & \text{for } q_1 \geq 0 \\ -\frac{2k_f(-q_1)^{\frac{5}{2}}}{5} & \text{otherwise} \end{cases} \quad (22.12)$$

```

Vk = sm.integrate(kk*(q3 - sm.pi/2), q3)
Vk

```

$$\frac{k_k q_3^2}{2} - \frac{\pi k_k q_3}{2} \quad (22.13)$$

```

V = (
  (mf*g*Pf.pos_from(O) +
  ma*g*Ao.pos_from(O) +
  mb*g*Bo.pos_from(O) +
  mu*g*Pu.pos_from(O)).dot(N.y) +
  Vf + Vk
)
V

```

$$gm_a q_1 + gm_b q_1 + gm_f q_1 + gm_u q_1 + \frac{k_k q_3^2}{2} - \frac{\pi k_k q_3}{2} + (\sin(q_2) \cos(q_3) + \sin(q_3) \cos(q_2)) (d_b g m_b + g l_b m_u) + (d_a g m_a + g l_a m_b - \frac{\pi k_k q_3}{2}) \quad (22.14)$$

The kinetic energy is made up of the translational kinetic energy of the foot and upper body particles K_f and K_u :

```

Kf = mf*me.dot(Pf.vel(N), Pf.vel(N)) / 2
Ku = mu*me.dot(Pu.vel(N), Pu.vel(N)) / 2
Kf, sm.simplify(Ku)

```

$$\left(\frac{m_f u_1^2}{2}, \frac{m_u \left(l_a^2 u_2^2 + 2l_a l_b (u_2 + u_3) u_2 \cos(q_3) + 2l_a u_1 u_2 \cos(q_2) + l_b^2 (u_2 + u_3)^2 + 2l_b (u_2 + u_3) u_1 \cos(q_2 + q_3) + u_1^2 \right)}{2} \right) \quad (22.15)$$

as well as the translational and rotational kinetic energies of the calf and thigh K_A and K_B :

```

KA = ma*me.dot(Ao.vel(N), Ao.vel(N)) / 2 + me.dot(me.dot(A.ang_vel_in(N), I_A_Ao), A.
ang_vel_in(N)) / 2
KA

```

$$\frac{I_a u_2^2}{2} + \frac{m_a (d_a^2 u_2^2 + 2d_a u_1 u_2 \cos(q_2) + u_1^2)}{2} \quad (22.16)$$

```

KB = mb*me.dot(Bo.vel(N), Bo.vel(N)) / 2 + me.dot(me.dot(B.ang_vel_in(N), I_B_Bo), B.
ang_vel_in(N)) / 2
sm.simplify(KB)

```

$$\frac{I_b (u_2 + u_3)^2}{2} + \frac{m_b (d_b^2 (u_2 + u_3)^2 + 2d_b l_a (u_2 + u_3) u_2 \cos(q_3) + 2d_b (u_2 + u_3) u_1 \cos(q_2 + q_3) + l_a^2 u_2^2 + 2l_a u_1 u_2 \cos(q_2) + u_1^2)}{2} \quad (22.17)$$

The total kinetic energy of the system is then $K = K_f + K_u + K_A + K_B$:

```

K = Kf + Ku + KA + KB

```

22.7 Simulation Setup

We will simulate the system to investigate the energy. Below are various functions that convert the symbolic equations to numerical functions, simulate the system with some initial conditions, and plot/animate the results. These are similar to prior chapters, so I leave them unexplained.

Simulation code

```
eval_kane = sm.lambdify((q, usd, us, r, p), kane_eq)
eval_holo = sm.lambdify((q, p), holonomic)
eval_vel_con = sm.lambdify((q, u, p), vel_con)
eval_acc_con = sm.lambdify((q, ud, u, p), acc_con)
eval_energy = sm.lambdify((q, us, p), (K.xreplace(u2_repl), V.xreplace(u2_repl)))

coordinates = Pf.pos_from(O).to_matrix(N)
for point in [Ao, Pk, Bo, Pu]:
    coordinates = coordinates.row_join(point.pos_from(O).to_matrix(N))
eval_point_coords = sm.lambdify((q, p), coordinates)
```

```
def eval_eom(t, x, xd, residual, p_r):
    """Returns the residual vector of the equations of motion.

    Parameters
    ==========
    t : float
        Time at evaluation.
    x : ndarray, shape(5,)
        State vector at time t: x = [q1, q2, q3, u1, u3].
    xd : ndarray, shape(5,)
        Time derivative of the state vector at time t: xd = [q1d, q2d, q3d, u1d, u3d].
    residual : ndarray, shape(5,)
        Vector to store the residuals in: residuals = [fk, fd, fh].
    r : function
        Function of [Tk] = r(t, x) that evaluates the input Tk.
    p : ndarray, shape(15,)
        Constant parameters: p = [Ia, Ib, cf, ck, da, db, g, kf, kk, la, lb,
        ma, mb, mf, mu]

    """
    p, r = p_r

    q1, q2, q3, u1, u3 = x
    q1d, _, q3d, u1d, u3d = xd  # ignore the q2d value

    residual[0] = -q1d + u1
    residual[1] = -q3d + u3
    residual[2:4] = eval_kane([q1, q2, q3], [u1d, u3d], [u1, u3], r(t, x, p), p)
    residual[4] = eval_holo([q1, q2, q3], p)
```

```
def setup_initial_conditions(q1, q3, u1, u3):
    q0 = np.array([q1, np.nan, q3])
```

(continues on next page)

(continued from previous page)

```

q0[1] = fsolve(lambda q2: eval_holo([q0[0], q2, q0[2]], p_vals),
               np.deg2rad(45.0))[0]

u0 = np.array([u1, u3])

u20 = fsolve(lambda u2: eval_vel_con(q0, [u0[0], u2, u0[1]], p_vals),
               np.deg2rad(0.0))[0]

x0 = np.hstack((q0, u0))

# TODO : use equations to set these
ud0 = np.array([0.0, 0.0])

xd0 = np.hstack(([u0[0], u20, u0[1]], ud0))

return x0, xd0

```

```

def simulate(t0, tf, fps, x0, xd0, p_vals, eval_r):

    ts = np.linspace(t0, tf, num=int(fps*(tf - t0)))

    solver = dae('ida',
                 eval_eom,
                 rtol=1e-8,
                 atol=1e-8,
                 algebraic_vars_idx=[4],
                 user_data=(p_vals, eval_r),
                 old_api=False)

    solution = solver.solve(ts, x0, xd0)

    ts = solution.values.t
    xs = solution.values.y

    Ks, Vs = eval_energy(xs[:, :3].T, xs[:, 3:].T, p_vals)
    Es = Ks + Vs

    Tks = np.empty_like(ts)
    for i, ti in enumerate(ts):
        Tks[i] = eval_r(ti, None, None)[0]

    return ts, xs, Ks, Vs, Es, Tks

```

```

def plot_results(ts, xs, Ks, Vs, Es, Tks):
    """Returns the array of axes of a 4 panel plot of the state trajectory
    versus time.

    Parameters
    =====
    ts : array_like, shape(n,)
        Values of time.
    xs : array_like, shape(n, 4)
        Values of the state trajectories corresponding to ``ts`` in order
        [q1, q2, q3, u1, u3].

```

Returns

(continues on next page)

(continued from previous page)

```
=====
axes : ndarray, shape(3,)
    Matplotlib axes for each panel.

"""
fig, axes = plt.subplots(6, 1, sharex=True)

fig.set_size_inches((10.0, 6.0))

axes[0].plot(ts, xs[:, 0])  # q1(t)
axes[1].plot(ts, np.rad2deg(xs[:, 1:3]))  # q2(t), q3(t)
axes[2].plot(ts, xs[:, 3])  # u1(t)
axes[3].plot(ts, np.rad2deg(xs[:, 4]))  # u3(t)
axes[4].plot(ts, Ks)
axes[4].plot(ts, Vs)
axes[4].plot(ts, Es)
axes[5].plot(ts, Tks)

axes[0].legend(['$q_1$'])
axes[1].legend(['$q_2$', '$q_3$'])
axes[2].legend(['$u_1$'])
axes[3].legend(['$u_3$'])
axes[4].legend(['$K$', '$V$', '$E$'])
axes[5].legend(['$T_k$'])

axes[0].set_ylabel('Distance [m]')
axes[1].set_ylabel('Angle [deg]')
axes[2].set_ylabel('Speed [m/s]')
axes[3].set_ylabel('Angular Rate [deg/s]')
axes[4].set_ylabel('Energy [J]')
axes[5].set_ylabel('Torque [N-m]')
axes[5].set_xlabel('Time [s]')

fig.tight_layout()

return axes
```

```
def setup_animation_plot(ts, xs, p):
    """Returns objects needed for the animation.

    Parameters
    ======
    ts : array_like, shape(n,)
        Values of time.
    xs : array_like, shape(n, 4)
        Values of the state trajectories corresponding to ``ts`` in order
        [q1, q2, q3, u1].
    p : array_like, shape(?,)
        Values of the parameters corresponding to ``ts`` in order
        [K, V, E].
    """
    x, y, _ = eval_point_coords(xs[0, :3], p)

    fig, ax = plt.subplots()
    fig.set_size_inches((10.0, 10.0))
    ax.set_aspect('equal')
```

(continues on next page)

(continued from previous page)

```

ax.grid()

lines, = ax.plot(x, y, color='black',
                  marker='o', markerfacecolor='blue', markersize=10)

title_text = ax.set_title('Time = {:.1f} s'.format(ts[0]))
ax.set_xlim((-0.5, 0.5))
ax.set_ylim((0.0, 1.5))
ax.set_xlabel('$x$ [m]')
ax.set_ylabel('$y$ [m]')
ax.set_aspect('equal')

return fig, ax, title_text, lines

```

```

def animate_linkage(ts, xs, p):
    """Returns an animation object.

    Parameters
    ======
    ts : array_like, shape(n,)
    xs : array_like, shape(n, 4)
        x = [q1, q2, q3, u1]
    p : array_like, shape(6,)
        p = [la, lb, lc, ln, m, g]

    """
    # setup the initial figure and axes
    fig, ax, title_text, lines = setup_animation_plot(ts, xs, p)

    # precalculate all of the point coordinates
    coords = []
    for xi in xs:
        coords.append(eval_point_coords(xi[:3], p))
    coords = np.array(coords)

    # define the animation update function
    def update(i):
        title_text.set_text('Time = {:.1f} s'.format(ts[i]))
        lines.set_data(coords[i, 0, :], coords[i, 1, :])

    # close figure to prevent premature display
    plt.close()

    # create and return the animation
    return FuncAnimation(fig, update, len(ts))

```

22.8 Conservative Simulation

For the first simulation, let's disable the ground reaction force and the passive and active knee behavior and simply let the leg fall in space.

```
p_vals = np.array([
    0.101,  # Ia,
    0.282,  # Ib,
    0.0,    # cf,
    0.0,    # ck,
    0.387,  # da,
    0.193,  # db,
    9.81,   # g,
    0.0,    # kf,
    0.0,    # kk,
    0.611,  # la,
    0.424,  # lb,
    6.769,  # ma,
    17.01,  # mb,
    3.0,    # mf,
    32.44,  # mu
])

x0, xd0 = setup_initial_conditions(0.2, np.deg2rad(20.0), 0.0, 0.0)

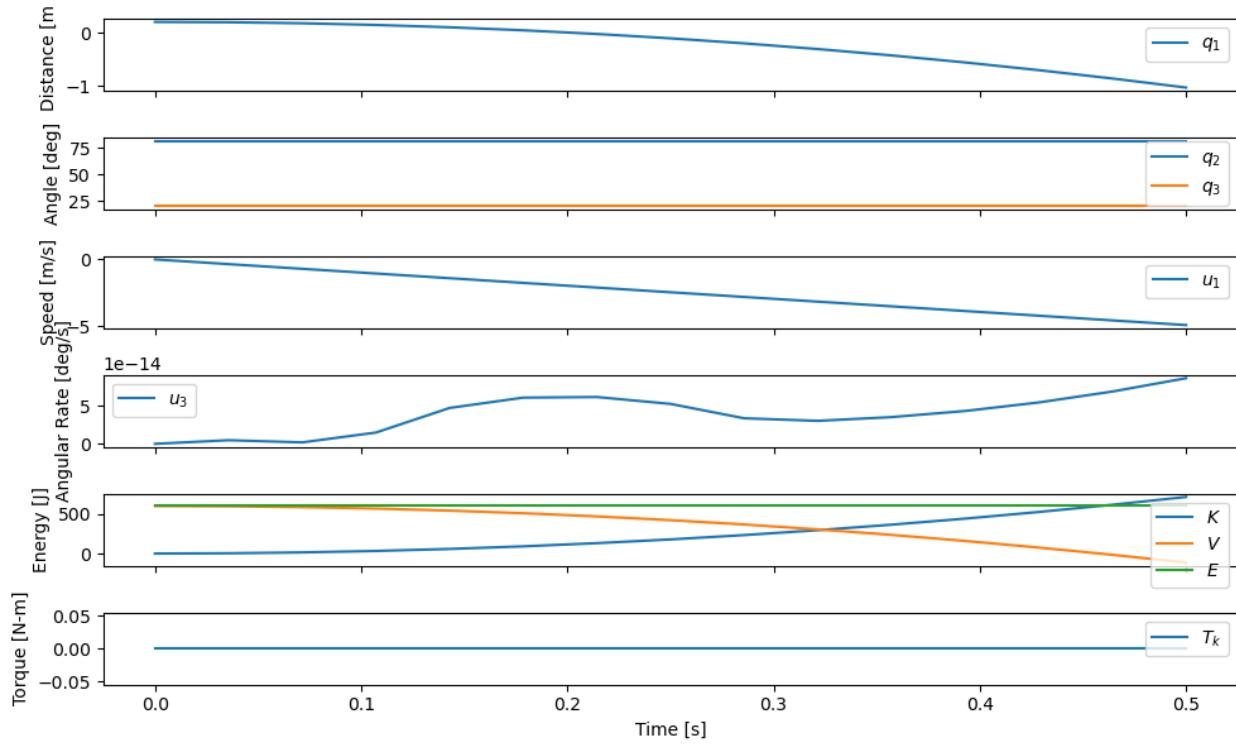
def eval_r(t, x, p):
    return [0.0] # [Tk]
```

```
t0, tf, fps = 0.0, 0.5, 30
ts_dae, xs_dae, Ks, Vs, Es, Tks = simulate(t0, tf, fps, x0, xd0, p_vals, eval_r)
```

```
HTML/animate_linkage(ts_dae, xs_dae, p_vals).to_jshtml(fps=fps))
```

```
<IPython.core.display.HTML object>
```

```
plot_results(ts_dae, xs_dae, Ks, Vs, Es, Tks);
```



With no dissipation and only conservative forces acting on the system (gravity), the total energy E should stay constant, which it does. Checking whether energy remains constant is a useful for sussing out whether your model is likely valid. So far so good for us.

22.9 Conservative Simulation with Ground Spring

For the second simulation of this model we will do the same thing but add only the conservative ground-foot stiffness force by setting $k_f = 5 \times 10^7$.

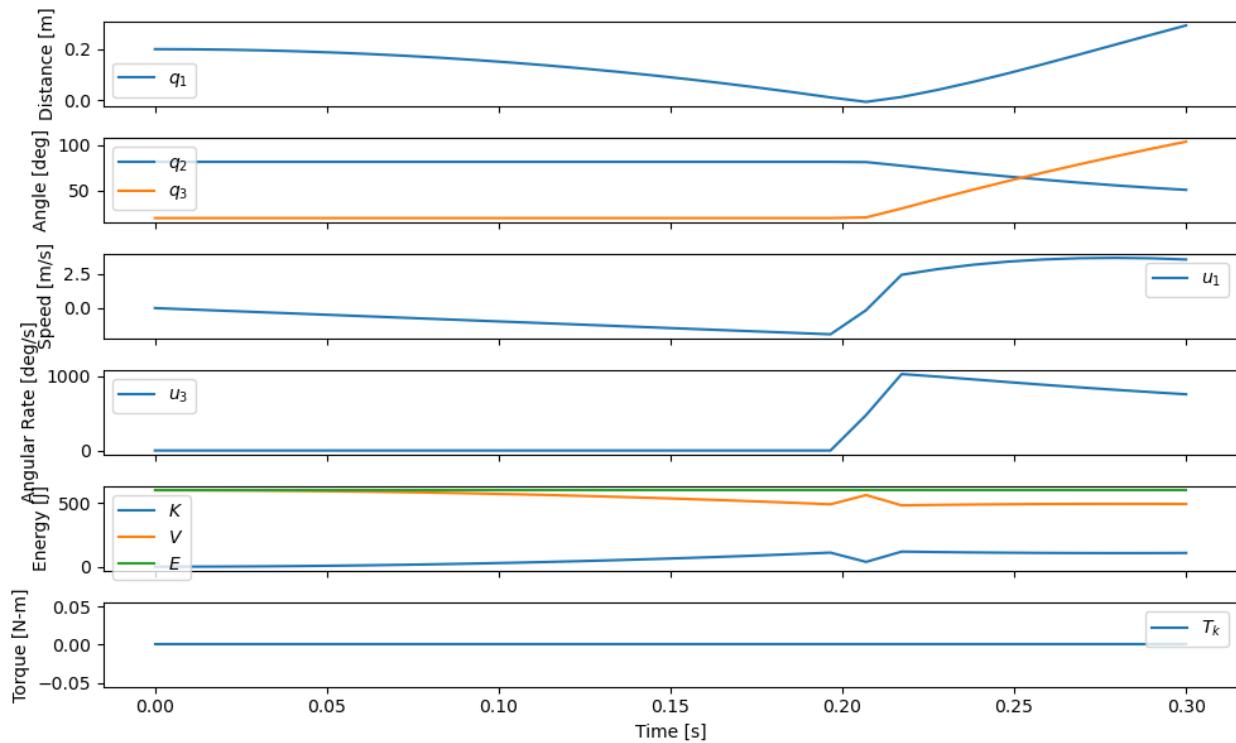
```
p_vals = np.array([
    0.101,  # Ia,
    0.282,  # Ib,
    0.0,    # cf,
    0.0,    # ck,
    0.387,  # da,
    0.193,  # db,
    9.81,   # g,
    5e7,    # kf,
    0.0,    # kk,
    0.611,  # la,
    0.424,  # lb,
    6.769,  # ma,
    17.01,  # mb,
    3.0,    # mf,
    32.44,  # mu
])
```

```
t0, tf, fps = 0.0, 0.3, 100
ts_dae, xs_dae, Ks, Vs, Es, Tks = simulate(t0, tf, fps, x0, xd0, p_vals, eval_r)
```

```
HTML(animate_linkage(ts_dae, xs_dae, p_vals).to_jshtml(fps=fps))
```

```
<IPython.core.display.HTML object>
```

```
plot_results(ts_dae, xs_dae, Ks, Vs, Es, Tks);
```



Now we get a bouncing jumper. This system should also still be conservative and we see that the energy stored in the foot spring is consumed from the loss of kinetic energy as the velocity goes to zero and that total energy is constant.

22.10 Nonconservative Simulation

Now we will give some damping to the Hunt-Crossley model by setting $c_f = 1 \times 10^5$.

```
p_vals = np.array([
    0.101,  # Ia,
    0.282,  # Ib,
    1e5,    # cf,
    0.0,    # ck,
    0.387,  # da,
    0.193,  # db,
    9.81,   # g,
    5e7,    # kf,
    0.0,    # kk,
    0.611,  # la,
    0.424,  # lb,
    6.769,  # ma,
    17.01,  # mb,
    3.0,    # mf,
```

(continues on next page)

(continued from previous page)

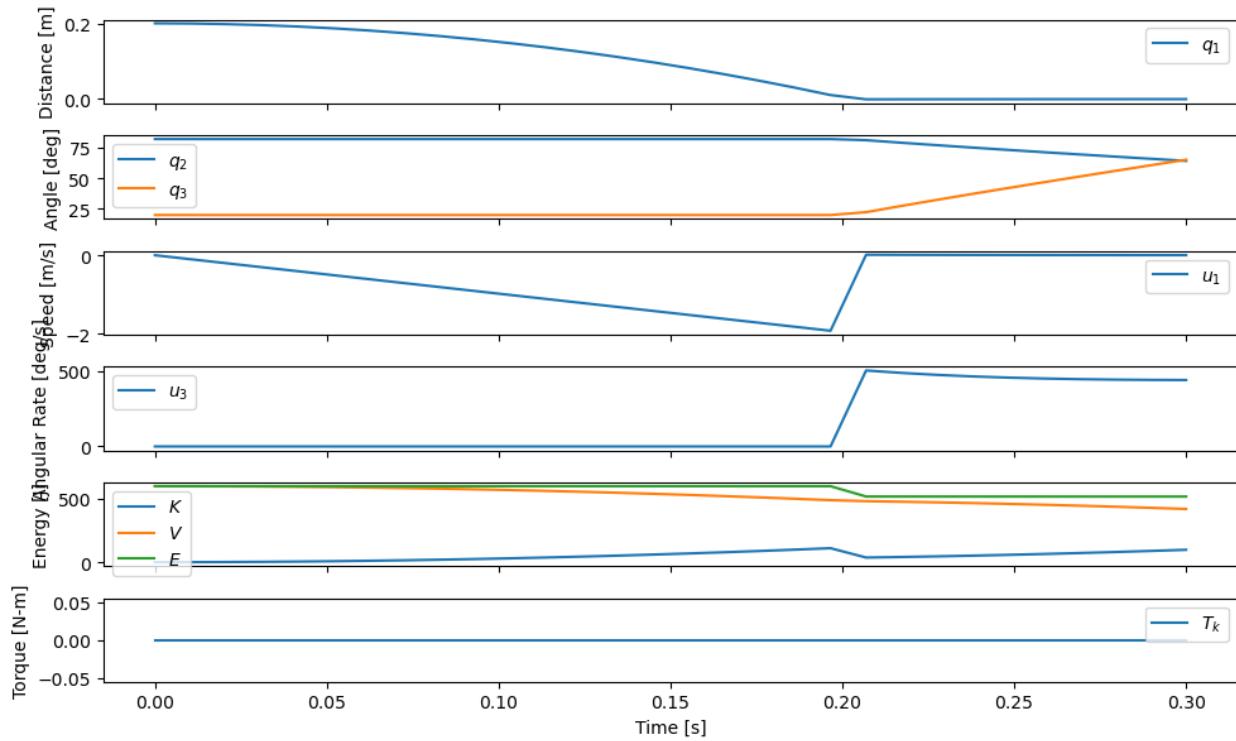
```
32.44, # mu
])

t0, tf, fps = 0.0, 0.3, 100
ts_dae, xs_dae, Ks, Vs, Es, Tks = simulate(t0, tf, fps, x0, xd0, p_vals, eval_r)
```

```
HTML(animate_linkage(ts_dae, xs_dae, p_vals).to_jshtml(fps=fps))
```

```
<IPython.core.display.HTML object>
```

```
plot_results(ts_dae, xs_dae, Ks, Vs, Es, Tks);
```



Now we see a clear energy dissipation from the system due to the foot-ground collision, i.e. the drop in E .

22.11 Simulation with Passive Knee Torques

In this simulation, we include some passive stiffness and damping at the knee joint.

```
p_vals = np.array([
0.101, # Ia,
0.282, # Ib,
1e5, # cf,
30.0, # ck,
0.387, # da,
0.193, # db,
9.81, # g,
5e7, # kf,
```

(continues on next page)

(continued from previous page)

```
10.0,    # kk,
0.611,   # la,
0.424,   # lb,
6.769,   # ma,
17.01,   # mb,
3.0,     # mf,
32.44,   # mu
])
```

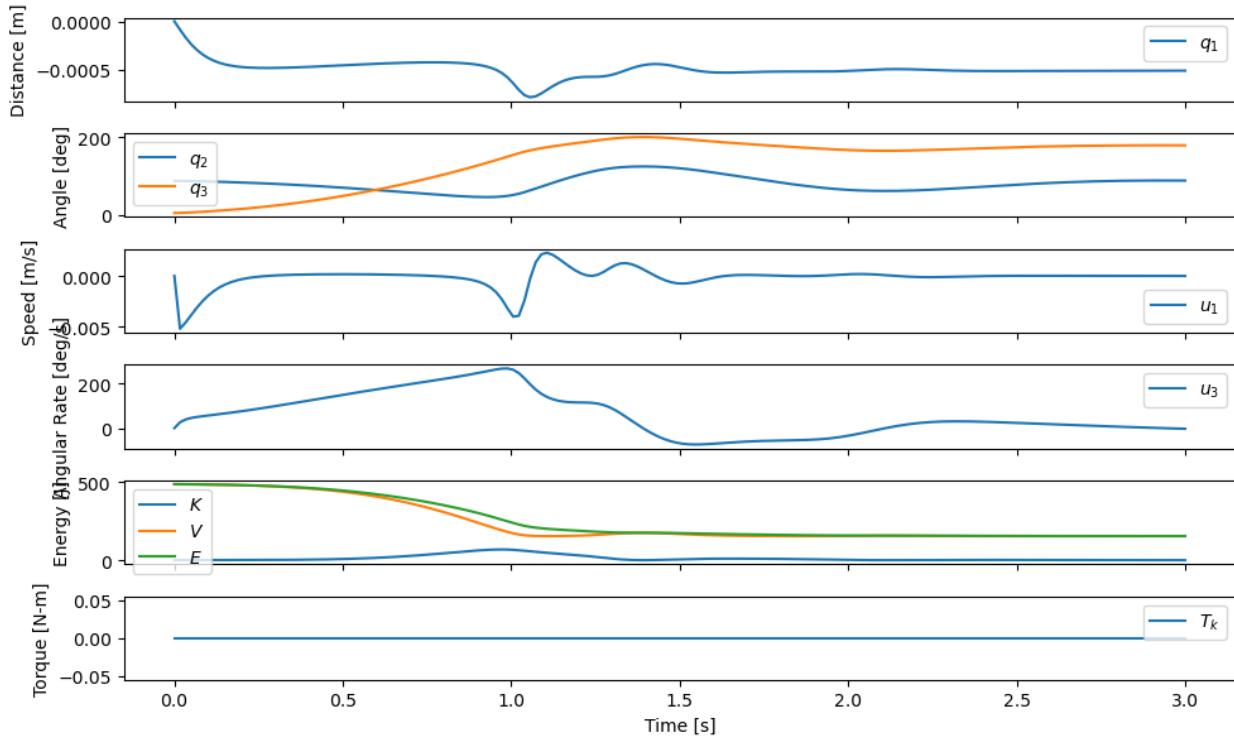
```
x0, xd0 = setup_initial_conditions(0.0, np.deg2rad(5.0), 0.0, 0.0)

t0, tf, fps = 0.0, 3.0, 60
ts_dae, xs_dae, Ks, Vs, Es, Tks = simulate(t0, tf, fps, x0, xd0, p_vals, eval_r)
```

```
HTML/animate_linkage(ts_dae, xs_dae, p_vals).to_jshtml(fps=fps))
```

```
<IPython.core.display.HTML object>
```

```
plot_results(ts_dae, xs_dae, Ks, Vs, Es, Tks);
```



Notice that the knee collapses more slowly due to the damping and in the total energy plot the energy loss due to the non-conservative knee damping can be clearly seen.

22.12 Simulation with Active Knee Torques

Now that we likely have a reasonable passive model of a jumper we can try to make it jump by added energy to the system through the knee torque T_k . We have a symbol for the specified time varying quantity and the simulation code has been designed above to accept a function that calculates T_k at any time instance. We'll let the thigh fall and then give a constant torque to drive the thigh back up for a just two tenths of a second.

```
def eval_r(t, x, p):  
  
    if t < 0.9:  
        Tk = [0.0]  
    elif t > 1.1:  
        Tk = [0.0]  
    else:  
        Tk = [900.0]  
  
    return Tk
```

```
p_vals = np.array([  
    0.101,    # Ia,  
    0.282,    # Ib,  
    1e5,       # cf,  
    30.0,      # ck,  
    0.387,    # da,  
    0.193,    # db,  
    9.81,      # g,  
    5e7,       # kf,  
    10.0,      # kk,  
    0.611,    # la,  
    0.424,    # lb,  
    6.769,    # ma,  
    17.01,    # mb,  
    3.0,       # mf,  
    32.44,    # mu  
])
```

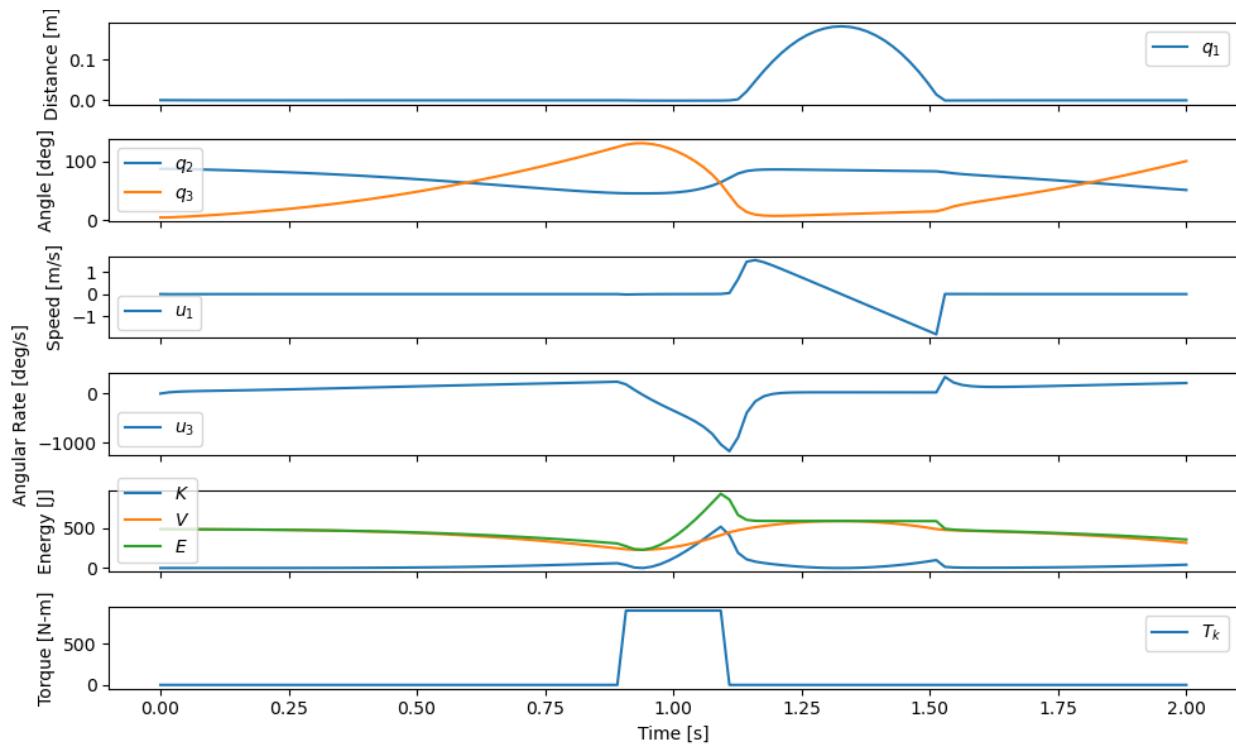
We'll start the simulation with the foot on the ground and with a slight knee bend.

```
x0, xd0 = setup_initial_conditions(0.0, np.deg2rad(5.0), 0.0, 0.0)  
  
t0, tf, fps = 0.0, 2.0, 60  
ts_dae, xs_dae, Ks, Vs, Es, Tks = simulate(t0, tf, fps, x0, xd0, p_vals, eval_r)
```

```
HTML/animate_linkage(ts_dae, xs_dae, p_vals).to_jshtml(fps=fps))
```

```
<IPython.core.display.HTML object>
```

```
plot_results(ts_dae, xs_dae, Ks, Vs, Es, Tks);
```



The final simulation works and gives a reasonably realistic looking jump. When examining the total energy E you can see how the applied knee torque adds energy to the system to cause the jump.

CHAPTER
TWENTYTHREE

EQUATIONS OF MOTION WITH THE LAGRANGE METHOD

Note: You can download this example as a Python script: `lagrange.py` or Jupyter Notebook: `lagrange.ipynb`.

```
import sympy as sm
import sympy.physics.mechanics as me
me.init_vprinting(use_latex='mathjax')

class ReferenceFrame(me.ReferenceFrame):

    def __init__(self, *args, **kwargs):
        kwargs.pop('latexs', None)

        lab = args[0].lower()
        tex = r'\hat{{}}_{{}}_{{}}'

        super(ReferenceFrame, self).__init__(*args,
                                             latexs=(tex.format(lab, 'x'),
                                                      tex.format(lab, 'y'),
                                                      tex.format(lab, 'z')),
                                             **kwargs)
me.ReferenceFrame = ReferenceFrame
```

23.1 Learning Objectives

After completing this chapter readers will be able to:

- Derive the Lagrangian for a system of particles and rigid bodies
- Use the Euler-Lagrange equation to derive equations of motions given a Lagrangian
- Use the method of Lagrange multipliers to add constraints to the equations of motions
- Know how to determine the generalized momenta of a system.

23.2 Introduction

This book has already discussed two methods to derive the equations of motion of multibody systems: Newton-Euler and Kane's method. This chapter will add a third: the [Lagrange method](#), originally developed by Joseph-Louis Lagrange. These materials focus on Engineering applications for multi-body systems, and therefore build the Lagrange method around the terms found earlier in Kane's equations. In other textbooks, the Lagrange method is often derived from the [Variational principles](#), such as virtual work or the principle of least action. A good starting point for studying the physical and mathematical background of the Lagrange approach is [\[Lanczos1970\]](#).

23.3 Inertial forces from kinetic energy

In Kane's method the negated generalized inertial forces equal the applied forces, see [Unconstrained Equations of Motion](#). A large part of Kane's method of deriving the equations of motions for a system is involved with finding the generalized inertial forces.

As an alternative, the following equation also calculates the generalized inertial forces of a system, now by starting from the kinetic energy $K(\dot{\bar{q}}, \bar{q})$ expressed as function of the generalized coordinates \bar{q} , and their time derivatives. See [Energy and Power](#) for the definition of kinetic energy.

$$-\bar{F}_r^* = \frac{d}{dt} \left(\frac{\partial K}{\partial \dot{q}_r} \right) - \frac{\partial K}{\partial q_r} \quad (23.1)$$

Warning: Note the two minus signs in the above equation

Note: In Kane's method, it is possible to choose generalized speeds \bar{u} that differ from the time derivatives of the generalized coordinates $\dot{\bar{q}}$. By convention $\bar{u} = \dot{\bar{q}}$ is assumed when using the Lagrange method. Therefore, throughout this chapter $\dot{\bar{q}}$ is used.

The generalized inertial forces computed in this manner are the same as when following Kane's method, or the TMT method used in the next chapter. This can be shown by carefully matching terms in these formulations, as is done for a a system of point-masses in [\[Vallery2020\]](#).

23.3.1 Example: freely moving 3D body

This example is largely the same as the example in [Body Fixed Newton-Euler Equations](#). A key difference is a difference between the generalized speeds describing the rotation. In the calculation with Kane's method, they were body-fixed angular velocities, whereas here they are the rates of change of the Euler angles.

First, set up the generalized coordinates, reference frames and mass properties:

```
t = me.dynamicsymbols._t
psi,theta, phi, x, y, z = me.dynamicsymbols('psi theta phi x y z')
q = sm.Matrix([psi, theta, phi, x, y, z])
qd = q.diff(t)
qdd = qd.diff(t)
N = me.ReferenceFrame('N')
B = me.ReferenceFrame('B')
B.orient_body_fixed(N, (psi, theta, phi), 'zxy')
```

(continues on next page)

(continued from previous page)

```
m, Ixx, Iyy, Izz = sm.symbols('M, I_{xx}, I_{yy}, I_{zz}')
I_B = me.inertia(B, Ixx, Iyy, Izz)
q
```

$$\begin{bmatrix} \psi \\ \theta \\ \phi \\ x \\ y \\ z \end{bmatrix} \quad (23.2)$$

Then compute the kinetic energy:

```

N_w_B = B.ang_vel_in(N)
r_O_P = x*N.x + y*N.y + z*N.z
N_v_C = r_O_P.dt(N)
K = N_w_B.dot(I_B.dot(N_w_B)) / 2 + m*N_v_C.dot(N_v_C) / 2
K

```

$$\frac{I_{xx} \left(-\sin(\phi) \cos(\theta) \dot{\psi} + \cos(\phi) \dot{\theta} \right)^2}{2} + \frac{I_{yy} \left(\sin(\theta) \dot{\psi} + \dot{\phi} \right)^2}{2} + \frac{I_{zz} \left(\sin(\phi) \dot{\theta} + \cos(\phi) \cos(\theta) \dot{\psi} \right)^2}{2} + \frac{M \left(\dot{x}^2 + \dot{y}^2 + \dot{z}^2 \right)}{2} \quad (23.3)$$

Use the kinetic energy to find the generalized inertial forces. Here we start with the generalized coordinate ψ

```
psid = psi.diff(t)
F_psi_s = K.diff(psid).diff(t) - K.diff(psi)
```

A similar derivation should be made for all generalized coordinates. We could write a loop, but there is a method to derive all the equations in one go. The vector of partial derivatives of a function, that is the gradient, can be created using the `jacobian()` method. The generalized inertial forces can then be found like this:

```
K_as_matrix = sm.Matrix([K])
Fs_transposed = K_as_matrix.jacobian(qd).diff(t) - K_as_matrix.jacobian(q)
Fs = Fs_transposed.transpose()
Fs
```

$$\begin{bmatrix} I_{xx} \left(-\sin(\phi) \cos(\theta) \dot{\psi} + \cos(\phi) \dot{\theta} \right) \sin(\phi) \sin(\theta) \dot{\theta} - I_{xx} \left(-\sin(\phi) \cos(\theta) \dot{\psi} + \cos(\phi) \dot{\theta} \right) \cos(\phi) \cos(\theta) \dot{\phi} - I_{xx} \left(\sin(\phi) \sin(\theta) \dot{\psi} - \cos(\phi) \sin(\theta) \dot{\theta} \right) \cos(\phi) \cos(\theta) \dot{\phi} \\ -I_{xx} \left(-\sin(\phi) \cos(\theta) \dot{\psi} + \cos(\phi) \dot{\theta} \right) \sin(\phi) \sin(\theta) \dot{\psi} - I_{xx} \left(-\sin(\phi) \cos(\theta) \dot{\psi} + \cos(\phi) \dot{\theta} \right) \sin(\phi) \cos(\theta) \dot{\theta} \end{bmatrix} \quad (23.4)$$

While these are correct equations of motion, the terms, particularly the terms involving \ddot{q}_r are mangled. It is common to extract the system mass matrix \mathbf{M}_d and velocity forces vector \bar{g}_d like before:

```

qdd_zerod = {qddr: 0 for qddr in qdd}
Md = Fs.jacobian(qdd)
gd = Fs.xreplace(qdd_zerod)
Md.simplify()
gd.simplify()
Md, gd

```

$$\begin{pmatrix}
I_{xx} \sin^2(\phi) \cos^2(\theta) + I_{yy} \sin^2(\theta) + I_{zz} \cos^2(\phi) \cos^2(\theta) & \frac{(-I_{xx} + I_{zz})(\sin(2\phi - \theta) + \sin(2\phi + \theta))}{4} & I_{yy} \sin(\theta) & 0 & 0 & 0 \\
\frac{(-I_{xx} + I_{zz})(\sin(2\phi - \theta) + \sin(2\phi + \theta))}{4} & I_{xx} \cos^2(\phi) + I_{zz} \sin^2(\phi) & 0 & 0 & 0 & 0 \\
I_{yy} \sin(\theta) & 0 & I_{yy} & 0 & 0 & 0 \\
0 & 0 & 0 & M & 0 & 0 \\
0 & 0 & 0 & 0 & M & 0 \\
0 & 0 & 0 & 0 & 0 & M
\end{pmatrix}, \quad (23.5)$$

23.4 Conservative Forces

Recall from [Energy and Power](#) that conservative forces, can be expressed using the gradient of a scalar function of the generalized coordinates, known as the [potential energy](#) $V(\bar{q})$:

$$\bar{F}_r = -\frac{\partial V}{\partial q_r} \quad (23.6)$$

Warning: Note the minus sign in the above equation.

Some examples of conservative forces are:

- a uniform gravitational field, for example on the surface of the earth, with potential $V = mgh(\bar{q})$,
- gravity from Newton's universal gravitation, with potential $V = -G \frac{m_1 m_2}{r(\bar{q})}$,
- a linear spring, with potential $V = \frac{1}{2}k(l(\bar{q}) - l_0)^2$.

For conservative forces, it is often convenient to derive the applied forces via the potential energy.

23.5 The Lagrange Method

Both the equation for computing the inertial forces from the kinetic energy, and the equation for computing the applied forces from a potential energy have a term in them with the partial derivative with respect to the generalized coordinate. Furthermore, the potential energy does not depend on the generalized speeds. Therefore, the resulting (inertial and conservative applied) forces can be derived in one go, by combining the two equations.

Step 1. Compute the so called [Lagrangian](#) L , the difference between the kinetic energy and potential energy:

$$L = K - V \quad (23.7)$$

Step 2. Use the Euler-Lagrange equations (the name for the equation (23.1)) to find the equations of motion:

$$\frac{d}{dt} \left(\frac{\partial L}{\partial \dot{q}_r} \right) - \frac{\partial L}{\partial q_r} = F_r, \quad (23.8)$$

while being careful to include a force either in the applied forces \bar{F}_r , or in the potential energy V , but never in both.

23.5.1 Example: Double pendulum with springs and sliding pointmass

This example will use the Lagrange method to derive the equations of motion for the system introduced in *Example of Kane's Equations*. The description of the system is shown again in Fig. 23.1.

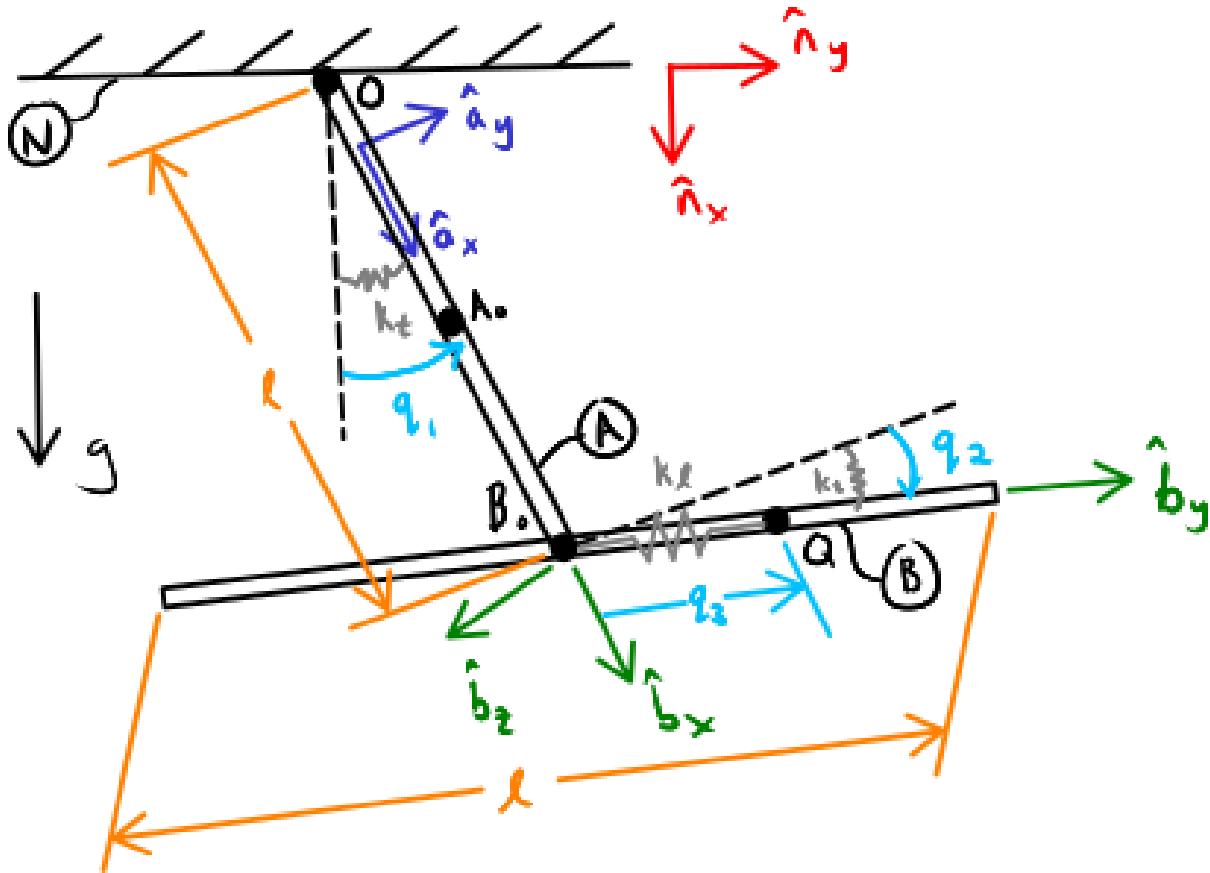


Fig. 23.1: Three dimensional pendulum made up of two pinned rods and a sliding mass on rod B . Each degree of freedom is resisted by a linear spring. When the generalized coordinates are all zero, the two rods are perpendicular to each other.

The first step is to define the relevant variables, constants and frames. This step is the same as for Kane's method.

Frames and Bodies Setup

```

m, g, kt, kl, l = sm.symbols('m, g, k_t, k_l, l')
q1, q2, q3 = me.dynamicsymbols('q1, q2, q3')

N = me.ReferenceFrame('N')
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')

```

(continues on next page)

(continued from previous page)

```

A.orient_axis(N, q1, N.z)
B.orient_axis(A, q2, A.x)

O = me.Point('O')
Ao = me.Point('A_O')
Bo = me.Point('B_O')
Q = me.Point('Q')

Ao.set_pos(O, 1/2*A.x)
Bo.set_pos(O, l*A.x)
Q.set_pos(Bo, q3*B.y)

O.set_vel(N, 0)

I = m*l**2/12
I_A_Ao = I*me.outer(A.y, A.y) + I*me.outer(A.z, A.z)
I_B_Bo = I*me.outer(B.x, B.x) + I*me.outer(B.z, B.z)

```

Then, set up the Lagrangian:

```

t = sm.symbols('t')
q = sm.Matrix([q1, q2, q3])
qd = q.diff(t)
qdd = qd.diff(t)

K = m/2*(Ao.vel(N).dot(Ao.vel(N)) + Bo.vel(N).dot(Bo.vel(N)) + Q.vel(N).dot(Q.
    .vel(N))) + (
    A.ang_vel_in(N).dot(I_A_Ao.dot(A.ang_vel_in(N))) + B.ang_vel_in(N).dot(I_B_Bo.
    .dot(B.ang_vel_in(N))))
)/2
V = m*g*(Ao.pos_from(O).dot(-N.x) + Bo.pos_from(O).dot(-N.x)) + kt/2*(q1**2) + kt/
    2*q2**2 + kl/2*q3**2

L = sm.Matrix([K - V])
sm.trigsimp(L)

```

$$\left[\frac{3g\cos(q_1)}{2} - \frac{k_l q_3^2}{2} - \frac{k_t q_1^2}{2} - \frac{k_t q_2^2}{2} + \frac{l^2 m \cos^2(q_2) \dot{q}_1^2}{24} + \frac{7l^2 m \dot{q}_1^2}{6} + \frac{l^2 m \dot{q}_2^2}{24} - l m q_3 \sin(q_2) \dot{q}_1 \dot{q}_2 + l m \cos(q_2) \dot{q}_1 \dot{q}_3 + \frac{m q_3^2 \cos^2(q_2) \dot{q}_1^2}{2} + \frac{m q_2^2 \dot{q}_2^2}{2} \right] \quad (23.9)$$

Finally, derive the equations of motion:

```

left_hand_side = L.jacobian(qd).diff(t) - L.jacobian(q)
qdd_zerod = {qddr: 0 for qddr in qdd}
Md = left_hand_side.jacobian(qdd)
gd = left_hand_side.xreplace(qdd_zerod)
me.find_dynamicsymbols(Md), me.find_dynamicsymbols(gd)

```

$$(\{q_2, q_3\}, \{q_1, q_2, q_3, \dot{q}_1, \dot{q}_2, \dot{q}_3\}) \quad (23.10)$$

The mass matrix \mathbf{M}_d only depends on \bar{q} , and \bar{g}_d depends on $\dot{\bar{q}}$ and \bar{q} , just as in Kane's method. Note that \bar{g}_d now combines the effects of the velocity force vector and the conservative forces. In this setting, \bar{g}_d is often called the dynamic bias.

It is often useful to use a vector of intermediate variables when finding the Euler-Lagrange equations. The variables are defined as:

$$p_r = \frac{\partial L}{\partial \dot{q}_r} \quad (23.11)$$

The variables are collected in a vector \bar{p} .

They are called the generalized momenta, as they coincide with linear momentum in the case of a Lagrangian describing a particle. Similar to the situation in the dynamics of particles, there can be conservation of generalized momentum. This is the case for the generalized momentum associated with ignorable coordinates, as defined in [Equations of Motion with Nonholonomic Constraints](#).

For the example pendulum, the generalized momenta are calculated as:

```
p = L.jacobian(qd).transpose()
sm.trigsimp(p)
```

$$\begin{bmatrix} m \left(\frac{l^2 \cos^2(q_2) \dot{q}_1}{12} + \frac{7l^2 \dot{q}_1}{3} - lq_3 \sin(q_2) \dot{q}_2 + l \cos(q_2) \dot{q}_3 + q_3^2 \cos^2(q_2) \dot{q}_1 \right) \\ m \left(\frac{l^2 \dot{q}_2}{12} - lq_3 \sin(q_2) \dot{q}_1 + q_3^2 \dot{q}_2 \right) \\ m (l \cos(q_2) \dot{q}_1 + \dot{q}_3) \end{bmatrix} \quad (23.12)$$

23.6 Constrained equations of motion

When using Kane's method, constraints are handled by dividing the generalized speeds into two sets: the dependent and independent generalized speeds. Depending on the type of constraints, the dependent generalized speeds are eliminated by solving the constraint equation (for non-holonomic constraints) or the time derivative of the constraint equation (holonomic constraints). Kane's method only gives rise to $p = n - m$ dynamical equations, one for each independent generalized speed.

The Lagrange method gives rise to n dynamical equations, one for each generalized coordinate. To eliminate the constraints, and end up with the right number of equations ($n - m$, one for each degree of freedom), both the generalized speeds and the generalized coordinates should be solved using the constraint equation. For non-holonomic constraints, this elimination is not possible (by the definition of non-holonomic), and for holonomic constraints this elimination requires solving often difficult non-linear equations for the generalized coordinates. The method of elimination is therefore not useful within the Lagrange method.

Instead, constraints are handled using a generalized version of the approach in [Exposing Noncontributing Forces](#). First the constraints are omitted. Then a constraint force is added, with a known direction, but unknown magnitude. Finally, the (second) time derivative of the constraint equation is then appended to the equations found with the Euler-Lagrange equation.

For example, consider a particle of mass m and position $\bar{r}^{P/O} = q_1 \hat{n}_x + q_2 \hat{n}_y + q_3 \hat{n}_z$ on a slope $q_1 = q_2$. The unconstrained Lagrangian is $L = \frac{1}{2}m(\dot{q}_1^2 + \dot{q}_2^2 + \dot{q}_3^2) - mgq_3$. The constraint force is perpendicular to the slope, so is described as $\bar{F} = F \hat{n}_x - F \hat{n}_y$. The appended equation is the second time derivative of the constraint equation $\ddot{q}_1 - \ddot{q}_2 = 0$. Combining all, gives:

$$\begin{aligned} m\ddot{q}_1 &= F \\ m\ddot{q}_2 &= -F \\ m\ddot{q}_3 + mg &= 0 \\ \ddot{q}_1 - \ddot{q}_2 &= 0 \end{aligned} \quad (23.13)$$

This can be put in matrix-form, by extracting the unknown acceleration and force magnitude:

$$\begin{bmatrix} m & 0 & 0 & -1 \\ 0 & m & 0 & 1 \\ 0 & 0 & m & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \ddot{q}_1 \\ \ddot{q}_2 \\ \ddot{q}_3 \\ F \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -mg \\ 0 \end{bmatrix} \quad (23.14)$$

It can be challenging to find the direction of the constraint force from the geometry of the system directly. There is a trick, called the method of the [Lagrange multipliers](#), to quickly find the correct generalized forces associated with the constraint forces.

Given a motion constraint (time derivatives of configuration constraint or a nonholonomic constraint) in the general form

$$\sum_r a_r(\bar{q}) \dot{q}_r = 0 \quad (23.15)$$

The generalized force is found as:

$$F_r = \lambda a_r(\bar{q}) \quad (23.16)$$

Here λ is a variable encoding the magnitude of the constraint force. It is called the Lagrange multiplier. The same λ is used for each r , that is, each constraint has a single associated Lagrange multiplier.

Due to how it is constructed, the power produced by the constraint force is always zero, as expected.

$$P = \sum_r F_r \dot{q}_r = \sum_r \lambda a_r(\bar{q}) \dot{q}_r = \lambda \sum_r a_r(\bar{q}) \dot{q}_r = \lambda \cdot 0 \quad (23.17)$$

For example, consider the pointmass to be constrained to move in a bowl $q_1^2 + q_2^2 + q_3^2 - 1 = 0$, [Fig. 23.2](#). Taking the time derivative gives: $a_1 = 2q_1$, $a_2 = 2q_2$, and $a_3 = 2q_3$. This results in generalized reaction forces $F_1 = 2\lambda q_1$, $F_2 = 2\lambda q_2$ and $F_3 = 2\lambda q_3$.

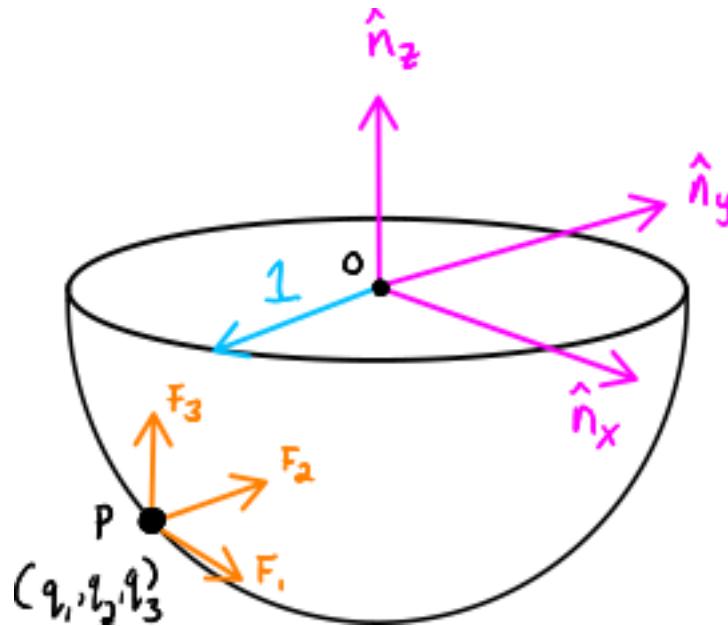


Fig. 23.2: Point mass P constrained to the surface of a spherical bowl with radius 1 and constraint force measure numbers F_1, F_2, F_3 .

Often, there are multiple constraints on the same system. For convenience, the handling of these constraints can be combined. Consider the $m + M$ dimensional general constraint equations consisting of the time derivatives of the holonomic

constraints and/or the non-holonomic constraints:

$$\bar{f}_{hn}(\bar{q}, \bar{\dot{q}}) = \mathbf{M}_{hn}\bar{\dot{q}} = 0, \quad (23.18)$$

the combined constraint forces are given as:

$$\bar{F}_r = \mathbf{M}_{hn}^T \bar{\lambda}, \quad (23.19)$$

where $\bar{\lambda}$ is a vector of $m + M$ Lagrange multipliers, one for each constraint (row in \mathbf{M}_{hn}).

23.6.1 Example: turning the freely floating body discussed earlier into a rolling sphere.

The non-slip condition of the rolling sphere is split into three constraints: the velocity of the contact point (G) is zero in the \hat{n}_x , the \hat{n}_y and the \hat{n}_z direction. The first two constraints are non-holonomic, the last constraint is the time derivative of a holonomic constraint. All three constraints are enforced by contact forces in their respective directions.

The contact point can be found according by $\bar{r}^{G/C} = -r\hat{n}_z$. By using the *Velocity Two Point Theorem*, the following constraints are found.

$$\begin{aligned} \bar{n}_x \cdot ({}^N\bar{v}^C + {}^N\bar{\omega}^B \times (-r\hat{n}_z)) &= 0 \\ \bar{n}_y \cdot ({}^N\bar{v}^C + {}^N\bar{\omega}^B \times (-r\hat{n}_z)) &= 0 \\ \bar{n}_z \cdot ({}^N\bar{v}^C + {}^N\bar{\omega}^B \times (-r\hat{n}_z)) &= 0 \end{aligned} \quad (23.20)$$

These can be used to derive the constraint force and the additional equations using the Lagrange-multiplier method as shown below. Note that here only the first time derivative of the constraint equation is used, again because the second time derivatives of the generalized coordinates appear.

Frames and Body Setup

Setting up reference frames

```
psi,theta, phi, x, y, z = me.dynamicsymbols('psi theta phi x y z')
N = me.ReferenceFrame('N')
B = me.ReferenceFrame('B')
B.orient_body_fixed(N, (psi, theta, phi), 'zxy')

# Mass and inertia
m, Ixx, Iyy, Izz = sm.symbols('M, I_{xx}, I_{yy}, I_{zz}')
I_B = me.inertia(B, Ixx, Iyy, Izz)
```

Finding the kinetic energy:

```
omega_B = B.ang_vel_in(N)
r_com = x*N.x + y*N.y + z*N.z
v_com = r_com.dt(N)
K = omega_B.dot(I_B.dot(omega_B)) / 2 + m*v_com.dot(v_com) / 2
```

Deriving equations of motion:

```
t = me.dynamicsymbols._t
q = sm.Matrix([psi, theta, phi, x, y, z])
qd = q.diff(t)
qdd = qd.diff(t)

L = sm.Matrix([K])
```

(continues on next page)

(continued from previous page)

```
left_hand_side = L.jacobian(qd).diff(t) - L.jacobian(q)
qdd_zerod = {qddr: 0 for qddr in qdd}
Md = left_hand_side.jacobian(qdd)
qd = left_hand_side.xreplace(qdd_zerod)
```

To make this free floating body a rolling wheel, three constraints are needed: the velocity of the contact point should be zero in \hat{n}_x , \hat{n}_y and \hat{n}_x direction.

```
lambda1, lambda2, lambda3 = me.dynamicsymbols('lambda1, lambda2, lambda3')
constraint = (v_com + B.ang_vel_in(N).cross(-N.z)).to_matrix(N)
M_hn = constraint.jacobian(qd)
diff_constraint = constraint.diff(t)
sm.trigsimp(constraint)
```

$$\begin{bmatrix} -\sin(\psi)\dot{\theta} - \cos(\psi)\cos(\theta)\dot{\phi} + \dot{x} \\ -\sin(\psi)\cos(\theta)\dot{\phi} + \cos(\psi)\dot{\theta} + \dot{y} \\ \dot{z} \end{bmatrix} \quad (23.21)$$

This constraint information must then be added to the original equations. To do so, we make use of a useful fact.

```
diff_constraint.jacobian(qdd) - M_hn
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (23.22)$$

This equality is true for all constraints, as can easily be shown by taking the time derivative of the constraint equation, using the chain rule.

The combined equations can now be written in a block matrix form:

$$\begin{bmatrix} \mathbf{M}_d & -\mathbf{M}_{hn}^T \\ \mathbf{M}_{hn} & 0 \end{bmatrix} \begin{bmatrix} \ddot{\vec{q}} \\ \dot{\lambda} \end{bmatrix} = \begin{bmatrix} \bar{F}_r - \bar{g}_d \\ -\frac{\partial \mathbf{M}_{hn} \dot{\vec{q}}}{\partial \vec{q}} \end{bmatrix}, \quad (23.23)$$

where \bar{g} is the dynamic bias, and the last term on the right hand side, called the constraint bias, can be quickly computed as:

```
constraint_bias = diff_constraint.xreplace({qddr : 0 for qddr in qdd})
```

We call the block matrix called the extended mass matrix, and the vector on the right hand side the extended dynamic bias.

With these $n + m + M$ equations, it is possible to solve for $\ddot{\vec{q}}$ and λ . It is therefore possible to integrate/simulate the system directly. However, because only the second derivative of the constraint is satisfied, numerical errors can build up, so the constraint is not satisfied. It is better to use a differential algebraic solver, as discussed in [Equations of Motion with Holonomic Constraints](#). See the [scikit.ode](#) documentation for a worked example.

The method of the Lagrange multiplier can of course also be used within Kane's method. However, it increases the number of equations, which is why the elimination approach is often preferred there. An exception being scenarios where the constraint force itself is a useful output, for instance to check no-slip conditions in case of limited friction.

23.7 Lagrange's vs Kane's

The is book has now presented two alternatives to the Newton-Euler method: Kane's method and Lagrange's method. This raises the questions: when should each alternative method be used?

For constrained systems, Kane's method has the advantage that the equations of motion are given for a set of independent generalized speeds only. In other words, Kane's method gives a minimal set of equations. This can give rise to simplified equations, additional insight, and numerically more efficient simulation. This also gives the benefit that Lagrange multipliers are not needed when solving constrained systems with Kane's method.

Furthermore, the connection from Kane's method to vector mechanics, that is, Newton's laws, is clearer, which can provide additional insight, and make it easier to incorporate non-conservative forces such as friction.

On the other hand, the Lagrange method only requires energies as input, for which only the velocities of the bodies are needed. Therefore, it can be simpler to derive than the accelerations which are needed for Kane's method.

Furthermore, the Lagrange method results in a set of equations with well understood structures and properties. These structures and properties are not studied further in these materials. A starting point for further study is [Noether's theorem](#), which extends the idea of ignorable coordinates to find conserved quantities like momentum and energy.

UNCONSTRAINED EQUATIONS OF MOTION WITH THE TMT METHOD

Note: You can download this example as a Python script: `tmt.py` or Jupyter Notebook: `tmt.ipynb`.

```
import numpy as np
import sympy as sm
import sympy.physics.mechanics as me
me.init_vprinting(use_latex='mathjax')

class ReferenceFrame(me.ReferenceFrame):
    def __init__(self, *args, **kwargs):
        kwargs.pop('latexs', None)
        lab = args[0].lower()
        tex = r'\hat{{}}_{{}}_{{}}'
        super(ReferenceFrame, self).__init__(*args,
                                             latexs=(tex.format(lab, 'x'),
                                                      tex.format(lab, 'y'),
                                                      tex.format(lab, 'z')),
                                             **kwargs)
me.ReferenceFrame = ReferenceFrame
```

There are [several mathematical methods](#) available to formulate the equations of motion of a multibody system. These different methods offer various advantages and disadvantages over Newton and Euler's original formulations and among each other. For example, [Joseph-Louis Lagrange](#) developed a way to arrive at the equations of motion from the descriptions of kinetic and potential energy of the system. [Sir William Hamilton](#) then reformulated Lagrange's approach in terms of *generalized momenta* instead of energy. Since then, [Gibbs & Appell](#), [Kane \[Kane1985\]](#), and others have proposed more methods. In this chapter, we present one of these alternative methods called the "TMT Method". The details and derivation of the TMT Method can be found in [\[Vallery2020\]](#).

Vallery and Schwab show how the collection of Newton-Euler equations for each individual rigid body can be transformed into the reduced dynamical differential equations associated with the generalized coordinates, speeds, and accelerations using the **T** matrix. This **T** matrix is populated by the measure numbers of the partial velocities expressed in the inertial reference frame.

Given ν rigid bodies in a multibody system described by n generalized coordinates and generalized speeds, the velocities of each mass center and the angular velocities of each body in an inertial reference frame N can be written in column

vector \bar{v} form by extracting the measure numbers in the inertial reference frame N of each velocity term.

$$\bar{v}(\bar{u}, \bar{q}, t) = \begin{bmatrix} {}^N\bar{v}^{B_{1o}} \cdot \hat{n}_x \\ {}^N\bar{v}^{B_{1o}} \cdot \hat{n}_y \\ {}^N\bar{v}^{B_{1o}} \cdot \hat{n}_z \\ {}^N\bar{\omega}^{B_1} \cdot \hat{n}_x \\ {}^N\bar{\omega}^{B_1} \cdot \hat{n}_y \\ {}^N\bar{\omega}^{B_1} \cdot \hat{n}_z \\ \vdots \\ {}^N\bar{v}^{B_{\nu o}} \cdot \hat{n}_x \\ {}^N\bar{v}^{B_{\nu o}} \cdot \hat{n}_y \\ {}^N\bar{v}^{B_{\nu o}} \cdot \hat{n}_z \\ {}^N\bar{\omega}^{B_\nu} \cdot \hat{n}_x \\ {}^N\bar{\omega}^{B_\nu} \cdot \hat{n}_y \\ {}^N\bar{\omega}^{B_\nu} \cdot \hat{n}_z \end{bmatrix} \in \mathbb{R}^{6\nu} \quad (24.1)$$

The measure numbers of the partial velocities with respect to each of the n generalized speeds in \bar{u} can be efficiently found by taking the Jacobian of \bar{v} with respect to the generalized speeds, which we will name matrix \mathbf{T} .

$$\mathbf{T} = \mathbf{J}_{\bar{v}, \bar{u}} \in \mathbb{R}^{6\nu \times n} \quad \text{where} \quad \bar{v} = \mathbf{T}\bar{u} \quad (24.2)$$

For each set of six rows in \bar{v} tied to a single rigid body, the associated mass and inertia for that body can be written as:

$$\mathbf{M}_{B_1} = \begin{bmatrix} m_{B_1} & 0 & 0 & 0 & 0 & 0 \\ 0 & m_{B_1} & 0 & 0 & 0 & 0 \\ 0 & 0 & m_{B_1} & 0 & 0 & 0 \\ 0 & 0 & 0 & \check{I}^{B_1/B_{1o}} \cdot \hat{n}_x \hat{n}_x & \check{I}^{B_1/B_{1o}} \cdot \hat{n}_x \hat{n}_y & \check{I}^{B_1/B_{1o}} \cdot \hat{n}_x \hat{n}_z \\ 0 & 0 & 0 & \check{I}^{B_1/B_{1o}} \cdot \hat{n}_y \hat{n}_x & \check{I}^{B_1/B_{1o}} \cdot \hat{n}_y \hat{n}_y & \check{I}^{B_1/B_{1o}} \cdot \hat{n}_y \hat{n}_z \\ 0 & 0 & 0 & \check{I}^{B_1/B_{1o}} \cdot \hat{n}_z \hat{n}_x & \check{I}^{B_1/B_{1o}} \cdot \hat{n}_z \hat{n}_y & \check{I}^{B_1/B_{1o}} \cdot \hat{n}_z \hat{n}_z \end{bmatrix} \quad (24.3)$$

Multiplying the velocities with this matrix gives the momenta of each rigid body.

$$\mathbf{M}_{B_1} \bar{v}_{B_1} = \begin{bmatrix} \bar{p}^{B_{1o}} \cdot \hat{n}_x \\ \bar{p}^{B_{1o}} \cdot \hat{n}_y \\ \bar{p}^{B_{1o}} \cdot \hat{n}_z \\ \bar{H}^{B_1/B_{1o}} \cdot \hat{n}_x \\ \bar{H}^{B_1/B_{1o}} \cdot \hat{n}_y \\ \bar{H}^{B_1/B_{1o}} \cdot \hat{n}_z \end{bmatrix} \quad (24.4)$$

The matrices for each rigid body can then be assembled into a matrix for the entire set of rigid bodies.

$$\mathbf{M} = \begin{bmatrix} \mathbf{M}_{B_1} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{M}_{B_2} & \dots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{M}_{B_\nu} \end{bmatrix} \quad (24.5)$$

Allowing the momenta of all the rigid bodies to be found by matrix multiplication of $\mathbf{M}\bar{v}$.

A vector \bar{F} of resultant forces and torques of couples acting on each rigid body can be formed in a similar manner as \bar{v} ,

by extracting the measure numbers in the inertial reference frame.

$$\bar{F} = \begin{bmatrix} \bar{R}^{B_{1o}} \cdot \hat{n}_x \\ \bar{R}^{B_{1o}} \cdot \hat{n}_y \\ \bar{R}^{B_{1o}} \cdot \hat{n}_z \\ \bar{T}^{B_1} \cdot \hat{n}_x \\ \bar{T}^{B_1} \cdot \hat{n}_y \\ \bar{T}^{B_1} \cdot \hat{n}_z \\ \vdots \\ \bar{R}^{B_{2o}} \cdot \hat{n}_x \\ \bar{R}^{B_{2o}} \cdot \hat{n}_y \\ \bar{R}^{B_{2o}} \cdot \hat{n}_z \\ \bar{T}^{B_2} \cdot \hat{n}_x \\ \bar{T}^{B_2} \cdot \hat{n}_y \\ \bar{T}^{B_2} \cdot \hat{n}_z \end{bmatrix} \quad (24.6)$$

The dynamical differential equations for the entire Newton-Euler system are then:

$$\frac{d\mathbf{M}\bar{v}}{dt} = \bar{F} \in \mathbb{R}^{6\nu} \quad (24.7)$$

We know that selecting n generalized coordinates for such a system allows us to write the dynamical differential equations as a set of n equations which is, in general, much smaller than 6ν equations due to the large number of holonomic constraints that represent the connections of all the bodies in the system. Vallery and Schwab show that the mass matrix \mathbf{M}_d for this reduced set of equations can be efficiently calculated using the \mathbf{T} matrix ([Vallery2020], pg. 349):

$$\mathbf{M}_d = -\mathbf{T}^T \mathbf{M} \mathbf{T} \quad (24.8)$$

and that the forces not proportional to the generalized accelerations is found with:

$$\bar{g}_d = \mathbf{T}^T (\bar{F} - \bar{g}) \quad (24.9)$$

where¹:

$$\bar{g} = \left. \frac{d\mathbf{M}\bar{v}}{dt} \right|_{\dot{u}=\bar{0}} \quad (24.10)$$

The equations of motion then take this form:

$$\bar{0} = \mathbf{M}_d \ddot{u} + \bar{g}_d = -\mathbf{T}^T \mathbf{M} \mathbf{T} \dot{u} + \mathbf{T}^T (\bar{F} - \bar{g}) \quad (24.11)$$

These equations are equivalent to Kane's Equations.

24.1 Example Formulation

Let us return once again to the holonomic system introduced in [Example of Kane's Equations](#).

Start by introducing the variables.

¹ Note that my \bar{g} is slightly different than the one presented in [Vallery2020] to make sure the time derivative of the angular momenta are properly calculated.

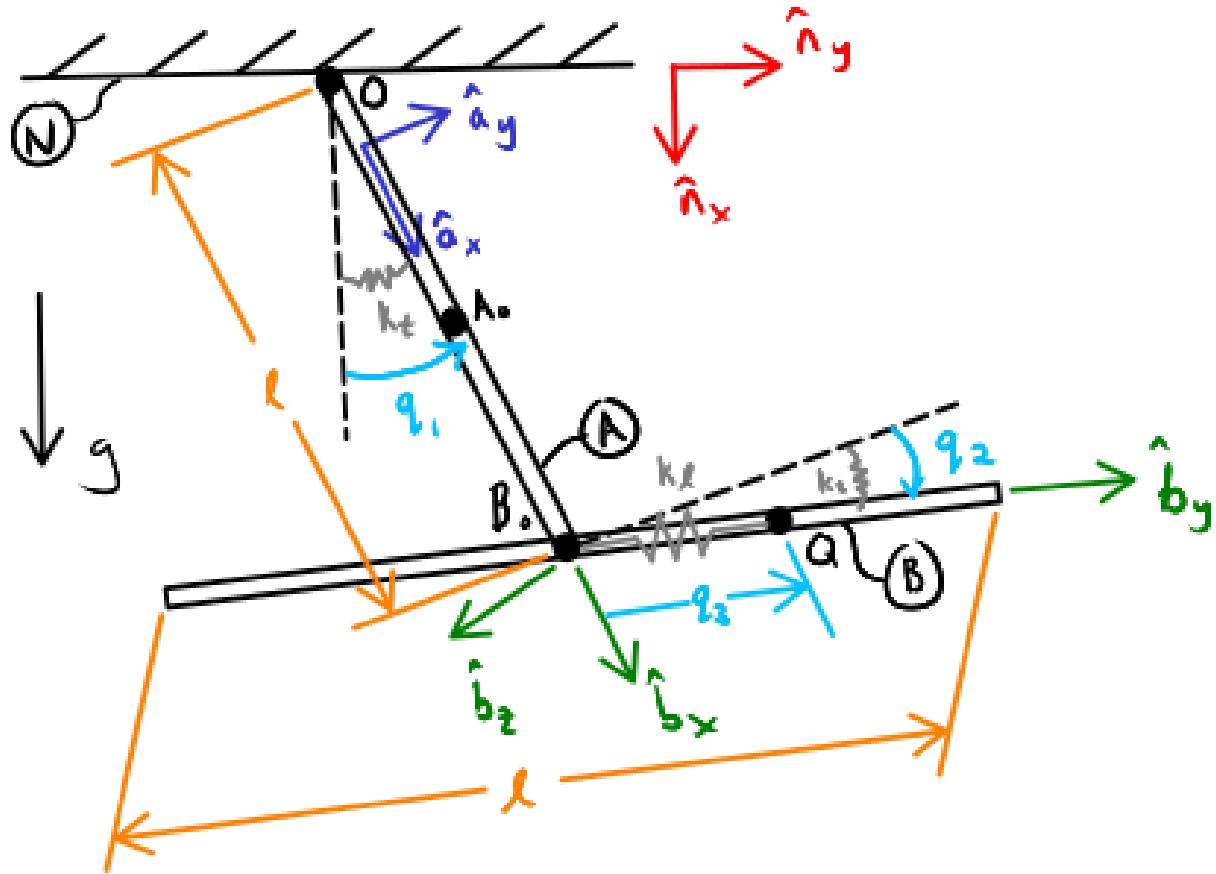


Fig. 24.1: Three dimensional pendulum made up of two pinned rods and a sliding mass on rod B . Each degree of freedom is resisted by a linear spring. When the generalized coordinates are all zero, the two rods are perpendicular to each other.

```

m, g, kt, kl, l = sm.symbols('m, g, k_t, k_l, l')
q1, q2, q3 = me.dynamicsymbols('q1, q2, q3')
u1, u2, u3 = me.dynamicsymbols('u1, u2, u3')
t = me.dynamicsymbols._t

q = sm.Matrix([q1, q2, q3])
u = sm.Matrix([u1, u2, u3])
p = sm.Matrix([g, kl, kt, l, m])
q, u, p

```

$$\left(\begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix}, \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}, \begin{bmatrix} g \\ k_l \\ k_t \\ l \\ m \end{bmatrix} \right) \quad (24.12)$$

The derivation of the kinematics is done in the same way as before.

```

N = me.ReferenceFrame('N')
A = me.ReferenceFrame('A')
B = me.ReferenceFrame('B')

A.orient_axis(N, q1, N.z)
B.orient_axis(A, q2, A.x)

A.set_ang_vel(N, u1*N.z)
B.set_ang_vel(A, u2*A.x)

O = me.Point('O')
Ao = me.Point('A_O')
Bo = me.Point('B_O')
Q = me.Point('Q')

Ao.set_pos(O, l/2*A.x)
Bo.set_pos(O, l*A.x)
Q.set_pos(Bo, q3*B.y)

O.set_vel(N, 0)
Ao.v2pt_theory(O, N, A)
Bo.v2pt_theory(O, N, A)
Q.set_vel(B, u3*B.y)
Q.v1pt_theory(Bo, N, B)

Ao.vel(N), A.ang_vel_in(N), Bo.vel(N), B.ang_vel_in(N), Q.vel(N)

```

$$\left(\frac{lu_1}{2}\hat{a}_y, u_1\hat{n}_z, lu_1\hat{a}_y, u_2\hat{a}_x + u_1\hat{n}_z, -q_3u_1 \cos(q_2)\hat{b}_x + u_3\hat{b}_y + q_3u_2\hat{b}_z + lu_1\hat{a}_y \right) \quad (24.13)$$

Only the contributing forces need be declared (noncontributing would cancel out in the TMT transformation if included). Do not forget Newton's Third Law and be sure to include the equal and opposite reactions.

```

R_Ao = m*g*N.x
R_Bo = m*g*N.x + kl*q3*B.y
R_Q = m/4*g*N.x - kl*q3*B.y

```

(continues on next page)

(continued from previous page)

```
T_A = -kt*q1*N.z + kt*q2*A.x
T_B = -kt*q2*A.x
```

The inertia dyadics of each body will be needed.

```
I = m*l**2/12
I_A_Ao = I*me.outer(A.y, A.y) + I*me.outer(A.z, A.z)
I_B_Bo = I*me.outer(B.x, B.x) + I*me.outer(B.z, B.z)
```

24.2 Create the TMT Components

The vector \bar{v} is formed from the velocities and angular velocities of each rigid body or particle.

```
v = sm.Matrix([
    Ao.vel(N).dot(N.x),
    Ao.vel(N).dot(N.y),
    Ao.vel(N).dot(N.z),
    A.ang_vel_in(N).dot(N.x),
    A.ang_vel_in(N).dot(N.y),
    A.ang_vel_in(N).dot(N.z),
    Bo.vel(N).dot(N.x),
    Bo.vel(N).dot(N.y),
    Bo.vel(N).dot(N.z),
    B.ang_vel_in(N).dot(N.x),
    B.ang_vel_in(N).dot(N.y),
    B.ang_vel_in(N).dot(N.z),
    Q.vel(N).dot(N.x),
    Q.vel(N).dot(N.y),
    Q.vel(N).dot(N.z),
])
v
```

$$\begin{bmatrix} -\frac{lu_1 \sin(q_1)}{2} \\ \frac{lu_1 \cos(q_1)}{2} \\ 0 \\ 0 \\ 0 \\ u_1 \\ -lu_1 \sin(q_1) \\ lu_1 \cos(q_1) \\ 0 \\ u_2 \cos(q_1) \\ u_2 \sin(q_1) \\ u_1 \\ -lu_1 \sin(q_1) - q_3 u_1 \cos(q_1) \cos(q_2) + q_3 u_2 \sin(q_1) \sin(q_2) - u_3 \sin(q_1) \cos(q_2) \\ lu_1 \cos(q_1) - q_3 u_1 \sin(q_1) \cos(q_2) - q_3 u_2 \sin(q_2) \cos(q_1) + u_3 \cos(q_1) \cos(q_2) \\ q_3 u_2 \cos(q_2) + u_3 \sin(q_2) \end{bmatrix} \quad (24.14)$$

The inertial matrices for each body and the particle Q are:

```
MA = sm.diag(m, m, m).col_join(sm.zeros(3)).row_join(sm.zeros(3).col_join(I_A_Ao.to_
→matrix(N)))
MA
```

$$\begin{bmatrix} m & 0 & 0 & 0 & 0 & 0 \\ 0 & m & 0 & 0 & 0 & 0 \\ 0 & 0 & m & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{l^2 m \sin^2(q_1)}{12} & -\frac{l^2 m \sin(q_1) \cos(q_1)}{12} & 0 \\ 0 & 0 & 0 & -\frac{l^2 m \sin(q_1) \cos(q_1)}{12} & \frac{l^2 m \cos^2(q_1)}{12} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{l^2 m}{12} \end{bmatrix} \quad (24.15)$$

```
MB = sm.diag(m, m, m).col_join(sm.zeros(3)).row_join(sm.zeros(3).col_join(I_B_Bo.to_
→matrix(N)))
sm.trigsimp(MB)
```

$$\begin{bmatrix} m & 0 & 0 & 0 & 0 & 0 \\ 0 & m & 0 & 0 & 0 & 0 \\ 0 & 0 & m & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{l^2 m (\sin^2(q_1) \sin^2(q_2) - \sin^2(q_1) + 1)}{12} & \frac{l^2 m \sin(q_1) \cos(q_1) \cos^2(q_2)}{12} & \frac{l^2 m (\cos(q_1 - 2q_2) - \cos(q_1 + 2q_2))}{48} \\ 0 & 0 & 0 & \frac{l^2 m \sin(q_1) \cos(q_1) \cos^2(q_2)}{12} & \frac{l^2 m (-\cos^2(q_1) \cos^2(q_2) + 1)}{12} & -\frac{l^2 m (-\sin(q_1 - 2q_2) + \sin(q_1 + 2q_2))}{48} \\ 0 & 0 & 0 & \frac{l^2 m (\cos(q_1 - 2q_2) - \cos(q_1 + 2q_2))}{48} & -\frac{l^2 m (-\sin(q_1 - 2q_2) + \sin(q_1 + 2q_2))}{48} & \frac{l^2 m \cos^2(q_2)}{12} \end{bmatrix} \quad (24.16)$$

```
MQ = sm.diag(m/4, m/4, m/4)
MQ
```

$$\begin{bmatrix} \frac{m}{4} & 0 & 0 \\ 0 & \frac{m}{4} & 0 \\ 0 & 0 & \frac{m}{4} \end{bmatrix} \quad (24.17)$$

Note that these matrices change with time because we've expressed the inertia scalars in the inertial reference frame N . The matrices for all of the bodies can be assembled into \mathbf{M} :

```
M = sm.diag(MA, MB, MQ)
```

\bar{F} is constructed to match the order of \bar{v} :

```
F = sm.Matrix([
    R_Ao.dot(N.x),
    R_Ao.dot(N.y),
    R_Ao.dot(N.z),
    T_Ao.dot(N.x),
    T_Ao.dot(N.y),
    T_Ao.dot(N.z),
    R_Bo.dot(N.x),
    R_Bo.dot(N.y),
    R_Bo.dot(N.z),
```

(continues on next page)

(continued from previous page)

```

T_B.dot(N.x),
T_B.dot(N.y),
T_B.dot(N.z),
R_Q.dot(N.x),
R_Q.dot(N.y),
R_Q.dot(N.z),
]
F
    
```

$$\begin{bmatrix}
 gm \\
 0 \\
 0 \\
 k_t q_2 \cos(q_1) \\
 k_t q_2 \sin(q_1) \\
 -k_t q_1 \\
 gm - k_l q_3 \sin(q_1) \cos(q_2) \\
 k_l q_3 \cos(q_1) \cos(q_2) \\
 k_l q_3 \sin(q_2) \\
 -k_t q_2 \cos(q_1) \\
 -k_t q_2 \sin(q_1) \\
 0 \\
 \frac{gm}{4} + k_l q_3 \sin(q_1) \cos(q_2) \\
 -k_l q_3 \cos(q_1) \cos(q_2) \\
 -k_l q_3 \sin(q_2)
 \end{bmatrix} \quad (24.18)$$

These are the components we need to form the reduced dynamical differential equations.

24.3 Formulate the reduced equations of motion

First find \mathbf{T} using the Jacobian:

```

T = v.jacobian(u)
T
    
```

$$\begin{bmatrix}
 -l \sin(q_1) & 0 & 0 \\
 \frac{l \cos(q_1)}{2} & 0 & 0 \\
 0 & 0 & 0 \\
 0 & 0 & 0 \\
 0 & 0 & 0 \\
 1 & 0 & 0 \\
 -l \sin(q_1) & 0 & 0 \\
 l \cos(q_1) & 0 & 0 \\
 0 & 0 & 0 \\
 0 & \cos(q_1) & 0 \\
 0 & \sin(q_1) & 0 \\
 1 & 0 & 0 \\
 -l \sin(q_1) - q_3 \cos(q_1) \cos(q_2) & q_3 \sin(q_1) \sin(q_2) & -\sin(q_1) \cos(q_2) \\
 l \cos(q_1) - q_3 \sin(q_1) \cos(q_2) & -q_3 \sin(q_2) \cos(q_1) & \cos(q_1) \cos(q_2) \\
 0 & q_3 \cos(q_2) & \sin(q_2)
 \end{bmatrix} \quad (24.19)$$

and then compute \bar{g} :

```
qd_repl = dict(zip(q.diff(t), u))
ud_repl = {udi: 0 for udi in u.diff(t)}
gbar = (M*v).diff(t).xreplace(qd_repl).xreplace(ud_repl)
sm.trigsimp(gbar)
```

$$\begin{bmatrix} -\frac{lmu_1^2 \cos(q_1)}{2} \\ -\frac{lmu_1^2 \sin(q_1)}{2} \\ 0 \\ 0 \\ 0 \\ 0 \\ -lmu_1^2 \cos(q_1) \\ -lmu_1^2 \sin(q_1) \\ 0 \\ \frac{l^2 m(u_1 \cos(q_1) \cos(q_2) - 2u_2 \sin(q_1) \sin(q_2))u_1 \sin(q_2)}{12} \\ \frac{l^2 m(u_1 \sin(q_1) \cos(q_2) + 2u_2 \sin(q_2) \cos(q_1))u_1 \sin(q_2)}{12} \\ -\frac{l^2 mu_1 u_2 \sin(2q_2)}{12} \\ m(-lu_1^2 \cos(q_1) + q_3 u_1^2 \sin(q_1) \cos(q_2) + 2q_3 u_1 u_2 \sin(q_2) \cos(q_1) + q_3 u_2^2 \sin(q_1) \cos(q_2) - 2u_1 u_3 \cos(q_1) \cos(q_2) + 2u_2 u_3 \sin(q_1) \sin(q_2)) \\ m(-lu_1^2 \sin(q_1) - q_3 u_1^2 \cos(q_1) \cos(q_2) + 2q_3 u_1 u_2 \sin(q_1) \sin(q_2) - q_3 u_2^2 \cos(q_1) \cos(q_2) - 2u_1 u_3 \sin(q_1) \cos(q_2) - 2u_2 u_3 \sin(q_2) \cos(q_1)) \\ \frac{m(-q_3 u_2 \sin(q_2) + 2u_3 \cos(q_2))u_2}{4} \end{bmatrix} \quad (24.20)$$

The reduced mass matrix is then formed with $-\mathbf{T}^T \mathbf{M} \mathbf{T}$:

```
Md = sm.trigsimp(-T.transpose() * M * T)
Md
```

$$\begin{bmatrix} -\frac{m(l^2 \cos^2(q_2) + 19l^2 + 3q_3^2 \cos^2(q_2))}{12} & \frac{l m q_3 \sin(q_2)}{4} & -\frac{l m \cos(q_2)}{4} \\ \frac{l m q_3 \sin(q_2)}{4} & -\frac{m(l^2 + 3q_3^2)}{12} & 0 \\ -\frac{l m \cos(q_2)}{4} & 0 & -\frac{m}{4} \end{bmatrix} \quad (24.21)$$

and the reduced remainder term is formed with $\mathbf{T}^T (\bar{F} - \bar{g})$:

```
gd = sm.trigsimp(T.transpose() * (F - gbar))
gd
```

$$\begin{bmatrix} -\frac{7glm \sin(q_1)}{4} - \frac{gmq_3 \cos(q_1 - q_2)}{8} - \frac{gmq_3 \cos(q_1 + q_2)}{8} - k_t q_1 + \frac{l^2 mu_1 u_2 \sin(2q_2)}{12} + \frac{l m q_3 u_2^2 \cos(q_2)}{4} + \frac{l m u_2 u_3 \sin(q_2)}{2} + \frac{m q_3^2 u_1 u_2 \sin(2q_2)}{4} - \frac{m q_3 u_2 u_3}{2} \\ \frac{gmq_3 \cos(q_1 - q_2)}{8} - \frac{gmq_3 \cos(q_1 + q_2)}{8} - k_t q_2 - \frac{l^2 mu_1^2 \sin(2q_2)}{24} - \frac{m q_3^2 u_1^2 \sin(2q_2)}{8} - \frac{m q_3 u_2}{2} \\ -\frac{gm \sin(q_1) \cos(q_2)}{4} - k_l q_3 + \frac{m q_3 u_1^2 \cos^2(q_2)}{4} + \frac{m q_3 u_2^2}{4} \end{bmatrix} \quad (24.22)$$

24.4 Evaluate the equations of motion

Now we can check to see if these dynamical differential equations are the same as the ones we found with Kane's Method by evaluating them with the same set of numbers we used in [Numerical Evaluation](#). The input values were:

```
u_vals = np.array([
    0.1,  # u1, rad/s
    2.2,  # u2, rad/s
    0.3,  # u3, m/s
])

q_vals = np.array([
    np.deg2rad(25.0),  # q1, rad
    np.deg2rad(5.0),  # q2, rad
    0.1,  # q3, m
])

p_vals = np.array([
    9.81,  # g, m/s**2
    2.0,  # k1, N/m
    0.01,  # kt, Nm/rad
    0.6,  # l, m
    1.0,  # m, kg
])
```

We can lambdify `Md` and `gd` to see if these give the same values as those found with Kane's Equations:

```
eval_d = sm.lambdify((u, q, p), (Md, gd))

Md_vals, gd_vals = eval_d(u_vals, q_vals, p_vals)
Md_vals, gd_vals
```

```
(array([[ -0.60225313,  0.00130734, -0.1494292 ],
       [ 0.00130734, -0.0325      ,  0.          ,  1.          ],
       [-0.1494292 ,  0.          , -0.25      ,  1.          ],
array([[ -4.48963535],
       [-0.02486744],
       [-1.1112791 ]]))
```

These numerical arrays are identical to our prior results. The state derivatives then should also be identical:

```
eval_d(u_vals, q_vals, p_vals)
ud_vals = -np.linalg.solve(Md_vals, np.squeeze(gd_vals))
ud_vals

array([-7.46056427, -1.06525862,  0.01418834])
```

which they are. We can be fairly confident that Kane's method and the TMT method result in the same equations of motion for this system.

CHAPTER
TWENTYFIVE

NOTATION

This explains the notation in the book. The mathematical symbol is shown and then an example of a variable name that we use in the code.

25.1 General

x, \mathbf{x}, X

Scalars are normal mathematical font.

\mathbf{R}, \mathbf{R}

Matrices are capitalized letters in bold font.

$\mathbf{J}_{\bar{v}, \bar{u}}$

The Jacobian of the vector function \bar{v} with respect to the entries in vector \bar{u} where the (i, j) entries of the Jacobian are $\mathbf{J}_{ij} = \frac{\partial v_i}{\partial u_j}$.

25.2 Orientation of Reference Frames

$A, \mathbf{A}, \mathbf{\hat{A}}$

Reference frame A .

$\hat{a}_x, \hat{a}_y, \hat{a}_z, \mathbf{A.x}, \mathbf{A.y}, \mathbf{A.z}, \hat{\mathbf{a}}_x, \hat{\mathbf{a}}_y, \hat{\mathbf{a}}_z$

Right handed mutually perpendicular unit vectors fixed in reference frame A .

${}^A\mathbf{C}^B, \mathbf{A.C.B}, \mathbf{\hat{C}}^B$

Direction cosine matrix relating reference frames (or rigid bodies) B and A where this relation between the right handed mutually perpendicular unit vectors fixed in the two reference frames follow this relationship:

$$\begin{bmatrix} \hat{a}_x \\ \hat{a}_y \\ \hat{a}_z \end{bmatrix} = {}^A\mathbf{C}^B \begin{bmatrix} \hat{b}_x \\ \hat{b}_y \\ \hat{b}_z \end{bmatrix} \quad (25.1)$$

25.3 Vectors and Vector Differentiation

$\bar{v}, \mathbf{v}, \bar{\mathbf{V}}$

Vectors are indicated with a bar.

$\hat{u}, \mathbf{u}\hat{\mathbf{u}} = \mathbf{u}.\text{normalize}(), \hat{\mathbf{u}}$

Unit vectors are indicated with a hat.

$|\bar{v}|, \mathbf{v}.\text{magnitude}(), |\bar{\mathbf{v}}|$

Magnitude of a vector; Euclidean norm (2-norm).

$\bar{u} \cdot \bar{v}, \mathbf{u}.\text{dot}(\mathbf{v}), \bar{\mathbf{u}} \cdot \bar{\mathbf{v}}$

Dot product of two vectors.

$\bar{u} \times \bar{v}, \mathbf{u}.\text{cross}(\mathbf{v}), \bar{\mathbf{u}} \times \bar{\mathbf{v}}$

Cross product of two vectors.

$\bar{u} \otimes \bar{v}, \mathbf{u}.\text{outer}(\mathbf{v}), \bar{\mathbf{u}} \otimes \bar{\mathbf{v}}$

Outer product of two vectors.

$\frac{^A\partial\bar{v}}{\partial q}, \mathbf{dvdqA} = \mathbf{v}.\text{diff}(\mathbf{q}, \mathbf{A}), \overset{A}{\frac{\partial\bar{v}}{\partial\mathbf{q}}}$

Partial derivative of \bar{v} with respect to q when observed from A .

$\frac{^A\bar{d}\bar{v}}{dt}, \mathbf{dvdtA} = \mathbf{v}.\text{dt}(\mathbf{A}), \overset{A}{\frac{d\bar{v}}{dt}}$

Time derivative of \bar{v} when observed from A .

25.4 Angular and Translational Kinematics

${}^A\bar{\omega}^B, \mathbf{A_w_B}$

Angular velocity vector of reference frame or rigid body B when observed from reference frame or rigid body A .

${}^A\bar{\alpha}^B, \mathbf{A_alp_B}$

Angular acceleration vector of reference frame or rigid body B when observed from reference frame or rigid body A .

$\bar{r}^{P/O}, \mathbf{r_O_P}, \bar{r}^P$

Vector from point O to point P .

${}^A\bar{v}^P, \mathbf{A_v_P}$

Translational velocity of point P when observed from reference frame or rigid body A .

${}^A\bar{a}^P, \mathbf{A_a_P}$

Translational acceleration of point P when observed from reference frame or rigid body A .

25.5 Constraints

N, M, n, m, p

N coordinates, M holonomic constraints, n generalized coordinates and generalized speeds, m nonholonomic constraints, and p degrees of freedom. These are related by the two equations $n = N - M$ and $p = n - m$.

$\bar{f}_h(q_1, \dots, q_N, t) = 0$ where $\bar{f}_h \in \mathbb{R}^M$, **fh**

Vector function of M holonomic constraint equations among the N coordinates.

$\bar{f}_n(u_1, \dots, u_n, q_1, \dots, q_n, t) = 0$ where $\bar{f}_n \in \mathbb{R}^m$, **fn**

Vector function of m nonholonomic constraint equations among the n generalized speeds and generalized coordinates.

A_r

Linear coefficient matrix for \bar{u}_r in the nonholonomic constraint equations.

A_s

Linear coefficient matrix for \bar{u}_s in the nonholonomic constraint equations.

\bar{b}_{rs}

Terms not linear in \bar{u}_s or \bar{u}_r in the nonholonomic constraint equations.

A_n

Linear coefficient matrix for \bar{u}_s in the equation for $\bar{u}_r = \mathbf{A}_n \bar{u}_s + \bar{b}_n$.

\bar{b}_n

Terms not linear in \bar{u}_s in the equation for $\bar{u}_r = \mathbf{A}_n \bar{u}_s + \bar{b}_n$.

25.6 Mass Distribution

$\bar{I}_a^{B/O}$, **I_B_O_a**

Inertia vector of rigid body B or set of particles B with respect to point O about the unit vector \hat{n}_a .

\breve{Q} , **Q**

Dyadics are indicated with a breve accent.

$\breve{I}^{B/O}$, **I_B_O**

Inertia dyadic of body B or set of particles B with respect to point O .

\breve{I}^{B/B_o} , **I_B_B_o**

Central inertia dyadic of body B or set of particles B with respect to mass center B_o .

${}^A\bar{H}^{B/O}$, **A_H_B_O**

Angular momentum of rigid body B with respect to point O in reference frame A .

25.7 Force, Moment, and Torque

\bar{R}^S , **R_S**

Resultant of the vector set S .

$\bar{R}^{S/Q}$, **R_S_Q**

Resultant of the vector set S bound to a line of action through point Q .

$\bar{M}^{S/P}$, **M_S_P**

Moment of the resultant of the vector set S about point P .

$\bar{T}^B, \mathbf{T}_{\mathbf{B}}$

Torque of couple acting on reference frame or body B .

25.8 Generalized Forces

${}^A\bar{v}_r^P, \mathbf{v}_{\mathbf{P}} \mathbf{r}$

r^{th} holonomic partial velocity of point P in reference frame A associated with the generalized speed u_r .

${}^A\tilde{\omega}_r^B, \mathbf{w}_{\mathbf{B}} \mathbf{r}$

r^{th} holonomic partial angular velocity of reference frame B in reference frame A associated with the generalized speed u_r .

${}^A\tilde{v}_r^P, \mathbf{v}_{\mathbf{P}} \mathbf{r}$

r^{th} nonholonomic partial velocity of point P in reference frame A associated with the generalized speed u_r .

${}^A\tilde{\omega}_r^B, \mathbf{w}_{\mathbf{B}} \mathbf{r}$

r^{th} nonholonomic partial angular velocity of reference frame B in reference frame A associated with the generalized speed u_r .

$F_r, \mathbf{F1}$

r^{th} holonomic generalized active force associated with the generalized speed u_r .

$\tilde{F}_r, \mathbf{F1}$

r^{th} nonholonomic generalized active force associated with the generalized speed u_r .

\bar{F}_r, \mathbf{Fr}

Column vector of all generalized active forces (holonomic or nonholonomic).

$F_r^*, \mathbf{F1s}$

r^{th} holonomic generalized inertia force associated with the generalized speed u_r .

$\tilde{F}_r^*, \mathbf{F1s}$

r^{th} nonholonomic generalized inertia force associated with the generalized speed u_r .

$\bar{F}_r^*, \mathbf{Fr}s$

Column vector of all generalized active forces (holonomic or nonholonomic).

25.9 Unconstrained Equations of Motion

$\bar{f}_k(\dot{\bar{q}}, \bar{u}, \bar{q}, t) = 0$

Kinematical differential equations.

\mathbf{M}_k

Linear coefficient matrix for $\dot{\bar{q}}$ in the kinematical differential equations.

\bar{g}_k

Terms not linear in $\dot{\bar{q}}$ in the kinematical differential equations.

$\bar{f}_d(\dot{\bar{u}}, \bar{u}, \bar{q}, t) = 0$

Dynamical differential equations.

\mathbf{M}_d

Linear coefficient matrix for $\dot{\bar{u}}$ in the dynamical differential equations, often called the “mass matrix”.

\bar{g}_d

Terms not linear in $\dot{\bar{u}}$ in the dynamical differential equations.

$$\bar{x} = [\bar{q} \quad \bar{u}]^T$$

State of a multibody system.

$$\mathbf{M}_m$$

Linear coefficient matrix for $\dot{\bar{x}}$ in the equations of motion.

$$\bar{g}_m$$

Terms not linear in $\dot{\bar{x}}$ in the equations of motion.

25.10 Equations of Motion with Nonholonomic Constraints

$$\bar{f}_n(\bar{u}_s, \bar{u}_r, \bar{q}, t) = 0$$

Nonholonomic constraint equations.

$$\mathbf{M}_n = \mathbf{A}_r$$

Linear coefficient matrix for \bar{u}_r in the nonholonomic constraint equations.

$$\bar{g}_n = \mathbf{A}_s \bar{u}_s + \bar{b}_{rs}$$

Terms not linear in \bar{u}_r in the nonholonomic constraint equations.

$$\dot{\bar{f}}_n(\dot{\bar{u}}_s, \dot{\bar{u}}_r, \bar{u}_s, \bar{u}_r, \bar{q}, t) = 0$$

Time derivative of the nonholonomic constraint equations.

$$\mathbf{M}_{nd}$$

Linear coefficient matrix for $\dot{\bar{u}}_r$ in the time differentiated nonholonomic constraint equations.

$$\bar{g}_{nd}$$

Terms not linear in $\dot{\bar{u}}_r$ in the time differentiated nonholonomic constraint equations.

25.11 Equations of Motion with Holonomic Constraints

$$\dot{\bar{f}}_h(\bar{u}, \bar{u}_r, \bar{q}, \bar{q}_r, t) = 0$$

Time derivative of the holonomic constraints.

$$\mathbf{M}_{hd}$$

Linear coefficient matrix for \bar{u}_r in the time differentiated holonomic constraints.

$$\bar{g}_{hd}$$

Terms not linear in \bar{u}_r in the time differentiated holonomic constraints.

25.12 Energy and Power

$$P, \mathbf{P}$$

Power

$$W, \mathbf{w}$$

Work

$$K, K_Q, K_B, \mathbf{K}, \mathbf{K}_Q, \mathbf{K}_B$$

Kinetic energy, kinetic energy of particle Q , kinetic energy of body B

$$V, \mathbf{v}$$

Potential energy

E, \mathbf{E}

Total energy, i.e. $E = K + V$

25.13 Lagrange's method

L, \mathbf{L}

Lagrangian the difference between the kinetic energy and the potential energy: $L = K - V$

a_r

Multiplicative term associated with generalized speed q_r in a constraint equation

λ

Lagrange multiplier, variable encoding the (scaled) magnitude of a constraint force

\bar{f}_{hn}

Combined time-derivatives of holonomic constraints and non-holonomic constraints

M_{hn}, \mathbf{M}_{hn}

Jacobian of constraint equations with respect to \dot{q}

\bar{p}, \mathbf{P}

Generalized momenta associated with the \bar{q} generalized coordinates

\bar{g}_d

Dynamic bias, the sum of terms not linear in \ddot{q} in the inertial forces and the generalized conservative forces considered in the Lagrangian.

CHAPTER
TWENTYSIX

REFERENCES

CHAPTER
TWENTYSEVEN

PRIOR VERSIONS

- Version 0.2: Final version after the 2023 course.
- Version 0.1: Final version after the 2022 course.

CHAPTER
TWENTYEIGHT

LECTURE VIDEOS

2023 Lecture Video Playlist

2022 Lecture Video Playlist

BIBLIOGRAPHY

- [Flores2023] Paulo Flores, Jorge Ambrósio, Hamid M. Lankarani, “Contact-impact events with friction in multibody dynamics: Back to basics”, *Mechanism and Machine Theory*, vol. 184, 2023. <https://doi.org/10.1016/j.mechmachtheory.2023.105305>
- [Hunt1975] K. H. Hunt, F. R. E. Crossley, “Coefficient of restitution interpreted as damping in vibroimpact”, *J. Appl. Mech.*, 42 (2) (1975), pp. 440-445.
- [Kane1985] Thomas R. Kane, and David A. Levinson. *Dynamics, Theory and Application*. McGraw Hill, 1985. <http://hdl.handle.net/1813/638>.
- [Meijaard2007] J. P. Meijaard, J. M. Papadopoulos, A. Ruina, and A. L. Schwab, “Linearized dynamics equations for the balance and steer of a bicycle: A benchmark and review,” *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 463, no. 2084, pp. 1955–1982, Aug. 2007.
- [Mitiguy1996] P. Mitiguy, “Motion variables Leading to Efficient Equations of Motion,” *The International Journal of Robotics Research*, vol. 15, no. 5, pp. 522–532, 1996.
- [Ostrowski1994] Jim Ostrowski, Andrew Lewis, Richard Murray, Joel Burdick *Nonholonomic Mechanics and Locomotion: The Snakeboard Example*. 1994
- [Vallery2020] Heike Vallery and Arend L. Schwab, “Advanced Dynamics”, 3rd edition, Delft University of Technology, 2020, ISBN/EAN 978-90-8309-060-3
- [Lanczos1970] Cornelius Lanczos, “The Variational Principles of Mechanics”, 4th edition, Dover Publications, 1970, ISBN/EAN 978-04-8665-067-8