PYTHON
for Classical Mechanics

PYTHON
for Classical Mechanics

Charlotte Christensen
Paul Tjossem

University Science Books
New York

University Science Books
An Imprint of AIP Publishing
uscibooks.aip.org

Publisher Jane Ellis
Production and Marketing Manager Barbara Dickson
Acquisitions Editor Katerina Heidhausen
Publishing Associate Felicity Henson
Manuscript Editor Kot Copyediting & Proofreading
Illustrator <TO COME>
Compositor Nova Techset
Cover Design <TO COME>
Printer & Binder <TO COME>

This book is printed on acid-free paper.

Casebound ISBN 978-1-940380-25-4

eBook ISBN 978-1-940380-26-1

Library of Congress Cataloging-in-Publication Data

Names: Christensen, Charlotte R., author. | Tjossem, Paul, author.
Title: Python for classical mechanics / Charlotte Christensen, Paul Tjossem.
Description: New York : University Science Books, [2024] | Summary: "Contemporary physics relies heavily on computer programming for analyzing data and modeling systems, yet time constraints often prevent undergraduate physics students from taking the computer science courses needed to develop these skills. This textbook integrates scientific programming instruction directly into a standard undergraduate classical mechanics physics course. Built to accompany John Taylor's popular Classical Mechanics, this text provides a series of interactive Python computational exercises that analyze classical mechanical systems from both analytical and numerical perspectives. The exercises guide students chapter-by-chapter through modeling classical physics systems such as the simple pendulum at high angle, two or more gravitational bodies in orbit, and damped, driven oscillators leading to period-doubling and chaos. The text uses guided instruction in critical programming techniques such as loops, logic, array manipulation, numerical integration, and data analysis and plotting to help intermediate physics students gain proficiency in both analytical and computational methods. It assumes no prior knowledge of programming on the part of the student and includes step-by-step instructions for starting the student programming in Python with the interactive Jupyter Notebook interface"-- Provided by publisher.
Identifiers: LCCN 2023045378 (print) | LCCN 2023045379 (ebook) | ISBN 9781940380254 (casebound) | ISBN 9781940380261 (ebook)
Subjects: LCSH: Mechanics--Textbooks. | Mechanics--Data processing--Textbooks. | Python (Computer program language)--Textbooks.
Classification: LCC QC125.2 .C488 2024 (print) | LCC QC125.2 (ebook) | DDC 531.0285/5133--dc23/eng/20231102
LC record available at https://lccn.loc.gov/2023045378
LC ebook record available at https://lccn.loc.gov/2023045379

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

Contents

Preface

Across the past several decades, computer programming has arisen as an essential tool in the study of physics. Within the study and application of physics, computer programming is used extensively to analyze data, control instruments, and model systems. Students are also acutely aware of the value of programming ability not only as a practical tool for their physics pursuits but also as an asset to their employability within physics and beyond.

The impetus for this work comes from the 2013 Grinnell Physics Department self-study, wherein physics students demanded increased programming instruction. Unfortunately, completing a standard undergraduate physics major leaves little time for additional computer science courses. Moreover, the content of most computer science courses is understandably focused on the interests of computer scientists. Consequently, students might spend semesters mastering topics like sorting algorithms, memory management, and data structures while missing out on essential physics-related programming tools, such as data plotting and numerical integration. While we fully encourage the interested student to delve into the former, we believe that all physics majors should be exposed to the latter. With this in mind, we have crafted this text to equip students with a fundamental tool kit of programming skills tailored to the realm of physics.

While the motivation of this text was to introduce students to the basic programming tools of computational physics, our second goal is no less important. This text serves as a vehicle to reinforce mechanics concepts and enhance students' physical intuition. It achieves this by guiding students through explorations of various physical systems, such as harmonic oscillators, gravitating bodies, and rotating objects. The computational analysis of these systems and the accompanying visualizations complement the standard analytical treatment by allowing students to "see" the motion described by the equations.

This text consists of a series of units that guide students through the exploration of various systems. It is designed to be interactive and to encourage inquiry-driven analysis. Programming instruction is interwoven throughout the text. Students are provided ample opportunities to hone their programming skills, with guidance gradually tapering off as they progress.

By its nature, this text straddles the line between programming and physics instruction. It provides little discussion of the theory of computer science or even algorithms beyond numerical integration (for that, we direct the student to a computer science course). Nor should it be considered a comprehensive Python reference (for that, we direct the student

to the extensive official online documentation of Python and its libraries). Similarly, it is not meant to be treated as a stand-alone mechanics physics text but rather to complement *Classical Mechanics*, by John Taylor. We hope, though, that it aptly fills its niche of providing interactive programming instruction within a physics context.

Acknowledgments

Development of these materials has been a communal endeavor across the past eight years at Grinnell College. We would like to warmly acknowledge the contributions of Professor Eliza Kempton, who co-led the first iteration of computational labs for Grinnell's classical mechanics course, including "Introduction to Python", "The Three-Body Problem", and "Orbits, Keplerian and Not". We are grateful to Sage Kaplan-Goland for providing extensive feedback on Python curricular development at Grinnell and writing the template for instruction on Fourier transforms. We also thank Rayn Samson for his feedback, especially on the oscillating wheel unit.

We are grateful to the Grinnell physics professors who led sections of mechanics lab with these notebooks: Professors Barbara Breen, Charles Cunningham, Keisuke Hasegawa, Shanshan Rodriguez, and James Zabel. Prof. Rodriguez helped streamline the exercises to make more extensive use of NumPy's array capabilities, while Prof. Zabel's suggestions improved the readability of the code. Prof. Charlie Duke has been a leader in incorporating Python programming into Grinnell's curriculum and helped us structure these materials. Students Evan Porter, Debanjali Pathak, and Zeineb Mezghanni gave detailed feedback on many of the exercises.

We appreciate all the students and teaching assistants who participated in early iterations of the labs. We extend our thanks to Prof. John Taylor, whose textbook *Classical Mechanics* clearly explains the physics concepts and inspired many of our exercises. Likewise, we are grateful to our editor, Jane Ellis, who encouraged us in this endeavor, and to the peer reviewers who provided such thoughtful feedback. And of course, we want to thank our spouses, Christopher Bowcutt and Paula Smith, whose love, encouragement, support, and advice was vital to this project and so much more.

Initial development of these exercises was supported by a grant to Grinnell College from the Howard Hughes Medical Institute through its Precollege and Undergraduate Science Education Program. NSF Grant AST-1848107 supported the creation of the appendices and more advanced exercises such as the modeling of coupled oscillators. Any opinions, findings, conclusions, or recommendations are ours and do not necessarily reflect the views of the National Science Foundation.

1 Introduction

1.1 Why Classical Mechanics? Why Python?

In designing a text that melds programming with physics, we had to decide both what physics content to focus on and what programming language to use.

Intermediate Classical Mechanics stood out as the natural place to incorporate programming into an undergraduate physics sequence. The content of Classical Mechanics is uniquely suited to computational modeling. A core concept underlying Classical Mechanics is the discovery and solving of the differential equations of motion that describe the behavior of a system. Programming a computer to numerically integrate the equations of motion is often the preferred or even only way to solve these equations. Consequently, Classical Mechanics provides an abundance of interesting computational problems. Additionally, the highly theoretical nature of the typical Classical Mechanics course makes computer simulations especially valuable for building a student's conceptual understanding. Such simulations aid students in bridging the mental gap between the equations describing a system and the system's observable behavior.

Moreover, students typically take their first Classical Mechanics course shortly after declaring their physics major. Placing programming instruction relatively early in their undergraduate physics education provides students with a skill base that can be used in future physics courses, research opportunities, and internships. Conversely, deferring programming instruction until after the first couple of introductory physics courses allows students to acclimate to physics coursework before adding additional programming challenges.

The selection of Python as the programming language was based on its broad utility to students and its approachable learning curve. Python is widely used across disciplines and at different proficiency levels. It is used by laboratory physicists controlling instruments, Google researchers developing natural language algorithms, data scientists developing network models, and many others. Therefore, basic knowledge of Python will position students well for a broad spectrum of career goals. In contrast, Mathematica, C++, and Fortran (all widely used in physics) provide a more specialized skill set.

The wide use of Python stems in part from its price (free!), in part from its intuitive syntax, and in part from the abundance of third-party packages that have been written for it. These packages include NumPy for manipulating arrays of data, SciPy for scientific computing, and Matplotlib for plotting and data visualization. The richness of these packages and their ongoing development mean that there are a large number of powerful tools available to the user. Finally, the use of Python allows for the Jupyter Notebook graphical, browser-based

interface. This notebook interface allows the user to compile and run code interactively and in small segments, making it easier to understand and debug. Together, these qualities of Python make it particularly friendly to the beginning programmer.

1.2 Who Is This Book For?

We wrote this text with the intermediate physics student with no or limited experience in programming in mind. The first two units, therefore, provide an overview of the basics of Python programming. More advanced instruction and opportunities to practice programming are progressively incorporated throughout the remainder of the text.

While the text primarily targets less experienced programmers, any Mechanics student should find the physics content engaging. In How to Use This Book, we suggest various entry points that will allow students with programming experience to bypass some of the introductory programming instruction. After the first couple of units, the physics concepts become the focus of the work and provide ample challenges in and of themselves. We also provide additional extension problems for those students who progress more swiftly or who seek greater challenges.

1.3 A Word of Encouragement to the Student

In our own Mechanics courses, we see students enter with a wide range of programming backgrounds. It is common for students who start with less programming experience to be self-conscious if they move more slowly through the material or desire more instructor support. From survey responses, we have learned that students who enter with less background find the material more frustrating but, ultimately, *find the most value in the experience*. Therefore, we wanted to conclude with a word of encouragement to the novice programmer.

Anyone can learn to program. Everyone has a picture in their mind of the stereotypical programmer. But it doesn't matter whether you know what a kilobyte is, whether you have built your own computer, or even whether you have seen all of the original *Star Trek* — if you can think through a sequence of actions you would like a computer to undertake, programming is within your grasp. If we could grant any qualities to the beginner programmer, therefore, it wouldn't be a particular knowledge base, but rather the curiosity to explore, the courage to try new things, and the resilience to try again when the first attempt (or five) fails.

Programming is a skill, and like any skill — be it riding a bike or baking a cake — it has to be learned by doing. The only way to improve is through practice and time. Frustration and error are inherent to the experience no matter how long you have been programming (which, as people who have been programming a long time, we can attest to). Encountering a challenge or making a mistake doesn't make you a bad programmer, it just means you have to spend more time debugging until you figure it out. Your reward is the unique satisfaction of having solved the puzzle.

2 How to Use This Book

We firmly believe that the most effective way to learn programming is through hands-on practice. Moreover, inquiry-based computational exploration of physical systems builds conceptual understanding akin to laboratory experiences. *Therefore, this text takes the form of interactive units structured around different physical systems.* Each unit takes the form of a series of unified exercises centered upon a particular physical system. Instruction in programming techniques and physical concepts is given when relevant, allowing the new ideas to immediately be put to use. The units are scaffolded such that techniques in earlier units are repeated in later ones and programming instruction is gradually withdrawn. While we believe this is the best pedagogical approach, it requires additional direction as to how the text may be incorporated into different physics classrooms, each with its own aims and course structure.

2.1 Course Structure

Rather than functioning as a stand-alone physics resource, this text is intended to be used in conjunction with an intermediate classical mechanics textbook. We specifically designed it to accompany John Taylor's *Classical Mechanics*: it follows the same content order, provides additional exploration of some of the systems that appear in examples and problems, and occasionally refers directly to specific figures and equations. The flowchart at the end of this section indicates how the units in this text pair with the chapters in *Classical Mechanics*, and the "Chapter 3, Description of Units" elaborates on the connection between the two.

In our experience, students require frequent, immediate assistance in both the programming and physics concepts when completing these units. They also benefit from the opportunity to discuss ideas and collaboratively problem solve. Therefore, while students with prior programming experience might tackle these units as independent homework assignments, we generally recommend that they be assigned during scheduled class time. At Grinnell College, they are completed during a weekly, three-hour Computational lab. A Teaching Assistant-led section would provide a similar experience. We also recommend that students work with a partner, i.e., "paired programming," a popular and effective software development technique. Partner work not only offers additional support but also encourages students to articulate their thought processes.

The addition of a new component to a course, such as a computational lab, generally entails the removal or reduction of a different component. In practice, at Grinnell we have successfully replaced a portion of the analytical problem solving with these computational

units. Although it is natural to balk at reducing the amount of time devoted to analytical problem solving in a Classical Mechanics class, our assessments indicate that this replacement *does not decrease students' ability to solve physics problems,* while *increasing their programming ability.* We believe that this lack of trade-off is because the units remain centered on the physics concepts and thus reinforce analytical analysis rather than diluting it.

2.2 Scheduling

For each of the chapters in the "Essentials" part in *Classical Mechanics*, this text provides one to two accompanying units. It concludes with three units exploring the Damped Driven Pendulum system that is the focus of Chapter 12, "Nonlinear Mechanics and Chaos", from *Classical Mechanics*. Although Chapter 12 is listed as one of the "Further Topics" chapters in *Classical Mechanics*, it is too well suited to numerical analysis for us to pass it by. Conversely, despite the importance of the Hamiltonian formalism to the study of classical mechanics, we do not include a computational unit on it. Not only is Chapter 13, "Hamiltonian Mechanics", not included in the "Essentials" part of *Classical Mechanics*, the concepts are almost entirely analytical and did not strongly benefit from additional numerical examination.

When we offer this course at Grinnell College, students complete twelve of the units during a semester course, one each week. While the assigned units roughly parallel the lecture content of the course, the content is sometimes offset by about a week in either direction. For example, students are assigned The Three-Body Problem while they are still completing Chapter 7, "Lagrange's Equations" from *Classical Mechanics* in preparation for Chapter 8, "Two-Body Central Force Problems." Similarly, the second and third units on Chaos (Bifurcation Diagram and Poincaré Diagram) are completed while they are covering Chapter 13, "Hamiltonian Mechanics" in the lecture. We find that this level of temporal offset is small enough to maintain the connection between the lecture content and the computational work while allowing for more or less computational analysis, as warranted.

We have designed these units to be modular to reflect the different scheduling needs of individual courses. While the scaffolding across units requires them to be completed roughly in order, not every unit needs to be assigned. Indeed, these units intentionally provide more content than could easily be covered in a semester-long course to allow the instructor to pick and choose among them. The flowchart indicates which units depend on which and provides varying points of entry, depending on student programming background. For instance, Python-savvy students might skip the initial one or two units — or even the initial four — if an in-depth exploration of numerical integration algorithms is not desired. Whenever a smaller programming concept, such as numerical root finding, is introduced, we provide a basic description in case the unit in which it previously appeared was omitted. For more complex programming techniques, such as numerical integration using the solve_ivp() solver from SciPy, we offer appendices as reference.

As students progress through the units, they will encounter more complex programming techniques with less guidance and example code. The flowchart specifies which core programming concepts are used in which units. Furthermore, each unit's introduction lists programming learning objectives (designated "PRO"), in addition to physics-focused learning objectives (designated "PHY"). Finally, the Index may be used to find where different programming techniques and Python tools are introduced.

Upon completing these units, we intend that all of the most important tools and techniques for the physics student will have been introduced. (Of course, not everything could be included, and some advanced topics such as data analysis via the Pandas library or symbolic mathematical analysis using SymPy are not included here.) Inherent in the structure of guided exercises, though, is that students will have had little practice in setting up their own programs from scratch. The "Challenge Exercises" address some of this gap by providing more open-ended prompts with minimal template code. We have also had success in assigning final projects. Following these units, students in our classes have modeled and analyzed such systems as the Duffing oscillator, a Hohmann transfer orbit, and the motion of a projectile subject to drag forces, the Coriolis force, and the centrifugal force. Such final projects encourage students to generate their own questions, make connections across concepts, and plan the structure of their program.

2.3 Working through the Units

Each unit consists of ten to twenty analytical and computational exercises unified around a particular physics concept or system (with the exception of the first unit, Introduction to Python, which acquaints students with Python basics and has no unifying physics theme). These exercises should be tackled in sequence, although occasionally we note an analytical exercise that may be assigned ahead of time. The units take on average about three hours to complete. However, there is some variation across units and a large variation among students. To address these differences and offer flexibility, we have distinguished between essential exercises (all those listed in black) and extension exercises (those listed in gray and indicated by an asterisk). Any extension exercise may be omitted for time without compromising the main ideas of the unit or precluding the completion of the later essential exercises. Additionally, almost all units include suggestions for "Challenge Problems". These problems include much less guidance and are intended to stretch the abilities of the more experienced or faster programmer.

Instruction for these exercises is provided in multiple different ways. Text descriptions (given within the "markdown cells" in the downloadable notebooks) explain the general concepts and provide the exercise prompts. Additionally, there is a large amount of example code, which demonstrates how different programming tools and techniques operate, and template code, which should be modified to complete the exercises. This code is interwoven with the text and appears as runnable code cells within the downloadable notebooks. Comments within the example code describe the function of different lines of code. They also provide directions to the student. Places where the student is asked to modify the code are generally indicated by three hashmarks: "###". Text, code, and comments should all be read carefully as each provides its own form of instruction.

The exercises are meant to provide direction in how to explore a physical system or experiment with a new programming technique. They are only a starting point, though, and are *not* meant to be an exhaustive list of possibilities. The more one takes the time to explore, the more profound the learning. One shouldn't hesitate to dive in, play around, and experiment.

2.4 Relationship between Units

A flowchart describing the relationship between the units is begun in Figure 2.1 and continued in Figure 2.2 and Figure 2.3.

Physics concepts are designated in green and programming techniques in blue. The corresponding chapters from *Classical Mechanics*, when applicable, are shown in black at the bottom of each block. The light blue flowlines show the suggested order of approach, as the programming topics in the source block are built on by the connected block. When the arrow extends from the side of the block, it indicates that the specific corresponding programming technique is used in both connected units. The first time the programming technique is introduced, it is listed in bold. Light green flowlines indicate a suggested order of approach for units that explore similar physics concepts. Even absent the light blue and green flowlines, the units should be completed approximately in vertical order, reading from the top down. The dark gray flowlines point to units that cannot be undertaken without completing the unit in the source block, as the exercises in the latter build on code developed in the former.

Three possible entry points are suggested by the mauve ovals. We anticipate that most students will start at the first unit, "Introduction to Python". However, students with a stronger background in Python may start at "Numerical Integration Applied to Projectile Motion" or later. Before proceeding to the units shown in Figure 2.2 and Figure 2.3, all students should either complete "Launching a Rocket" or have read through "Appendix: Using solve_ivp() for Numerical Integration," as this technique for solving differential equations numerically is critical to all subsequent units other than "Principal Axes of a Cuboid". Appendices, which may be used to fill in gaps in understanding caused by missed units or simply for a more in-depth treatment of a topic, are shown in solid gray blocks. They may be referenced at any point, although they will be most useful for those units whose flowlines pass beneath the block.

While the light blue and green arrows may imply dependency, this is not strictly the case. We anticipate that most classes will only work through a subset of these units. Therefore, we generally provide some guidance on the more advanced programming tools and techniques each time they appear. As such, one should feel encouraged to work through a unit without completing a preceding unit. In fact, one might even choose to work through a set of less-connected units in order to maximize the variety of programming tools and techniques encountered. Light green flowlines similarly indicate a conceptual connection and an increase in complexity but *not* a dependency.

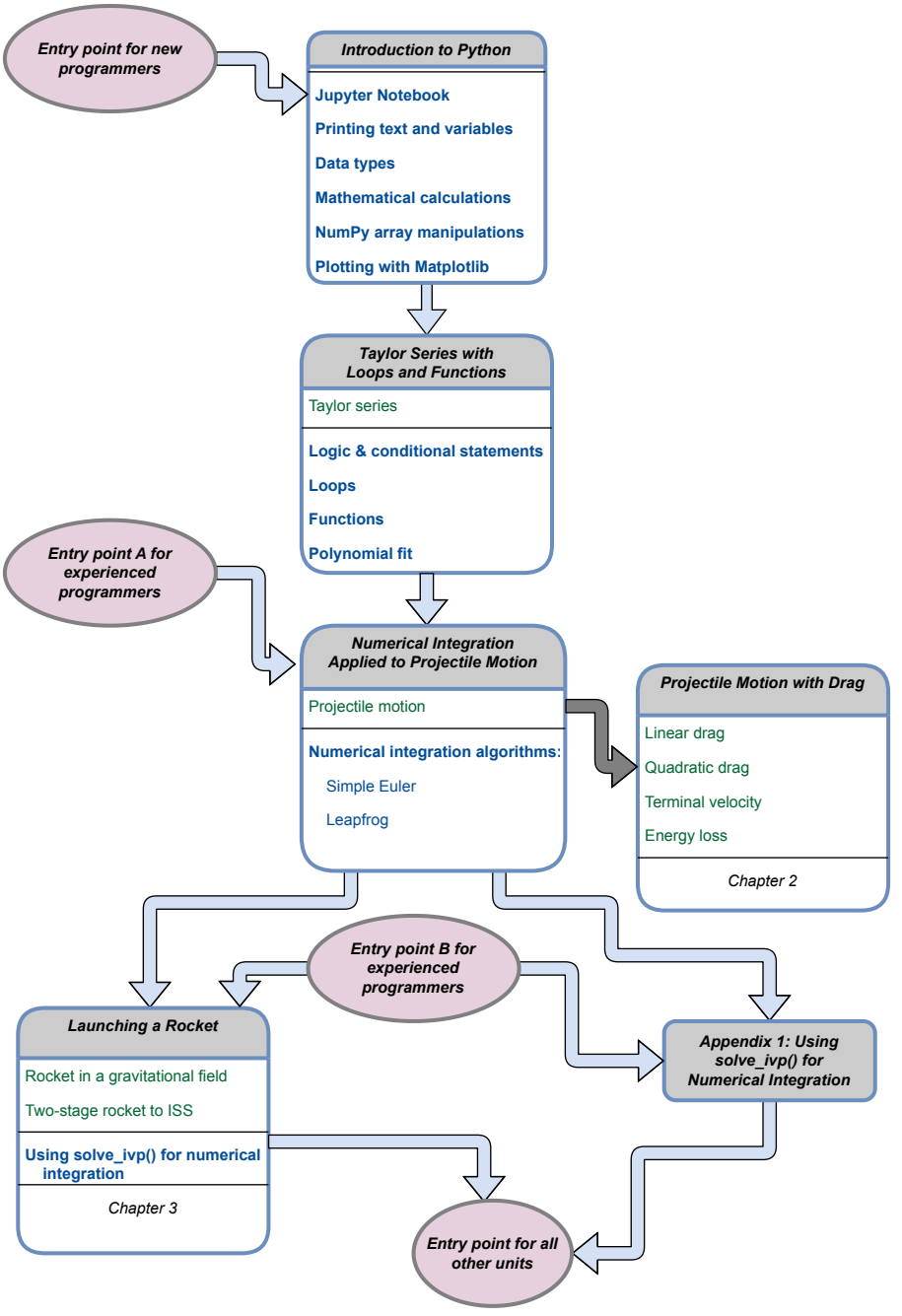


Figure 2.1 Flowchart illustrating the order of the units and the main topics covered by them.

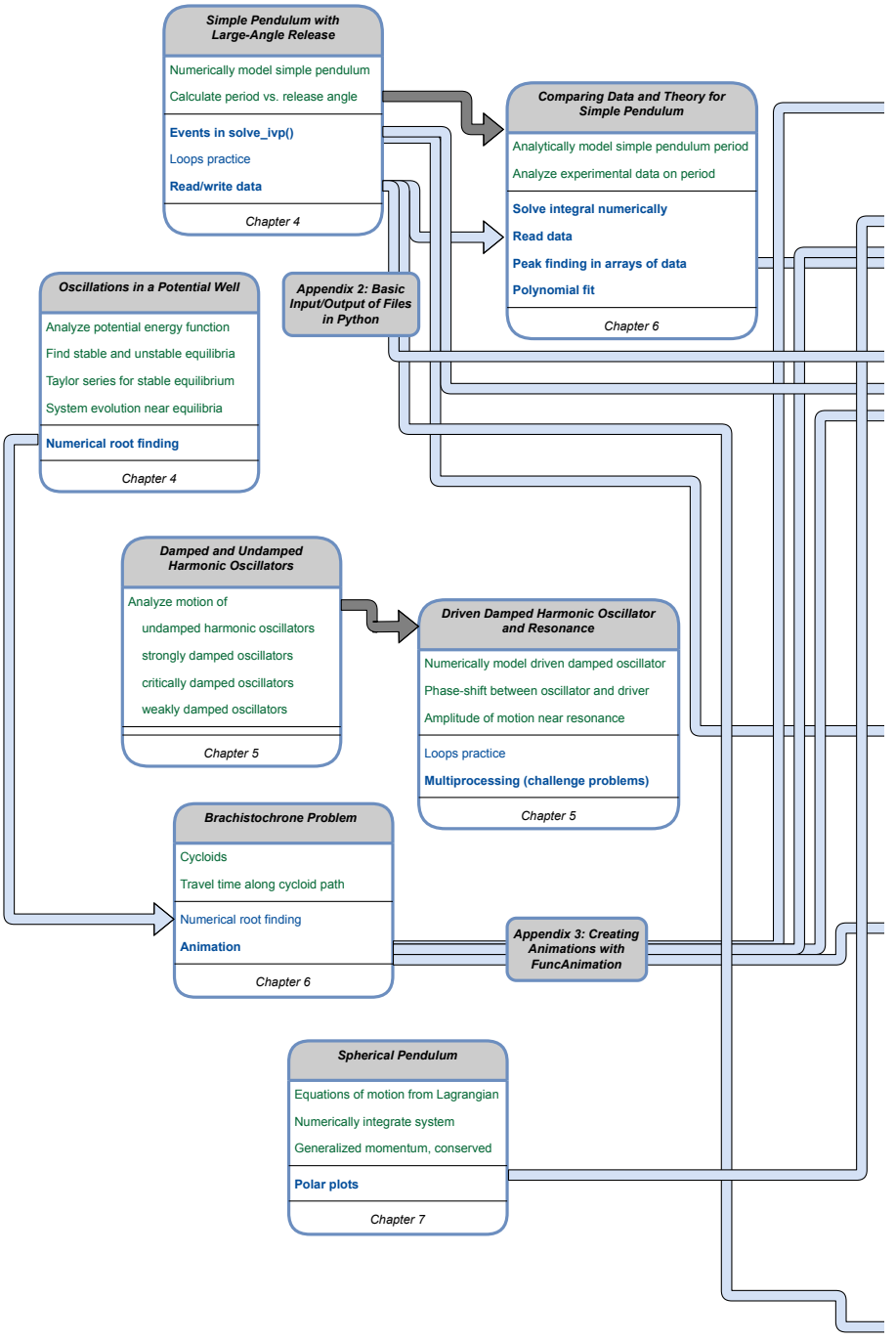


Figure 2.2 Flowchart illustrating the order of the units and the main topics covered by them.

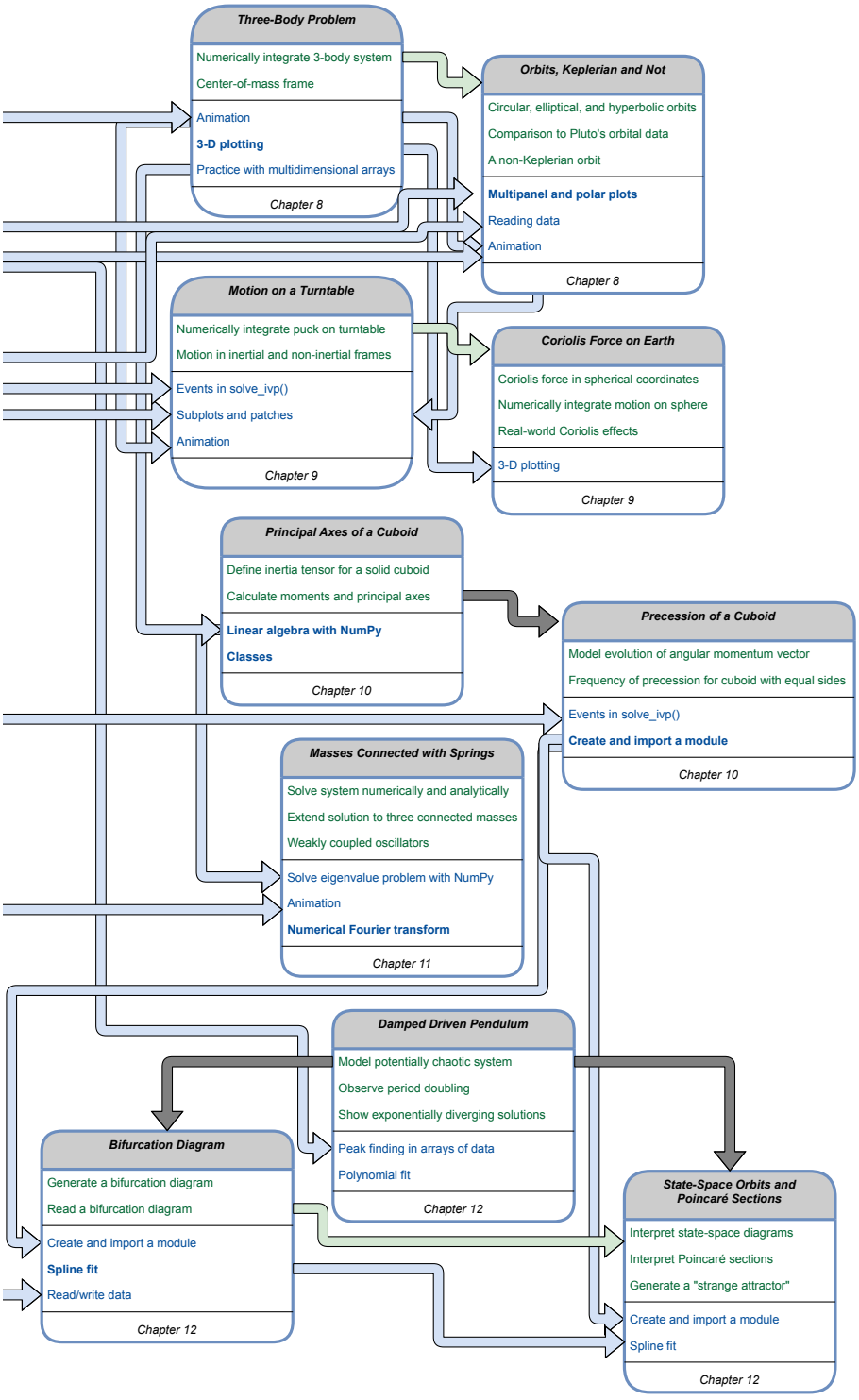


Figure 2.3 Flowchart illustrating the order of the units and the main topics covered by them.

3 Description of Units

Preparing to Use Jupyter Notebook with Python

This section provides a "quick-start" guide to accessing and using the Jupyter Notebook interface with Python.

Introduction to Python

This unit introduces the reader to Python and Jupyter Notebook. It includes information on programming basics such as data types, mathematical calculations, manipulation of NumPy arrays, and plotting using the Matplotlib library.

The unit is best completed at the start of the course, likely in conjunction with Ch. 1 of *Classical Mechanics*. The introduction of arrays as a data type is well-aligned with the discussion of mathematical vectors in Ch. 1.2.

Files: u01_introductionToPython.ipynb, u01_Ex8.png

Taylor Series with Loops and Functions

This unit continues the introduction to Python programming basics by providing instructions in logical operators, loops, and functions. Students apply these techniques to plot Taylor series approximations to different functions at increasing levels of accuracy. They also learn how to fit a curve to data.

Along with the first unit, this unit lays the groundwork for future programming. It also builds student understanding and appreciation for Taylor series, an approximation that is critical for both future analytical and computational techniques in physics.

While *Classical Mechanics* assumes that students have already learned Taylor series in a different context, such as a math class, the concept is used throughout the textbook, starting early on (e.g., Equation 2.40 and later Problem 2.18). The graphical illustration of Taylor series included in "Taylor Series with Loops and Functions," therefore, will prepare the students for later content in the textbook. Note that **Exercise 9** is similar to Problem 2.18 from *Classical Mechanics*, and the latter could be assigned as homework in preparation.

Files: u02_taylorSeries.ipynb

Numerical Integration Applied to Projectile Motion

Students modify code to write their own computational solver for ordinary differential equations (ODEs). Students first explore a first-order Taylor series solver and then increase the

accuracy by modifying the code to use the "improved Euler" or "leapfrog" method. This solver is applied to 2-D projectile motion.

Students gain their first experience writing an algorithm in this unit, and it ensures that students are familiar with the techniques and limitations of numerical ordinary differential equation (ODE) solvers before they use the built-in Python/SciPy ODE solvers.

Although students will likely already be familiar with the physics concepts in this unit (specifically, projectile motion without air resistance), this unit serves as a precursor to the following unit, Projectile Motion with Drag, and is therefore best completed in conjunction with or before Ch. 2 of *Classical Mechanics*.

Files: u03_numericalIntegrationProjectileMotion.ipynb

Projectile Motion with Drag

Students use the numerical integration algorithm they developed in Numerical Integration Applied to Projectile Motion to determine how the trajectory of a projectile is affected by air resistance. Students first model motion subject to linear drag and quadratic drag separately before combining both. Students compare their results to different analytical calculations, including to the terminal velocity resulting from quadratic drag. When determining the 2-D projectile motion with quadratic drag, students also gain greater familiarity with the position unit vector, $\hat{r}$.

This unit is aligned with the content in Ch. 2.1, 2.2, and 2.4 from *Classical Mechanics*. Students numerically model the linear and quadratic air resistance introduced in these chapters and create plots analogous to Figures 2.7 and 2.8. References are also made to the situation described in Problem 2.1, allowing for synergy with a homework assignment.

This unit also guides students through using a Riemann sum integration to find the work done by the drag force. This content is covered in greater depth in Ch. 4 of *Classical Mechanics*, but students should have prior knowledge of it from an earlier physics class. If desired, though, this work calculation could be either removed or delayed.

Files: u04_projectileMotionWithDrag.ipynb

Launching a Rocket

This unit synthesizes material on conservation of linear and angular momentum, using the case study of the Space Shuttle launch. Students model the initial acceleration of the rocket to find the velocity as a function of mass and time in the cases of zero gravity and constant gravity. For the final problem, students model a two-stage rocket launch of the Space Shuttle to the International Space Station.

This unit introduces students to the SciPy ODE solver, `solve_ivp()`, as they are asked to determine the motion of the rocket computationally and analytically and to compare the solutions.

This unit covers the concepts discussed in Ch. 3.1–3.4 of *Classical Mechanics*. The analytical work references Problems 3.11 and 3.13, while the parameters for the Space Shuttle are defined as in Problem 3.7.

Files: u05_launchingRocket.ipynb

Simple Pendulum with Large-Angle Release

Students undertake an in-depth exploration of this simple, yet rich, classical system. They numerically integrate to find the motion of a pendulum released from large angles and compare the resulting period to the one found using the small-angle approximation. They also examine the flow of energy through the system, providing a connection to Ch. 4 from *Classical Mechanics*.

In order to analyze the period as a function of release angle, students are asked to write a loop and to use events in `solve_ivp()` to identify zero-crossings. This unit also introduces reading and writing data to a file.

Files: u06_simplePendulum.ipynb

Comparing Data and Theory for Simple Pendulum

Students compare the results of their numerical analysis from the previous unit to theoretical analysis and experimental data. They show that all three physics techniques are able to reveal the same relationship between pendulum period and release angle.

In completing their analytical analysis, students encounter an elliptic integral of the first kind and tackle it using two different tools from the SciPy library. The associated problem from *Classical Mechanics* is Problem 4.38, which could be completed beforehand.

Students read in and process a set of experimental data. The data set comprises the times when the pendulum bob blocked and unblocked a laser beam aimed at the nadir of its path. Students are asked to manipulate the data array to determine the period of the pendulum, its speed, and its release angle.

Conservation of energy, as covered in Ch. 4, provides a framework for both the analytical analysis and the analysis of the experimental data.

Files: u07_simplePendulumComparison.ipynb, simplePendulum.csv, u07_fig1.png, u07_fig2.png

Oscillations in a Potential Well

Students create potential energy diagrams to explore the behavior of an oscillating wheel with attached masses. Students then relate these energy diagrams to the force and determine the Taylor series approximation for the force felt near the stable equilibrium. Students computationally model the trajectory when the system is released near stable and unstable equilibria.

This unit contrasts the numerical solution for this non-linear differential equation to the analytical solution in the first-order approximation. From this comparison, students gain a greater understanding of the power and limitations of the commonly used analytical technique of taking first-order approximations.

This unit is based on the system from Problem 4.37 and covers material from Ch. 4 of *Classical Mechanics*. This is a longer unit, which may benefit from being shortened or divided across two sessions.

Files: u08_oscillationsInPotentialWell.ipynb

Damped and Undamped Harmonic Oscillators

Students simulate harmonic oscillators with and without damping. This unit reinforces student understanding of the analytical solution by guiding them through an exploration of the effects of under-, over-, and critical damping.

This unit covers material from Ch. 5.1, 5.2, and 5.4 of *Classical Mechanics*, including the generation of plots similar to Figures 5.11 and 5.12.

Files: u09_dampedAndUndampedHO.ipynb

Driven Damped Harmonic Oscillator and Resonance

Students build upon their simulation of simple harmonic oscillators to include a driving term. This unit reinforces the idea of transient and long-term behavior by comparing the trajectories resulting from different initial conditions. It also explores the changes to phase and amplitude as the driving term approaches resonance. Because some of the calculations take more than a few seconds to compute, the Challenge Problems provide an introduction to multiprocessing.

This unit covers material from Ch. 5.5 and 5.6 of *Classical Mechanics*, including the creation of a resonance curve, similar to that in Figure 5.17.

Files: u10_drivenDampedHOResonance.ipynb, resonanceForMultiProcessingSolvers.py

Brachistochrone Problem

This unit explores the brachistochrone problem, the most famous example of using the calculus of variations. Students are asked to simulate the amount of time it would take an object to fall according to its own gravity along a straight-line and cycloidal path. They also numerically explore the isochrone property of a cycloid (the analytical equivalent is Problem 6.25 from *Classical Mechanics*). Students gain their first experience creating animations to model the motion of a mass along a cycloid path.

This unit uses the same physical situation as in Example 6.2 and provides motivation for the calculus of variations.

Files: u11_brachistochrone.ipynb

Spherical Pendulum

Students explore the spherical pendulum initially by using Lagrangian mechanics to determine the equations of motion and then by describing the trajectory using both an analytical examination of the equations of motion and calculating the numerical solution. Students look at special cases, such as planar and conical motion of the pendulum. They also show analytically and numerically that the generalized momentum in ϕ is conserved.

As part of their examination of the numerical solution, students create a projection of the trajectory onto the horizontal plane using a polar plot.

This unit is based on Problem 7.40 from *Classical Mechanics* and reinforces concepts of generalized coordinates covered in Ch. 7.

Files: u12_sphericalPendulum.ipynb

The Three-Body Problem

Students simulate the motion of three gravitationally interacting masses, which is a non-linear and potentially chaotic system. The students are first asked to simulate two stable situations (two planets orbiting a star; one distant planet orbiting a binary star system) before exploring chaotic systems.

In this unit, students are introduced to a commonly used numerical technique of including gravitational softening, and they become familiar with the center-of-mass frame of reference. Students are also introduced to more advanced NumPy array manipulation, including using two-dimensional arrays. They also create their first three-dimensional plot.

This unit can be completed as a lead-in to the orbital discussion of Ch. 8 of *Classical Mechanics*. In particular, the calculations of circular orbits and the introduction of the center-of-mass frame of reference provide a useful grounding in the material. However, it may also be completed in conjunction with Ch. 4.7 and 4.8, due to its discussion of central forces and the energy of multiple particle systems.

Files: u13_threeBody.ipynb

Orbits, Keplerian and Not

Students numerically model circular, elliptical, and hyperbolic Keplerian orbits, followed by an orbit produced by a non-$1/r$ potential that cannot be analytically solved. Students approximate Pluto's orbit as a two-body system and compare their results to more accurate data obtained from the NASA Jet Propulsion Laboratory.

This unit reinforces the ideas of center-of-mass frame and reduced mass. Students also learn how to create polar plots and practice animations and reading data from a file.

The unit covers material from Ch. 8.1–8.7 and includes the prompt from Problem 8.25 from *Classical Mechanics*.

Files: u14_orbits.ipynb, pluto_ephemeris.csv

Motion on a Turntable

Students model the motion of a puck on a turntable in inertial and non-inertial reference frames. This unit builds conceptual understanding of the centrifugal and Coriolis forces and how they shape the trajectory in the non-inertial reference frame. By comparing visualizations of the inertial and non-inertial reference frames, students practice mentally shifting between reference frames.

This unit covers material from Ch. 9.1–9.7 of *Classical Mechanics*, using the situation and some of the prompts from Problems 9.19, 9.20, 9.21, and 9.24.

Files: u15_turntable.ipynb

Coriolis Force on Earth

Students model the trajectory of an object constrained to the surface of a sphere subject to the Coriolis force. Students first determine the appropriate equations of motion in spherical

coordinates before completing the numerical integration. They check their code and build physical intuition by first computing trajectories for special cases and then exploring isolated changes to the initial conditions. Finally, students are guided through the modeling of real-world situations.

This unit provides students with extensive practice with spherical coordinates, in both their analytical calculations and their visualizations. It also provides instruction in three-dimensional plotting.

This unit continues the exploration of material from Ch. 9.1–9.7, but extrapolates the ideas from a plane to the surface of a sphere.

Files: u16_coriolis.ipynb

Principal Axes of a Cuboid

In this unit, students determine the principal axes of different rectangular cuboid bodies using different axes by computationally calculating the eigenvalues and vectors. They then use three-dimensional plotting to visualize the principal axes.

This unit introduces students to object-oriented programming and walks them through creating their own classes and objects. It also introduces students to the NumPy's linear algebra capabilities, and it builds student understanding of the NumPy array structure.

This unit covers material from Ch. 10.1–10.8 of *Classical Mechanics*.

Files: u17_principalAxesCuboid.ipynb

Precession of a Cuboid

This unit builds on the previous unit by having students use the calculated principal moments in Euler's equations to compute the angular momentum over time. It leads students to examine the stability of motion of a cuboid with three unequal sides spun around different axes. Students also explore the conical motion of a body with two equal principal moments and calculate the frequency of oscillation analytically and numerically.

This unit guides students in writing and importing their own module.

Files: u18_precessionCuboid.ipynb, principal_axes.py

Masses Connected with Springs

Students analyze the oscillations of two equal-mass carts and three equal springs, attached between two rigid walls. They first solve and plot the analytical solutions following the procedure detailed in Ch. 11 of *Classical Mechanics*, then repeat the problem numerically both as an eigenvalue problem and as a direct integration of the equations of motion. They explore the weak-coupling regime wherein the center spring constant is much smaller than those of the two outer springs. They computationally calculate the eigenvalues, as in "Principal Axes of a Cuboid," to determine the normal modes, and visualize the combined motion of normal modes resulting from different initial conditions. A framework is developed for extending the analysis to any number of masses and springs. It also includes an (optional) introduction to fast Fourier transforms.

This unit covers material from Ch. 11.1–11.3 of *Classical Mechanics* and uses prompts from Problem 11.7.

Files: u19_massesConnectedSprings.ipynb

Damped Driven Pendulum

Students model the trajectories of a damped driven pendulum in order to explore the approach to chaos. By changing the driving force and the initial conditions of the system, they gain familiarity with chaotic and non-chaotic regimes, including an examination of period doubling. This lab also sets the stage for further investigations into chaotic behavior.

This unit covers material from Ch. 12.1–12.5 of *Classical Mechanics*, including asking students to reproduce many of the plots in these chapters and following the prompts of Exercises 12.6–12.10.

Files: u20_dampedDrivenPendulum.ipynb

Bifurcation Diagram

Students use their simulation of the Damped Driven Pendulum to construct the bifurcation diagram of Figure 12.17 of *Classical Mechanics*. To do this, they create and import a module containing functions developed in Damped Driven Pendulum. This unit builds familiarity with this important visualization and reinforces their understanding of period doubling and the approach to chaos. It also provides instruction for a class-wide group project in which teams of students solve subsets of the bifurcation diagram and share their data to create a composite diagram.

This unit explores material from Ch. 12.6 of *Classical Mechanics*.

Files: u21_bifurcation.ipynb

State-Space Orbits and Poincaré Sections

Students construct Poincaré Plots for the Damped Driven Pendulum system in the chaotic (e.g., Figure 12.29 of *Classical Mechanics*) and non-chaotic regime. By working through the construction of a Poincaré Plot, students become more fluent with state space. They observe state space mixing within this classic visualization. Once again, students use functions created in Damped Driven Pendulum and stored in a module.

Students recreate many of the state space diagrams in Ch. 12.7 of *Classical Mechanics*. Concepts covered by this unit are introduced in Ch. 12.7–12.8.

Files: u22_stateSpacePoincare.ipynb

Appendix 1: Using solve_ivp() for Numerical Integration

A description and example of how to use the numerical solver, `solve_ivp()`, to solve differential equations. This technique is first referenced in Launching a Rocket and is used in all subsequent units.

Files: a1_solvingODEs.ipynb, a1_fig1.png

Appendix 2: Basic Input/Output of Files in Python

A summary of several different methods for reading and writing documents with Python.
Files: a2_basicInputOutput.ipynb

Appendix 3: Creating Animations with FuncAnimation

A guide to how to make animations using the FuncAnimation method from Matplotlib.
Files: a3_creatingAnimations.ipynb

Preparing to Use Jupyter Notebook with Python

4

Your first task before beginning these exercises is to obtain access to the Python programming language, several standard libraries, and an interactive computing platform. Specifically, you will need the following:

- Python 3. These exercises have been developed with Python 3.9, but they should be largely backward compatible to version 3.4. They may also be run with Python 4, with small modifications.
- The latest versions of the NumPY, SciPY, Matplotlib, and iPython libraries associated with Python 3.
- A platform to run the interactive Jupyter notebooks.

There are two primary methods for accessing all of the above. The first is to install them on a personal computer. The second is to use a web server, such as CoCalc or Google Colab. Both methods are explained below.

4.1 Installation on a Personal Computer

Your first option is to install your own version of Python on a personal computer and run Jupyter Notebook using that version. This approach offers the advantage of offline work, potentially faster code execution, and somewhat more control over Python and library versions. If you use this method, we recommend you start by downloading the interactive Jupyter notebooks (files ending in .ipynb) and the supporting files (files ending in .py, .csv, and .png) associated with this text to your computer from XXXX. Alternatively, you may instead choose to transcribe the contents of this text to Jupyter notebooks that you create.

4.1.1 Python installation

If you are installing Python for the first time, our recommendation is to use the Anaconda distribution of Python. With the continuous development of Python and, in particular, its libraries means it can be challenging to ensure compatibility among them. Anaconda eases this process enormously. Because the packages needed for these exercises are widely used, the default programming environment created for you by Anaconda during installation should include everything you need. However, if you eventually decide to use Python more extensively, Anaconda allows you to easily install other packages and to create and manage environments with distinct sets of packages.

You may download and install Python 3.x for free through Anaconda by using the link and following the instructions on the website here: `https://www.anaconda.com/products/distribution`.

4.1.2 Running Jupyter Notebook

After installing Anaconda Python, use it to start Jupyter Notebook. On a Microsoft PC, you will find Jupyter Notebook within the "Anaconda3" folder accessible through the Start Menu. On a Macintosh computer, locate "Anaconda-Navigator" using the spotlight search or some other method. Once you have started the Anaconda-Navigator, select "Jupyter Notebook" (or JupyterLab, if you prefer) from the menu.

When you start Jupyter Notebook, a terminal window running Python will open. Do not close this window; you can minimize or move it aside. You should be able to ignore it for the remainder of your session. After the terminal window opens, a Jupyter notebook will automatically open as a tab in your default web browser window.

In the Jupyter Notebook tab, you will see a list of folders in your current directory. At this point, you will want to open the desired interactive unit (for example, `tableOfContents.ipynb`). The easiest way to access all the units is to download all the files into a single folder. Then you can navigate through the directory tree to locate that folder by selecting the appropriate folders listed in the Jupyter Notebook tab. Once in that folder, you can open individual units by selecting them. We recommend starting by opening `tableOfContents.ipynb` and then using the links within that document to open individual units.

Alternatively, you may click on the "Upload" button located toward the upper right of the screen or drag and drop files to upload individual units into the current working directory. The uploaded file should then appear in the list, and you will be able to open it by selecting it.

Should you prefer to create your own Jupyter notebooks and write code into those files rather than downloading the interactive exercises, you may do so. To create a new Jupyter notebook, select the New pull-down menu located in the upper right of the screen. Then select "Python 3" to create a new notebook using that version of Python code.

4.1.3 Closing Jupyter Notebook

When you believe you have completed a unit, it is good practice to restart the kernel (found under the Kernel menu) and rerun the entire notebook in order to verify the code's functionality.

Before closing Jupyter Notebook, remember to save the notebook you're working on. You can do this by selecting the floppy disk "Save" icon or opting for "Save" within the Jupyter "File" menu within the browser tab. Following this, close the tab containing the notebook. Next, navigate to the browser tab running Jupyter Notebook and showing the directory contents. Select "Quit" in the upper-right corner of the screen to close the program. You may then close this browser tab and the terminal window that opened upon starting Jupyter Notebook.

4.2 Using a Web Server to Run Jupyter Notebook

If you prefer to not install Python onto your computer, you can run Jupyter Notebook through your browser using a web server, for example Google Colab https://colab.research.google.com/ or CoCalc https://cocalc.com/. This approach eliminates the need for software installation and offers the advantage of cloud storage accessibility from any computer via the web. It can also facilitate collaboration among different users. To make use of this feature, add users to your project.

One of the primary limitations to using a web server is that the platforms differ in appearance from both Anaconda3 and each other. This is particularly the case with Google Colab, which uses a simplified version of Markdown for text cells called "Text," such that some of the formatting of the notebooks may appear different. Google Colab also does not support internal links between notebooks, making file navigation more difficult. Since these platforms are under continuous development, their appearance may change over time. With these caveats in mind, we have attempted to provide general instructions. However, please note that some adjustments may be necessary when running the notebooks on these platforms.

Begin by creating an account on the web server and logging in. Once logged in, we recommend you upload all the files, both notebooks (ending in .ipynb) and supporting files (ending in .py, .csv, and or .png). Instructions for doing this using both CoCalc and Google Colab are below. Alternatively, though, you can manually transcribe the text contents into new Jupyter notebooks.

4.2.1 CoCalc

Select "Your Projects" from the navigation bar near the top of your screen. Create a new project. Upload files to your project by scrolling to the bottom of the screen and dragging files into it from your file browser. You can also create new notebooks.

Then select "Explorer" on the left-hand toolbar. From this window, start with `tableOfContents.ipynb` and use the embedded links to navigate to the units you desire.

4.2.2 Google Colab

Upon opening Google Colab, a window may automatically open giving you the option to open various notebooks. If so, select the "Upload" tab and either use the "Choose File" option or drag and drop from your file navigator to upload a notebook. Once a notebook is open, you may load an additional one by selecting "Upload notebook" from the File menu within Google Colab. Supporting files, such as those ending in .py, .csv and .png, must be added to your "Colab Notebooks" folder on Google Drive.

Unfortunately, at the time of writing, Google Colab lacks an easy method for uploading multiple files at once. Additionally, internal hyperlinks between notebooks, such as in tableOfContents.ipynb, also fail once the notebooks have been uploaded to Colab. Given these constraints, the most straightforward approach is to upload and open the Jupyter Notebook for the desired unit using its filename. Filenames of notebooks and associated supporting files are listed in the table of contents. You may also always see the filename linked to from a notebook by double clicking on the cell.

Sometimes you will wish to access supporting files, such as images, data, or modules, from within your notebooks. To do this, you will need to make sure that file is within your "Colab Notebooks" folder on Google Drive. Then you must "mount" your Google Drive folder by adding the following commands to a cell in your notebook and running it.

5 Introduction to Python

Welcome to the computational component of your Mechanics course! This first unit is designed to get you started programming in Python. Through it you will learn how to edit and run Jupyter Notebook (an interface for interactively coding in Python). You will also learn basic Python data types and how to perform calculations, print, and plot your results.

If you are familiar with Python, you may wish to skip some or all of this unit. For example, you may wish to skip to "Lists and Arrays" if you are familiar with Python syntax but would like an introduction to NumPy. Similarly, "Plotting" provides an introduction to plotting using the Matplotlib library and can be completed independently of the rest of the unit. Finally, if you are already well-versed in Python, Jupyter Notebook, NumPy, and Matplotlib, and are interested only in the physics-related content, you can move directly to the next unit.

The basic tools introduced in this unit will set you on the path to using programming in your physics studies. There is no better way to learn to program than to start. Therefore, rather than reading much introductory text, you will immediately start executing commands and working from examples.

5.1 Objectives

In this unit, you will:

- learn how to open, edit, run, and save in Jupyter Notebook (PRO);
- become familiar with different Python data types and how to print variables and use them in calculations (PRO);
- manipulate Python lists and NumPy arrays (PRO);
- plot data (PRO).

5.2 Working with Jupyter Notebook

These exercises are written using the Jupyter Notebook interface, which provides a web-based interactive computer platform for Python programming. "Interactive" means that the programmer (you) will be writing parts of the code while it is already active. This is a handy format for developing code and learning to program as it provides immediate feedback and ready access to different variables. We provide a description of it here.

While we have written these instructions for a local installation of Jupyter Notebook, these interactive units should run easily using JupyterLab and web servers such as CoCalc and Google Colab. Be aware, though, that these have slightly different interfaces and may differ somewhat in appearance and functionality from what is described here.

The menu bar near the top of the notebook includes a File menu for creating new or opening existing notebooks, saving, renaming, and printing. The Edit menu allows you to copy notebook cells, move them up or down, split at the cursor, or merge two cells (of the same type) together. All operations work on the currently highlighted cell, indicated with a colored outline on the screen. The Insert menu allows you to add a blank cell above or below the highlighted cell. The Cell menu permits you to run different cells, select the cell type (for example Code or Markdown), or clear all the output cells. The Kernel menu controls the starting and stopping of the Python calculation engine. Go to the Help menu for all questions about the various features and functions of Python and its additional libraries. Many of these menu functions are duplicated in the pictorial toolbar below it, and all of them have keyboard shortcuts (as found in the Help menu).

5.2.1 Cell types

Jupyter notebooks consist of sets of cells that can be run by the user. The cells are either Markdown (or Text) cells or Code cells. The type of a given cell is listed in the pulldown menu in the toolbar above (this one is labeled Markdown). You can change the type of cell by selecting a different type from the pulldown menu.

Markdown cells

Markdown cells (a.k.a. text cells) generally contain text, for instance, instructions or explanations. Markdown refers to a markup language with syntax designed for writing formatted text. It is easily converted to HTML and other formats. Markdown syntax includes headers of various sizes, lists, paragraphs, color, ***text emphasis***, formatted math LaTeX using MathJax, etc. You can also use HTML tags. For more detailed information see: https://www.markdownguide.org/cheat-sheet/.

As an example of how Markdown syntax can be used to format text, here's your favorite equation using MathJax math LaTeX:

$$\sum \mathbf{F} = m\ddot{\mathbf{x}}$$

Try this:

- To see the syntax of this Markdown cell, click on this cell with your cursor and push `Return` (or `Enter` on a PC).
- Then run the cell (type `Shift-Return`) to show the marked-up cell. Alternatively, click the triangle icon ▶ Run in the toolbar above.

➤ *Note: If you're interested in more Markdown details, the above link is an excellent place to start.*

Code cells

Python code and Jupyter code extensions go into code cells, which you can add using the Insert menu. The entire cell can be executed by typing `Shift-Return` (or `Shift-Enter`). In the next section, you will run some example code.

➤ *Note: To get help on other such notebook commands you may be able to press* `Esc-H`*. The* `Esc` *puts you into command mode; the* `h` *calls for a help page. This command currently works in Jupyter Notebook and CoCalc. It does not currently work in Google Colab or when using JupyterLab.*

It helps to have line numbers in each cell so that you can refer easily to specific lines. If they aren't already on by default, hit `Esc-Shift-L`, or go into the View menu and click "Toggle Line Numbers" in your code cells.

Try this:

- Insert a cell below this one and type something in it.
- Change the cell type from Code to Markdown.
- See if you can move the cell up and down.
- Explore splitting a cell, merging cells, and deleting a cell.

5.2.2 The kernel

The kernel is the process that actually executes (a.k.a. "runs") the code in the code cells of your Jupyter notebook. Knowing how to interrupt, shut down, and restart kernel is, therefore, important for being able to execute your notebooks. (In Google Colab, the kernel is referred to as "runtime," but the principle is the same.)

The kernel is programming language specific, so while this notebook should be run in Python 3, other notebooks could be run with other versions of Python, provided you had those versions available to you (for instance, installed on your computer). The language the kernel is running in is listed in the upper right of your screen.

Occasionally, you may want to halt the execution of a code cell, for example, if it is taking too long to run, or if you discover a bug in it you want to fix. You can do this by selecting "Interrupt Kernel" from the Kernel menu or clicking the Stop button on the toolbar.

The kernel stores the information gathered by running the notebook in its memory. This information includes such things as the value of variables and the definition of functions. If you shut down or restart the kernel, this information will be wiped from the memory. While clearing the memory may sound scary, it is often very useful. For example, coding problems can easily occur if the computer has an incorrect or outdated value stored in a variable. A basic debugging practice is to use the Kernel menu and select "Restart" to restart the kernel. Then you can use the command under the Cell menu to execute all the cells in order.

The kernel may also sometimes shut down on its own, usually because it has run out of memory. If this happens, select "Restart" under the Kernel menu and use the commands under the Cell menu to execute the cells up until the point where the problem occurred.

5.3 Introduction to Python Programming

5.3.1 Libraries

One of the benefits of Python as a programming language is that many people have written extensive code to perform basic (and not so basic) tasks. This code is stored in Python libraries. For example, the NumPy (Numerical Python) library includes a large number of

mathematical functions and support for useful datatypes, such as "arrays". The Matplotlib library allows for the creation of data visualizations, in particular, plots and graphs.

In order to have access to the code stored in these Python libraries, they need to be imported using the `import` command shown below. Once the following cell has been executed, you will have access to all the functionality of the NumPy and Matplotlib.pyplot libraries. Note that in the latter case, the ".pyplot" selects a subset of the Matplotlib library most useful for generating plots.

```
# The '#' key is the comment character. Any line that starts with it, such as
#    this one, will not be executed.
#    (You can comment out a set of lines by highlighting, then pressing
#    Ctrl+'/'.  Repeat to uncomment.)

# The following are import statements, which we always put at the
#    beginning of a notebook. These bring in packages (also called
#    libraries) which add functionality to Python
# numpy (or NumPy), short for Numerical Python, covers a huge number of
#    mathematical functions and datatypes
# matplotlib.pyplot is part of a library that makes it easy to make
#    nice-looking plots and graphs

import numpy as np  # "as np" means that numpy commands will be prefaced by
                    #    "np", for example, np.array(a).
import matplotlib.pyplot as plt  # "as plt" means that matplotlib.pyplot
                                 #    commands will be prefaced
                                 #    by "plt", for example, plt.show()
```

5.3.2 Variables

In programming, *variables* are used to store information. These variables can be recalled or manipulated by the program to accomplish the desired goal.

Changing variables

Work through the following cells of code, executing each one by one:

```
# This cell defines a variable, aObj, sets it equal to 234, and prints a
#    statement about it

aObj = 234
print("The variable aObj is equal to ", aObj)
```

```
# Now that aObj has been set to a value, other cells in the notebook will also
#    know its value.

print("The variable aObj is still equal to ", aObj)
```

```
# If we change the value of aObj in this cell and then rerun the previous cell
#    it will update to the newest value.

### DO THIS: Try it for yourself by executing this cell and re-executing the
#    previous one.

aObj = 1
```

If you want to clear all your variables, you can run the magic command `%reset` in a code cell. Or you can clear your current session (including any imported packages) by selecting "Restart & Clear Output" in the Kernel menu. Try this now, and then execute the following code cell. Notice that the number next to the code cell is now 1, indicating that it is the first cell that has been executed. But you'll also encounter an error because `aObj` is undefined. Once you have convinced yourself that the variable was cleared, rerun the notebook to this point (to pick up all imports and variable assignments). You can do this easily by selecting "Run All Above" in the Cell menu.

```
print("The variable aObj is equal to ", aObj)
```

As you can see, the value of a variable can change depending on the order in which the previous cells were executed. *This is one of the tricky aspects of programming in an interactive environment.* Watch out for how your program changes when cells are run out of order, as failing to keep track of how different cells have modified a variable can easily lead to unintended behavior.

The final version of your code should run correctly when all the cells are executed in order. To check this, when you finish a unit, clear your output by selecting from the pulldown menu at the top "Cell"->"All Output"->"Clear". Then run the entire notebook by selecting "Cell"->"Run All" from the pulldown menu.

If you would like to print out your completed notebook, you may find that some web browsers handle margins and extended output windows better than others.

5.3.3 Using Python for calculations

Python can easily be used for computations. The cell below gives three examples. Be sure you understand the syntax used.

```
a = 20.0
b = 55.2
print(a*b, a/b, a**2)
```

Exercise 1

In the next cell, calculate and print the following for $c = 4$ and a and b as defined above. You'll need to look up the syntax of the square root function (Hint: It's part of NumPy, so

should be referred to as `np.sqrt()`). And note that Python uses ** as the exponentiating, or power, operator rather than ^.

1. $\frac{a+b}{c}$
2. $\sqrt{c}$
3. $a^c b^c$

Your answers should be 18.8, 2.0, and approximately 1.5 trillion.

```
# < Exercise 1 >
```

5.3.4 Order of operations, and left-right associativity

Expressions in Python are evaluated in PEMDAS order:

```
Parentheses
Exponentiation
Multiplication & Division
Addition & Subtraction
```

For expressions containing multiplication and division, the expression is evaluated in left-right order. This can have some unusual behavior. When in doubt, use parentheses to clarify the order of operations.

```
a = 1
b = 2
c = 3
print("a, b, c = ", a, b, c)
print("a+b/c = ", a+b/c)
print("(a+b)/c = ", (a+b)/c)
print("a/b/c = ", a/b/c)
print("(a/b)/c = ", (a/b)/c)
print("a/(b/c) = ", a/(b/c))
print("a/b*c = ", a/b*c)
print("(a/b)*c = ", (a/b)*c)
print("a/(b*c) = ", a/(b*c))
```

5.3.5 Data types

Numbers in Python can be either integers, floating point (i.e., numerical approximations of real numbers), or complex numbers. `type()` is a built-in Python function that will return the data type of an object.

```
a = 20
print("a = ", a, "  and a's datatype is ", type(a))
```

```
b = 55.2
print("b = ", b, "  and b's datatype is ", type(b))
c = 5+100j
print("c = ", c, "  and c's datatype is ", type(c))
```

(Note that Python follows the engineering convention that $\sqrt{-1} \equiv j$.)

```
### DO THIS: Try modifying the following line of code to add and multiply
#    different numerical data types.
#    Make sure you understand the output

print("a+b = ", a+b, "  and a+b's datatype is ", type(a+b))
```

Data may also be stored as a *string* (i.e., a set of characters, such as a word or sentence). Either single (e.g., 'word') or double (e.g., "word") quotation marks can be used in Python to designate a string; *triple* quotation marks can create a multiline string. Generally, double quotation marks are preferred for the purposes of these exercises, although you may encounter cases where single quotation marks are needed for clarity, such as when the string itself contains double quotation marks, as shown in the example below:

```
string = "This is a string"
print(string)

string = '"If I have seen further it is by standing on the sholders [sic] of
            Giants." -- Newton'
print(string)
```

```
### DO THIS: Modify the following line of code to set d equal to
#    "Classical Mechanics"

d = "0"
print("d = ", d, ", and d's datatype is ", type(d))
```

5.3.6 Formatting output

Sometimes you may want to format numerical output (for example, the a/b output from the previous section), so it does not print so many decimal places. The cell below shows you how, by placing the string object within single or double quotation marks and then invoking the `format()` method.

```
# Examples of formatting

# 6 places, with 3 numbers after the decimal point.
#    The "f" signifies that the number is a float (a number with a decimal
#    point).
```

```
aOverbFormatedString = "(a/b) = {:6.3f}".format(a/b)
print(aOverbFormatedString)

# 6 places of an integer with leading spaces.  (The d signifies an integer.)
aFormattedString = "a = {:6d}".format(a)

# \t is the same as hitting the tab key. \n would give a new line
newString = aOverbFormatedString + "\t\t" + aFormattedString

### DO THIS: Modify this cell to print each of the variables above to see the
#    results.
#    Hint: Try hitting tab when you are typing the name of variables,
#    especially those with long names.
```

While in the example above the strings were defined in a separate line prior to printing, these commands can be combined into one line:

```
print("(a/b) = {:6.3f}".format(a/b))
```

The `format` method also allows you to include multiple variables in your formatted output by using multiple sets of curly brackets. By default, the first curly bracket refers to the first variable, the second to the second, and so on. If you would like to print the variables in a different order, a number can be included inside the curly brackets indicating which variable to use, with "0" indicating the first element.

```
# Examples of formatting multiple variables

print("{}/{} = {:6.3f}".format(a, b, a/b))  # format "a/b" as a float with
                                            #    three numbers after the
                                            #    decimal but use the default
                                            #    formatting for a and b

# This format statement refers to the variables out of order but produces the
#    same result as above.
print("{0}/{2} = {1:6.3f}".format(a, a/b, b))
```

Exercise 2*

In the cell below, print out the first 10 digits of π.

➤ *Hint: NumPy has a stored version of π. See if you can find how to call it. The web might be especially useful here.*

```
# < Exercise 2, first 10 digits of pi >
```

5.4 Lists and Arrays

Frequently it is useful to store sets of data. Python allows you to do this using "lists" while NumPy provides the added functionality of "arrays". Lists allow you to store multiple types of data together. Arrays store only one type of numerical data, but they have the advantage that you can perform vector (array) operations on them, as illustrated below. Calculations on arrays are usually much faster than corresponding calculations on lists because the Python interpreter does not need to check the datatype for each element.

Arrays are defined in the NumPy package, the fundamental package for scientific computing in Python. This package has many built-in functions that you will find useful. You can get a list of all NumPy built-in functions from `np.<tab>`.

```
# A list of objects can contain variables of any datatype

# Here is how you define a list:
List1 = [1, 2, 3, 4, 5, 6]  # Note the square brackets surrounding the elements
                            #    of the list, separated by commas
print("Here's a list of integers: ", List1)
print("This list is of type: {}, and length {}".format(type(List1),len(List1)))

List2 = [a, "b", "cdefg", "h", "i", 3, 5, 7]
print("This list is of mixed datatype, consisting of integer variables, "
      + "strings, and integers: ", List2)
print("It is of type: {}, and length {}".format(type(List2), len(List2)))
```

```
nArrayConverted = np.array(List1, float)  #  use np.array() to convert a list
                                          #    of identical objects into an
                                          #    array
print("Here's the same integer list turned into a floating point array: ",
      nArrayConverted)

print("We get an error if we try to convert List2 into floating point numbers")

### DO THIS: Try uncommenting the following line. Then recomment it out to
#    avoid the error.
# nArrayConverted = np.array(List2, float)
```

When defining an array, the easiest way to do this is to send a list (elements surrounded by square brackets and separated by commas) to the `np.array()` function, for example:

```
array1 = np.array([3.3, 4, 8.2, 1.0])
```

or

```
array2 = np.array(List1)
```

In the above code cell, an optional second parameter "float" was also included in the `np.array()` function call:

```
nArrayConverted = np.array(List1, float)
```

This optional parameter tells Python that all the data stored in the array should be considered a `float`. If this parameter is not included, Python will make its best guess as to the appropriate data type.

Below is an example to illustrate another important distinction between lists and arrays. See if you can tell the difference.

```
# Multiplication of an array by a constant gives completely different results
#    from multiplying a list by a constant

a = [1, 2, 3] # a list
print("a = ", a)
print("a is a list, indicated by the comma separators and the [] enclosure.")
print("First, let us multiply a by 5: ")
print("\t5*a = ", 5*a)

b = np.array([1, 2, 3]) # an array
print("\nHere is an array b: ", b)
print("An array has no commas, but retains the [] enclosure.")
print("We can multiply b by 5: ")
print("\t5*b = ", 5*b)

print("\nWe can perform other mathematical operations on arrays.  "
      + "For example, we can multiply array b by itself: \n\tb*b = ", b*b)

print("But we cannot multiply a list by itself: ")
### DO THIS: Uncomment the following line and rerun this cell to see the error
#    message
# print(a*a)
```

Similarly, you can add a constant to an array but not to a list, as demonstrated below:

```
# Addition of a number to an array (but not a list)

print(5 + b)

### DO THIS: Try uncommenting out the next line to see. Then recomment it out
#    to avoid the error.
# print(5 + a)
```

Likewise, you can multiply two arrays together, provided the arrays are the same length. Notice that this results in element-wise multiplication, neither a dot product nor a cross product. NumPy has separate routines for those manipulations that you can read about at np.dot and np.cross.

```
# Array multiplication

print('b = ', b)
```

```
c = np.array([4, 5, 6])
print('c = ', c)
print('b*c = ', b*c)
```

Exercise 3

In the cell below define two arrays: `even = (2,4,6,8)` and `odd = (1,3,5,7)`. Add 3 to the even array and multiply the odd array by 2. Then multiply the two arrays together and print your answer. See if you can do everything except defining the arrays in one line of code.

```
# < Exercise 3, Defining and manipulating arrays >
```

The added mathematical functionality of arrays means that they will generally be more useful than lists for computation. You should default to using arrays over lists in these exercises.

5.4.1 Making arrays

You have seen that we can convert a list of numbers into an array using `np.array()`. We can avoid that intermediate step and declare arrays to make evenly spaced numbers. There are two main ways to generate arrays in Python: `np.linspace()` and `np.arange()`. An example of each is shown below.

Try to understand the purpose of each argument in these two calls. The main difference is that `np.arange()` excludes the right-hand endpoint, whereas `np.linspace()` includes the right-hand endpoint.

```
array1 = np.arange(0, 3, 0.1)
print("array generated with 'np.arange()': ", array1)

array2 = np.linspace(0, 3, 30)
print("array generated with 'np.linspace()': ", array2)
```

Exercise 4

Generate a regularly spaced array with values ranging from 1 to 5 (inclusive) and spacing between array values of 0.1. Do this with both `np.arange()` and with `np.linspace()` and print out your results in the cell below.

➤ *Coding style tip: Rather than doing any necessary calculations elsewhere (say, on a piece of scratch paper), include those calculations in the cell block. This way you will have a record of what you did, and it will be easy to reproduce the calculations for different values, for instance an array that goes from 1 to 6.*

```
# < Exercise 4, Generating arrays with linspace and arange >
```

5.4.2 Indexing and slicing

Data are stored in lists and arrays by *index*, with the index count starting at zero. One tool that makes lists and arrays especially powerful is the ability to select different elements from them. The block of code below will show you several examples of how to do so, including how to identify non-zero elements using the `.nonzero()` method, find the maximum value using the `.max()` method, find all elements greater than some value, and count how many times a value occurs.

```
### DO THIS: Read this code first and try to predict what the output will look
#    like below.
#    Write the predictions on a piece of paper,
#    then execute this cell and scroll down to see if you were correct.

arrayA = np.array([1, -6, 5, 0, 0, 9, 5, -4, 1, 2, 2, 5, 20, 3])

print("We begin with an array of integers: ", arrayA)
print("The element at index 3 is:  ", arrayA[3])
print("The first element is: ", arrayA[0])
print("The last element is: ", arrayA[-1])
print("The first 4 elements are: ", arrayA[0:4])
# The fifth element from the end to the end
print("The last 5 elements are: ", arrayA[-5:])
 # argmax() method gives the index, not the value
print("The index of the largest element in the array is: ", arrayA.argmax())
print("The value of the largest element in the array is: ",
      arrayA[arrayA.argmax()])
print("\tOr more simply, ", arrayA.max(), "using the .max() method directly.")

print("\nWe can find an array containing the indices of arrayA that are not"
      + " zero, using the .nonzero() method:")
print(arrayA.nonzero())
print("But note that this result is itself a tuple, so we extract the"
      + " first element via index [0]:")
print("arrayA.nonzero()[0]: ",arrayA.nonzero()[0])
notzeroindices = arrayA.nonzero()[0]
print("\nThese are the indices of arrayA that are not 0: ", notzeroindices)
print("These are the non-zero values in arrayA: ", arrayA[notzeroindices])
print("These are the indices of arrayA that have zeros: ",
      (arrayA == 0).nonzero()[0]) #  (==is a logical test of equality)
print("  which will (of course) have values ",
      arrayA[(arrayA == 0).nonzero()[0]])

print("\nnp.arange() can take 3 values: the start value, the end value,"
      + " and the step size:")
print("Here is a better way to make an array of even values, using"
      + " np.arange(0,10,2): \n", np.arange(0,10,2))
print("Or odd values, using np.arange(1,11,2): \n", np.arange(1,11,2))
print("If you want to find the indices of all elements greater than 4,"
      +" simply do this:")
```

```
# Below is a logical test.  Use >, <, ==, >=, <=, != (not equal)
print("arrayA > 4", arrayA > 4)
print("  and so here they are: ", arrayA[arrayA > 4])
print("Here are the elements that are not equal to 5: ", arrayA[arrayA != 5])
print("And you can count the number of 5s like so, using the .sum() method")
print("(arrayA==5).sum(): ", (arrayA == 5).sum())

print("\nHere is an illustration of slicing, selecting only certain elements"
      + " in an array.")
print("Slicing uses three optional index values representing the start index,"
      + " the end index,")
print("and the increment of the index, separated by colons, like so:")
print("\tarr[start_index:end_index:increment]")
arr = np.arange(0, 20)
print("We illustrate with a new array arr as the first 20 non-negative"
      + " integers: \n\tarr = ", arr)
print("And now we slice out every third element from the array arr"
      + " using arr[::3] \n", arr[::3])
print("Or we write out arr in reverse order using arr[::-1] \n", arr[::-1])
print("Or slice out every other element of arr in reverse order"
      + " using arr[::-2]\n", arr[::-2])
print("\nFor completeness, note that arr[:] or arr[::] is the entire array:\n")
print(arr[:])
print(arr[::])
```

Exercise 5*

Write code to find and print the largest element in the first half of arrayA. Do not hard-code in any numbers; your code should work just as well if you were to change arrayA to a different length.

➤ *Hint:* The function `len()` will return the length of an array, and `round()` will round a float to an integer, suitable for indexing the elements within the array (which you cannot do with a floating-point number).

```
# < Exercise 5, Practice slicing arrays >
arrayA = np.array([1, -6, 5, 0, 0, 9, 5, -4, 1, 2, 2, 5, 20, 3])
```

Arrays can also have more than one dimension. When a multidimensional array is defined, as below, indices can be specified to select individual elements or entire columns and rows. When referencing an element, the row index is listed first and the column index is listed second.

```
# Referencing elements of multidimensional arrays

arrayB = np.array([[0,0],[2,3],[4,5]])
print("Define a new 2-dimensional array as arrayB:\n", arrayB)
print("The zeroth element of row 1 in arrayB: ", arrayB[1, 0])
```

```
print("Slice the zeroth row of arrayB: ", arrayB[0, : ])

print("Slice row 1 of arrayB: ", arrayB[1, : ])
print("Slice the zeroth column of arrayB: ",arrayB[:, 0])
print("Pick out the first two rows (and all the columns) of arrayB: \n",
      arrayB[0:2, : ])

#  And finally, arrays can be reshaped from one-dimensional to
#    multidimensional, as long as the number of elements stays the same:
arrayC = np.arange(0, 12, 1)
print("\nDefine a new 1-dimensional array as arrayC:\n", arrayC)
print("Reshape arrayC to have 3 rows and 4 columns:\n", arrayC.reshape(3, 4))
print("Reshape arrayC to have 6 rows and 2 columns:\n", arrayC.reshape(6, 2))
```

Exercise 6

Write code to print out the following for arrayB:

1) the element in column 1 and row 2;
2) the entire column 1; and
3) the entire row 2.

```
# < Exercise 6, Slicing 2-D arrays >
```

5.4.3 An aside on additional data types

If you progress further into Python programming you may encounter `tuples`, `sets`, and `dictionaries`. Like `lists`, these are built-in (meaning you don't need to load an additional library like NumPy to use them) data types that can store collections of data items. A `tuple` is very similar to a `list` but has the additional restriction that once it is defined it cannot be changed without redefining it. (For instance individual items cannot be changed nor can items be appended to it.) Tuples are defined using parentheses instead of square brackets:

```
ex_tuple = (0, 1, 2, 3)
```

A `set` is like a `tuple` but the items within it have no set order, and it cannot contain duplicate items. They are written with curly brackets. In a `dictionary`, each item consists of both a "value" and a "key," and specific values are referenced by their key rather than their location within the dictionary. Dictionaries are also defined with curly brackets, and colons are used between the values and the keys. For example, the code below would print 9 AM.

```
course = {"name": "Classical Mechanics", "Department": "physics", \
"capacity": 32, "time": "9 AM"}
print(course["time"])
```

You will not be required to use any of these data types in these units. In particular, lists can generally be used in place of tuples. When explaining what a function does, though, we will occasionally use this terminology because functions often return tuples.

5.5 Plotting

One of the benefits of using Python is that it makes it very easy to create attractive graphics. In particular, we will be using it to plot data in these units. In learning how to generate a plot, the examples in the `matplotlib` references (in particular http://matplotlib.org/) are good places to start.

There are several different ways to make plots in Python, but we will focus on the `matplotlib` object-oriented method, as it provides a nice balance between ease of use, readability, and adjustability.

Object programming is a particular approach in which programs are based on the use of "objects" — programming elements that can contain both code and data. Python is especially well suited to this approach, and many of the Python libraries you will use rely on objects. For example, the NumPy arrays that were introduced earlier are objects. These arrays contain data (e.g., the numbers stored in them) but also procedures (e.g., code to transpose the array, sum its terms, or compute its average value). You will learn more about object-oriented programming later when you write your own *classes*.

The core concept of object-oriented plotting methods is that you create Python objects, and then you can add new objects inside of those or change elements of those objects. The final plot you see is a `Figure` object. This is the top-level container for all other objects. The elements in this are called `Artists`. Think of the figure as a canvas on which the artists draw. Below the figure object are `Axes`, which hold the actual data for display. (You can have multiple `Axes` in each figure.) Let's look at a simple example in which we make a basic plot of `sin(x)`.

Exercise 7

Fix the code below to generate example data to be plotted. Your x data should be an array of 1,000 elements that runs from 0 to 4π. The y data should be the result of $y = \sin(x)$.

```
# < Exercise 7, Generating data for a plot >

n_points = 1  ### FIX THIS number of points
x = np.linspace(0, n_points)  ### FIX THIS Create array from 0 to 4pi with
                              #    n_points total points
y = x  ### FIX THIS to create sine data



print("Beginning and ending elements of x: ", x[0:5], ". . .", x[-1])
print("Beginning and ending elements of y: ", y[0:5], ". . .", y[-1])
```

Now that we have some data to work with, let's make the simplest possible plot with it. The code below does the following:

1. We use `plt.subplots()` to create a `Figure` and an `Axes` object simultaneously.
2. We use `ax.plot()` to create a `Line` object on our `Axes`. A `Line` is a type of `Artist`.
3. We call `plt.show()` in order to generate the final figure.

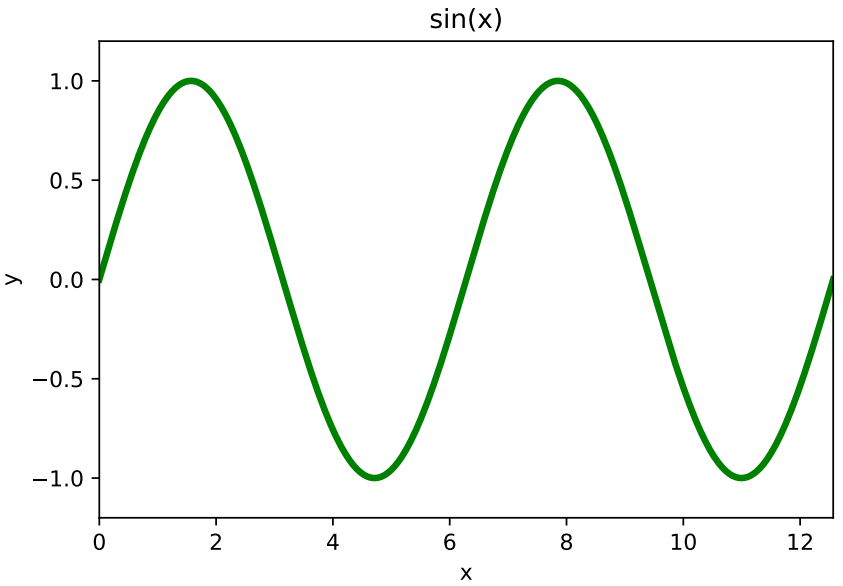
```
fig, ax = plt.subplots()  # Create Figure and Axes
ax.plot(x, y)  # Add plot
plt.show()  # Show the plot
```

This is OK, but not great for a number of reasons. The minimum and maximum values chosen on the *x* and *y* axes are not ideal, and there are no labels to denote what you are plotting. We can improve it by, for example, setting the *x*-axis limits using `ax.set_xlim(min, max)`, which sets the limits, rather than letting Matplotlib try to choose them automatically.

```
fig, ax = plt.subplots()  # Create Figure and Axes
ax.plot(x, y)  # Add plot
ax.set_xlim(0, 4*np.pi)  # Define x limits
plt.show()  # Show the plot
```

Exercise 8*

Try to make a nicer version of the same plot that looks like the one pasted in here:



(You can also find a copy of this image stored as intro_Ex_8.png.)

➤ *Hint:* You can type `ax.[tab]` to list all of the commands you can put after it. Once you have typed in a new command, typing `Shift-Tab` will provide you with additional help for that command, e.g., the types of data that can be passed as arguments. Also, Google and ChatGPT can be extremely helpful for finding sample code and examples. Just make sure to obey the rules of academic honesty and never pass off someone else's work as your own!

```
# < Exercise 8, Improving a plot >
# Plot commands go here:
```

5.5.1 A few more plotting tricks

Matplotlib offers many, many ways to customize your plots. The code cell below illustrates how one can include a plot legend, add horizontal and vertical lines, and create custom labels for the ticks on the *x*-axis.

```
fig, ax = plt.subplots()  # Create Figure and Axes
# Plot y vs. x using a thick, green line
ax.plot(x, y, color = 'g', linewidth=3.0)
ax.set_xlim(0, 4*np.pi)  # Set the limits for the x-axis
ax.set_ylim(-1.2, 1.2)  # Set the limits for the y-axis
ax.set_xlabel('x')  # Label the x-axis
ax.set_ylabel('y')  # Label the y-axis

# Include cos(x) as well:
y2 = np.cos(x)
ax.plot(x, y2, 'b',lw = 3.0)
ax.set_title('sin(x) and cos(x)')

# Generate a legend
ax.legend(['sin(x)', 'cos(x)'])  # This line associates text using the color
                                 #    of each curve plotted on ax, in the
                                 #    order in which they were plotted
                                 # Note that if you plot more than one curve,
                                 #    separate the text strings with commas
                                 #    and place them into a list []

# Add horizontal and vertical lines to a plot
#    A dashed red line at y = 0
ax.axhline(0.0, color = 'red', linestyle ='--')
#    A solid light blue line at x = 2*pi
ax.axvline(2*np.pi, color = 'DodgerBlue', linestyle = '-')

# To get really fancy, label the x-axis in units of pi by providing a list of
#    labels at the correct locations.
#    Here we make a list of 5 labels that we pass to the tick method, which
#    expects an array of values, followed by a list of tick labels.
#    See matplotlib documentation for other options, including minor tick
#    placement.
labels = ["$0$", "$\pi$", "$2\pi$", "$3\pi$", "$4\pi$"]  # The text between $ $
                                                         #    symbols is
                                                         #    interpreted
                                                         #    as inline LaTeX,
                                                         #    where \pi is the
                                                         #    Greek letter pi
```

```
# If you want your title or legend to look better, you could write
#    ax.set_title('$\sin(x)$ and $\cos(x)$'), etc.
ax.set_xticks(np.pi*np.arange(0,5,1))
ax.set_xticklabels(labels)

plt.show()
```

You may find the default figure size of inline plots to be too small for your preference. There are two ways around this. The first is to create each new figure with the `figsize` option, as shown below.

```
fig, ax = plt.subplots(figsize = (10, 6))  # select a 10 horizontal by 6
                                           #    vertical figure
ax.plot(x, y, color = 'g', linewidth = 3.0)  # Plot y vs. x
ax.legend(['Sin(x)'])  # This line creates a legend by associating text using
                       #    the color of each curve plotted on ax
plt.show()
```

Alternatively, one may change the default figure size for any new figure, using the global `plt.rcParams[]` method:

```
plt.rcParams["figure.figsize"] = (12, 8)
# To revert to the small default size, uncomment the following line:
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]

### TRY THIS After you have executed this cell, regenerate one of your plots
#    above, for instance by re-executing your work in Exercise 8.
#    Then you can try reverting back to the smaller figure size and executing
#    your work in Exercise 8 again.
```

➤ *Coding style tip:* Once you are comfortable using the `plt.rcParams[]` method, you may find it useful to generate large plots while working through a notebook and then globally change the plot sizes for easier printing.

5.6 Check-out

Exercise 9

Briefly summarize in the cell below the ideas in this unit.

Taylor Series with Loops and Functions

6

In this unit, you will continue your introduction to Python within the context of a very important mathematical concept: Taylor series. Taylor series are so useful in Mechanics (and physics, in general) that they appear inside the front cover of *Classical Mechanics*!

The way Taylor series will be primarily used in this course is to approximate a difficult-to-solve problem with a more easily solvable one. For example, in Section 2.3 of *Classical Mechanics*, Taylor series are used to find the horizontal range of a projectile subject to air resistance. More often, you will encounter situations in which non-linear differential equations are approximated as *linear* differential equations. This is done by assuming that a function, $f(x)$, can be described as a polynomial of infinite degree and then generating an approximation of the function by ignoring the higher-order terms.

Your final goal in this unit will be to determine what degree of Taylor series is needed to approximate a particular function to a desired degree of accuracy. To accomplish this, you will need to understand logical statements, loops, and functions. These three techniques allow you to control the flow of a program, making them critical to any programming beyond the most basic tasks. Therefore, a large fraction of this unit will be devoted to introducing you to them.

6.1 Objectives

In this unit, you will learn how to:

- use Taylor series to approximate a function (PHY);
- use logical statements to write conditional commands (PRO);
- write programming loops to enable the easy repetition of code (PRO);
- write and call functions (PRO);
- use the above programming techniques to write code to determine what order of Taylor series is sufficient to approximate a particular function to a desired accuracy (PHY);
- fit a polynomial function to data (PRO).

As usual, we will start by importing the appropriate Python libraries. This same cell also increases the default figure size using the global `plt.rcParams[]` method described at the end of Introduction to Python.

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = (12,8) # make figures big (12, 8) for easier
                                       #    viewing, (4,3) for inline printing
                                       #    (width, height in inches)
# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

6.2 Taylor Series

Taylor's theorem states that, for any reasonable function $f(x)$, the value of f at a point (x) can be expressed as an infinite series involving f and its derivatives at the point a:

$$f(x) = f(a) + f'(a)(x-a) + \frac{1}{2!}f''(a)(x-a)^2 + \frac{1}{3!}f'''(a)(x-a)^3 + \cdots$$

When a location of interest x is close to the reference location a, $\delta = (x-a)$ is small, so the higher order terms in the polynomial become very small and can frequently be ignored.

6.2.1 Derivation of Taylor series

We can derive Taylor Series for the specific case when $a = 0$.

Assume a function $f(x)$ can be approximated by a power series of the form:

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_n x^n.$$

We would like to find the value of the coefficients a_0, a_1, a_2, etc., that satisfy this equation.

We can start by taking a derivative of $f(x)$ with respect to x:

$$f'(x) = a_1 + 2a_2 x + 3a_3 x^2 \cdots \; n a_n x^{n-1}.$$

Next we'll take the second and third derivatives:

$$\begin{aligned} f''(x) &= 2a_2 + (3\times 2)a_3 x + (4\times 3)a_4 x^2 \;\cdots\; n(n-1)a_n x^{n-2}, \text{and} \\ f'''(x) &= (3\times 2)a_3 + (4\times 3\times 2)a_4 x + (5\times 4\times 3)a_5 x^2 \;\cdots \\ &\quad n(n-1)(n-2)a_n x^{n-3}, \text{ etc.} \end{aligned}$$

We can now determine the values of the a_n coefficients by choosing to approximate $f(x)$ about $x = 0$. Substituting $x = 0$ into the above equations gives:

$$\begin{aligned} a_0 &= f(0) \\ a_1 &= f'(0) \\ a_2 &= \frac{1}{2}f''(0) \\ a_3 &= \frac{1}{6}f'''(0) \\ &\vdots \\ a_n &= \frac{1}{n!}f^{(n)}(0), \end{aligned}$$

and so

$$f(x) = f(0) + \frac{f'(0)}{1!}x + \frac{f''(0)}{2!}x^2 + \cdots \frac{f^{(n)}(0)}{n!}x^n$$

is the n^{th}-order Taylor approximation of the function f near $x = 0$.

6.2.2 Taylor series for sin(*x*)

We can try this out for the example of $y = \sin(x)$:

$$\begin{aligned} y(0) &= 0, \quad y'(0) = 1, \\ y''(0) &= 0, \quad y'''(0) = -1, \\ &\text{etc.} \end{aligned}$$

Therefore,

$$\sin(x) = x - \frac{1}{6}x^3 \cdots \approx x, \text{ if } |x| << 1,$$

which is commonly referred to as the small-angle approximation.

Taking just the first-order term in a Taylor series is typically called a first-order Taylor approximation. Taking all terms up to the third order would be called a third-order Taylor approximation. Let's try plotting some Taylor approximations of $\sin(x)$ about the point $x = 0$.

Exercise 1

On a separate sheet of paper, complete the derivation of the Taylor series approximation of $\sin(x)$ out to fifth-order.

Then plot the original function and its first-order Taylor approximation, using `ax.legend()` to label the curves.

Modify your plot to include higher-order Taylor approximations up to the fifth order, again labeling the curves.

➤ *Hint:* There is no need to write your own factorial calculator — Python has got you covered. Confusingly, there are several ways to compute factorials in the Python/NumPy environment. We recommend not importing the math library (because of potential clashes with NumPy), so if you need a single value, use this form from the NumPy library:

```
np.math.factorial(3)
```

which takes an integer argument and returns an integer.

Over what range of x values would you say that the fifth-order Taylor series is a good (say, within 10%) approximation for sin(x)?

(Write your response here.)

```
# < Exercise 1, Taylor series for sin(x) >

n_points = 1000  # Number of points
# Create array from 0 to 2pi with n_points total points
x = np.linspace(0.0, 2*np.pi, n_points)
y = np.sin(x)  #  Create sine data array
y1 = x # first-order Taylor approximation

# Set up plot
fig, ax = plt.subplots()
ax.set_xlim(0, 2*np.pi)
ax.set_ylim(-2., 2.0)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('sin(x) and Taylor series approximations')

# Plot data
ax.plot(x, y, color='red')    # Plot sin(x)
ax.plot(x, y1, color='blue')  # Plot the first-order Taylor approx for sin(x)
```

6.2.3 Taylor series about an arbitrary *x* value

So far, we have only looked at Taylor series about the point $x = 0$. This is technically a special case of a Taylor series, called a Maclaurin series. In general, to compute a Taylor series about an arbitrary location of $x = a$, we must make the following alteration to our Taylor series expression:

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots \frac{f^{(n)}(a)}{n!}(x-a)^n.$$

Exercise 2*

Plot the third-order Taylor approximation to $\sin(x)$ around $x = 3$. Include $\sin(x)$ on your plot, as well.

```
# < Exercise 2, Taylor series around non-zero value >

n_points = 1000  # Number of points
# Create array from 0 to 2pi with n_points total points
x = np.linspace(0.0, 2*np.pi, n_points)
a = 3

### INCLUDE your calculation and plot below
```

An aside on factorials

For future reference, if you want to compute an array of factorials, say up to n = 5, you can't do this using `np.math.factorial()`, because that method only works on single values.

Instead, in your toolbar above, search for "factorial" under the Help menu/SciPy Reference/Search the Docs. You'll find it under `scipy.special.factorial`. You will need to import the library of special functions as shown in the cell below.

```
arr5 = np.arange(0,6)
import scipy.special
print("Array of integers from 0 to 5: ", arr5)
print("Factorial of each element in arr5", scipy.special.factorial(arr5))
```

Note that this form of factorial will act on an entire array of floating-point numbers and will return floats as an approximation via the gamma function. If you want exact values of integers, use the keyword argument `exact = True`:

```
print(scipy.special.factorial(arr5, exact = True))
```

Finally, if you forget the syntax of these function calls, you could calculate, say 5!, using NumPy's function `prod` (which calculates the product of all elements in an array) like so:

```
print("5! is the product of 1*2*3*4*5: ", arr5[1:].prod()) # better skip the
                                                           #    first element!
```

6.3 New Programming Tools

6.3.1 Logic

In order to write useful code, you will want your program to behave differently in different situations. To accomplish this, your program will

1) need to be able to evaluate whether certain conditions are satisfied and
2) evaluate different code depending on whether these conditions are or are not satisfied.

Logical and comparison operators allow us to test whether variables satisfy certain conditions. Comparison between two or more variables can be made using the following symbols:

- >, < ("greater than" and "less than")
- >=, <= ("greater than or equal to" and "less than or equal to")
- ==, != ("equal to" or "not equal to") Note that a single equals sign assigns the value of the variable or number or array on the right to the variable on the left. Two equal signs are needed for a comparison, yielding a logical `True` or `False`.

Multiple conditions can be combined together using

- `and` (if both the operands are true, then the condition becomes true)
- `or` (if any of the two operands are true, then the condition becomes true) and

- `not` (used to reverse the logical state of its operand; for instance, a `True` statement becomes `False`).

Parentheses may be used to specify the order of operations.

```
# Examples of how to use conditional statements
### DO THIS: Before running this cell, predict the outcomes of each of the
#    statements.

a = 10
b = 20
c = 30

# Comparisons are used to test whether a equals b
print("a equals b: ", a == b)
# Comparisons are used to test whether a is less than or equal to b
print("a is less than or equal to b: ", a <= b)
# Tests whether a is less than b and b is less than c
print("a is less than b and b is less than c: ", a < b and b < c)
# Tests whether either a or c is less than b
print("a or c is less than b: ",(a < b) or (c < b))
# Tests whether a is less than b and b is not less than c
print("a is less than b and b is not less than c: ", (a < b)
      and not(b < c))
# Note that the lengthy line of code above is continued onto a second line.
#    Because the parentheses from the print statement are not closed on
#    the first line, Python knows the statement continues.  In other
#    situations, a single backslash, '\', at the end of the first line
#    may be used to indicate that the line continues.

print(not False, not 0, not True, not 1)  # The number "zero" evaluates to
                                          #    False and the number "one"
                                          #    evaluates to True.
# What do you think this line will produce if a = 10?
print(not (False and a < 20))

### DO THIS: Change the values of a, b, and c and then rerun this cell.
```

Exercise 3

Write a line of code that will evaluate to *True* only if *a* is equal to *b* or if *a* is equal to *c* but *not* if both *a* is equal to *b* and *a* is equal to *c* (i.e., all three variables are the same). Test your code using different values for *a*, *b*, and *c*.

```
# < Exercise 3, Combinational logic >

a = 20
b = 20
c = 10
print("a = {:}; b = {:}; c = {:}\n".format(a, b, c))
```

```
print("(a equals b) or (a equals c): ", (a == b) or (a == c)) # example
### FIX
print("(a equals b) or (a equals c) but not if  (a equals b equals c): ")
```

6.3.2 Conditional statements

Once your program has evaluated whether a condition is `True` or `False`, it can make a decision to execute different code. This is done using `if` and `if ... else` statements, as shown below. In these statements, the condition (e.g., `a == 10`) is separated from the outcome by a colon, and indentation is used to separate the code inside the `if` statement from the rest of the code. Indentation must be consistent throughout a cell (typically set at four spaces). Typically your Jupyter notebook will automatically create the indentation after you press Enter/Return after typing the colon character:

`if` statement

```
if expression:
    statement(s)
```

`if ... else` statement

```
if expression:
    statement(s)
else:
    statement(s)
```

```
# Examples of "if" and "if ... else" statements.
### DO THIS: Before running this cell, predict the outcomes of each of the
#    statements.

a = 10

# If the outcome consists of only a single line of code, it may go on the same
#   line as the "if" statement.
if a == 10 : print("a equals 10")

# If the outcome consists of multiple lines of code, it is separated from the
#   rest of the code by indentation and forms a block paragraph:
if a == 10 :
    print("a equals 10")
    print("hooray")

# Once the "if" statement is done, the left-justification of the code returns
#   to as it was before.
name1 = "Alice"

# "if ... else" statements allow the programmer to specify an outcome if the
#   condition is not met.
if name1 == "Carlos" : print("name1 is Carlos")
```

```
else :
    print("name1 is not Carlos.")
    print("Who is it?")

# "elif" (else-if) can be used to check multiple conditions.
#   Note that if the "if" statement evaluates to "True", the "elif" statement
#   will not be evaluated.
if name1 == "Carlos" : print("name1 is Carlos")
elif name1 == "Alice" : print("name1 is Alice")
else :
    print("name1 is not Carlos or Alice.")
    print("Who is it?")

# Multiple "if" and "if ... else" statements may be nested together to create
#    more complex conditionals.
name2 = "Bob"
if name1 == "Alice" :
    if name2 == "Bob" : print("Alice and Bob are both here.")
    else : print("Alice is here but where is Bob?")
else : print("We are missing Alice.")

### DO THIS: Change the values of a, name1, and name2; reevaluate this cell
```

Exercise 4

In the cell below, use nested (cascaded) `if ... else` statements to write code that will compare three variables, *a*, *b*, and *c*, and produce the following outcomes:

- print "*a* is False" if *a* equals 0;
- print "*a* is the greatest" if *a* is not equal to zero and *a* is greater than *b* and *c*;
- print "*a* is less than or equal to *b* or *c*" and print the values of each of the variables if *a* is less than or equal to either *b* or *c* and *a* is not equal to 0.

Set *a*, *b*, and *c* equal to different values to test that your code produces the expected results.

```
# < Exercise 4, Conditionals >

a = 10
b = 20
c = 30

### Your code here
```

6.3.3 Loops

In other situations, you will want your program to complete the same actions multiple times. Loops are one way to accomplish this sort of task.

`for` loops enable you to iterate through each item in an array (or list or string). The syntax looks like

```
for iterating_var in sequence:
    statement(s)
```

As with `if` statements, the colon separates the statement(s) from the condition, and indentation is used to separate the code inside the loop from that outside of it. An example is below.

```
# Example for loop
### DO THIS: Before running this cell, predict the outcome.

# Create an array from zero to five (inclusive) in steps of 1.
iterator = np.arange(0, 6, 1)

for i in iterator:
    print(i**2) # print the squared value of each element in the array
```

Sometimes, you will want to stop iterating (i.e., break out of the loop) before you have gone through each item in the array. A `break` statement inserted into the loop (typically used with an `if` statement) will do this for you, as shown below.

```
# Example for loop with a break
### DO THIS: Before running this cell, predict the outcome.

# Create an array from zero to five (inclusive) in steps of 1.
iterator = np.arange(0, 6, 1)

for i in iterator:
    if i**2 > 10 : break  # if i^2 is greater than 10, break out of the loop
    print(i, i**2)  # otherwise print the squared value of each element in the
                    #   array
```

There are times that you will wish to iterate over multiple variables in the same loop. For instance, you may have two different lists or arrays that you would like to manipulate elements on at the same time. This can be done using `zip`, as shown in the cell below, behaving like a zipper to match the two arrays term-by-term.

Specifically, what `zip` does is combine the matched elements in the array (or list or other iteratable object) into tuples. If you are curious, you can add

```
print(list(zip(array1, array2)))
```

to the code block below to see the operation.

```
# Example of using zip to iterate through two arrays simultaneously

### DO THIS: Try changing the arrays to make them different lengths.
#   What happens if the length of array1 is less than the length of array2 and
#   vice versa?
```

```
### DO THIS: See if you can add a third array to iterate through

array1 = np.arange(0, 6, 1)
array2 = np.arange(8, 2, -1)

for a, b in zip(array1, array2):
    print("{0} x {1} = {2}".format(a, b, a*b))
```

In other circumstances, you may wish to keep track of how many iterations the loop has currently completed. For example, you may wish to know how many times the loop iterated before a break condition was met. This can be accomplished elegantly using the `enumerate` function, which returns both the *count* of the current iteration and the *value* of the item in the current iteration, as shown below.

```
# Example of using enumerate to preform a sequential search for "cabbage" in a
#    list

veg_list = ["radish", "celery", "carrot", "cabbage", "asparagus"]
for count, value in enumerate(veg_list):
    print(count, value)
    if value == "cabbage" : break

print("Cabbage is element {} in veg_list".format(count))
```

There is much more that could be said about loops. For example, in addition to `for` loops, Python also supports `while` loops (loops which are iterated until a condition is met instead of evaluating for each iterator). Loops can also be nested (indented blocks) inside each other to allow one to iterate through multiple variables at once. You are encouraged to research both of those ideas as they come up, but for now you know enough to get started.

Exercise 5

Write a loop that will iterate through an array, printing for each element in the array the sum of it and all previous elements. Break out of the loop if your sum ever exceeds 50. Test your loop on an array that goes from 0 to 20 in steps of 1. You should find that it stops iterating when it reaches 55.

➤ *Hint:* Consider defining a variable that will be updated inside of the loop.

```
# < Exercise 5, Loops >
### Your code here
```

6.3.4 An aside on speed

One last word to the wise: Loops can be a natural way to do computations in Python, but they are often not the fastest way. NumPy arrays, in particular, allow a user to do a calculation

on each element in an array "all at once".[1] This can result in cleaner code and also faster code. For example, the block of code below does the same calculation as the first two loop examples above.

For the types of code you will be writing for these exercises, the speed usually does not matter, so you should feel free to use loops whenever it feels right. It is important to be aware that there may be other options, though.

```
# Example NumPy array calculations that mimic the loop examples above

# Create an array from 0 to 5 (inclusive) in steps of 1.
array1 = np.arange(0, 6, 1)

# The squared elements of array1
array1_2 = array1**2
print(array1_2)

# Print only those elements of array1_2 that are less than 10 (The logical
#   test creates True and False, and then we print only the elements whose
#   indices are True)
print(array1_2[array1_2 < 10])
```

Similarly, NumPy has a method for cumulatively summing an array, as you did in **Exercise 5**. This method is referred to as `np.cumsum` (*cumulative sum*), and an illustration of how it could be used to accomplish the same task as **Exercise 5** is below.

```
#  Example of using np.cumsum to sum all the elements in an array

iterator = np.arange(0,22,1)  # make the array to be added up
cumulative_sum = iterator.cumsum()  # perform the cumulative sum
print("All elements of the running sum: ", cumulative_sum)
print("All elements less than 50: ", cumulative_sum[cumulative_sum < 50])
```

Exercise 6*

To compare the efficiency of doing calculations using NumPy arrays rather than loops, we can use the magic command `%%timeit`, which runs any entire code cell several times and reports the amount of time taken for the whole cell (reported as a loop). Below, all the integers from 0 to 5,000 are added up using a `while` loop and the amount of time taken is reported. Write and time how long it takes to do a similar calculation using a `for` loop and with the `np.sum()` method.

You'll find that the NumPy version of the calculation is 10–40x faster than when using loops.

[1] This is not to imply that calculations on arrays are done in parallel. Arrays are merely collections of numbers of the same type, which makes the underlying calculations particularly efficient because each element of the array can be stored sequentially in the computer's memory, allowing fast access.

```
# < Exercise 6 >

import timeit
#  %%timeit must appear by itself at the top of the code cell to be timed,
#    so we import it into a separate cell first.
```

```
%%timeit
# < Exercise 6 >

# calculate the sum of all integers from 0 to 5000 inclusive using a "while"
#   loop
sum5000 = 0
i = 0
while i < 5000:
    i+= 1
    sum5000 = sum5000+i
# comment out the print statement when using %%timeit; otherwise it will take a
#    long time
#print(sum)
```

```
%%timeit
# < Exercise 6 >
# calculate the sum of all integers from 0 to 5000 inclusive using a "for" loop
```

```
%%timeit
# < Exercise 6 >
# Calculate the sum of all integers from 0 to 5000 inclusive using a NumPy array
#   and np.sum()
```

```
# For future reference, note that %%timeit will not actually return any values
#    to the notebook except for its timing results.  Here, for example, sum5000
#    is undetermined after running the last cell.  You need to comment out the
#    %%timeit in order to return any values from the code cell.

### UNCOMMENT this line to see the error
#print(sum5000)
```

6.3.5 Functions and other subroutines

In other situations, you may want to repeat the same operations in different contexts. Defining your own functions[2] will allow you to do this without writing the same code over again.

[2] Technically, a function returns values, a procedure returns nothing, and a method is a function or procedure bound to a class. Because the syntax is the same, we will refer to functions and procedures interchangeably.

In programming, repetitive code is bad — not only does repetition clutter your code, making it harder to read, but repetition also makes your code harder to debug and harder to update.

Python and NumPy come with a huge number of built-in functions, many of which you are already using (e.g., `print()`, `len()`, `np.arange()`, and `np.linspace()`). *Caution: `print()` and `len()` are Python primitives, but `np.arange()`, `np.linspace()` come from NumPy*. Each of these functions performs some action and may return a value. For example, `len()` returns the length of a list or array. Functions frequently have parameters, which are listed inside the parentheses. When the function is called, a value (known as an argument) can be passed to that function and stored as a variable with that parameter name. For example, `len()` takes a list or array as its argument. Therefore, when I write

```
length = len(array1)
```

I am calling the function `len()` with the argument of *`array1`*. The `len()` function does some magic to determine the number of elements in *`array1`*, then assigns *`length`* to that returned value. When you define your own functions, you will call them the same way. (For future reference, NumPy gives more information about multidimensional array sizes via `np.size()`.)

Defining a function

Many times, Python will not have a built-in function that accomplishes what you want, so you will have to define your own function. To define a function, you must follow a specific form:

- The function block begins with the keyword *`def`* followed by the function name and then a pair of parentheses.
- Any input parameters or arguments are listed inside the parentheses. If the function does not take any parameters or arguments, the parentheses are left empty.
- A colon separates the function header from the code block, and the entire code block is indented.
- The statement *`return [expression]`* exits the function, optionally passing back an expression to the caller. A return statement with no arguments is the same as returning nothing.

As a result, your function will look like

```
def function_name ( parameters ):
    statement(s)
    return [expression]
```

and will be called like

```
function_name( parameters )
```

```
# Example of a simple user-defined function (technically a "procedure") that
#   returns nothing
```

```
# Define the procedure "say_hello" that takes one argument, "name".
def say_hello(name):
    """Print: Hello [name]."""  # Three double quotation marks are another way
                                #    of designating a comment
    print("Hello ", name)
    return

say_hello("Dave")

### DO THIS: Add a line of code to call the say_hello procedure with your name
```

It is good practice to document the purpose of a function in a comment immediately following the function definition. This comment forms a *docstring*, which is readily visible from the coding interface as shown below. Typically, we enclose it in pairs of triple quotation marks to allow for multiline commenting. This level of commenting may seem a bit overboard for the simple functions that you will initially be writing, but it is good to get in the habit now.

```
# Print the docstring associated with the procedure say_hello()
#     Note that the word "doc" is surrounded by double underlines
print(say_hello.__doc__)
```

Another way you can access the docstring for this function is to add a question mark to the end of it:

```
say_hello?
```

Happily, this syntax will not only provide documentation for the functions you wrote but also for procedures, methods, objects, and so on, including ones you imported. This makes it a very powerful tool when learning how to use new libraries.

```
# Print information about the function/procedure
say_hello?
```

```
# Print information about a function that you imported
np.arange?
```

To get an even deeper level of understanding of the code, you can use two question marks to access the source code for the function (or procedure, method, or object). (Sometimes the documentation information will be printed but not the source code. This is typically because the object was not implemented in Python, but in some other language such as C with Python providing the interface.)

```
# Print information about the function/procedure AND the source code
say_hello??
```

It is possible to set default values for the parameters in the function. The following block of code defines a function similar to `say_hello` but sets a default name so that it can be called without a parameter.

```
# Example of a simple user-defined procedure, with a default value for the
#    parameter

# Define the procedure "say_hello" that takes one optional argument, "name".
def say_hello_default(name = "you"):
    """Print: Hello [name]."""
    print("Hello " + name)
    return

say_hello_default()
say_hello_default("Dave")

### DO THIS: Uncomment the following line of code
# say_hello_default()
```

Functions can take multiple optional parameters with default values. In these circumstances, the function call can contain the name of the parameters to designate which one is being referred to. If a parameter name is not given, the first parameter in the function definition will be set to the first value passed in the function call and so on.

```
# Example of a simple user-defined function, with two optional arguments

# Define the function "say_hello" that takes two optional
#   arguments, "greeting" and "name".
def greet_default(greeting = "Hello", name = "you"):
    """Print: [greeting] [name]."""
    print(greeting + ", " + name)
    return

greet_default()
greet_default(name = "Dave")
greet_default(greeting = "Hi")
greet_default("Hey there")
greet_default("Greetings", "Earthling")
```

Return values

The previous examples have been of procedures that are very simple and return no information, so here is a slightly more complex one involving multiple return statements.

```
# Example of a slightly more complex function that returns a value

# Define a function to determine the distance between two points (which may be
#    integers, arrays, or lists and have any number of dimensions).
def find_distance(point1, point2):
    """Return the distance between two points."""
    # If point1 and point2 are both integers, the distance between them is just
    #    the absolute value of their difference
    if (type(point1) == int and type(point2) == int):
        return abs(point1 - point2) # Exit the function and return the absolute
                                    #    value of the difference

    # If the function is called with two parameters that do not have the same
    #    number of elements, print an error and exit gracefully
    if (type(point1) == int or type(point2) == int
        or len(point1) != len(point2)):
        print("The two points do not have the same dimension")
        return # Exit the function and return nothing

    if type(point1) == list: point1 = np.array(point1) # If they are lists,
                                                       #    convert to arrays
    if type(point2) == list: point2 = np.array(point2)

    # If point1 and point2 are arrays of the same length, use array
    #    calculations to find the distance. np.sum calculates the sum of all
    #    elements in the array; alternatively, array_name.sum()
    #    np.sqrt(num) returns the square root of num
    dist = np.sqrt(np.sum((point1 - point2)**2))
    return dist

# And we now give an example of usage of the function 'find_distance()'
p1 = np.array([1, 2])
p2 = np.array([2, 7])
distance = find_distance(p1, p2)
print("Distance: ", distance)

### DO THIS: Test this function by calling it with different arguments.
# See if you can come up with arguments that will result in each of the three
#    different return statements.
```

Of course, NumPy includes an efficient function that calculates the Euclidean distance between arrays of any dimension by using the `norm` function from its built-in linear algebra package `np.linalg`. It is less flexible than the user-defined function above, because it requires NumPy arrays as input, and the dimensions of the inputs need to be greater than 1:

```
# Example using np.linalg.norm to calculating the distance between NumPy arrays

p1 = np.array([1, 2])
p2 = np.array([2, 7])
```

```
distance = np.linalg.norm(p1 - p2)
print("Distance between {} and {}: {}".format(p1, p2, distance))

p1 = np.array([0, 0, 0])
p2 = np.array([1, 1, 1])
print("\nHere's the well-known distance between diagonally opposite corners"
      + " of a unit cube:")
print(np.linalg.norm(p1 - p2))
print("which should equal the square root of 3: ", np.sqrt(3))
```

Exercise 7

Write a function that will solve a 1-D constant-acceleration, kinematic problem analytically, meaning that you will use the familiar equations of motion for constant acceleration. Your function should take an initial position, an initial velocity, an acceleration, and a time and return the position after that time. Test your function by running it with a variety of different parameters, including $x_0 = 2$, $v_0 = 1$, $a = -10$, and $t = 3$ (which should produce -40). Include a docstring for your function. Try sending your function an *array* of times rather than a single value. Does it still work?

```
# < Exercise 7, 1-D kinematic function >
### Your code here
```

Scope of variables and functions

One of the beautiful things about functions is that they are their own little environments. This means that any variable defined inside the function will not be available outside of the function. Such variables are considered *local variables* in contrast to *global variables*, which are defined outside of functions and available everywhere, even inside other functions. Where a given variable is available is referred to as the *scope* of the variable. An example is below.

```
# Example of local vs. global variables
### DO THIS: Before running this cell, predict the outcome.

# Defining two "global" variables. These two variables will be available
#    everywhere
globalvar_a = 1
var_b = 2

def scope_function1():
    """Print the value of two variables to illustrate scope"""
    var_b = 3
    localvar_c = 4
    print("I'm inside the function!")
    print("\t globalvar_a equals ", globalvar_a)
```

```
    # Since globalvar_a is defined outside of a function, it is available
    #    everywhere

    print("\t var_b equals ", var_b)
    print("\t localvar_c equals ",localvar_c)
    return

scope_function1()
print("\nI'm outside the function!")
print("\t globalvar_a equals ", globalvar_a)
print("\t var_b equals ", var_b)

# Even though var_b was modified inside the function, var_b outside the
#    function is unchanged. What happened is that the line "var_b = 3" actually
#    defined a local copy of the variable inside of the function, rather than
#    altering the global variable var_b.

### DO THIS: Uncomment the following line
# print("\t localvar_c equals ",localvar_c)

# Because localvar_c was only defined inside the function, it is not available
#    outside of it.
```

If you would like to see all presently used global variables and functions, use the `%who` magic command.

```
# Print all presently used global variables
%who

### TRY THIS: Restart the kernel (using the command under the Kernel pulldown
#    menu). Doing this will clear the global variables, as you will see
#    when you rerun this cell. When you rerun the cell after restarting the
#    kernel, notice that the number next to this cell has changed to 1. Cells
#    are numbered in the order they have been executed starting with 1 each
#    time the kernel starts.

#    Before you continue to Exercise 8, you will need to import your libraries
#    again (i.e., rerun the first code cell of this notebook). Even better, you
#    can use the command "Run all above" located under the Cell menu to rerun
#    all the code up until this point, then run this cell.
```

As you can see, keeping track of the scope of variables can get confusing. To keep things clean, it is smart to limit the use of variables that were defined outside of the function, *unless* they were passed as parameters.

6.4 Taylor Series Using Loops, Functions, and Conditional Statements

Exercise 8

For this exercise, you will practice putting together the skills you have learned by writing code to approximate the function $ln(1 + x)$ as a series. As you may have discovered in the "Taylor Series" exercises, by using Taylor series we can show that

$$ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots$$

If you have not already done so, verify on a separate sheet of paper that the above equation is true.

Your goal is to write code that will determine how many elements of the series are needed to approximate $ln(1 + x)$ to a certain level of precision, given a value of x. To do this

- define a function that will return the nth element of the series, given the parameters x and n;
- write a `for` loop that will calculate the sum by calling your user-defined function, breaking out when the sum of the series is within some ϵ of the actual value of $ln(1 + x)$;
- use your loop to determine how many elements of the series are needed to achieve the desired level of precision.

Please test your code with a variety of different values for x and ϵ, including $x = 0.8$ and $\epsilon = 0.001$ (which should require 16 elements in the series).

➤ *Hint 1:* The smartest way to write code is to build it from smaller pieces. In this case, I suggest writing and testing your function before using it in the loop.

➤ *Hint 2:* Printing variables inside your loop may help you determine if your code is working correctly.

➤ *Hint 3:* You'll have to look up the NumPy function for the natural logarithm.

```
# < Exercise 8, Approximating ln(1+x) >
### YOUR CODE here
```

Exercise 9*

One way to validate a Taylor series approximation is to compare it to a polynomial fit to the data. First, find the Taylor series expression for $\cos(x)$ near $x = 0$.

(write your answer here)

Now compare to a polynomial fit to the data by using scipy.optimize.curve_fit to fit a second-order polynomial. Then compare the coefficients returned by curve fit for a second-order polynomial to the values of a_0, a_1, and a_2 generated for the Taylor series.

To use `scipy.optimize.curve_fit()`, you must provide a function in the form of the desired fit. This function should take as arguments your dependent variable array, x, along with the parameters that will be adjusted to provide a fit. In this exercise, the functional form is a second-order polynomial and the parameters are a_0, a_1, and a_2:

$$y = a_0 + a_1 x + a_2 x^2$$

Fix the template code in the cell below to define this function.

The syntax to call `curve_fit()` is then

```
popt, pcov = curve_fit(fitFunc, x, y, p0)
```

where `fitFunc` is the name of the fitting function you defined (`poly_fit()` in this case), `x` is the array of independent variables, and `y` is the array of dependent variables. The variable `p0` is an array of initial guesses for the parameters in your fitting function, in this case, a guess for each of the coefficients. The returned values are 1) the array of fitted parameters (`popt`) optimized such that the square of the residuals (fitted y-values - data y-values) is minimized and 2) the estimated approximate covariance of the fitted parameters (`pcov`), which is a measurement of the quality of the fit.

In the second cell below, use `curve_fit()` to fit a second-order polynomial to the function $\cos(x)$. You will get the best comparison to the Taylor series if you limit the range of x values, for example, $-\pi/8 \leq x \leq \pi/8$. This task can be accomplished by writing a logical statement that returns whether a given element in an array fulfills that statement:

```
ind = ((-np.pi/8 < x) & (x < np.pi/8))
```

This returned array of True/False values can then be used as the indices of an array: `x[ind]` or `y[ind]`.

Finally, plot the results of your best-fit polynomial along with the original function and the Taylor series approximation on the same set of provided axes.

➤ *Hint:* To plot your polynomial fit, you may, of course, write a polynomial using each of the values of `popt` as the coefficients: `popt[0] + popt[1]*x + popt[2]*x**2`. However, a similar and cleaner way to accomplish this is to use the `poly_fit()` function along with your best-fit values: `poly_fit(x, *popt)`. Here the asterisk before `popt` tells the computer to unpack the values of the `popt` array and pass each of the values individually as arguments to the `poly_fit()` function.

How similar are the coefficients for the Taylor series to those found using `curve_fit()`?

How do the coefficients to the polynomial fit change if the range of x-values used in the fitting is decreased?

```
# < Exercise 9, Curve fitting >

#  Define the functional form to be used by curve_fit
```

```
def poly_fit(x, a0, a1, a2):
    """Returns the result of a second-order polynomial for the provided
    coefficients"""
    result = 0  ### FIX

    return result
```

```
# < Exercise 9, Curve fitting, continued >

# Import the curve_fit method from the scipy.optimize library
from scipy.optimize import curve_fit

# An array of independent values
x = np.arange(-np.pi, np.pi, 0.01)
# cos(x)
y = np.cos(x)
# Taylor series approximation
y3 = y ### FIX


# Provide an array of initial guesses for each of the coefficients to be sent
#    to curve_fit
p0 = np.array([0.0, 0.0, 0.0])  ### SET your initial guesses
# Limit the range of x-values over which the fit will happen
ind = ((-np.pi/2 < x) & (x < np.pi/2))  ### ADJUST the range
# Call curve_fit
popt, pcov = curve_fit(poly_fit, x[ind], y[ind], p0)

### PRINT fit results


# Set up plot
fig, ax = plt.subplots()
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('y = cos(x) near x = 0')

### PLOT cos(x), Taylor fit, and second-order polynomial fit

plt.show()
```

Exercise 10*

This exercise is similar to Problem 2.18 of *Classical Mechanics*.

1. On a separate piece of paper, find the Taylor series expressions around $x = 0$ for
    1. $\ln(1 + x)$
    2. e^x,
    3. $(1 + x)^n$ (where n is a constant).

2. In each of the cases, comment on the range of x and n values that will provide a convergent solution.

3. Next, plot the first three terms of the Taylor approximations for each function in separate plot windows in the cells below.

➤ *Note: The Taylor series that you just derived in this exercise and for* $\sin(x)$ *and* $\cos(x)$ *earlier are used often enough in physics that it is worth committing them to memory!*

```
# < Exercise 10 A >
# ln(1+x)
x = np.arange(0, 1, 0.01)
y = np.log(1 + x)
# y3 =

# Set up plot
fig, ax = plt.subplots()
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('y = ln(1 + x) near x = 0')

# Plot data
# ax.plot(x, y, color='red') # plot y(x)
# ax.plot(x, y3, color='blue') # plot Taylor series approximation
```

```
# < Exercise 10 B >
# e^x
x = np.arange(-2, 2, 0.01)
```

```
# < Exercise 10 C >
# (1+x)^n
x = np.arange(-2, 2, 0.01)
```

6.5 Check-out

Exercise 11

Briefly summarize in the cell below the ideas in this unit.

Numerical Integration Applied to Projectile Motion

7

One of the fundamental questions in mechanics is how to solve the equation of motion (such as $\ddot{x} = -\frac{k}{m}x$) for a system in order to determine how it evolves over time. Equations of motion typically relate the position of a mass to its time derivatives (velocity and acceleration), meaning that they are *differential equations*. In this course, you will be learning a number of techniques to solve analytically (i.e., via algebra and calculus) these differential equations, but oftentimes an analytical solution cannot be found. In these circumstances, scientists may turn to computers for numerical solutions to the differential equations. In this unit, and in most of the exercises hereafter, you will be doing exactly that — numerically solving the differential equations of motion. These solutions, say $x(t)$ or $v(t)$, arise from integrating a force law via a set of time derivatives, from known starting values of $x(0)$ and $v(0)$. As such, these are referred to as "initial value problems."

Specifically, in this unit you will write a numerical integrator to solve the trajectory of a projectile subject to constant acceleration. Later on, you will use the numerical integrator from the SciPy Python package, but it is instructive to write your own.

7.1 Objectives

In this unit, you will:

- write a numerical integrator that relies on the simple Euler method (PRO);
- use your simple Euler integrator to find the trajectory of a projectile (PHY);
- improve your simple Euler integration method by calculating the derivatives halfway between time steps (PRO).

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = (12,8)
# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

7.2 Simple Euler Integration of 2-D Projectile Motion

Euler integration is a method for numerically integrating to solve a differential equation.

The main idea of Euler integration while solving kinematic problems is to use Taylor series approximations to update the position of a projectile by

1) applying a constant acceleration for a small time interval to determine a new velocity,
2) applying the new velocity for a small time interval to get the new position, then
3) rinse and repeat steps 1) and 2), building up the solution via iteration.

➤ *Note:* We will be using only first-order Taylor approximations here.

You will need to store x, v_x, y, and v_y at each point in time. Rather than hold them as individual one-dimensional arrays, we use the following convention:

- `r_soln[0]` will be the x position. We'll call it `pos_x`.
- `r_soln[1]` will be the x component of the velocity. We'll call it `vel_x`.
- `r_soln[2]` will be the y position. We'll call it `pos_y`.
- `r_soln[3]` will be the y component of the velocity. We'll call it `vel_y`.

So this `r_soln` array contains four elements for each time step. You will create solutions on an array of time points that we will call `t_arr`.

Note that the complete solution will be found by pairing `t_arr` with pieces of the solution array `r_soln`. For example, we can plot `y(t)` via a call to `ax.plot(t_arr, r_soln[2])`; similarly we plot the trajectory using `ax.plot(r_soln[0], r_soln[2])`.

➤ *Programming style tip:* Why the long variable names? Variable names should be as descriptive as possible, and yet not standard words, which might be used in the future to represent something else (for instance, time-keeping functions in Python are in a library called `time` so we want to avoid redefining, a.k.a. "clobbering" that name). We'll need to distinguish between purely computational results and those derived algebraically, also known as analytical results. So, later, we'll have `pos_x_analyt`, `vel_x_analyt`, etc. Finally, if you are working on a large project, but you want to change the variable names throughout the notebook (or indeed, an entire directory of notebooks), you can use Search/Replace to substitute some other unique name for your chosen variable, without interfering with comments, which will generally be written in complete sentences. For ease of searching, it is also recommended that variables be longer than single characters, e.g., `grav` rather than `g` for the gravitational acceleration near the surface of Earth.

7.2.1 Using simple Euler integration

The following cells will use first-order Euler integration to find the trajectory of a projectile without drag force. Examine and run each of them in turn.

Exercise 1

The following cell will set the initial conditions and parameters for the system. Please edit it to set the following (global) variables:

- $g = 9.8$ m s^{-2},
- $t_{start} = 0.0$ s,

- $t_{end} = 15.0$ s,
- and the number of time steps to 3000.

Then edit it to define the initial conditions such that the projectile starts from (0, 0) with velocity = (20 m/s, 20 m/s), i.e., a launch from the origin at an angle of 45 degrees above horizontal.

```
# < Exercise 1, Setting parameters for projectile motion >

# No damping, numerical integration, projectile motion over level ground:
#    defining variables.

# Define the variables that will be used throughout the integration
grav = 0 # [m s^-2]; the acceleration due to gravity.  By declaring it here it
         #    will be available anywhere in the rest of the notebook

# Create
t_start = 0  # [s]; starting time
t_end = 1    # [s]; ending time
n_steps = 1  # number of time steps
h_step = (t_end-t_start)/n_steps  # h is the time step, delta_t
t_arr = np.arange(t_start, t_end, h_step)  # an array of evenly spaced time
                                           # points at which to find solutions

# Initialize x and v.  For a complete solution to the differential equations, we
#    need to specify the starting values of x, vx, y, and vy
pos_x0 = 0 # initial x position
pos_y0 = 0 # initial y position
vel_x0 = 0 # initial x-component of the velocity
vel_y0 = 0 # initial y-component of the velocity
r_init = np.array([pos_x0, vel_x0, pos_y0, vel_y0])  # this array contains the
                                                     #    initial values of x,
                                                     #    vx, y, vy, in order


### DO THIS: Add a line of code to print the variable h_steps and the array
#       t_arr so that you can see what they contain.

# After you have done so, remove the line of code and rerun the cell so that
#    you don't clutter up the output.
```

In a first-order Euler integrator, the time derivative is used to propagate the motion forward by assuming $x(t + \delta_t) \approx x(t) + \delta_t \frac{dx(t)}{dt}$. The following cell defines a function to return the time derivative of all the variables.

In order to examine how the derivative function works, add the following line to the end of the cell before you run it:

```
print(deriv(0, r_init))
```

This line will call the derivative function for $t = 0$ and print the derivatives of x, v_x, y, and v_y. Convince yourself that the values will print correctly.

```
# No damping, numerical integration, projectile motion over level ground:
#    derivatives.

# We need to define a function that returns the time derivatives of all the
#    variables.

# The syntax for this defined function begins with the reserved word *def*,
#    then the function name.  In parentheses go the variables (which are
#    typically arrays and scalars).  Following the variables, we put a colon.
#    The colon tells the Python interpreter that the indented block that
#    follows falls under the scope of the defined function.  Finally, the
#    reserved word *return* tells us the output of the function.  In this case,
#    we return the derivatives in a 4-element array (x, vx, y, vy), using the
#    np.array() function.

def deriv(t_now, r_now):
    """Return the time derivatives at time t_now of pos_x, vel_x, pos_y,
    vel_y assuming constant velocity.
    r_now contains 4 elements: x, vx, y, vy at this point in time"""
    # First, extract the variables from the r-array
    pos_x = r_now[0]
    vel_x = r_now[1]
    pos_y = r_now[2]
    vel_y = r_now[3]

    # Now create the derivatives. Note: by definition dx_dt = vel_x
    dx_dt = vel_x
    dvx_dt = 0
    dy_dt = vel_y
    dvy_dt = -grav
    dr_dt = np.array([dx_dt, dvx_dt, dy_dt, dvy_dt])
    return dr_dt # we need to specify the output of this function

### DO THIS: Add the line of code to call the deriv function here.
```

In order to continue propagating the trajectory forward in time, you will need to loop through the entire array of times (`t_arr` from two code cells ago). Each time you do so, you will calculate the derivatives of position and velocity at the previous step and use them to find the new values of position and velocity. In order to keep track of all of the positions and velocities, store them in an array to which you append each new set of `r_soln` values as they are calculated. For more information on the `np.append()` method, click on this link: numpy.append.

```
# No damping, numerical integration, projectile motion over level ground:
#    define the integration function

def simple_euler(t_arr, r_arr):
    """Use the simple Euler method to integrate from initial state,
    r_arr = r_init, and return the x, vx, y, and vy values for each time step
    in t_arr"""
```

```
    # Accumulate arrays that hold the solutions.  Let each array accumulate in
    #    order to create x(t), vx(t), y(t), vy(t)

    # Create the empty arrays
    pos_x = np.array([])  # an empty array that will hold pos_x solutions
    vel_x = np.array([])  # this will hold vel_x solutions
    pos_y = np.array([])  # pos_y
    vel_y = np.array([])  # vel_y

    # The engine: initialize values, then step through with simple Euler method
    #    (uses a first-order Taylor series)
    for time_step in t_arr:  # This *for* loop sequentially takes on all the
                             #    elements of t_arr, first to last
        pos_x = np.append(pos_x, r_arr[0]) # Extend the array by tacking the
                                           #    latest x on to the end of it
                                           #    starting from r_arr
        vel_x = np.append(vel_x, r_arr[1])
        pos_y = np.append(pos_y, r_arr[2])
        vel_y = np.append(vel_y, r_arr[3])

        # In words:  a new 4-element array r_arr is created that takes the
        #    previous r_arr and adds a piece that is the first-order Taylor
        #    approximation of the extrapolated values at one time step later,
        #    using the derivatives from the deriv() function applied to each
        #    one of the elements.
        r_arr = r_arr + h_step*deriv(time_step, r_arr)  # update all 4 r_arr
                                                        #    elements by using
                                                        #    a first-order
                                                        #    Taylor series
                                                        #    expansion in time,
                                                        #    advanced by one
                                                        #    time step h_step.

        # Test to see when to cut off the solution by noting when the
        #    projectile hits the ground
        if (time_step > 0.0) and (r_arr[2] <= 0.0):
            break  # quit out of the loop

    # Pack all 4 values for each time point
    r_soln = np.array([pos_x, vel_x, pos_y, vel_y])
    return r_soln # return pos_x, vel_x, pos_y, vel_y
```

```
# No damping, numerical integration, projectile motion over level ground:
#    complete the integration

print("Initial values of the parameters: ", r_init)
r_soln = simple_euler(t_arr, r_init)
```

Exercise 2

Now that we have completed the Euler integration, it is time to examine the results. The following block of code will output interesting values and plot the trajectory. Modify it to include values of:

- initial speed,
- launch angle,
- time in the air, and
- range of the projectile.

Some of these values are directly available from the code in the cell above. Others you will have to make a simple calculation to find.

Additionally, add the code to graph the trajectory by plotting y vs. x.

➤ *Note:* In the plotting portion of the code below, you'll see an example of a long line of code that has been split across two lines. Python knows the line of code should continue because the outer set of parentheses hasn't closed yet. In cases where it doesn't make sense to use parentheses, a backslash can be inserted at the line break to indicate a continuation.

```
# < Exercise 2, Interpreting the results >

print("\nProjectile motion without air drag, over level ground: ")
print("Initial speed = {:5.2f} m/s".format(0)) ### FIX
print("Launch angle = {:5.2f} degrees".format(0)) ### FIX
print("Numerical range = {:5.2f} m".format(0)) # ### FIX



fig, ax = plt.subplots()
### DO THIS: Add the line of code to plot the trajectory here

ax.set_xlabel('Distance [m]')
ax.set_ylabel('Height [m]')
# An example of implicit line continuation. Python knows to keep on reading to
#    the next line because the set of parentheses hasn't been closed by the end
#    of the first line.
ax.set_title("Projectile motion no drag\nv_x ={:3.2f}, v_y ={:3.2f}".
             format(r_init[1], r_init[3]))
plt.show()

#  Compare with the 'range formula' over level ground
print('Analytical range = ', (2/grav)*vel_y0*vel_x0)
```

7.2.2 Accuracy of simple Euler integration

Congratulations, you have now just walked through your first numerical integration. The next step is to examine its accuracy, something that can be easily done by comparing the numerically calculated trajectory to the analytical solution.

Exercise 3

For comparison, set up the analytical (exact) solution evaluated at the same time points and plot it on the same axes as the numerical solution, along with an array representing the difference between the y values of the two solutions (called the *residuals*).

➤ *Hint:* You will want to make sure that your numerical and analytical solutions have the same number of points, evaluated at the same times, so that the y values of the numerical solution and the residuals can be plotted against the x values of the numerical solution. Remember that `len()` is the function that will return the length of an array and that you can select the first n elements in an array using square brackets and indexing:

```
arrayA[:n]
```

Although you began with n = 3,000 points, you will have truncated your solution when the y-value got back to 0. The following cell illustrates this difference.

```
# initial length of t_arr
print("Length of the entire time array: " ,len(t_arr))

print("Actual length of pos_y: ",len(r_soln[2]))  # length of pos_y
print("Corresponding length of pos_x: ",len(r_soln[2]))  # length of pos_x
```

```
# < Exercise 3, Finding and plotting residuals >

# Compute the analytical solution

# Plot analytical trajectory in blue, numerical trajectory in red.

# Then create an array

# resid = pos_y - pos_y_analyt

# that represents the difference between these two y-values as a function of x,
#    and plot 100* this difference vs. x in green.
```

Exercise 4*

You should observe that there is a small linear error term that arises because of when and where we evaluate the derivatives. How big was the error term by the end of the calculation (when the projectile returned to $y = 0$)? Be sure to include units in your answer.

***Accumulated error* =**

7.3 Improving the Simple Euler Integration Method: Euler Half-Step Integration

We can reduce the accumulated error in the simple Euler integration method if we choose a smarter place to evaluate our derivatives. The following blocks of code show how evaluating the derivatives *halfway between* the time steps greatly improves the accuracy.

7.3.1 Illustrated example

Consider the following simple quadratic function: the solution to a 1-D constant acceleration problem in which acceleration equals -1.0 m s^{-2} and the projectile starts from a height of 1.0 m with velocity 1.5 m s^{-1}.

```
t_arr = np.arange(0,1,0.01)
pos_y0 = 1.0 # initial height
vel_y0 = 1.5 # initial velocity
acc_y0 = -1.0 # constant acceleration
# analytical value of height over time
pos_y_analyt = pos_y0 + vel_y0*t_arr + 0.5*acc_y0*t_arr**2

fig, ax = plt.subplots()
ax.plot(t_arr, pos_y_analyt, '-b') # analytical solution
ax.set_xlabel("Time [s]")
ax.set_ylabel("Height [m]")
ax.set_title("Analytical Solution for a = -1.0 m/s^2")
plt.show()
```

Using the Derivative at the Start of the Time Step

Now imagine trying to approximate that trajectory by using the local slope at $t_0 = 0.0$ s to take a single coarse step between $t_0 = 0.0$ s and $t_1 = 0.5$ s. That's what a simple Euler approximation with a step size of $\delta_t = 0.5$ s would do, as demonstrated by the following blocks of code.

```
# The numerical solution: Defining the derivative function

def deriv_example(t_now, r_now):
    """Return the derivatives for a 1-D, constant accel. problem"""
    pos_y = r_now[0]
    vel_y = r_now[1]
    dy_dt = vel_y
    dvy_dt =  acc_y0
    return np.array([dy_dt, dvy_dt])
```

```
# The numerical solution: Simple Euler

# Defining the initial state
h_step = 0.5  # the step size, delta_t
```

```
r_init = np.array([pos_y0, vel_y0])  # array containing the initial values of y,
                                     #    vy, in order
pos_y = np.array([r_init[0]])  # y
vel_y = np.array([r_init[1]]) # vy
t_arr_trim = np.array([0, h_step]) # only look at 1 time step so there will be
                                   #    no "for" loop

# Integrating along a single time step
#    this is y1 = (delta_t)*(v0) + y0 and vy1 = (delta_t)*a
r_soln = r_init + h_step*deriv_example(t_arr, r_init)
pos_y = np.append(pos_y, r_soln[0]) # append the new y position
vel_y = np.append(vel_y, r_soln[1]) # append the new y velocity
```

```
# Comparing the analytical and numerical solutions

fig, ax = plt.subplots()
ax.plot(t_arr, pos_y_analyt, '-b')  # analytical solution
# Plot extrapolated position to get y1 at t = delta_t = 0.5 s
ax.plot(t_arr_trim, pos_y,'--r')
ax.plot(t_arr_trim, pos_y,'ko', ms = 5)
ax.set_xlabel("Time [s]")
ax.set_ylabel("Height [m]")
ax.set_title("Illustration of the Simple Euler Method for a = -1.0,\n"
             + "with 0.5 s time step")
ax.legend(["Analytical","Simple Euler"])  # creates a legend for plot. Legend
                                          #    keys are listed in order of
                                          #    plotting, and Python is usually
                                          #    smart enough to identify the
                                          #    corresponding linestyle and
                                          #    color.  Note that these are a
                                          #    list of strings within square
                                          #    brackets

# Evaluated at full h_step
pos_y_analyt_1step = pos_y0 + vel_y0*h_step + 0.5*acc_y0*h_step**2
# Creates a vertical line at t = 0.5 extending between the simple Euler and
#    analytical solution
ax.vlines(0.5, pos_y_analyt_1step, r_soln[0], color = 'purple')
ax.annotate("error", (0.51, 1.7))  # annotate adds text at the specified (x, y)
                                   #    location on the plot
plt.show()
print("Simple Euler error [meters]:  ",
      r_soln[0] - pos_y_analyt_1step )  # remember that r[0] contains the
                                        #    latest value of y
```

The black dot at the end of the red dashed line represents the result of taking a whole step of h = 0.5, using the slope that was evaluated back at the origin. The error in y (0.125 m) is the vertical distance between the black dot and the analytical solution at time h = 0.5 s.

This is a substantial error, especially for only one step. Where might be a better place to evaluate the derivative? *Halfway through the time step!*

Using the Derivative Halfway Through the Time Step

Instead of finding the derivatives at the start of the time step, let's take a half step ($h/2 = 0.25$ s), evaluate the derivatives there, and then use those derivatives for making the whole step. This is essentially a two-step process:

1. Use the derivative at t_0 to estimate the state of the system at $t_0 + h/2$. [Half step]
2. Use the derivatives at $t_0 + h/2$ to integrate from t_0 to $t_0 + h$. [Whole step]

You can envision this like a game of leapfrog, and that, in fact, is another name for this algorithm.

You might recall the mean-value theorem from calculus, where the slope of the function matches the secant line somewhere in between its end points. For quadratic functions, we will see that halfway is exactly the right place for us to evaluate the derivatives.

```
# The numerical solution: Improved Euler, or Euler Half-Step Method

# Defining the initial state
h_step = 0.5  # the step size, delta_t
r_init = np.array([pos_y0, vel_y0]) # the initial values of y, vy, in order

pos_y = np.array([r_init[0]]) # y
vel_y = np.array([r_init[1]]) # vy
# Only look at 1 time step so there will be no "for" loop
t_arr_trim = np.array([0, h_step])

# Integrating along a single time step
# 1) Evaluate the derivative at start of the time step; then multiply by length
#    of half time step to estimate the value at half time step
k1 = (h_step/2)*deriv_example(t_arr_trim, r_init)
# 2) Evaluate derivative at half time step; integrate from start of time step by
#    multiplying half-step derivative by full time step
r_soln = r_init + h_step*deriv_example(t_arr_trim, r_init + k1)

pos_y = np.append(pos_y, r_soln[0]) # append the new y position
vel_y = np.append(vel_y, r_soln[1]) # append the new y velocity
```

```
# Comparing the analytical and numerical solutions

# Evaluated on a fine time vector
pos_y_analyt = pos_y0 + vel_y0*t_arr + 0.5*acc_y0*t_arr**2

fig, ax = plt.subplots()
ax.plot(t_arr, pos_y_analyt, '-b')  # analytical solution
ax.plot(t_arr_trim, pos_y, '--r')  # plot extrapolated position to get y1 at
                                   #    t = delta_t = 0.5 s
```

```
ax.plot(t_arr_trim, pos_y, 'ko', ms = 5) # black dot

ax.annotate("Evaluate the derivative at t = 0.25 s", xy=(0.25, 1.34375),
            xytext=(0.05,1.65), arrowprops={"arrowstyle":"->", "color":"gray"})
ax.annotate("and then apply it to the full 0.5 s step here", xy=(0.5, 1.625),
            xytext=(0.6,1.45), arrowprops={"arrowstyle":"->", "color":"gray"})


ax.set_xlabel("Time [s]")
ax.set_ylabel("Height [m]")
ax.set_title("Illustration of the Improved Euler Method for a = -1.0")
ax.legend(["Analytical Soln", "Improved Euler"])
plt.show()
# Evaluated at the single half step
pos_y_analyt = pos_y0 + vel_y0*h_step + 0.5*acc_y0*h_step**2

print("Half-step Euler error [m]:  {:15.14e}".
      format(r_soln[0] - pos_y_analyt)) # the difference, or residual
```

7.3.2 Applying improved Euler to projectile motion

So now we know that there is a better time step at which to evaluate the derivatives before making the full step. By *substituting* these lines inside your integration loop we make the Euler half step, or improved Euler method:

```
# Evaluate the increment on the half step
k1 = (h_step/2)*deriv(t_arr, r_soln[0])
# This is the whole step after sniffing out the derivatives at the midpoint
r_soln[0] = r_soln[0]+ h_step*deriv(t_arr, r_soln[0] + k1)
```

```
# Improved Euler integration function
def improved_euler(t_arr, r_arr):
    """Integrate using the Improved Euler method"""
    # Create the empty arrays
    pos_x = np.array([]) # this is an empty array that will hold x solutions
    vel_x = np.array([]) # this will hold vx solutions
    pos_y = np.array([]) # y
    vel_y = np.array([]) # vy
    h_step = (t_arr[1] - t_arr[0]) # time step

    # The engine: initialize values, then step through
    #    (uses first-order Taylor series twice)
    for t_step in t_arr:  # this *for* loop sequentially takes on all the
                          #    elements of t_arr, first to last
        pos_x = np.append(pos_x, r_arr[0])  # extend the array by tacking the
                                            #    latest x-position on to the
                                            #    end of it, starting from r_arr
                                            #    which is originally a copy of
                                            #    r_init
```

```
        vel_x = np.append(vel_x, r_arr[1])
        pos_y = np.append(pos_y, r_arr[2])
        vel_y = np.append(vel_y, r_arr[3])

        # Here is the half-step Euler:
        # Evaluate the increment on the half step
        k1 = (h_step/2)*deriv(t_step, r_arr)
        # This is the whole step after sniffing out the derivatives at the
        #    midpoint
        r_arr = r_arr + h_step*deriv(t_step, r_arr + k1)
        # Test to see when to cut off the solution by noting when the
        #    projectile hits the ground
        if (t_step > 0.0) and (r_arr[2] <= 0.0):
            break  # break out of the loop
    r_soln = np.array([pos_x, vel_x, pos_y, vel_y])
    return r_soln
```

Exercise 5

Use this new `improved_euler()` function to analyze the projectile motion problem from **Exercise 1**. To do this, you'll want to copy the relevant blocks of code from above into the cell below. You may use as few or as many code cells as you like. Try to copy only what is necessary. For example, you will need to redefine the initial conditions and time arrays, because those were redefined (a.k.a. "clobbered") during the improved Euler example. However, your derivative function from earlier should still be usable.

- *Coding style tip:* Especially when developing new code, it is a good practice to define functions in separate cells. Then they can be edited and reevaluated separately from the rest of your code (or vice versa).

- *Hint: We suggest making sure your copied code produces results for simple Euler integration before modifying it for improved Euler.*

As before, compare your result with the analytical solution when the projectile hits the ground. What is the error now? If you've done everything correctly, you'll find that the error array will need to be multiplied by a *big* number to display in a useful fashion on the same plot as the trajectory.

Accumulated Error (Improved Euler) =

```
# < Exercise 5, Projectile motion w/o damping, Improved Euler;
#    Set initial conditions (ICs) & call function to integrate the solution >
```

```
# < Exercise 6, Projectile motion w/o damping, Improved Euler; Plot solution >
```

7.4 Check-out

Exercise 7

Briefly summarize in the cell below the ideas in this unit.

7.5 Challenge Problem

Exercise 8

You may wonder whether the increased accuracy you observed with the improved Euler method is simply a result of taking the derivatives twice as many times (twice for each time step). In a word, *no*!

Prove this fact to yourself by *increasing* the step size in your improved Euler method above by a factor of two. Then find the accumulated error and compare it to what you found for simple Euler. Describe your findings below.

Accumulated Error (Improved Euler, twice the step size) =

8 Projectile Motion with Drag

This unit builds upon the numerical integration code you wrote in Numerical Integration Applied to Projectile Motion. Now, though, you will include linear and quadratic velocity drag into your calculations of projectile motion. The inclusion of quadratic drag means that for the first time you will have the opportunity to model systems that cannot be analytically solved. Not only will completing this unit give you a greater understanding of the effect of drag on projectile motion, you will also solidify your understanding of $\hat{r}$, the unit vector of position, as it appears in the linear and quadratic drag equations.

These exercises are aligned with the content in Ch. 2.1–2.4 of *Classical Mechanics*. You will create plots analogous to Figures 2.7 and 2.8. References are also made to the situation described in Problem 2.1, which may be completed as a homework assignment.

This unit also includes a calculation of the work done by the drag force in **Exercises 6** and **7**. This content is covered in greater depth in Ch. 4 of *Classical Mechanics*, but you should have prior knowledge of it from an earlier physics class. These two exercises may also be skipped without affecting **Exercise 8**.

8.1 Objectives

In this unit, you will:

- include linear and quadratic drag into your equations of motion for a projectile (PHY);
- validate your code by comparing your numerical solution to the analytical solution in several special cases (PHY);
- calculate terminal velocity and compare it to a plot of velocity over time (PHY);
- calculate the mechanical energy loss due to drag (PHY).

```
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = (12,8) # make figures big for easier viewing
                                        #    (width, height in inches)
# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

8.2 Improved Euler Numerical Integration

Exercise 1

Into a new code cell, copy your improved Euler method code from Numerical Integration Applied to Projectile Motion to solve projectile motion without air resistance. As before, include the analytical solution for projectile motion without air resistance on your graph (there is no need to plot the residuals, though). Try not to include extraneous code. Before proceeding, make sure that your copy produces reasonable results by plotting the trajectory.

The first cell includes a template for the `deriv()` function. Unlike the `deriv()` function in the previous unit, this function definition allows for two additional *optional* parameters, `b_drag` (the coefficient for linear drag) and `c_drag` (the coefficient for quadratic drag), to be passed into it. Later in this unit, you will modify your `deriv()` function to include the effects of these parameters, but for now you may ignore them in both your function and your function call. In other words, the interior of your `deriv()` function should remain the same as in Numerical Integration Applied to Projectile Motion for now. In cases where there is no drag, you may continue to call the function as

```
deriv(t_now, r_now)
```

Similarly, the second cell includes a template for an `improved_euler()` function, which will compute the integration. As with the `deriv()` function, the *optional* parameters, `b_drag` and `c_drag`, are included so that you can use them later in the unit.

You might wonder why `deriv()` needs to know about `t_now`. It doesn't at the moment, but you could imagine a situation where the object is acted upon by an external time-varying force; perhaps your projectile is *rocket powered*. For that future reason we include `t_now` in the calling signature.

```
# < Exercise 1, 2-D Projectile Motion, deriv() function >

# The deriv() function should take as parameters the present time t_now and a
#    4-element array defining the state (r_arr = np.array([x, vx, y, vy])).
#    It should return a new 4-element array defining the derivatives at time t
#    (r = np.array([dx/dt, dvx/dt, dy/dt, dvy/dt])).

grav = 9.8 # [m s^-2], gravitational acceleration near the surface of Earth
mass = 0.5 # [kg], the mass of the projectile

def deriv(t_now, r_now, b_drag = 0, c_drag = 0):
    """Return the time derivatives at time t_now of x, v_x, y, v_y.
    b_drag is the coefficient for linear drag and c_drag is the coefficient
    for quadratic drag."""

    ### UNPACK the variables stored in array r_arr

    ### CALCULATE the derivatives for x, v_x, y, and v_y

    dr_dt = np.array([0, 0, 0, 0]) ### FIX


    return  dr_dt
```

```
# < Exercise 1, 2-D Projectile Motion, Improved Euler Integration function >

def improved_euler(t_arr, r_arr, b_drag = 0, c_drag = 0):
    """Integrate using the Improved Euler method
    b_drag is the coefficient for linear drag, and c_drag is the coefficient
    for quadratic drag."""

    # Create the empty arrays
    pos_x = np.array([]) # this is an empty array that will hold x solutions
    vel_x = np.array([]) # this will hold vx solutions
    pos_y = np.array([]) # y
    vel_y = np.array([]) # vy
    h_step = (t_arr[1] - t_arr[0]) # time step

    ### ADD integration code below


    r_soln = np.array([pos_x, vel_x, pos_y, vel_y])

    return r_soln # return the 4 x n element solution (probably a little
                  #    shorter than n)
```

```
# < Exercise 1, 2-D Projectile Motion, initial conditions and integration >
```

```
# < Exercise 1, 2-D Projectile Motion, analytical calculation >
```

```
# < Exercise 1, 2-D Projectile Motion, plot of projectile to test code >
```

8.3 Projectile Motion with Linear Drag

In order to include a drag force, you will need to adjust your `deriv()` function to include velocity-dependent damping coefficients of the form:

$$\vec{F}_{drag} = -b\vec{v} - cv^2\hat{v}$$

You will start with the simpler and analytically solvable case of just linear drag (i.e., $c = 0$). In later exercises, you will add a non-zero quadratic drag component.

Exercise 2

Redo your improved Euler projectile problem with the same initial conditions and time steps as before to show the trajectory and find the range (over level ground) for $b = 0.05$ kg/s and $c = 0.0$. Assume that the projectile has mass $m = 0.5$ kg, as defined above in the second code cell. Please *do not* make a new copy of either `deriv()` or `improved_euler()`. Instead, make the necessary modifications to your work above to include the effect of linear drag.

In your final plot, include the analytical solution for the motion *without air resistance*. This will allow you to compare the effect of linear drag on the trajectory. There is no need to recalculate the analytic solution from **Exercise 1**.

➤ *Hint:* Make sure that `b_drag` is passed to the `deriv()` function from inside your integration function (`improved_euler`). Also, make sure that you are passing your integration function a fresh copy of `r_init`.

Drag-restricted range (for b = 0.05 kg/s, c = 0.0, m = 0.5 kg) =
I get approximately 64 m. Show your result to 4 significant figures.

How is the range and height of the projectile affected by the inclusion of linear drag? Explain ***here***.

```
# < Exercise 2, 2-D projectile motion with linear drag, using improved Euler >

# Your code should include:
#    Setting the initial conditions
#    Calling the deriv function with a parameter for linear drag
#    Calculating the analytic solution for the motion without linear drag
#    A determination of the range of the trajectory
#    A plot of the trajectory with and without linear drag
```

8.4 Quadratic Drag

8.4.1 One-dimensional motion

You will start your exploration of quadratic drag by examining situations in which the linear drag term is negligible and the motion is entirely in one direction (e.g., something dropped or thrown vertically). Choosing to work in one dimension will allow easy comparison with the analytically derived terminal velocity, permitting you to validate your code in the simpler case before adding the complexity of two-dimensional motion.

Exercise 3

Now modify your `deriv()` function to include quadratic drag. In your integration function call, let $b = 0$ and $c = 0.002$ kg/m; keep $m = 0.5$ kg. Because it is only possible to find an analytical solution with quadratic drag for one-dimensional motion (why is that?), start by using the following initial conditions (equivalent to a rock dropped vertically from a cliff): position ($t = 0$ s) = (0 m, 500 m) and velocity ($t = 0$ s) = (0 m/s, 0 m/s). Because the motion will be in one dimension, plot y vs. t rather than vs. x. Once again, include the analytical solution for the motion with no air resistance (this time, you will have to recalculate the analytic trajectory because the initial conditions have changed).

- *Hint 1:* Make sure that the quadratic drag force in your `deriv()` function is opposite the direction of motion.
- *Hint 2:* Remember that your `t_arr` array will likely not have the same number of elements as your `r_soln` array. How can you select the portion of `t_arr` that is relevant?
- *Hint 3:* Only copy the minimal amount of code into the cell below. For instance, there is no need to make a copy of your `deriv()` or `improved_euler()` functions.

```
# < Exercise 3 Solution for dropped rock, 1-D motion with quadratic drag,
#    using improved Euler >
```

Exercise 4

What terminal velocity would you expect the 0.5 kg rock to reach when the quadratic drag coefficient is $c_{\text{drag}} = 0.002$ kg/m? Provide an analytical calculation showing your work:

Expected terminal velocity:

Plot y velocity vs. time and include the analytical solution for the velocity without drag. Estimate the terminal velocity from the plot and compare it to the last value of `vel_y`.

- *Hint:* You might want to add a grid to your plot, via `ax.grid()`. If you would like to plot a horizontal line, you can use axhline. The syntax to plot at a particular y-value (for example, your analytically calculated terminal velocity) on an already-defined Matplotlib axis object, `ax`, is

  ```
  ax.axhline(y = yvalue)
  ```

  where `yvalue` is the desired value. If this parameter is not included, the line will default to $y = 0$. Similarly, `ax.axvline(x = xvalue)` will give a vertical line at some `xvalue`.

Is your plot consistent with the terminal velocity you calculated? Explain ***here***.

```
# < Exercise 4, v_y vs. t for dropped rock, 1-D motion with quadratic drag,
#    using improved Euler >
```

Next, you will test your quadratic-drag code with a different set of initial conditions that will allow for the rock to have both positive and negative velocity. *Make sure the direction of your drag force is always opposite to the direction of motion.* As before, try to determine the minimal code you will need to copy into a new cell.

Exercise 5

Calculate and plot y vs. t and v_y vs. t for these initial conditions: position ($t = 0$ s) = (0 m, 0 m) and velocity ($t = 0$ s) = (0 m/s, 28 m/s), and find how long the rock remains in the

air. Keep all other parameters the same as in the previous two exercises. This is the situation where the rock is thrown vertically up into the air. Once again, include the analytical solution for the motion with no air resistance in your plots of y vs. t and v_y vs. t.

Time in air (for b = 0, c = 0.002, m = 0.5 kg) =
I get approximately 5 s. Show your result to 3 significant figures.

```
# < Exercise 5, Up and down projectile motion with quadratic drag, using
#    improved Euler >

### COMPUTE the solution for the trajectory subject to quadratic drag for the
#    given initial conditions

### PLOT y vs. t

### PLOT v_y vs. t
```

8.4.2 Energy loss in one-dimensional motion

Exercise 6*

Drag forces are nonconservative forces so work done by them results in loss of mechanical energy from the system. To see this, first calculate and graph the kinetic energy, potential energy, and total energy vs. time (all on one plot). Then find the amount of mechanical energy (in joules) lost during the rock's entire trajectory.

Mechanical energy loss =
I get approximately 48 J. Show your result to 3 significant figures.

Notice that at one point in the trajectory, the rate of energy loss goes to zero. Explain ***here*** why that makes sense.

```
# < Exercise 6, Mechanical energy loss >
```

Exercise 7*

Because work done on a system is equal to the change in energy, it is possible to equate the change in mechanical energy of the rock to the work done by the drag force. For this exercise, estimate the work done by the drag force across each time step and sum to find the total work done. Essentially, you will be finding

$$W = \int \vec{F} \cdot d\vec{r}$$

using the simple rectangle method (Riemann sum) for estimating the integral. Verify that the total work you calculate is close to the change in energy.

- *Hint 1:* The method np.sum() will calculate the sum over an array. This is cleaner and easier than writing a loop to sum over the elements, as you did in Taylor Series with Loops and Functions.
- *Hint 2:* You'll need to find the displacement across each time step. One way to do this is to subtract `y[:-1]` from `y[1:]`. Can you see why? Remember that referencing element -1 refers to the last element in the array. Alternatively, you can use the np.diff() function.
- *Hint 3:* Be aware of the negative/positive signs of your displacement.

***Work done by drag force* =**

```
# < Exercise 7, Work done by drag force >
```

8.5 Linear and Quadratic Drag

Now for two-dimensional motion with both linear and quadratic drag. *This is a trajectory that cannot be found analytically and so must be calculated numerically!*

Exercise 8

Looking at Example 2.1 in Taylor's *Classical Mechanics* (p. 45), match the coefficients for a baseball of diameter 7 cm, and mass $m = 145$ g and again plot the trajectory and find the range. In order to see clearly the effect of the drag, include the analytical solution for the case of zero-drag on your plot of the trajectory.

- *Hint:* If your results seem suspect, check that the quadratic drag term in your `deriv()` function is correctly accounting for both the x and y components of velocity. You may want to consider how you could write the x and y components of the quadratic drag term using $\hat{v}_x$ and $\hat{v}_y$ and do that in your definitions of $\mathrm{d}v_x/\mathrm{d}t$ and $\mathrm{d}v_y/\mathrm{d}t$.

***Drag-restricted range for the baseball* =**
I get approximately 55 m. Show your result to 4 significant figures.

```
# < Exercise 8, Solution for baseball from Taylor, p. 45. Projectile motion,
#    improved Euler, with drag >
```

Exercise 9*

Try alternately setting both b and c to zero.

Which has a greater effect on the trajectory of the baseball: the linear or quadratic term? Why do you think this is?

8.6 Check-out

Exercise 10

Briefly summarize in the cell below the ideas of this unit.

8.7 Challenge Problem

Repeat the work-energy analysis you did in **Exercises 6** and **7** for the baseball. Note that you'll have to calculate the change in position for two-dimensional motion and that you'll have to include both forms of drag.

9 Launching a Rocket

In the previous exercises, we showed how numerical integration can be implemented with a simple Euler or improved Euler method. More sophisticated algorithms achieve higher accuracy by using more terms in the Taylor series approximation while taking the next step. (For example, one such easily coded algorithm is the fourth-order Runge-Kutta.) This exercise introduces the higher-accuracy ordinary differential equation solver included in Python, `solve_ivp()`. You will apply it to the problem of a rocket launching in a gravitational field. Specifically, you will model the launch of the Space Shuttle, a two-stage rocket, with the simplifying assumption that the trajectory is vertical, and the acceleration due to gravity remains constant.

While this problem can be solved analytically, it is a good illustration of how sets of first-order differential equations can be rewritten to be solved with a computer.

Exercises 9 and **10** ask students to compare the solutions from `solve_ivp()` to those generated using an improved Euler method similar to that used in Numerical Integration Applied to Projectile Motion and Projectile Motion with Drag. These exercises may be omitted to save time without affecting the rest of the unit.

9.1 Objectives

In this unit, you will

- learn how to use `solve_ivp()` to solve ordinary differential equations (PRO);
- analytically and computationally solve the motion of a rocket in an external gravitational field (PHY);
- model a two-stage rocket launch (PHY).

```
#%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = (12,8) # make figures big for easier viewing
                                        #    (width, height in inches)
# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

9.2 Motion of Rocket without Gravity

In order to familiarize yourself with numerically solving differential equations, you will start with the simplest case: a rocket experiencing no external forces.

In the next section, you will use the Python numerical solver, `solve_ivp()`, to find the velocity as a function of mass. Before that, though, you will set up the problem and graph the analytical solution.

Exercise 1

In the code block below, please define variables for the initial mass of the rocket (`mass_0`), final mass of the rocket that remains after the payload is exhausted (`mass_final`), the amount of time during which the rocket is expelling mass (`burn_time`), exhaust speed (`vel_ex`), and the initial velocity of the rocket (`vel_0`). Use the data on the Space Shuttle from Problem 3.7 of *Classical Mechanics*.

```
# < Exercise 1, Defining the variables >

### FILL IN the information below:

mass_0 = 0 # [kg]; Initial mass of the Space Shuttle
mass_final = 0 # [kg]; Final mass after fuel in the Space Shuttle Solid Rocket
               #    Booster has been expended
mass_propel = mass_0 - mass_final # [kg]; Mass of propellant contained in
                                  #    Space Shuttle Solid Rocket Booster
burn_time = 0 # [s]; Amount of time before propellant is exhausted from the
              #    first-stage boosters.
vel_ex = 0 # [m/s]; Exhaust velocity
vel_0 = 0 # [m/s]; Initial velocity
```

Exercise 2

Using the analytical solution given in Equation 3.8 of *Classical Mechanics*, graph the velocity as a function of mass. For the independent variable, `mass`, define an array of at least 50 elements that runs from the initial mass to the final mass.

```
# < Exercise 2, Plots >

# Array of rocket masses
mass = 0 ### FIX

# Analytically calculated velocities
vel = mass ### FIX

### INCLUDE code for plotting velocity versus mass
```

Exercise 3

Sometimes it is more natural to think in terms of velocity as a function of time, rather than mass of the rocket. This conversion can be done *if* the change in mass as a function of time is known. Assume here that mass is expelled at a constant rate, $\dot{m} = -k$, and that the entire process takes place over the "burn time" defined in **Exercise 1**. Therefore,

$$m(t) = m_0 - kt.$$

In the cell below, calculate k (`burn_rate`). Then rearrange the above equation for $m(t)$ to find $t(m)$, and use that relation to plot rocket velocity as a function of time.

```
# < Exercise 3, Convert mass to time >

### FIX the two definitions below
burn_rate = 0 # [kg/s]; Rate at which mass is expelled
time_arr = mass # [s]; Array of times

### INCLUDE code for plotting velocity versus time
```

Exercise 4

Perhaps just as interesting as velocity as a function of time is the position as a function of time. To find this, analytically integrate Equation 3.8 from *Classical Mechanics* for velocity to find the position as a function of time. Use your equation for $m(t)$ assuming a constant burn rate over 2 minutes. Then plot position versus time. You should find that the height already exceeds 100 km after 2 minutes. Assume that the initial height is zero.

Height after 2 minutes =

```
# < Exercise 4, Position vs. time >

### INCLUDE code to plot the position versus time.
```

9.3 Introducing solve_ivp() for a Rocket

The differential equation solver, solve_ivp (short for "solve initial value problem"), is part of the SciPy package, which must be loaded at run time using the command `import`.

Previously, you solved differential equations by defining an "r-array" that consisted of the values of each variable to be integrated over (e.g., generalized positions and velocities) at a particular time. You then used a function to return the derivatives for each of those variables, determined the value of those variables at the subsequent time step, and appended the r-array for that subsequent time step to your solution. If you had n variables and N time steps, the solution would be a two-dimensional nxN array.

With `solve_ivp()`, though, the solver determines the array of time points (using a user-defined time span as one of its arguments) and outputs an array of r-arrays (really, a two-dimensional nxN array), eliminating the `for` loop we used in the previous units. Thus, you will no longer be building up the solution by repeatedly appending to an array; instead, the

solve_ivp solver will output the entire solution as a two-dimensional array, along with the array of times over which it was evaluated. The output of `solve_ivp()` is in the form of a programming "object": if the output is `r_soln`, the times will be accessible as `r_soln.t` and the two-dimensional array as `r_soln.y`.

The exercises below will walk you through how to find and interpret the numerical solution to the rocket problem generated by `solve_ivp()`.

For a more comprehensive and general introduction to `solve_ivp()`, refer to Appendix 1: Using `solve_ivp()` for Numerical Integration.

```
# Import the differential equation integrator
#    To reduce demand on memory, we import only the subset of the SciPy
#    package that contains solve_ivp
from scipy.integrate import solve_ivp
```

9.3.1 Defining the `deriv()` function

To use the `solve_ivp()` solver, the user must provide the system of first-order ordinary differential equations in the form of a function (what will typically be called `deriv()`) that returns the derivative for each variable. The `deriv()` function takes as its parameters a single time, `t_now`, an array of the variables, `r_arr`, defining the state of the system at `t_now`, and any other constants of the system necessary for solving the derivatives. It then returns an array containing the derivatives of each of those variables in the same order as they were passed to the function.

Exercise 5

In this case, the variables to be integrated are *mass*, *position*, and *velocity.* Modify the code below to create a function `deriv()` that will return the derivatives of each of these variables at a particular time, `t_now`. The variables *mass*, *position*, and *velocity* are all passed to the function as elements of an r-array, `r_now`, and the return of the function is an array of the derivatives of those variables in the same order.

In the base code below, the `deriv()` function also takes *burn_rate* and *vel_ex* as optional parameters. This will allow you to compute solutions for situations with different values of *burn_rate* and *vel_ex*. Note that if *burn_rate* and *vel_ex* are not passed to the `deriv()` function, they will default to zero (and the rocket will not move).

```
# < Exercise 5, Computational solution for rocket, the deriv function >

# Deriv function for use by solve_ivp

# As you did when writing your own integrator, create a function to evaluate
#    the derivatives
def deriv(t_now, r_now, burn_rate = 0, vel_ex = 0):
    """ Supplies the derivatives of mass, position, and velocity to be used
    by the differential equation integrator, solve_ivp. The optional
    parameters burn_rate, and vel_ex are set to zero by default. """
```

```
    # Unpack the r-array for the mass, position, and velocity
    mass, pos, vel = r_now # Notice that this syntax is equivalent to
                           #    separately defining
                           #    mass = r_now[0],
                           #    pos = r_now[1]
                           #    vel = r_now[2] but is shorter and cleaner

    dmass_dt = 0  ### REPLACE 0 with the appropriate expression
    dpos_dt = 0  ### REPLACE 0 with the appropriate expression
                 #    Hint:  This is probably simpler than you think.
    dvel_dt = 0  ### REPLACE 0 with the appropriate expression


    return np.array([dmass_dt, dpos_dt, dvel_dt])
```

9.3.2 Finding the solution with `solve_ivp()`

Now that the derivatives have been defined, the user must define the initial conditions (the eponymous initial values in `solve_ivp()`). This is done by defining an array of the initial states of the variables to be integrated, in the same order as assumed for the r-array in `deriv()`.

Next, the integration time must be defined. Unlike other solvers, `solve_ivp()` is special in that it can determine the time steps over which to find the solution to minimize the computational time (this is an example of an *adaptive* solver). Remarkably, these time steps typically have uneven spacing. While it is possible to ask `solve_ivp()` to use a particular array of user-defined time steps, by default the user only passes the range of times. Generally, the user will define a two-element list stating the minimum and maximum time. *(Alternatively, the range can be passed in the form of a tuple rather than a list. Tuples are like lists except that they are defined with parentheses rather than square brackets and are immutable, i.e., unchangeable. Using tuples rather than lists is slightly more memory efficient, but feel free to continue to use lists for the sake of simplicity.)*

Finally, the name of the function containing the derivatives, the time span for the solution, and the initial conditions are all passed to `solve_ivp()` along with any optional parameters for the `deriv()` function.

Exercise 6

Fill in the requested information in the code cell below to find the computational solution. Use the same parameters as you did in **Exercise 1** and complete the integration for the entire burn time.

```
# < Exercise 6, Computational solution for rocket, calling solve_ivp >

# Set the initial conditions
r_init = np.array([0, 0, 0]) ### FIX # initial mass, position, and velocity
                                     #    These must be in the same order as
                                     #    accessed in deriv()
```

```
# Define the range of times over which a solution is desired
t_span = [0.0, 0.0] ### FIX # List of minimum and maximum times

# Call solve_ivp
#    solve_ivp takes the name of the function of derivatives (deriv), the time
#    span for the solutions (t_span), an array of initial conditions (r_init),
#    and optional parameters to be passed to the deriv function
r_soln = solve_ivp(deriv, t_span, r_init, args = (burn_rate, vel_ex))
```

9.3.3 Accessing the solution from `solve_ivp()`

The return from `solve_ivp()` is a programming object that contains lots of information. Soon you will see an example of how to access the data in it. Initially, though, a complete description of the contents of `r_soln` may be seen by using the `help()` function:

```
help(r_soln)
```

The `help()` function in Python returns information about variables, functions, classes, and so on, as illustrated in the cells below. It is a very useful debugging tool and a quick way to learn about functions and classes from Python libraries.

```
# Print out information about the function, solve_ivp()
help(solve_ivp)

# Once you have run this cell, you may wish to comment out the above commands
#    to avoid cluttering your output
```

```
# Print out information about the variable r_soln returned by solve_ivp()
help(r_soln)

# Once you have run this cell, you may wish to comment out the above commands
#    to avoid cluttering your output
```

The cell below shows you the information contained in this particular instance of `r_soln`, including how to access specific quantities from it. Most important is the success of the integration (accessible as `r_soln.success`), the time steps (accessible as `r_soln.t`), and the two-dimensional array of solutions (accessible through `r_soln.y`). The first row of `r_soln.y` (accessible as `r_soln.y[0]` or `r_soln.y[0,:]`) gives the values for the first variable listed in the `r_init` and in the variable `r_now` used by `deriv()`. The second row gives the values for the second variable and so on.

If you look closely at the values for the time step, you will notice that they initially increase exponentially.

```
print(r_soln) # You may want to comment this line out after
              #    successfully running the cell once to avoid cluttering your
              #    output
print("\nWas the solver successful? ", r_soln.success)
print("\nArray of time steps: ", r_soln.t)
print("Two-dimensional array of solutions: ", r_soln.y)
print("Array of masses: ", r_soln.y[0])
print("Array of positions: ", r_soln.y[1])
print("Array of velocities: ", r_soln.y[2])
```

Exercise 7

In order to practice using the output from `solve_ivp()` and to see the accuracy, plot the mass, velocity, and position of the rocket as a function of time. Include both your numerical and analytical solutions for each of these. Finally, determine the accuracy of the solution by finding the difference between the analytical and computational position at the end of the burn time.

Analytical Position after 120 s - Numerical Position after 120 s =

```
# < Exercise 7, Computational solution for rocket, plots >

### PLOT the mass vs. time

### PLOT the position vs. time

### PLOT the velocity vs. time
```

You should have found that the difference in final position for the analytical and computational solutions is less than 0.0001%! This is a very high level of accuracy.

However, you also probably noticed that the time steps in the computational solution are very coarse, resulting in a choppy-looking plot. What happened is that `solve_ivp()` did not need to use small time steps to find an accurate final step. For this exercise, though, we would like to know the solution with much finer time spacing. To request that `solve_ivp()` use no time step spacing greater than 0.1 seconds, set `max_step` in `solve_ivp()` to 0.1.

Exercise 8

Copy the code from the previous two exercises to again find and plot the computational solution. This time, though, replace the call to `solve_ivp()` with the following line of code:

```
r_soln = solve_ivp(deriv, t_span, r_init, args = (burn_rate, vel_ex), max_step = 0.1)
```

to define a maximum time step size.

```
# < Exercise 8, Computational solution with smaller step sizes >

### INCLUDE code to find and plot the solution
```

9.3.4 Comparison between `solve_ivp()` and improved Euler

It is clear that `solve_ivp()` can produce accurate results. However, you have also previously had success using your own improved Euler integrator. How do these two methods compare in terms of both accuracy and efficiency (how long it takes to run)? While the answer will vary depending on the problem, the following exercise provides a simple comparison for the case of the rocket.

Exercise 9*

The following cell contains the template for an improved Euler integration of the rocket in zero gravity. It has been modified from your work in Numerical ODE & Projectile Motion and Projectile Motion with Drag to enable the integration of mass, position, and velocity, as well as to include variables for the burn rate and the exhaust speed. Basing your edits on your work in the previous units, add the integration code to the `improved_euler` function adjusted to fit this problem.

➤ *Hint: You do not need to (and should not) define a new derivative function. Simply call* `deriv()`*, as defined previously in this unit.*

```
# < Exercise 9, Rocket w/o gravity, Improved Euler Integration function >

def improved_euler(t_arr, r_arr, burn_rate = 0, vel_ex = 0):
    """Integrate a rocket in zero gravity using the Improved Euler method.
    burn_rate is the rate at which the mass is expelled, and vel_ex is the
    velocity at which it is expelled."""

    # Create the empty arrays
    mass = np.array([])  # an empty array that will hold the mass solutions
    pos = np.array([])   # positions
    vel = np.array([])   # velocities
    h_step = (t_arr[1] - t_arr[0]) # time step

    ### ADD integration code below


    r_soln = np.array([mass, pos, vel])

    return r_soln # return the solution
```

The next three cells import the `timeit` library and then run the `solve_ivp()` and `improved_euler()` integration for a fixed array of times (`time_arr`, previously defined to be used in the analytical calculation). To use a fixed array of times with `solve_ivp()`, set the optional parameter `t_eval` in `solve_ivp()` to the desired array, in this case `time_arr`.

```
import timeit
```

```
%%timeit
r_soln_imp_euler = improved_euler(time_arr, r_init, burn_rate, vel_ex)
```

```
%%timeit
r_soln_solve_ivp = solve_ivp(deriv, t_span, r_init, t_eval = time_arr,
                             args = (burn_rate, vel_ex))
```

Exercise 10*

Plot the positions versus time for the rocket, as calculated using `solve_ivp()`, your improved Euler integrator, and analytically. Then calculate the difference between the position of the rocket after 120 seconds for each computational solution and the analytical solution.

➤ *Hint:* Because you will be plotting both the solution from your `improved_euler()` function and `solve_ivp()`, you should make sure to give those solutions different names.

```
# < Exercise 10, Rocket w/o gravity, Integration method comparison >

# Using Improved Euler to calculate trajectory
r_soln_imp_euler = improved_euler(time_arr, r_init, burn_rate, vel_ex)
# Using solve_ivp to calculate trajectory
r_soln_solve_ivp = solve_ivp(deriv, t_span, r_init, t_eval = time_arr,
                             args = (burn_rate, vel_ex))

### PLOT the position vs. time
```

Comparing the results of `solve_ivp()` and your improved Euler integration when calculated over the same set of time steps reveals that `solve_ivp()` produces much more accurate results. While improved Euler was able to produce incredibly accurate results for a projectile subject to constant acceleration, its accuracy decreases as the complexity of the problem increases. By using a higher-order Taylor series approximation as the basis for the integrator `solve_ivp()` is able to achieve much higher accuracy.

There are a number of additional advantages to using `solve_ivp()`. For example, the adaptive time stepping means that `solve_ivp()` can appropriately reduce the number of time steps during slowly changing portions of the integration to save time without significantly reducing the accuracy. Additionally, you probably noticed that it was much easier to use `solve_ivp()` than to adapt your improved Euler integration method to a new problem.

9.4 Motion of a Rocket in a Constant Gravitational Field

For the remainder of this unit, you will model the launch of the Space Shuttle to the International Space Station (ISS) located in Low Earth Orbit.

Suppose the rocket is taking off vertically from the surface of Earth. Because Earth's gravitational field is roughly constant in Low Earth Orbit, this situation can be approximated as

a rocket subject to a constant external force along the same directional line as the velocity. (OK, we acknowledge that you ought to launch it at some angle so as to match the angular momentum required to keep it in a circular orbit, but that is a further refinement. Here we're illustrating why you want to use a *staged* rocket.)

9.4.1 Analytical solution

Exercise 11

On a separate sheet of paper, show that the equation of motion for this circumstance can be written as

$$m\dot{v} = -\dot{m}v_{ex} - mg \tag{9.1}$$

Then rewrite the equation assuming a constant rate of mass ejection such that $\dot{m} = -k$ and $m = m_0 - kt$. Use separation of variables to solve for v as a function of t by rearranging the equation such that all terms involving v are on the left and all terms involving t are on the right and integrating.

This problem is equivalent to part *b* of Problem 3.11 from *Classical Mechanics*.

Exercise 12*

Now integrate your above equation for $v(t)$ (again assume the initial height is zero) to find that the rocket's height as a function of t is

$$y(t) = v_{ex}t - \frac{1}{2}gt^2 - \frac{mv_{ex}}{k}\ln\left(\frac{m_0}{m}\right) \tag{9.2}$$

You should be able to repeat much of your derivation from **Exercise 4**.

This problem is equivalent to Problem 3.13 from *Classical Mechanics*.

Exercise 13*

Use the equations you derived above to find the velocity and position of the rocket released vertically in a uniform gravitational field as a function of time. Continue to assume the same parameters for the rocket's initial mass, rate of mass expulsion, and velocity of mass expulsion. Plot the position and velocity as a function of time for the first 120 seconds of the launch.

How does the height of the rocket after 120 seconds change when gravity is included?

```
# < Exercise 13, Analytical solution for rocket >

grav = 9.8 # [m/s^2]; gravitational acceleration near the surface of Earth

### PLOT the position vs. time

### PLOT the velocity vs. time
```

9.4.2 Computational solution

Exercise 14

Now that you have found the analytical solution, find the computational solution. To do this, you will need to define a new `deriv()` function (I suggest renaming it `deriv_g()`) with the appropriate modification to the calculation of the derivatives. Include plots for the mass, position, and velocity of the rocket as a function of time for the first 120 seconds of the launch. Overplot the analytical solution for the velocity and (provided you completed **Exercises 12** and **13**) the position on top of the computational solution. As in **Exercise 8**, tell `solve_ivp()` to use a maximum time step spacing of 0.1 seconds.

```
# < Exercise 14, Computational solution for rocket in a gravitational field,
#    derivative function >

### DEFINE a new derivative function that includes the effect of gravity
```

```
# < Exercise 14, Computational solution for rocket in a gravitational field >

### CALCULATE the trajectory of the rocket using solve_ivp()

### PLOT mass, position, and velocity versus time
```

9.5 Launching to the International Space Station

The launch of the Space Shuttle to the International Space Station actually takes place over two stages. During the first stage (what you have modeled thus far), the two solid rocket boosters provide the majority of the thrust. After 120 seconds, the empty tanks are jettisoned (and recovered via parachute), further reducing the rocket's mass. Over the following 6.5 minutes, the main engine continues to burn fuel from the external tank.

Exercise 15

Computationally model the two-stage launch of the Space Shuttle using the following approximate parameters. As in **Exercise 8,** tell `solve_ivp()` to use a maximum time step of 0.1 seconds. When setting the initial conditions for the second stage, *use the final conditions from the first stage, but be sure to reduce the mass of the rocket by the mass of the two solid rocket boosters.* For defining the time span of the integration, `t_span`, for the second stage, it is easiest to set the initial time equal to the final time from the first stage. For example

```
t_span = [0.0, 120] # Time span for integration of the first stage
t_span_2 = [120, 120 + 6.5*60] # Time span for integration of the second stage
```

- Stage 1: Let the initial mass of the rocket be 2×10^6 kg. As in the previous exercises, assume a 120 s burn time and an expulsion of 1×10^6 kg of fuel.
- Stage 2: After the two solid rocket boosters (each with mass 87,500 kg) are emptied, they are jettisoned (separated via explosive bolts). A further 733,500 kg of fuel is then expelled from the external tank across the next 6.5 minutes. This mass is expelled at a considerably

higher velocity of 4,447 m/s because of the higher burning speed of hydrogen/oxygen compared with the solid-fuel boosters.

For ease of calculation, some approximations have been made. For instance, you will assume a constant burn rate during each of the two stages. In reality, though, the burn rate is initially higher in the second phase than stated and then reduced part way through to prevent the rocket from exceeding 29.4 m/s^2 (3 g), which would damage the structure and needlessly compress the occupants. Additionally, we are modeling a radial launch, when in reality the trajectory of the Space Shuttle curved soon after leaving the launchpad to be roughly tangential to the surface of the Earth.

➤ *Hint:* When plotting the trajectories during the first and second stage, there is no need to combine the arrays. You can simply plot the trajectory from each stage individually, perhaps in different colors.

```
# < Exercise 15, Two-stage rocket launch in a gravitational field, finding the
#    solution >

### CALCULATE the trajectory across the two stages
```

```
# < Exercise 15, Two-stage rocket launch in a gravitational field, Plots >

### PLOT the mass vs. time

### PLOT the position vs. time

### PLOT the velocity vs. time
```

Exercise 16*

Verify that the final velocity of the rocket is *similar* to the orbital velocity at the radius of the International Space Station. To do this, you will need to calculate the circular velocity for an object in a gravitational orbit around Earth 140 km above the surface of Earth. You should find that after Stage 2 in your model is complete, the Space Shuttle is traveling at about 90% speed of the International Space Station.

➤ *Hint 1:* Remember that the equation for Newtonian gravity is

$$\vec{F}_g = -G\frac{Mm}{d^2}\hat{d}$$

where G is the gravitational constant, M and m are the masses of the two objects, and $\hat{d}$ is the vector between the two centers of the masses.

➤ *Hint 2:* You will find the final velocity you calculate to be somewhat lower than that of the International Space Station. This is because of the approximations alluded to in **Exercise 15** and because we are neglecting the additional velocity boost the rocket receives from the rotational motion of the Earth as the rocket enters a near-equatorial orbit. A somewhat higher-energy launch is used for spy satellites, which tend to be launched into *polar orbits*. It's considered *bad form* to launch into a *retrograde orbit* (east-to-west) because potential head-on collisions are spectacularly un-neighborly.

```
# < Exercise 16, Calculating the orbital velocity of the ISS >

### INCLUDE your calculation here
```

Exercise 17*

Would the Space Shuttle have been able to reach the orbital velocity of the International Space Station if the two solid rocket booster tanks had not been jettisoned before Stage 2 of the burn? Repeat your calculations from **Exercise 15** with this adjustment to find the effect of jettisoning the solid rocket booster tanks.

```
# < Exercise 17, Two-stage rocket launch in a gravitational field, without
#    jettisoning first stage >

### CALCULATE the trajectory across the two stages
```

9.6 Check-out

Exercise 18

Briefly summarize the ideas in this unit.

9.7 Challenge Problem

Most of the flight of the Space Shuttle is at very high elevations where air resistance is negligible. To see the effect of air resistance, instead model the flight of a 0.2 m diameter water rocket. Create a new derivative function that includes the effect of quadratic drag and assume that $\gamma = 0.25\ \mathrm{N\ s^2/m^4}$.

Let the initial mass of the water rocket be 0.7 kg and the final mass be 0.1 kg. Assume that all the fuel is expelled over 0.1 s at a constant exhaust speed of 20 m/s. Note that the latter approximation is a particularly poor one — in real life, the launch of a water rocket proceeds over two stages, one during which water is expelled and one during which the pressurized gas is expelled, and the speed of expulsion depends on the changing pressure inside the bottle. For more context, see: C. J. Gommes, "A more thorough analysis of water rockets: Moist adiabats, transient flows, and inertial forces in a soda bottle," *Am. J. Phys.* **78**, 236–243 (2010). https://doi.org/10.1119/1.3257702

Computationally determine the flight of a vertically launched water rocket subject to air resistance and gravity during the first 0.1 seconds and compare it to the analytical solution for the flight without air resistance.

You may also wish to extend the trajectory to see what happens after the fuel is expended. One way to do this is to numerically calculate the trajectory for the time span between 0.1 and 3.5 seconds, using the final state of the rocket from the earlier stage as your new initial conditions and letting `burn_rate = 0` and `vel_ex = 0`.

Simple Pendulum with Large-Angle Release

10

The simple pendulum is a classic example of a nonlinear system. Despite its simplicity, it offers a depth of analysis, as you will explore in this and the following unit. At small angles, the simple pendulum acts as a simple harmonic oscillator with a period almost independent of amplitude. As the release angle is allowed to increase, though, the sine dependency in the force causes the period to change. In this unit, you will numerically calculate how period depends on release angle for this nonlinear system.

10.1 Objectives

In this unit, you will:

- model the motion of a simple pendulum (PHY);
- determine how the period of a simple pendulum depends upon the release angle (PHY);
- use events in `solve_ivp()` to determine the period at a given angle (PRO);
- learn how to write and read in a table of data (PRO).

```
#%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
# Import the differential equation integrator
#    To reduce demand on memory, we import only the subset of the SciPy
#    package that contains solve_ivp
from scipy.integrate import solve_ivp

plt.rcParams["figure.figsize"] = (12,8) # make figures big for easier viewing
                                        #    (width, height in inches)
# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

10.2 Simple Pendulum, Theory

To orient you, we will start by analytically analyzing the system.

10.2.1 Period of pendulum released from small angles

Exercise 1

Let us look at a simple pendulum of length = 0.816 m. Assuming local $g = 9.803$ m/s^2, calculate the period it ought to have for small oscillations. Show your result to 4 significant figures.

Period of 0.816 m long pendulum for small oscillations, T_0 =

10.2.2 Simple pendulum equation of motion

You will recall that the usual expression for the simple pendulum makes a small-angle approximation, whereby $\sin(\theta) \approx \theta$. Let us relax that requirement in order to get an equation of motion that is valid for any angle.

Exercise 2

Work through the following analytical derivation on a separate sheet of paper.

- Starting with Newton's second law in angular form, $\sum \vec{\tau} = I\vec{\alpha}$, with $\tau = \vec{r} \times \vec{F}$, determine the restoring torque for a simple pendulum (just a point mass m on a frozen massless string):
- Write down the moment of inertia of a point mass at radius equal to the length of the pendulum:
- Hence the second derivative of the angle is some function of the angle. Write it down.
- What additional force (torque) term would need to go on the force (torque) side of the equation to account for damping?

10.3 Numerical Solution for $\theta(t)$

Now that you have the equation of motion for a simple pendulum released from an arbitrary angle, you will use `solve_ivp()` to solve it numerically and find the angle as a function of time.

Exercise 3

Start by defining the variables for this system. Fill in the code below to describe a 1.5 kg, 0.816 m pendulum.

```
# < Exercise 3, Defining the variables >

# Global variables
# length of pendulum, in meters
length = 1.0 ### FIX
# mass of pendulum in kg (may not drop out because of damping force)
mass = 1.0 ### FIX
# local gravitational acceleration in m/s^2
grav = 1.0 ### FIX
```

```
# optional damping parameter, such that the damping force is -(gamma_damp)*v
gamma_damp = 0.00 ### FIX
```

10.3.1 Defining the `deriv()` function

The next step is to use the equation of motion you determined in **Exercise 2** for a simple pendulum without damping to write a function, `deriv()`, that will return the derivative of each of the variables.

The equation of motion you found is a *second-order* differential equation in which the second derivative of the angle ($\ddot{\theta}$) is expressed as a function of the angle, θ. To use the `solve_ivp()` solver, you will need to turn that single second-order differential equation into a system of coupled first-order differential equations somewhat like the following:

$$\dot{\theta} = \omega$$
$$\dot{\omega} = ?$$

where ω is the angular velocity and you must provide the right-hand term of the second equation.

Exercise 4

The code cell below contains an outline of the `deriv()` function. Look over it carefully and make the necessary fixes to the calculations of the derivatives.

This function also takes the optional parameter, *gamma_damp*, which allows for the possibility of viscous damping. By default, this parameter is set to zero, and you may ignore it for now.

```
# < Exercise 4, Creating the deriv() function >

def deriv(t_now, r_now, gamma_damp = 0):
    """ Supplies the derivatives of angle and angular velocity to be used by
    the differential equation integrator, solve_ivp(), for a simple pendulum.
    The optional parameter, gamma_damp, allows for viscous damping and
    is set to zero by default. """

    theta = r_now[0]      # unpack the r-array for the angle
    theta_dot = r_now[1]  # unpack the r-array for the angular velocity

    d_Theta_dt = 0  ### FIX:  replace 0 with the appropriate expression from
                    #    Exercise 2
                    #    Hint:  This is probably simpler than you think

    d_ThetaDot_dt = 0  ### FIX:  replace 0 with the appropriate expression from
                       #    Exercise 2

    return np.array([d_Theta_dt,d_ThetaDot_dt])
```

10.3.2 Finding the solution with `solve_ivp()`

Now that the derivatives have been defined, the user must define the initial conditions and the time span over which to find the solution. The variables, along with the name of the function containing the derivatives and any optional parameters in the `deriv()` function, are then passed to `solve_ivp()`.

Exercise 5

Fill in the requested information in the code cell below to find the computational solution for a pendulum released from rest at 5 degrees. Calculate your solution for the time period from 0 to 5 seconds using 0.02-second-long integration time steps.

```
# < Exercise 5, Integrating the solution >

# Create the time span for the solution.  Just 5 seconds will be enough to
#    illustrate the technique for now
t_start = 0.0
t_end = 0.0  ### FIX

t_span = [t_start, t_end]
delta_t = 0.02  # time step of integration.  We will force solve_ivp() to work
                #    on fixed time intervals, rather than let it determine its
                #    own times.  This constraint will simplify plotting
                #    solutions for different starting angles
# Because we want the solution to be calculated at fixed time intervals, create
#    this array of times:
t_arr = np.arange(t_start, t_end, delta_t)

# Set the initial conditions
start_angle_degrees = 0  # starting angle, in degrees ### FIX

start_angle_radians = start_angle_degrees*np.pi/180.0  # convert to radians
start_anglev = 0  # [rad/s] start the pendulum from rest
#  The solver needs an array of initial conditions
r_init = np.array([start_angle_radians, start_anglev])
#     Integrate to get the solution array r_soln:
r_soln = solve_ivp(deriv, t_span, r_init, t_eval = t_arr,
                   rtol = 1e-8, atol = 1e-8)
    # deriv is the name of the function deriv() that returns the derivatives
    #    of the variables
    # t_span contains the minimum and maximum times for the integration
    # r_init is the array of the initial conditions
    # t_eval is an optional parameter.  If you omit t_eval, solve_ivp() will
    #   pick its own times within t_span
    # rtol and atol set the relative and absolute tolerance on the error
```

Exercise 6

In order to practice using the output from `solve_ivp()` and to see the accuracy, make the following two labeled plots. On both, please also include the small-angle solution for the motion of a pendulum with the same initial conditions.

1. Pendulum angle vs. time
2. Pendulum angular velocity vs. time

Optionally, you may also be interested in the plot of pendulum angular velocity vs. pendulum angle, also known as the "state-space" trajectory.

➤ *Hint:* What is the functional form of the analytical solution for the angle and angular velocity? Remember, you calculated the period of oscillations in **Exercise 1**.

```
# < Exercise 6, Plotting the trajectory >

### PLOT the angle vs. time, analytical and computational solutions

### PLOT the angular velocity vs. time, analytical and computational solutions
```

Exercise 7*

Further check your computational solution by examining the exchange of the system's energy between kinetic and potential over time. To do this, plot the Kinetic Energy (KE, red), Potential Energy (PE, blue), and Total Energy (TE, green) vs. time all on the same axes.

➤ *Hint:* How should the total energy behave as a function of time, if there is no damping in the system? This result for total energy will be a powerful check on your values of potential energy and kinetic energy.

```
# < Exercise 7, Plotting the energy >

# Placeholder (reminder) that when you're done, you'll need to make a solution
#    vector of KE, PE, and TE.
#    In order to be able to plot, we've prefilled this with *nonsense* copies
#    of the time array, which at least will be the correct length:
KE = 1.0*t_arr.copy()
PE = 2.0*t_arr.copy()
TE = 3.0*t_arr.copy()

### PLOT the kinetic, potential, and total energy vs. time
```

10.4 Comparing Solutions for Simple Pendulum at Different Initial Angles

Next you will use your numerical solver to compare the behavior of a pendulum when released at a large angle to the same pendulum released at a small angle.

Exercise 8

How do you expect the behavior of the pendulum to change as the initial angle is increased? Write down your prediction with justification in the space below.

Exercise 9

The cell below includes code that will call the numerical solver to find the solution for the behavior of a pendulum released from 5 degrees for the span of 20 seconds. Copy those lines of code and modify them to solve for the behavior of a pendulum released from 60 degrees. Then plot θ vs. time for both cases on the same figure.

➤ *Hint:* If you are having trouble seeing the difference in the behavior of the two cases, try finding the solution across a longer time period.

```
# < Exercise 9, Large-angle release >

# Create a new time array for the solution.
#    This code is included so that you can change the time period over which
#    the solution is calculated.
t_start = 0.0
t_end = 0 # [s] ### FIX

t_span = [t_start, t_end]
delta_t = 0.05
t_arr = np.arange(t_start, t_end, delta_t)

start_angle_degrees = 5  # [deg] initial angle
# Convert the initial angle to radians
start_angle_radians = start_angle_degrees*np.pi/180.0
start_anglev = 0  # [rad/s] start the pendulum from rest
r_init = np.array([start_angle_radians, start_anglev])  # initial conditions
r_soln = solve_ivp(deriv, t_span, r_init, t_eval = t_arr,
                   rtol = 1e-8, atol = 1e-8)

### DO THIS: Copy and modify the above code to find the solution to the
#    60-degree case.
# Don't forget to save a copy of the solution vector of the 5-degree case
#    before solving for the 60-degree one
```

```
# < Exercise 9, Large-angle release, Plots >

### DO THIS: Your plot here
```

You should see that the two pendula (small angle and high angle) get out of phase with one another. After 12.5 seconds, the 5-degree pendulum will go through about 7 periods, whereas the 60-degree pendulum, running more slowly, is closer to 6.5 periods. A nice way to observe this difference is to plot the high-angle solution vs. the small-angle solution.

10.5 Period of Simple Pendulum as a Function of Release Angle

Our goal is to determine how the period of a pendulum changes as a function of its maximum angle. However, it is rather inconvenient to determine the period manually, so you will streamline your code and determine the periods automatically.

As it happens, `solve_ivp()` can keep track of the times when certain things ("events") happen during the integration, such as when a projectile crosses a certain height, or in this unit, when a pendulum crosses through zero angle. To define such an event, you must define a function whose name is the same as the event and that takes the same parameters as the `deriv()` function:

```
def event_name(t_now, r_now, param1, param2, etc.):
```

When the function returns zero, the event will be triggered.

To use the events, add the parameter `events = event_name` to your `solve_ivp()` call. The time of the event triggering is recorded in `r_soln.t_events`, if `r_soln` is the name of the return from `solve_ivp()`. The contents of `r_soln.t_events` will actually be a list of arrays, with each element in the list corresponding to a different event definition. So if you wanted to access those times the first event was triggered, you would use `r_soln.t_events[0]`.

An additional functionality you will want to use here is that you can request events only be triggered if the zero point is crossed from a particular direction (either positive to negative or negative to positive). To use this functionality, set `event_name.direction` to a positive float if the event should trigger when going negative to positive and a negative float if the opposite is desired.

The following two cells illustrate how events can be used to calculate the period of a pendulum.

```
# Define an event by creating a function that returns zero when you want the
#    event triggered
def zero_cross(t_now, r_now):
    """ Event triggered when the omega_1 is zero """
    return r_now[0]

# Set the direction of the event so that it is only recorded if the pendulum is
#    crossing zero in the positive direction
zero_cross.direction = 1.0
```

```
# Illustrate how to use events to calculate period with the simple pendulum at
#    small angle (Exercise 3):

# Define the initial conditions
start_angle_degrees = 5
start_angle_radians = start_angle_degrees*np.pi/180.0
r_init = np.array([start_angle_radians, 0])
```

```
# Solve for the trajectory
r_soln = solve_ivp(deriv, t_span, r_init, t_eval = t_arr, events = zero_cross,
                   rtol = 1e-8, atol = 1e-8)

print("Here are the times of the zero-crossings:")
print(r_soln.t_events[0])

diffs = np.diff(r_soln.t_events[0])
print("Here are the differences in times between these crossings:")
print(diffs)

print("which yields an average period of {:7.4f} s".format(diffs.mean()))
```

Exercise 10

Putting it all together:

1. Use a loop to make a numerical solution for the trajectory of a pendulum for release angles from 1 to 89 degrees in 1 degree steps. For each solution, calculate the average period of the pendulum. To build up an array of periods for each release angle, start with an empty array and append the periods to it, similarly to how you built up your solution in Numerical Integration Applied to Projectile Motion. Our time step of 0.05 seconds is probably a little coarse for this exercise, and we have been finding only about ten positive-to-negative zero-crossings in our 20-second-long time array. You'll want to use a finer time step (your choice) and a longer time range (again, your choice). Example code to get you started is below.
2. Plot the period as a function of release angle, using the arguments `marker = 'o'` and `linestyle = ''` within your `ax.plot()` call to use unconnected circles for the data.
3. Create a second plot that normalizes the period by dividing by the small-angle value T_0 (which should be what you found in **Exercise 1**). Once again, plot your data as unconnected circles.

```
# < Exercise 10, Calculate the pendulum period for different release angles >

# Array of initial angles to calculate the periods for
start_angles = np.arange(1, 1, 1) ### FIX

### DEFINE the time span and array of times to be used by solve_ivp()

period_arr = np.array([])  # Empty array to store the periods
print("angle  period (s)")

# Loop through the angles from 1 to 89 degrees
for pendulum_angle in start_angles:
    start_angle_radians = pendulum_angle*np.pi/180.0
    start_anglev = 0  # [rad/s] start the pendulum from rest

    ### SOLVE for the trajectory of the pendulum at the desired ICs
```

```
    ### CALCULATE the period and save it as the variable "this_period"
    this_period = 0 ### FIX

    print("{:}:     {:7.5f}".format(pendulum_angle, this_period))
    period_arr = np.append(period_arr, this_period)
```

```
# < Exercise 10, Plot the pendulum period for different release angles >

### PLOT period versus release angle

### PLOT period/"small-angle period" vs. release angle
```

Exercise 11*

Find the angle at which the period has increased by 10 % over its small-angle value. You may find the Python numpy.where method useful for returning the indices in an array that fulfill some criteria. For example, given an array `arr`, the indices of all the elements greater than or equal to a fixed `value` are given by `np.where(arr >= value)[0]`. (The `[0]` indicates the first element of the returned values, which is what is desired here.)

An example of the `np.where()` usage is contained in the code cell below.

Angle (in degrees) at which the period is 10% greater than its small-angle period =

```
# Illustration of the np.where() method

arr = 2.6*np.arange(0.0, 12.0, 1)
print("The array:  ", arr)
print("The indices for which the elements are >= 15:  ",
      np.where(arr >= 15)[0])
print("The index of the first element that is >= 15:  ",
      np.where(arr >=15)[0][0])
print("The value of the first element that is >= 15:  ",
      arr[np.where(arr >=15)[0][0]])
```

```
# < Exercise 11, Angle for which the period increased by 10% >

### USE your results from Exercise 10 and np.where to find the release angle
#    that results in a period 10% greater than that for the small-angle
#    approximation
```

10.5.1 Writing data to a file

After running a time-consuming calculation, such as the period of the pendulum as a function of release angle, it can be advantageous to save the results to a file. These results can then be accessed later on, for instance in the following unit, Simple Pendulum Comparison to Data and Theory.

There are many options for reading and writing data with Python, some of which are explored in Appendix 2: Basic Input/Output of Files in Python. Because in these circumstances you'll want to write NumPy arrays of data, we recommend using numpy.savetxt(). The required arguments for `savetxt()` are the filename and a one- or two-dimensional array:

```
np.savetxt(filename, data_array)
```

The data can be formatted using the keyword `fmt` and the same format codes used when printing statements, preceded by "%". The keyword `delimiter` can be used to set the character-separating columns, and the keyword `newline` can be used to set the character-separating rows. The `header` and `footer` keywords are used to set strings that will be written at the start or end of the file, respectively.

Exercise 12

Use `np.savetxt()` to write the arrays of release angles and periods to a file. Because `np.savetxt()` takes only one data argument and you would like to write two arrays, the easiest thing to do is to combine them into one two-dimensional array by stacking them together as columns. This can be accomplished using numpy.column_stack, which takes a sequence of one-dimensional arrays and combines them into a two-dimensional array:

```
data_arr = np.column_stack([a,b])
```

Note that `np.hstack()` has a similar functionality, and `np.concatenate()`, `np.stack()` and `np.block()` provide more general stacking and concatenation operations.

For this exercise, save your data as a comma-separated values (CSV) file by using a filename with a .csv filename extension and commas for your delimiter:

```
np.savetxt("dataFile.csv", data_arr, delimiter=',')
```

For better documentation, you can use the `header` keyword to add the names of the columns separated by commas to the file, e.g., `header = "Release Angle [deg],Period [s]"`.

Once you have written your data to a file, ***try opening it up using a text editor and/or a spreadsheet program such as Excel.***

Finally, ***practice reading the data*** back into this notebook using numpy.loadtxt. Because you used comma delimiters when writing the data, set the `delimiter` argument to ','. If you used the header argument in `np.savetxt()`, then skip reading the first row using `skiprows = 1`. Finally, set the `unpack` argument to `True` so that the returned data are transposed and arguments may be unpacked using `angle_read_arr, period_read_arr = np.loadtxt(...)`. Altogether, your line of code will look like the following:

```
angle_read_arr, period_read_arr = np.loadtxt('dataFile.csv', delimiter = ',',
                                             skiprows = 1, unpack = True)
```

```
# FOR GOOGLE COLAB ONLY

# < Exercise 12, Write data to a file >

### UNCOMMENT the following lines if you are using Google Colab

#from google.colab import drive ## Mount your Google drive
#drive.mount('/content/drive/')

#import sys  # Append your Google drive "Colab Notebooks" directory to your
             #    path so that files contained in it may be found
#sys.path.append('/content/drive/My Drive/Colab Notebooks/')
#file_path = '/content/drive/My Drive/Colab Notebooks/'
```

```
# < Exercise 12, Write data to a file >

### CREATE a 2-D array of data using np.column_stack

### WRITE data using np.savetxt

### READ data using np.loadtxt
#    If you are using google.colab, use file_path + "periodData.csv" for the
#    filename in the call to np.savetxt
```

```
Checking elements:  [1. 2. 3. 4. 5.] [1.81282 1.81292 1.81309 1.81333 1.81364]
```

10.6 Check-out

Exercise 13

Briefly summarize in the cell below the ideas in this unit.

10.7 Challenge Problems

Consider taking a peek at the effect of small damping.

Verify, by direct substitution, that the solution to this equation of motion: $\frac{dv}{dt} = -\gamma v$ is a velocity that decays exponentially in time, like so:

$$v(t) = v_0 e^{-\gamma t}.$$

Given a damping parameter of $\gamma = 0.05$, when might the velocity be expected to fall to its 1/e point?

At what time will $v_0 \frac{1}{e} \approx 0.37 v_0$ ***?***

Returning to your computational solution for a pendulum released from 5 degrees, modify your code to include damping, generate the plots from Exercises 6 and 7, and describe what you see.

In what part of the motion is the energy loss the greatest? You might want to zoom in on a few cycles of the plot. Note that the instantaneous power loss due to the frictional damping is $P = -\vec{F} \cdot \vec{v}$, so $P = -\gamma v^2$. The cycle average power is half this, insofar as the average value of either $\sin^2(\omega t)$ or $\cos^2(\omega t)$ is $\frac{1}{2}$, thus $\langle P \rangle = -\frac{1}{2}\gamma v_{\max}^2$. Given that the total energy is the maximum of the kinetic energy, $\frac{1}{2}mv_{\max}^2$, the fractional energy loss goes as $-\frac{\frac{1}{2}\gamma v_{\max}^2}{\frac{1}{2}mv_{\max}^2} = -\frac{\gamma}{m}$. Therefore, you might compare your plot to $E_{(t=0)}e^{-(\gamma/m)t}$.

```
# < Damping, Exercises 3-7 revisited with small angle, small damping
#    (gamma_damp = 0.05) >

# Set the global variables
# length of pendulum, in meters
length = 1.0 ### FIX
# mass of pendulum (may not drop out because of damping force)
mass = 1.0 ### FIX
# local gravitational acceleration in m/s^2
grav = 1.0 ### FIX
# optional damping parameter, such that the damping force is -(gamma_damp)*v
gamma_damp = 0.0 ### FIX

# Create the time span for the solution.  Just 20 seconds will be enough to
#    illustrate the technique for now
t_start = 0.0
t_end = 20.0
t_span = [t_start, t_end]
delta_t = 0.01   # time step of integration
t_arr = np.arange(t_start,t_end,delta_t)

# Set the initial conditions
#   starting angle, in degrees
start_angle_degrees = 0  ### FIX

start_angle_radians = start_angle_degrees*np.pi/180.0
# Start at rest at start_angle, after converting to radians
rinit = np.array([start_angle_radians, 0],float)

# As usual, create a function to evaluate the derivatives:
def deriv(t_now,r_now):   # note that time is now the first element in the
                          #    derivative function
    theta = r_now[0]      # unpack the r-array for the angle
    theta_dot = r_now[1]  # unpack the r-array for the angular velocity

    d_Theta_dt = 0      # replace 0 with the appropriate expression from
                        #    Exercise 2
                        #    Hint:  This is simpler than you think -- refer to
                        #    Appendix: Using solve_ivp for Numerical Integration
```

```
    d_ThetaDot_dt = 0   # replace 0 with the appropriate expression from
                        #    Exercise 2, including the drag force (torque).

    return np.array([d_Theta_dt,d_ThetaDot_dt],float)

# This is it, the Python ODE solver, all in one line:
#    integrate to get the solution array r_soln at all points in time
r_soln = solve_ivp(deriv, t_span, rinit, rtol = 1e-8, atol = 1e-8)
        # in which variable r_soln.y[0] is the angle
        #   and variable r_soln.y[1] is the angular velocity
        # The times at which the solution is evaluated are given
        #   by r_soln.t

# Placeholder (reminder) that when you're done, you'll need to make a solution
#    array of KE, PE, and total E for plot #4.  In order to be able to plot,
#    we've prefilled this with *nonsense* copies of the time array,
#    which at least will be the correct length:
KE = 1.0*t_arr.copy()
PE = 2.0*t_arr.copy()
TE = 3.0*t_arr.copy()
```

Comparing Data and Theory for Simple Pendulum

11

Previously, you analyzed the simple pendulum through a simulation. The system's simplicity means that it offers many opportunities for analytical and experimental analysis, too. Here you will compare the relationship between period and release angle using these three physics techniques.

11.1 Objectives

In this unit, you will:

- analytically calculate the period of the simple pendulum (PHY);
- compute an integral using numerical integration or determining the value from a table of special functions (PRO);
- read in and analyze a table of experimental data (PRO);
- compare the analytical, computational, and experimental results (PHY).

```
#%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
# Import the differential equation integrator
#    To reduce demand on memory, we import only the subset of the SciPy
#    package that contains solve_ivp
from scipy.integrate import solve_ivp

plt.rcParams["figure.figsize"] = (12,8) # make figures big for easier viewing
                                        #    (width, height in inches)
# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

11.2 Computational Solution

Exercise 1

In the previous unit, Simple Pendulum with Large Angle Release, you simulated the motion of a simple pendulum. The first step in comparing that simulation to either the analytical solution or a real experiment is to grab information from that unit. In the next cell, copy over

your definitions for the gravitational acceleration and length of the pendulum. Calculate the period of the pendulum in the small-angle limit.

Rather than repeat your calculation for the period as a function of maximum angle, it is better to read in the data you calculated in the previous lab. Follow the instructions from Simple Pendulum with Large Angle Release **Exercise 12** to read in the contents of the data file you created in that unit.

```
# < Exercise 1, System parameters >
file_path = './'  # path to current directory

grav = 1 ### FIX # [m/s^2] gravitational acceleration
length = 1 ### FIX # [m] measured length of the pendulum
t0 = 0  ### FIX # [s] Set equal to the period in the small-angle limit

### CALCULATE the period in the small-angle limit
```

```
# FOR GOOGLE COLAB ONLY

# < Exercise 1, Load data >

### UNCOMMENT the following lines if you are using Google Colab

#from google.colab import drive ## Mount your Google drive
#drive.mount('/content/drive/')

#import sys  # Append your Google drive "Colab Notebooks" directory to your
             #    path so that files contained in it may be found
#sys.path.append('/content/drive/My Drive/Colab Notebooks/')
```

```
# < Exercise 1, Load data >

### READ data from the .csv file you created in Exercise 12 of "Simple Pendulum
#    with Large-Angle Release"
```

11.3 Analytical Solution

Although using the small-angle approximation greatly simplifies the math, it is possible to find the exact analytical solution to the period of a pendulum released at a particular angle. The following three exercises will lead you through the steps to do this.

Exercise 2*

Fill in the steps outlined in Problem 4.38, Equation 4.103 of *Classical Mechanics*, leading to the analytical solution for the simple pendulum at any angle:

$$\tau = \frac{2\tau_0}{\pi} \int_0^1 \frac{1}{\sqrt{1-u^2}} \frac{du}{\sqrt{1-A^2u^2}}$$

for some A (Eq. 4.103 from *Classical Mechanics*).

The resulting dimensionless definite integral is known as the complete elliptic integral of the first kind. It can be solved via direct numerical integration (using an efficient *quadrature* algorithm), or its values looked up in tables of special functions. The SciPy package calls it `scipy.special.ellipk()`, and it is a function of the variable A^2, usually referred to as k^2.

Exercise 3*

First, let's attempt to find the period by integrating Eq. 4.103 from *Classical Mechanics* using a quadrature formula.

The scipy.integrate.quad function numerically computes a definite integral by essentially computing a Riemann sum using adaptively subdivided intervals. To use `quad()`, one must first define the function that one wishes to integrate (in this case, Eq. 4.103). Then the required arguments for `quad()` are the name of the function, the lower limit of integration, and the upper limit of integration. The user must also pass to `quad()` using the keyword `args` any arguments the function takes in addition to the integration variable.

Fill in the code in the cells below to define the function. Then use `quad()` to calculate the integral for initial release angles used in the computational solution.

```
# < Exercise 3, Integrate using quad >

# Define a function equivalent to Eq. 4.103. It should take u (the integration
#    variable), tau_0, and A as parameters.

def dtau(u, tau0, A):
    """Eq. 4.103. u is the variable of integration
    tau0 is the zero-angle period T0
    A is a function of the starting angle
    """

    result = 0 ### FIX so that it is set to the value of Eq. 4.103

    return result
```

```
# < Exercise 3, Integrate using quad, cont. >

from scipy.integrate import quad  # Import the function from scipy.integrate

# Example of how to use quad() to find the period
start_angle_degrees = 0  # Start_angle in degrees
start_angle_radians = (start_angle_degrees/180)*np.pi
```

```
A = np.sin(start_angle_radians/2)

period = quad(dtau, 0, 1, args = (t0, A))  # The quad() integrator takes the
                                           #    function, the limits of the
                                           #    integral, and the parameters in
                                           #    a tuple called 'args' and
                                           #    returns a tuple

# The integrator returns the result and the error
print("Result and error: ", period)
print(period[0]) # extract the result

# Loop to calculate the period for an array of initial angles
print("\nangle, period")
for start_angle_degrees in angle_arr:

    ### FILL in to calculate the period using quad

    print(start_angle_degrees, period[0])
```

Exercise 4

As noted in Problem 4.38 in *Classical Mechanics*, the dimensionless integral with limits (0,1) is known as the *complete elliptic integral of the first kind*, and it is a function of A^2. Because this is a well-known integral, you don't actually have to calculate it yourself. In fact, SciPy includes it as a special function, scipy.special.ellipk.

Use `ellipk()` to calculate the period for the start angles in your computational data. All you need to do to run `ellipk()` is to pass it the value of `A`, e.g.,

```
ellipk(A**2)
```

Even better, you can pass `ellipk` an array of values for `A` and have it do all the calculations together.

```
# < Exercise 4, Using ellipk to calculate period >

# Import the library
from scipy.special import ellipk

### USE ellipk to calculate the period for all the release angles in the
#    computational data
```

Exercise 5

How well do the results of your analytical calculation compare to the simulation results? To find out, create a well-labeled plot of the elliptic integral solution for period vs. angle and your `solve_ivp()` solutions. Make a second plot showing the differences between the solutions, a.k.a. the residuals, as a function of release angle.

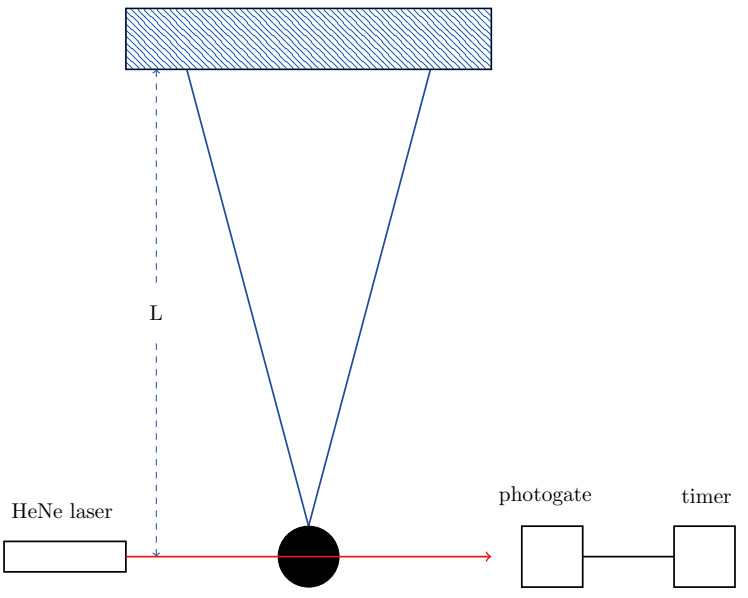
```
# < Exercise 5, Elliptic integral, solve_ivp() and their difference >

### PLOT period vs. angle for your simulation and analytical calculations.

### PLOT the difference between these solutions vs. angle
```

11.4 Experimental Solution

You have now calculated the period of the pendulum using two different theoretical approaches: analytically and with a simulation. How does your calculated period compare to the actual period, though? For that you will need experimental data.

We have collected data for you using the following experimental setup and included it in the file "simplePendulum.csv". You are welcome, though, to replicate this experiment to collect your own data!

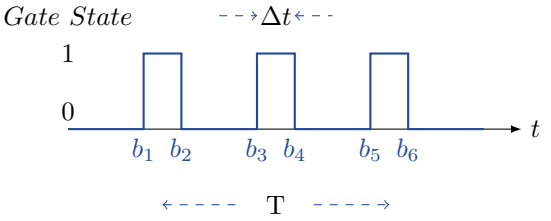
11.4.1 Experimental arrangement



As shown in the figure above, a 25.4 mm steel ball was attached to a rigid support via a pair of lightweight strings, forming a V-shaped bifilar pendulum of length 0.816 meters constrained to move in a fixed vertical plane (into and out of the page). The ball was drawn back 45 degree and set in motion so that it crossed a photogate (Vernier) at its lowest point. The pendulum was allowed to wind down due to weak damping, while the times of blocking and unblocking of the photogate were recorded. The photogate was used with an external

HeNe laser that was aligned with the retroreflection of the ball at the equilibrium position, ensuring that the beam passed through the center of the ball.

The data for this example were exported from Vernier Logger Pro as a CSV (comma separated value) file, "simplePendulum.csv". The Logger Pro software records photogate data as the time at which the photogate is blocked (Gate State = 1, called gsH) or unblocked (Gate State = 0, called gsL), along with a column of digital gate states at these times.



As noted by Kostov et al.,[1] timing a pendulum in this way can be divided into six transitions: b1 are the times when the ball first blocks the beam, b2 are the times when it next unblocks the beam, b3 is blocking on the first retrace, b4 is unblocking of the first retrace, b5 is blocking again in the forward direction, and b6 is unblocking in the forward direction. The center of the ball is at the equilibrium position at times (b1+b2)/2 and at (b5+b6)/2; the difference between these two times gives the period for that cycle. The speed of the ball is proportional to the blocking time of the midpoint readings, (b4-b3). It is given approximately by the diameter of the ball divided by the blocking time.

We ignore some small corrections: the reflection of the laser beam tends to smear out the image at the detector as the beam grazes the ball making it appear slightly bigger than its diameter, the beam falls along an arc of radius L, rather than straight through the center of the ball, and the ball is not actually at the lowest point when it is measured (making the measured speed slightly smaller than it would be). By attenuating the beam and placing a small aperture in front of the photogate, the smearing of the beam was largely controlled. The arc error is much less than 1%. The height error correction is smaller and is expected to be significant only at very small amplitude.

11.4.2 Experimental data

Exercise 6*

In order to interpret the experimental data, we must account for the fact that the measurements are for a physical, rather than ideal, pendulum. As such, there is a tiny correction to the effective length of the pendulum due to the moment of inertia of the 25.4 mm steel ball at the end of a 0.816 meter string (assumed to be massless).

Calculate the effective length of the pendulum and the period in the small-angle limit below.

[1] Y. Kostov, R. Morshed, B. Holing, R. Chen, and P.B. Siegel, "Period-speed analysis of a pendulum," Am. J. Phys. **76**, 956-962 (2008).

```
# < Exercise 6, Effective length of the pendulum >

### CALCULATE the effective length of the pendulum using the moment of inertia
#    for a 25.4 mm steel ball at the end of a 0.816 m massless string
```

Exercise 7

Read in the experimental data from `simplePendulum.csv`. When reading data in from a file, it is helpful to first open the file in a text or spreadsheet reader in order to see the format. When you do this, you'll notice that `simplePendulum.csv` contains two columns of comma-separated data. At the start of the file are two rows of header. The first lists information about when the file was created and the second indicates the contents of the columns. The first column contains the time (in seconds) when there was a gate change (i.e., when the bob passed in front of the laser, breaking the beam, or when it exited the beam). The second contains the gate state, 1 for blocked and 0 for open.

Because this is comma-separated value data, you can use `np.loadtxt()` to read it in, as you did in **Exercise 1**. Once you have read in the data, print the first ten entries.

Other ways of reading in files of different formats are discussed in Appendix 2: Basic Input/Output of Files in Python.

```
# < Exercise 7, Reading data >

### REPLACE the dummy placeholder definitions for t_arr and gs_arr below by
#    data read in from simplePendulum.csv
t_arr = np.array([])  # [s] Times at which there was a gate change
gs_arr = np.array([])  # Gate state
```

The data file only contains the instances when the gate state changes. For the purposes of illustration, it is nice to be able to plot the gate state versus time, similar to the figure on p. 118. This can be accomplished by manipulating the data arrays, as done in the cell below.

```
# This block of code makes two modified arrays, t_plt_arr and gs_plt_arr, for
#    use in plotting the gate state over time

# Make a new array of gate state by duplicating each one except the last.
#    To do this, a new two-row array with duplicate rows is produced.
#    The array is then flattened into one array. The parameter "F" in the
#    flatten call tells it to flatten in "column-major" order, meaning that
#    columns of data appear contiguous in the new array
gs_plt_arr = np.array([gs_arr[:-1], gs_arr[:-1]]).flatten("F")

# Make a new array of times that provides duplicate values for all but the
#    first time. As for gs_plt_arr, an array of two new rows is constructed
#    and then flattened in "column-major" order
t_plt_arr = np.array([t_arr[0:-1], t_arr[1:]]).flatten("F")
```

```
print("Here are the ten gate changes and times:")
for row in range(20):
    print("{:10.6f}, {:1.0f}".format(t_plt_arr[row], gs_plt_arr[row]))
print("")

# Plot of Gate State vs. Time for the first 5 seconds
fig, ax = plt.subplots(2,1)
ax[0].plot(t_plt_arr, gs_plt_arr)
ax[0].plot(t_arr, gs_arr, '*r')
ax[0].set_xlim([0, 5])
ax[0].set_ylabel("Gate State")
ax[0].set_title("First and Final 5 seconds\nTransitions marked as red stars")
# Plot of Gate State vs. Time for the final 5 seconds
ax[1].plot(t_plt_arr, gs_plt_arr)
ax[1].plot(t_arr, gs_arr, '*r')
ax[1].set_xlim([t_plt_arr[-1] - 5, t_plt_arr[-1]])
ax[1].set_xlabel("Time [s]")
ax[1].set_ylabel("Gate State")
plt.show()
```

Exercise 8

Consider the plots above showing the gate state versus time for the first and final five seconds of the experiment. ***Based on these data, how is the period and maximum angular velocity of the pendulum changing over time? How can you tell?***

As discussed in Experimental Arrangement, each complete back-forth-back swing of the pendulum results in six transitions: b1, b2, b3, b4, b5, and b6. To collect the maximum amount of data, we can imagine that these complete swings are interleaved together. In other words, a complete left-right-left swing is one period but so is a right-left-right swing. As a result, b3 and b4 for one period become b1 and b2 for the next period.

We slice the time arrays to produce the arrays of values for these transitions below. Remember that the double colon notation, `[::n]`, means that every nth data point should be selected. Therefore, `[::2]` selects every other element starting from index 0 (i.e., 0, 2, 4 …), while `[1::2]` selects every other element starting from index 1 (i.e., 1, 3, 5 …).

We then have to trim the last two elements off of b1 and b2 because these transitions for the initial swing don't have data for the return swings. Similarly, we must trim off the final elements of b3 and b4.

```
#  There are six edges to detect the period and the velocity.  The first edge
#    is when the ball blocks the beam, then a short while later the ball
#    unblocks, blocks and unblocks, then blocks and unblocks a third time.
#    Label edges b1, b2, b3, b4, b5, b6 respectively.
print("Length of t_arr: ", len(t_arr), len(t_arr)%6)  # make sure it is
                                                      #    divisible by 6
```

```
b1 = t_arr[::2]  # first blocking in forward direction
b2 = t_arr[1::2] # first unblocking
b3 = t_arr[2::2] # retrace blocking
b4 = t_arr[3::2] # retrace unblocking
b5 = t_arr[4::2] # second blocking in forward direction
b6 = t_arr[5::2] # second unblocking in forward direction
print("Lengths of arrays in which photogate changes sign:\n",
      len(b1),len(b2),len(b3),len(b4),len(b5),len(b6))

# Trim the final two elements off of b1 and b2 and the final elements of b3
#    b4 so that they are the same length as b5 and b6
b1 = b1[:-2]
b2 = b2[:-2]
b3 = b3[:-1]
b4 = b4[:-1]

### CAUTION: If you rerun this cell, you will have to read in the data again,
#    because we overwrite the edge times (i.e., trim their length)!
```

Exercise 9

The period of the pendulum (time for one complete swing back and forth) is the time between the midpoint between b1 and b2 and the midpoint between b5 and b6. Calculate the array of periods in the cell below.

```
# < Exercise 9, Calculate the array of periods >

period_expt = np.array([]) ### FIX
```

Exercise 10

The velocity of the bob can be calculated by determining how long the beam is blocked during a middle swing (b4-b3) and assuming that the distance traveled during that time is the diameter of the bob. Make the calculation in the cell below.

```
# < Exercise 10, Calculate the array of velocities >

velocity_expt = np.array([]) ### FIX
```

Exercise 11

By knowing the velocity of the bob, you can use conservation of energy to calculate the initial angle of that swing. Make that calculation for each swing of the pendulum in the cell below.

```
# < Exercise 11, Calculate the array of angles >

### APPLY conservation of energy to deduce the maximum angle at which the
#   pendulum started each swing
angles_expt = velocity_expt  # FIX
```

Exercise 12

You now have all the information you need to plot period versus release angle for the experimental data. Do so by plotting both the period versus release angle and normalized period (period/period in the small-angle limit) versus angle.

```
# < Exercise 12, Plot experimental period versus angle >

### PLOT experimental period versus release angle

### PLOT normalized experimental period versus release angle
```

Exercise 13

Now compare the relationship between period and angle for the experimental data with your analytical solution and data from your simulations. To do this, make the following plots:

1) a plot showing the normalized periods versus angle generated experimentally, from simulations, and analytically and
2) a plot showing the residuals for the experimental and analytical data (analytical period - experimental period) versus angle.

➤ *Hint:* When calculating the residuals, it is easiest to first recalculate the analytical solution using `ellipk()` for the angles with experimental data.

If you did not complete the portion of this unit on the analytical solution, you should leave that curve off of the first plot and skip the second plot.

```
# < Exercise 13, Compare period versus angle for experiment, analytical
#    and simulation results >

### CALCULATE the analytical solution for the angles for which there are
#    experimental data

### PLOT normalized period versus angle for experiment, analytical solution,
#    and simulations

### PLOT the difference between the normalized periods for the experimental and
#    analytical data as a function of angle
```

Exercise 14*

Using the same dataset as above, make a plot of amplitude vs. time; better yet, try plotting the natural log of the amplitude vs. time.

➤ *Hint:* What times should you use for your *x*-axis? Consider what gate state changes best represent the time of the swing.

Compare the shape with the damped harmonic oscillator for when the amplitude is low. ***What happens at high amplitude?***

```
# < Exercise 14, Plot experimental amplitude vs. time >

### PLOT Experimental amplitude vs. time
```

Exercise 15

At small angles, the plot of $\log(\theta)$ versus time looks very much like a line, which would indicate an exponential decay of the amplitude. Such an exponential decay rate is consistent with linear velocity-dependent drag. At high amplitude, the loss is greater, indicating a v^2 or more complicated regime as the flow around the ball becomes turbulent.

Calculate the decay rate by fitting a line to the small-angle portion of the graph using scipy.optimize.curve_fit, similarly to what you did in **Exercise 9** from Taylor Series with Loops and Functions. As you saw in that exercise, the first step is to define the functional form of the fit. To do this, in the cell below define a function that returns $m * x + b$ when passed the arguments x, b, and m. This will enable you to fit a line to the log of the amplitude. While it is possible to fit an exponential directly to the data, the method is less robust.

Then, use scipy.optimize.curve_fit to find the best-fit values of m and b for times after 500 seconds. The form of the function call is

```
popt, pcov = curve_fit(fit_func, x, y, p0)
```

where `fit_func` is the name of the fitting function you defined above, `x` is your independent variable (time), and `y` is your dependent variable (θ). The variable `p0` is an array of initial guesses for the parameters in your fitting function, in this case, a guess for b and a guess for m. The returned values are 1) the array of fitted parameters (`popt`) optimized such that the square of the residuals is minimized and 2) the estimated approximate covariance of the fitted parameters (`pcov`), which is a measurement of the quality of the fit.

After you have found the best-fit curve, plot it atop your graph of $\log(\theta)$ versus time.

What is the exponential decay rate of the amplitude?

```
# For fitting the decay of the amplitude
from scipy.optimize import curve_fit
```

```
# < Exercise 15, Linear fit to log(angle) >

### DEFINE a fitting function: a function that takes x, the independent
#    variable, and two parameters (b, the y-intercept, and m, the slope) as
#    arguments and returns m*x + b
```

```
# < Exercise 15, Linear fit to log(angle), cont. >

### DEFINE an array of initial guesses for b and m

### CALL CURVE_FIT for log(angle) vs. t, assuming a linear fit

### PLOT your best-fit over the graph of log(angle) vs. t
```

11.5 Check-out

Exercise 16

Briefly summarize in the cell below the ideas in this unit.

12 Oscillations in a Potential Well

When faced with a differential equation that cannot be analytically solved, physicists typically take one of two routes: simplifying the equations so that they can be analytically solved or using a computer to numerically find the solution. In this unit, you will compare both of those methods. When completing your analysis, you will refer to the potential energy function. In this way, you will gain practice linking the shape of a potential energy function to the behavior of a system.

This is a long unit, and it may be desirable to split it across two sessions (after **Exercise 8** or **9** are natural break points) or to shorten it by adopting one or more of the following suggestions. Many of the analytical calculations (in particular, **Exercises 1**, **5**, and **6**) could be completed ahead of time by the student. The unit could also be truncated after **Exercise 9** and still fulfill the first three objectives below. **Exercises 14** through **16** compare small- and large-angle oscillations. They are similar to Simple Pendulum with Large-Angle Release and could be omitted, especially if that unit has already been completed.

12.1 Objectives

In this unit, you will:

- analyze a system, including determining stable and unstable equilibria, by graphing the potential energy function (PHY);
- learn how to use the `scipy.optimize.root()` function to solve an equation (PRO);
- use a Taylor series approximation to find an approximate analytical solution (PHY);
- determine how the behavior of the system is affected by its initial energy and its location relative to stable and unstable equilibria (PHY).

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp  # the differential equation integrator

plt.rcParams["figure.figsize"] = (12,8)  # make figures big for easier viewing
#    (width, height in inches)
# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

12.2 Potential Energy Function for an Oscillating Wheel

Imagine a wheel with a mass m_1 mounted to the rim at angle ϕ. Another mass, m_2, hangs below on the opposite side of the wheel from a string wrapped around the rim. The wheel is free to rotate back and forth. As it turns one way, m_1 rises while m_2 falls and vice versa.

This situation is the subject of Problem 4.37 of *Classical Mechanics* and is illustrated in Figure 4.28.

12.2.1 Identifying stable and unstable equilibria

Exercise 1

Determine the potential energy function.

To do this, write a function, `pot_wheel()`, that takes as variables:

- value(s) of `phi` (angle at which m_1 lies on the wheel),
- `radius`, radius of the wheel,
- `m1`, mass of m_1, and
- `m2`, the mass of m_2.

The function should return the value(s) for the potential energy corresponding to the different value(s) of `phi`.

Note that Python and NumPy are clever enough that you should be able to send the function either a single value for `phi` or a NumPy array of values. The form of the potential energy returned will match that of `phi` and thus be either a single value or an array.

```
# < Exercise 1, Function to calculate the potential energy >

grav = 9.8 # [m s**-2]; gravitational constant

def pot_wheel(phi, radius, m1, m2):
    """ Return the potential energy associated with the system in Fig. 4.28 for
    a particular value or values of phi."""

    pot_energy = 0*phi ### FIX

    return pot_energy
```

Exercise 2

Using your function, calculate and plot the potential energy as a function of ϕ for -180 degrees $< \phi <$ 180 degrees. Let the radius of the wheel be 1 m and let m_1 = 1 kg. Show on the same set of axes the potential energy curves for $m_2 = 0.7m_1$, $0.8m_1$, $0.9m_1$, $1.0m_1$, and $1.1m_1$. Include a legend labeling the different potential energy curves.

➤ *Hint:* You will likely have to convert between degrees and radians as `np.cos()` and similar NumPy trigonometric functions assume the angle is given in radians.

What differences in the potential energy curves appear when m_2/m_1 increases? How will this affect the behavior of the system? Consider differences such as the number, location, and type of equilibria, and depth and width of the potential well (where applicable).

```
# < Exercise 2, Plot the potential energy >

radius = 1 # [m]; the radius of the wheel
m1 = 1 # [kg]; the mass mounted on the edge of the wheel

phi = np.array([0,0]) ### FIX

fig, ax = plt.subplots()
ax.axhline(0.0, color = 'black', linestyle ='--')  # horizontal line indicating
                                                   #    zero potential energy
ax.set_xlabel("$\phi$ [degrees]")
ax.set_ylabel("U [J]")
ax.set_title("Potential Energy Well for Wheel")

# Legend
ax.legend(["U = 0","m2 = 0.7*m1", "m2 = 0.8*m1",  "m2 = 0.9*m1", "m2 = m1",
           "m2 = 1.1*m2"], loc = "lower left")
plt.show()
```

Exercise 3

For $m_1 = 1$ kg and $m_2 = 0.7$ kg, calculate the value of $\phi > 0$ that corresponds to the *stable* and *unstable* equilibria.

➤ *Hint:* `np.arcsin()` only returns values between $-\pi/2$ and $\pi/2$ so you will need to make an adjustment to find values outside of that range.

ϕ_{stable} =
$\phi_{unstable}$ =

```
# < Exercise 3, Stable and unstable equilibria >

### USE this cell for your calculation
print("For m2 = 0.7:\n")
```

Exercise 4

Find the critical value for m_2/m_1 at which on one side the system oscillates and on the other side it does not, when released from rest at $\phi = 0$. Plot the potential energy function for this mass ratio, and explain how the graph illustrates your result.

➤ *Hint 1:* Unlike in the previous exercise, you will not be able to solve the resulting equation analytically. Instead, simplify your equation as far as possible and arrange it such that one side of the equation is zero. Then use the scipy.optimize.root function to find the root of the equation numerically (i.e., find the value of x at which $y(x) = 0$). An example of how to use `scipy.optimize.root()` can be found below.

➤ *Hint 2:* Give some thought to the initial guess you will use with `scipy.optimize.root()`, as there are multiple possible solutions that result in zero potential energy but only one meaningful one. You can use your plot from **Exercise 2** to do this intelligently.

$\left(\frac{m_2}{m_1}\right)_{critical} =$

When looking at the graph of $U(\phi)$ ***for*** $\left(\frac{m_2}{m_1}\right)_{critical}$***, it is apparent that . . .***

```
# Example of using scipy.optimize.root() to solve an equation numerically
#    Make sure you understand what is happening here.

from scipy.optimize import root # Import root from the scipy.optimize package

def func(x):
    """ A quadratic function for which we want to find the root """
    return -10 + 2*x**2

# To call root, provide the name of the function as the first argument
    # and a guess at the value of the root as the second argument
result = root(func, 2)  ### TRY using a different guess to find the other root.
                        #    What happens if your guess is right between the
                        #    two roots?
                        #    root can return multiple roots:  try [-5,2]
                        #    instead of 2
#
# Print the return of root()
#    This contains a lot of information
#    in the form of an object. For more of a description, see
#    https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.
       OptimizeResult.html#scipy.optimize.OptimizeResult
print(result)

# For our purposes, the key information is the value of the root, accessible as
#    "result.x[0]"
root_val = result.x[0] # The value of the root
print("\nThe root: {}".format(root_val))

# The block of code below provides a graph of func() to allow you to visualize
#    the solution
fig, ax = plt.subplots()
x_array = np.linspace(-10, 10, 200) # Set of x values for plotting
y_array = func(x_array) # The corresponding y values
```

```
ax.plot(x_array, y_array) # The function the root was calculated for
ax.axhline(0.0, color = 'black', linestyle ='--') # horizontal line at zero
ax.plot([root_val, root_val], [0,0], 'ko')  # A point at the determined root of
                                            #    the solution
                                            #    The "k" indicates to make it
                                            #    black, the "o" indicates to
                                            #    make it a circle. Linestyles,
                                            #    marks, and colors can often be
                                            #    indicated together in the
                                            #    plotting call this way.
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_title("Example of root finding")
plt.show()
```

```
# < Exercise 4, Use scipy.optimize.root to find the value of m2/m1 for
#    oscillation when E = 0 >

radius = 1 # [m]; the radius of the wheel
m1 = 1 # [kg]; the mass mounted on the edge of the wheel

def func(phi):
    """ The simplified equation for which you want to find the root """
    value = 0 ### FIX

    return value

guess = 0 ### FIX
print("The root lies at an angle of {:6.4f} radians ({:5.2f} degrees)".
    format(root_val, root_val*180/np.pi))

result = root(func, guess)
root_val = result.x[0]
m2_crit = 0 ### FIX

print(r"(m2/m1), critical = ", m2_crit)

# Plot of the potential energy function for the critical value of m2/m1
fig, ax = plt.subplots()
ax.axhline(0.0, color = 'black', linestyle ='--')  # horizontal line indicating
                                                   #    zero potential energy
ax.set_xlabel("$\phi$ [degrees]")
ax.set_ylabel("U")
ax.set_title("Potential Energy Well for Wheel")

phi = np.array([0,0]) ### FIX

plt.show()
```

12.3 Analytical Approximation Using Taylor Series

The presence of a stable equilibrium for many mass ratios indicates that the system can undergo oscillations. But what do these oscillations look like, and specifically, what is their period? In this following section, you use a Taylor series expansion of the force to determine the approximate behavior of the system when oscillating.

12.3.1 Approximating the force and potential energy

Exercise 5

On a separate sheet of paper, write down a second-order Taylor series approximation for the potential energy expanded *around the stable equilibrium point*. You may include the angle of the stable equilibrium, ϕ_{stable}, as a term in your final equation or expand it in terms of m_1 and m_2 as you did in **Exercise 3**.

➤ *Hint:* Remember that the first derivative of potential energy is zero at equilibrium points.

$U_{approx}(\phi) =$

Exercise 6

Plot the *approximate* potential energy function along with the *true* potential energy function for $m_1 = 1$ kg and $m_2 = 0.7$ kg.

What this graph demonstrates is that Taylor series may be used to fit a parabola to a potential energy function around a stable equilibrium.

```
# < Exercise 6, Exact and Taylor approximation for the potential energy >

fig, ax = plt.subplots()
ax.set_xlabel("$\phi$ [degrees]")
ax.set_ylabel("U [ J ]")
ax.set_title("Potential Energy Well for System: $m_1$ = 1 kg, $m_2$ = 0.7 kg")

plt.show()
```

Exercise 7

Making a second-order Taylor series approximation of the potential energy, as you did above, means that you are modeling the derivative of it with a first-order Taylor series approximation.

To show this equivalency, first determine the exact expression for the force on the system as a function of ϕ.

➤ *Hint:* Recall that in polar coordinates $\nabla f = \frac{1}{r}\frac{\partial f}{\partial \phi}\hat{\phi} + \frac{\partial f}{\partial r}\hat{r}$. Furthermore, since this is a curvilinear system, the potential energy is only changing with ϕ so its partial derivative with respect to r is zero.

$F(\phi) =$

Then, once again working in polar coordinates, find the negative of the gradient of the Taylor series approximation to the potential energy from **Exercise 6**.

$F_{approx}(\phi) =$

Exercise 8

Now, plot on the same axes the exact force as a function of ϕ and your Taylor series approximation from **Exercise 7** as a function of ϕ for $m_1 = 1$ kg and $m_2 = 0.7$ kg. Keep the radius of the wheel equal to 1 meter.

```
# < Exercise 8, Exact and Taylor approximation for the force >

m1 = 1
m2 = 0.7

fig, ax = plt.subplots()
ax.set_xlabel("$\phi$ [degrees]")
ax.set_ylabel("Force [N]")
ax.set_title("Force Felt by System: $m_1$ = 1 kg, $m_2$ = 0.7 kg")

ax.axhline(0.0, color = 'black', linestyle ='--') # horizontal line at force = 0
plt.show()
```

12.3.2 Theoretical analysis of oscillations

You have used Taylor series to find an approximate equation of motion of the system. Now you can analyze that equation of motion to determine the behavior of the system. To do this, you will compare it with a familiar system: the simple harmonic oscillator made from a mass and a spring.

In Chapter 5 of *Classical Mechanics*, you will derive the solution to the simple harmonic oscillator. In previous physics classes, however, you have likely learned that the equation of motion for a mass on an ideal spring is

$$\ddot{x} = -\frac{k}{m}(x - x_{equilib}),$$

and that the solution to this differential equation if the mass is released from rest at distance A from equilibrium is

$$x(t) - x_{equilib} = A\cos\left(\sqrt{\frac{k}{m}}t\right)$$

i.e., oscillations of amplitude A at an angular frequency of $\sqrt{\frac{k}{m}}$, resulting in a period of $T = 2\pi\sqrt{\frac{m}{k}}$.

Exercise 9

Express the force on the system in terms of the angular acceleration, $\ddot{\phi}$ and the inertial mass, $(m_1 + m_2)$. Then set this expression for force equal to your first-order Taylor series

approximation for the force. By slightly rearranging your equation and matching constants from the above equation for a mass on a spring, determine the period of oscillations. Then graph your approximate solution for ϕ(t) if the system is released from rest at an angle of 5 degrees away from the stable equilibrium. Show the solution for $0 \leq t \leq 10$ s using time steps of 0.05 s.

➤ *Hint:* Remember `np.cos()` and `np.sin()` assume the parameters are in radians; be sure to do the conversion from degrees to radians.

Period [s] =

```
# < Exercise 9, Analytical approximation of oscillations >

### INCLUDE your calculation of the period here

# Define an array of times to calculate the solution for
t_arr = np.arange(0, 1, 1) ### FIX
# Calculate angle(t)
angle = t_arr ### FIX

fig, ax = plt.subplots()
ax.set_xlabel("Time [s]")
ax.set_ylabel("Angle [degrees]")
ax.set_title("Oscillations with Taylor series:"
             + "radius = 1 m, $m_1$ = 1 kg, $m_2 = 0.7$ kg")
### PLOT angle vs. t

plt.show()
```

12.4 Computational Solution for $\phi(t)$

As you can see, Taylor series expansion provides a simple way to find an approximate solution for oscillations around a stable equilibrium point. However, another option is to solve the exact equations of motion numerically. In this next section, you will use the Python numerical solver, `solve_ivp()`, to find the angle as a function of time.

12.4.1 Computational solution for small angles

Exercise 10

The following code blocks are missing a few pieces, which you will have to supply. In particular, the function that evaluates the derivatives needs some work. To determine the appropriate expressions for the derivatives, remember that $F = (m_1 + m_2)R\ddot{\phi}$ and that you found an exact expression for force in **Exercise 7**. Please supply those missing pieces of code, and create these two labeled plots for a wheel with radius = 1 m, m_1 = 1 kg and m_2 = 0.7 kg:

1. Angle vs. time for the numerical and Taylor series approximation solutions
2. Angular velocity vs. time for the numerical and Taylor series approximation solutions

Let the wheel start from rest 5 degrees from the stable equilibrium. As in your approximate solution, your numerical solution should run from $0 \leq t \leq 10$ s using time steps of 0.05 s. In this exercise and throughout this notebook, please set the *t_eval* parameter in `solve_ivp()` to an array of times for which a solution is defined. Requiring that `solve_ivp()` use a fixed array of times rather than selecting its own adaptive time steps will simplify comparisons among different solutions.

How does your approximate analytical solution compare to the computational one? Can you explain any differences by referencing the potential energy diagram? Pay attention to the period, amplitude, and symmetry of the oscillations.

```
# < Exercise 10, Computational solution for phi(t), the deriv function >

# Deriv function for use by solve_ivp()

# Create a function to evaluate the derivatives
def deriv(t_now, r_now, radius = 1, m1 = 1, m2 = 0.7):
    """ Supplies the derivatives of phi and dphi/dt to be used by the
    differential equation integrator, solve_ivp. The optional parameters
    radius, m1, and m2 are set to default values for the system. """

    phi = r_now[0]    # unpack the r-array for the angle
    phiDot = r_now[1] # unpack the r-array for the angular velocity

    dphi_dt = 0  ### replace 0 with the appropriate expression from Exercise 7
                 #    Hint:  This is probably simpler than you think.

    dphiDot_dt = 0  ### replace with the appropriate expression from Exercise 7


    return np.array([dphi_dt, dphiDot_dt])
```

```
# < Exercise 10, Computational solution for phi(t), the integrator >

# Create the time array for the solution going from 0 to 10 seconds in steps of
#    0.05 s
delta_t = 0  ### FIX the time step of integration
t_end = 0  ### FIX the maximum time

# Set the time span to run from t = 0 to t_end.
t_span = (0.0, t_end)
# Define an array of times for which a solution will be found
t_arr = np.arange(0.0, t_end, delta_t)

# Set the initial conditions by defining the starting angle
start_angle = 0  ### FIX
```

```
r_init = np.array([start_angle, 0])  # Start at rest at start_angle, after
                                     #    converting to radians.
                                     #    In general, r_init contains an array
                                     #    of the initial states of the
                                     #    variables you'll be integrating.
                                     #    Here, you have phi and
                                     #    dphi/dt (angular velocity)

# This is it, the Python ODE solver, all in one line:
#    "deriv" is the name of the function that returns the derivatives of the
#    two variables, angle and angular velocity.
#    t_span is a list or tuple containing the range of times to solve over.
#    Variable r_init[0] is the initial angle
#    and variable r_init[1] is the initial angular velocity.
#    The variable t_arr is the specific array of times to solve over.
#    args is set to the values of the additional parameters used in the call to
#    the deriv function.
r_soln = solve_ivp(deriv, t_span, r_init, args = (radius, m1, m2),
                   t_eval = t_arr ) # integrate to get the solution vector
                                    # r_soln across the time span

# r_soln holds the time array and the values of angle and velocity as:
#    t_array = r_soln.t[0]
#    phi solution for all time is r_soln.y[0]
#    phiDot solution is r_soln.y[1]
```

```
# < Exercise 10, Computational solution for phi(t) & dphi/dt(t), the plots >

# Plot the angle vs. time for the approximate analytical and computational
#    solutions
fig, ax = plt.subplots()
ax.set_title("Angle vs. time")
ax.set_xlabel('Time [s]')
ax.set_ylabel('$\phi$ [degrees]')


# Plot the angular velocity vs. time for the approximate analytical and
#    computational solutions
fig, ax = plt.subplots()
ax.set_title("Angular velocity vs. time")
ax.set_xlabel('Time [s]')
ax.set_ylabel('$\omega$ [degrees/s]')

plt.show()
```

Exercise 11*

Validate your code by showing that energy is conserved in your computational solution. To do this, plot potential energy, kinetic energy, and total energy over time on the same axes.

➤ *Hint:* Your total energy will likely be a negative number (why is this?). If you would like to modify your plot so that the kinetic energy, total energy, and potential energy all cover the same range, you can adjust your potential energy by adding a constant offset. One reasonable choice for such an offset is to make the potential energy at the bottom of the potential well zero.

```
# < Exercise 11, Plot the potential, kinetic, and total energy vs. time >

p_energy = r_soln.y[0]  ### FIX the potential energy calculation
k_energy = r_soln.y[0]  ### FIX the kinetic energy calculation
t_energy = p_energy + k_energy  # Total energy

fig, ax = plt.subplots()
ax.set_title("Energy vs. time")
ax.set_xlabel('Time [s]')
ax.set_ylabel('Energy [J]')

plt.show()
```

12.4.2 Computational solution for different mass ratios

Next you will use your numerical solver to compare the behavior of the system for different values of m_2/m_1.

Exercise 12

In the cell below, copy your *relevant* code from above to call the numerical solver to find the solution for the behavior of the system released from rest at $\phi = 0$ when $m_2/m_1 = 0.7$. Then, copy those lines of code again and modify them to solve for the behavior if $m_2/m_1 = 0.8$ and if $m_2/m_1 = \left(\frac{m_2}{m_1}\right)_{critical}$ from Exercise 4.

Note that you will want to save the solutions for each of the different mass ratios so that you can plot them together. One way to do this is to set the output of `solve_ivp()` to different variable names:

```
r_soln_07 = solve_ivp(deriv, t_span, r_init, args = (radius, m1, m1*0.7),
 t_eval = t_arr)
r_soln_08 = solve_ivp(deriv, t_span, r_init, args = (radius, m1, m1*0.8),
 t_eval = t_arr)
```

Another way to do this is to have the output of `solve_ivp()` always set to the same variable name (e.g. `r_soln`) and to save the solution vector by copying it to another variable. Unfortunately, in some versions of Python with arrays one can't just use an equals sign to make a copy (e.g., `phi_07 = r_soln.y[0]`), or Python will think that there are now two names for the same underlying phi vector, and future modifications of `r_soln.y` will also modify `phi_07`. To be safe, use numpy.copy. For example, include the following line after solving for the $m_2/m_1 = 0.7$ case before solving for the $m_2/m_1 = 0.8$ case:

```
phi_07 = r_soln.y[0].copy()
```

Plot all of your solutions for ϕ versus t of the same axes and include a legend labeling them.

➤ *Hint:* To use `solve_ivp()` to solve for different values of m_2, you'll need to modify the "args" parameter in the function call.

Are these trajectories what you would expect? Explain why or why not with reference to the potential energy diagram. What is the effect of finite numerical accuracy on the behavior of the system with the critical ratio of m_2/m_1?

```
# < Exercise 12, Plotting the behavior for different m2/m1 >

### DO THIS: Copy and modify the above code to find the solutions
#    when the system is released from rest at phi = 0
#    for m_2/m1 = 0.7, m_2/m1 = 0.8. and the ratio found in Exercise 4

### Don't forget to save different versions of the solution vectors for each
#    case

### PLOT all three trajectories on the same axes
```

12.4.3 Computational solution for different release angles

Next you will use your numerical solver to compare oscillations of the system when released at a large angle to the same system released from a small angle.

Exercise 13

So far you have examined initial conditions that lead to oscillations. Now, using your solution from **Exercise 3**, try starting the system from rest at the unstable equilibrium.

In the cell below, copy your *relevant* code from above to call the numerical solver to find the solution for the behavior of the system released from rest from the unstable equilibrium.

After you have found the solution when starting the system from rest, try giving it a small initial angular velocity. $|\dot{\phi}| = 0.001$ rad s^{-1} will be sufficient, but consider what sign for $\dot{\phi}$ to use to produce the most interesting behavior. To do this, copy those lines of code to numerically integrate the system again and modify them for the different initial conditions.

Finally, plot ϕ versus time for both sets of initial conditions on the same set of axes. As in **Exercise 12**, you will need to save solutions for both cases. You can accomplish this by either setting the output of `solve_ivp()` to different variable names or by using `np.copy()`. You may need to adjust either your integration time or the range of the time axis in your plot to see the full behavior.

Using the information from the graph, describe the behavior of the system when started from the unstable equilibrium at rest and with some initial velocity.

```
# < Exercise 13, A non-oscillating solution >

### DO THIS: Copy and modify the above code to find the solutions for the
#    system when released from rest at the unstable equilibrium and when
#    released with some very small initial velocity
```

Exercise 14*

Let us return to conditions that lead to oscillating behavior.

> ***How do you expect the behavior of the system to change as the initial angle is increased slightly away from the stable equilibrium point? Explain your reasoning. You may find it helpful to recall the behavior of the simple pendulum explored in Simple Pendulum with Large-Angle Release.***

Exercise 15*

In the cell below, numerically solve for behavior of the system when it is released from rest at both 5 degrees and 40 degrees away from the stable equilibrium. As in **Exercise 13**, plot ϕ versus time for both cases on the same figure. You may find it interesting to also include the solution from the Taylor series approximation on your plots.

➤ *Hint:* If you are having trouble seeing the difference in the behavior of the two cases, try finding the solution over a longer time period. Twenty seconds is a good length.

```
# < Exercise 15, Plotting small- and large-angle solution >

### DO THIS: Copy and modify the above code to find the solutions for the
#    5-degree and 40-degree cases
#    Don't forget to save a copy of the solution vector of the 5-degree case
#    before solving for the 40-degree one

### DO THIS: Your plot here
```

You should see that the two solutions (small-angle and large-angle) get out of phase with one another. After 20 seconds, the 5 degrees release will go through a little over six and a half periods, whereas the 40 degrees release, running a bit slower, is closer to six periods.

Exercise 16*

Repeat the above exercise for a release angle of 66 degrees from equilibrium. Include an additional plot showing your 66 degrees solution versus the 5 degrees solution. This plot produces a *Lissajous* figure, useful for identifying how different frequencies relate to each other. You will encounter Lissajous figures again in Section 5.4 of *Classical Mechanics*.

What is the approximate ratio between the number of periods completed within 15.5 seconds for the 5-degree angle release versus the 66-degree angle release? How is this ratio apparent in the Lissajous figure? How would the Lissajous figure change if a slightly different pair of angles were compared?

➤ *Hint:* You should notice that this ratio is close to a rational number.

```
# < Exercise 16, Plotting small- and large-angle solution >

### DO THIS: Copy and modify the above code to find the solutions for the
#    5-degree and 66-degree cases.
#    Don't forget to save a copy of the solution vector of the 5-degree case
#    before solving for the 66-degree one



### DO THIS: Plot phi vs. t for both situations and phi vs. phi,
#    the Lissajous figure
```

12.5 Check-out

Exercise 17

Briefly summarize the ideas in this unit.

12.6 Challenge Problems

If you have time, consider completing one or more of the following problems:

- Determine the period as a function of release angle.
  - Using the zero-crossing detector and associated technique introduced in Simple Pendulum with Large-Angle Release, write a function to determine the period of oscillations around the stable equilibria.
  - Loop through different release angles between 1 and 89 degrees from the stable equilibrium and calculate the period.
  - Plot the period as a function of angle from 1 to 89 degrees and create a second plot that normalizes the period by the small-angle value (which should be what you found using a Taylor series approximation in **Exercise 9**).
- In **Exercise 16** you were asked to plot the Lissajous figure for the interesting case where the ratio between the periods for the small- and large-angle releases was close to a rational number. Such systems may be considered to be in *resonance* with each other. Using your

results from the first challenge problem, identify a different large released angle that will also result in resonance; plot the resulting Lissajous figure.

- Using a similar technique as in the first example problem, plot the period for the system when released from rest 5 degrees from the stable equilibrium for different values of m_2/m_1.

Damped and Undamped Harmonic Oscillators

13

Simple harmonic oscillators (SHO) and variations on them show up everywhere in physics. This is because, as you saw in Oscillations in a Potential Well, motion around a stable equilibrium can be approximated as simple harmonic motion. By becoming very familiar with them here, you will be well-positioned to work with them the next time they appear, for example, in quantum mechanics. In this unit, you will explore how the addition of damping affects harmonic motion.

This unit covers material from Ch. 5.5 and 5.6 of *Classical Mechanics*, including the creation of a resonance curve, similar to that in Figure 5.17.

13.1 Objectives

In this unit, you will analyze the equations of motion and model the behavior for:

- undamped (PHY);
- overdamped (PHY);
- critically damped (PHY);
- and underdamped oscillators (PHY).

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp  # the differential equation integrator

plt.rcParams["figure.figsize"] = (12,8) # make figures big for easier viewing
                                        #    (width, height in inches)
# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

13.2 Harmonic Motion without Damping

Consider the mass-spring problem from introductory physics:

$$F = ma = -kx,$$

so

$$\frac{d^2x}{dt^2} = -\frac{k}{m}x,$$

where the x values are functions of time.

13.2.1 Analytical solution

Assume a solution

$$x = A\cos(\omega t) + B\sin(\omega t).$$

Exercise 1*

On a separate piece of paper, do the algebra to find the analytical solution by substituting your assumed solution (and its derivatives) into the differential equation and determining the value of ω.

Then, make a Python plot of the analytical result for $x(t)$ and $v(t)$, assuming that $x_0 = 3$ m, $v_0 = 1$ m/s, $k = 8$ N/m, and $m = 2$ kg. See if you can show both x and v as different color lines on the same plot. Make the plot run for 10 seconds, with 10,000 points.

```
# < Exercise 1, Analytical solution, no damping >
```

13.2.2 Numerical solution

As usual, we convert this second-order equation into two coupled first-order equations by defining a velocity function, which is itself a first derivative of the position function:

$$v = \frac{dx}{dt}.$$

Then we seek simultaneous solutions to these two first-order equations:

$$v = \frac{dx}{dt},$$

$$\frac{dv}{dt} = -\frac{k}{m}x = -\omega_0^2 x.$$

Now, for each moment in time you will get solutions for x and v, which will be stored as the usual `r_now` for each time step. Following the conventions of the previous units using `solve_ivp()`, let x be the first variable in `r_soln.y[0]`), and v be the second variable in `r_soln.y[1]`).

Exercise 2

Modify the code below to calculate a numerical solution using `solve_ivp()` for the same initial conditions as your analytical solution from above, $x(t = 0) = 3$m, $v(t = 0) = 1$ m/s. When defining your `deriv()` function, note that it takes an optional parameter `omega0`. When `solve_ivp()` is executed, it passes along `omega0` to `deriv()`, via a command that looks like:

```
r_soln = solve_ivp(deriv, t_span, r_init, args = (omega0,))
```

where the comma after `omega0` in `(omega0,)` is syntactically necessary. Technically, the comma tells the program that `(omega0,)` is a one-element tuple. Remember, tuples are immutable (i.e., unchangeable) lists.

To ease the comparison with your analytical solution, pass `solve_ivp()` the parameter `t_eval` set to the same time array you used for the analytical solution. This means that, as before, your solution will be generated for 10 seconds, with 10,000 points.

The code below also sets the desired accuracy for `solve_ivp()` through the use of the optional parameters `rtol` and `atol`. These parameters set the maximum relative and absolute tolerance, respectively. Specifically, local error estimates are required to be less than `atol + rtol * abs(y)`. In the code, they have been decreased from the default values of 10^{-3} for `rtol` and 10^{-6} for `atol` to 10^{-11} to achieve an extremely accurate solution.

Plot $x(t)$ and $v(t)$ as different colored lines in the same figure.

```
# < Exercise 2, Numerical solution to SHO without damping, deriv function >

def deriv(t_now, r_now, omega0 = 1):
    """ Supplies the derivatives of position and velocity for an undamped
    harmonic oscillator to be used by the differential equation integrator,
    solve_ivp(). The optional parameter, omega0, is the natural frequency,
    which is set to one by default. """

    # unpack the variables
    pos_x = r_now[0]
    vel_x = r_now[1]

    # perform the derivatives
    dx_dt = 0  ### FIX
    dv_dt = 0  ### FIX

    return np.array([dx_dt, dv_dt])
```

```
# < Exercise 2, Numerical solution to SHO without damping >

# Global variables
k_spring = 8.0 #  [N/m] Spring constant
mass = 2.0 #  [kg] Mass of oscillator
omega0 = 0.0  ### FIX # [s^-1] Natural frequency

t_end = 1 ### FIX # [s] Final time
t_span = [0, t_end] # Time span

pos_x0 = 0 ### FIX # [m] Initial position
vel_x0 = 0 ### FIX # [m/s] Initial velocity

r_init = np.array([pos_x0 ,vel_x0])  # two-element array that gives the initial
                                     #    position and velocity
```

```
r_soln = solve_ivp(deriv, t_span, r_init, args = (omega0,), t_eval = t_arr,
                   rtol = 1.0e-11, atol = 1.0e-11)  # to maintain reasonable
                                                    #    accuracy, we need
                                                    #    to tighten these
                                                    #    tolerances (compare
                                                    #    error if you omit these
                                                    #    keywords).  If you are
                                                    #    patient, you could set
                                                    #    these to 1E-13 to
                                                    #    reduce the residual
                                                    #    further
```

```
# < Exercise 2, Plots of Numerical solution to SHO without damping >

### PLOT x and v versus t here
```

Exercise 3*

Check the accuracy of your code by comparing your numerical and analytical solutions. To do this, calculate the residual (difference) in position (`x_numerical - x_analytical`) and plot it as a function of time. If desired, you can control this accumulated error by reducing `atol` and `rtol` even further, taking smaller steps, or by applying a different integration method.

```
# < Exercise 3, Residuals >

### PLOT the difference between the analytical and computational positions
#    versus time
```

Exercise 4*

Use your numerical solution to make a plot showing the kinetic energy, potential energy, and total energy versus time. Just for fun, make a second plot showing velocity vs. position, called a "state space" plot.

Are these two plots what you would expect? Explain below.

```
# < Exercise 4, Plots of energy etc. for SHO >

### PLOT the kinetic energy, potential energy, and total energy vs. time and
#    the velocity versus the position
```

13.3 Damped Harmonic Oscillator

The damped harmonic oscillator is most easily modeled as a velocity-dependent drag force of the form

$$F_{damp} = -b\dot{x}(t),$$

which means that our equations of motion now look like:

$$F = ma = -kx - b\dot{x},$$

so

$$\frac{d^2x}{dt^2} = -\frac{k}{m}x - \frac{b}{m}\dot{x}.$$

We saw earlier in Chapter 2 of *Classical Mechanics* and Projectile Motion with Drag that a viscous damping force gave exponentially decaying solutions. In this situation, we expect some combination of exponentially decaying and oscillating solutions.

But first, anticipating the notation from *Classical Mechanics*, we define a new variable $\beta = \frac{b}{2m}$, and we can substitute the undamped natural frequency $\omega_0^2 = \frac{k}{m}$ so that

$$\ddot{x} + 2\beta\dot{x} + \omega_0^2 x = 0.$$

Now given Euler's theorem, $e^{it} = \cos(t) + i\sin(t)$, we may compactly write solutions in the form:

$$x(t) = e^{rt}.$$

Exercise 5

On a separate sheet of paper, substitute this trial solution, and solve (analytically) the resulting quadratic equation, yielding a pair of solutions:

$$r_{\pm} = -\beta \pm \sqrt{\beta^2 - \omega_0^2}.$$

The complete solution can then be found by matching initial conditions in the form

$$x(t) = C_1 e^{r_+ t} + C_2 e^{r_- t}.$$

We now consider the four cases of no damping, large damping, critical damping, and small damping.

13.3.1 Case 1: No damping, $\beta = 0.0$

Exercise 6*

On a separate sheet of paper, show that setting $\beta = 0$ is identical to the undamped case, recalling that $\sqrt{-1} = i$.

Exercise 7

You should already have plotted this case above. For completeness, though, edit the code below to solve the damped oscillator numerically for a given value of β, and then set the value of β to zero. In later sections, you will change the value of β, so make sure your newly defined `deriv_damp()` function is set up correctly to take both `omega0` and `beta_damp` as

optional parameters. Run your solution from $t = 0$ s to $t = 40$ s with 10,000 steps by defining an array of times and passing that array to `solve_ivp()` through `t_eval`. Keep the same initial conditions and values for k and m.

Plot $x(t)$ and $v(t)$ on the same figure using different line colors. It will also be handy for you to save a copy of the undamped solution $x(t)$ for comparison with the other cases, so include the following line at the end of your code to save your solution:

```
undamped_x_solution = r_soln.y[0].copy()
```

```
# < Exercise 7 No damping, define deriv_damp >

def deriv_damp(t_now, r_now, omega0 = 1, beta_damp = 0):
    """ Supplies the derivatives of position and velocity for a damped
    harmonic oscillator to be used by the differential equation integrator,
    solve_ivp. The optional parameters and their defaults are  (omega0 = 1)
    and (beta_damp = 0). """

    # Unpack the variables
    pos_x = r_now[0]
    vel_x = r_now[1]

    # Perform the derivatives
    dx_dt = 0  ### FIX
    dv_dt = 0  ### FIX

    return np.array([dx_dt,dv_dt])
```

```
# < Exercise 7, No damping >

t_start = 0
t_end = 1 ### FIX # [s], Final time
t_span = [0, 0] ### FIX  [s], Time span
t_arr = np.linspace(t_start, t_end, 1000)  ### FIX Array of times for which to
                                           #    solve


pos_x0 = 3.0  # [m] Initial position
vel_x0 = 1.0  # [m/s] Initial velocity
r_init = np.array([pos_x0, vel_x0])  # set initial conditions
# The damping parameter. In later exercises, you will change this value
beta_damp = 0.0

# Call solve_ivp with optional parameters for the natural frequency and damping
r_soln = solve_ivp(deriv_damp, t_span, r_init, args = (omega0, beta_damp),
                   t_eval = t_arr,
                   rtol = 1.0e-11, atol = 1.0e-11)

### CODE to save undamped solution here
```

```
# < Exercise 7, Plot no damping case >

### CODE to plot x(t) and v(t) here
```

13.3.2 Case 2: Large damping, $\beta^2 > \omega_0^2$

We call this the *overdamped* case, and it has exponentially decaying solutions. For example, when you opened the door to this room, you may have had to do work against a door-closing spring. In many doors when you let go, a pneumatic or hydraulic soft-closing mechanism makes the door overdamped up until the moment that it needs to latch shut, whereupon the damping is suddenly reduced (a valve opens) to snap the door closed.

Note that in the overdamped case you get two solutions as usual,

$$r_{\pm} = -\beta \pm \beta\sqrt{1 - \left(\frac{\omega_0}{\beta}\right)^2}.$$

The overall shape will be determined by the *more-slowly* decaying exponential.

Exercise 8

For a concrete illustration of the overdamped case, let $\beta = 10\omega_0$ and numerically calculate and plot the position and velocity versus time. Because you have already defined your time array and your `deriv_damped()` function in the cell above (and your initial conditions and values for k, m, and ω_0 much earlier), you do not need to replicate that code. However, you will need to include a line of code to set β and call the integrator

```
r_soln = solve_ivp(deriv_damp, t_span, r_init, t_eval = t_arr,
 args = (omega0, beta_damp))
```

to solve the equations of motion, in addition to writing code to plot *x(t)* and *v(t)*.

```
# < Exercise 8, Overdamping, beta_damp = 10*omega0 >

### ADD CODE to solve the overdamped case

### ADD CODE to plot position and velocity vs. time
```

Exercise 9

On a separate sheet of paper, analytically calculate the timescale over which the exponential decays, assuming $\beta = 10\omega_0$. To do this, note that $(\frac{\omega_0}{\beta})^2$ is « 1. Taylor expand the square root in the expression for $r_{\pm}$ to first order (you'll see that this is equivalent to a first-order binomial approximation). Now compare the two solutions. You will see that one of them is much more highly damped than the other. Assuming one of them dies off really quickly, the other one must determine the overall shape of $x(t)$, that is, r_+ dominates. Solve for the *time constant* τ, which is the time that it takes a decaying exponential to fall to 1/e of its initial value.

➤ *Hint: It should be about 10 seconds — show this, and look for it on your plot above, where* $1/e \approx 0.37$.

τ has units of seconds, and it is a measure of the timescale over which the amplitude falls off. You may have encountered in a previous physics course an identical term while discharging a capacitor through a resistor, where $Q(t) = Q_0 e^{-t/\tau}$ represents the charge on the plates of the capacitor, and $\tau = RC$ is the RC time-constant.

τ =

Is this value of τ consistent with your plot?

13.3.3 Case 3: Critical damping, $\beta^2 = \omega_0^2$

When $\beta^2 = \omega_0^2$, this leads to a double root in the characteristic equation, which requires solutions of the form

$$x(t) = Ae^{-\beta t} + Bte^{-\beta t}.$$

This *critical damping* case is the quickest way to get an oscillator to shed its energy. Car suspensions employ struts or shock absorbers that are designed to be close to critically damped for the average passenger load.

Exercise 10

Numerically calculate the solution to the critical damping case for the same initial conditions as before, and plot `x(t)` and `v(t)`. Use the same initial conditions, time span, and natural frequency as before.

From your graph, about how long does it take for the amplitude to decrease by a factor of 1/e?

```
# < Exercise 10, Critical damping >

### ADD CODE to solve the overdamped case

### ADD CODE to plot position and velocity vs. time
```

Note that for the initial conditions $x_0 = 3$ m and $v_0 = 1$ m/s, the oscillator starts at 3 m and approaches the origin $x = 0$, reaching it in the limit of t approaching ∞. For other cases, though, the oscillator may first overshoot, crossing through $x = 0$, reapproach it, and finally reach 0 in the limit as t approaches ∞.

Exercise 11*

Find one possible pair of A and B for which the critically damped oscillator hits $x = 0$ before $t \to \infty$, and solve analytically for the time at which the oscillator will first hit $x = 0$. To check your calculations, numerically solve for and plot $x(t)$ and $v(t)$ given those initial conditions.

Justify your solution ***below***. (Non-trivial solutions, please, so you are not allowed $A = 0$, $B = 0$.)

```
# < Exercise 11, Critical damping with zero-crossing >

### REDO your plot for critical damping, but now with the pair of A, B values
#    that you listed above
```

13.3.4 Case 4: Underdamping, $\beta^2 < \omega_0^2$

The small damping case is mathematically the most interesting because the system oscillates and decays. The general result for real solutions can be matched to

$$x(t) = e^{-\beta t}(A\cos(\omega_d t) + B\sin(\omega_d t)),$$

with A and B real constants, chosen to fit the initial conditions.

Exercise 12

For a concrete illustration of the underdamped case, keep $\omega_0 = 2\ \mathrm{s}^{-1}$ and let $\beta = \frac{\omega_0}{10}$. On a separate sheet of paper, calculate the time at which the amplitude will have fallen to its 1/e point (roughly 1/3 of its initial amplitude). Calculate, as well, the frequency of the oscillations, ω_{damp}.

Time when $A = A_0 e^{-1}$***:***
Frequency of oscillation:

Exercise 13

Numerically calculate the solution to the underdamping case for the same initial conditions and natural frequency as before ($x = 3$ m, $v = 1$ m/s, $\omega_0 = 2\ \mathrm{s}^{-1}$) and $\beta = \frac{\omega_0}{10}$. Plot $x(t)$ and $v(t)$.

In order to compare the period of the oscillations to the undamped case, overlay your saved solution for $x(t)$ for zero damping.

Justify that your plots match your results from Exercise 12.

```
# < Exercise 13, Underdamping >

### ADD CODE to solve the underdamped case

### ADD CODE to plot position and velocity vs. time for underdamped case.
#    INCLUDE position vs. time for the zero damping case
```

13.4 Check-out

Exercise 14

Briefly summarize the ideas in this unit.

13.5 Challenge Problems

If you have time, consider completing one or more of the following problems:

- For the underdamped case, how big does β have to be to make ω_{damp} 2% less than the frequency of the undamped oscillator? You should be able to solve this problem quickly by using the first-order Taylor series approximation in exactly the same way as you approached **Exercise 8**, though in this case ω_0 is the term you pull out of the radical, and you are expanding the radical for small β. Do not feel that you need to do this numerically!
- Make plots of velocity vs. position for the underdamped, overdamped, and critically damped oscillators.

Driven Damped Harmonic Oscillator and Resonance

14

In this unit you will build on the work you did in Damped and Undamped Harmonic Oscillators to characterize and model damped harmonic motion. Specifically, you will add a sinusoidal forcing term in order to measure the resonant response at different driving frequencies. This unit covers material from Ch. 5.5 and 5.6 of *Classical Mechanics*, including the creation of a resonance curve, similar to that in Figure 5.17.

14.1 Objectives

In this unit, you will learn about:

- the long-term and transient motion of driven damped harmonic oscillators (PHY);
- the phase-shift between the oscillator and the driver (PHY);
- the amplitude of the resulting motion due to different driving frequencies (PHY);
- the effect of damping on the resonance curve (PHY).

The first and third challenge problems also provide an introduction to generator expressions and multiprocessing. This content is motivated by the resonance curves for **Exercises 10** and **12** taking a few minutes to compute. If instruction in multiprocessing computing is particularly desired, **Exercises 7** and **8** could easily be omitted to save time.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp  # the differential equation integrator

plt.rcParams["figure.figsize"] = (12,8)  # make figures big for easier viewing
                                         #    (width, height in inches)
# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

14.2 Motion of a Driven Damped Harmonic Oscillator

Exercise 1

You will be modifying your code from Damped and Undamped Harmonic Oscillators. Therefore, start by copying and pasting your code to model a damped harmonic oscillator into the cell below. Make sure it runs as you would expect by regenerating one of your plots.

➤ *Hint:* You will probably find most of the code you need in your response to **Exercise 7** of Damped and Undamped Harmonic Oscillators. However, you may still need to define some constants of the system (e.g., m and ω_0).

```
# < Exercises 1 and 2, Numerical solution to SHO with damping and
#    later driving >

# Deriv function definition
```

```
# < Exercises 1 and 2, Numerical solution to SHO with damping and
#    later driving >

# ADD code to numerically calculate the solution
```

```
# < Exercises 1 and 2, Plot numerical solution to SHO with damping and
#    later driving >

### ADD code to plot x(t) and v(t) here
```

Exercise 2

Modify your code above (the damped harmonic oscillator) to include a driving term of the form $F_0\cos(\omega t)$, where ω is the driver frequency. Make the natural frequency, damping term, drive frequency (ω) and the amplitude of the driver (F_0/m) parameters in your `deriv()` function, similarly to how the natural frequency and damping term were incorporated in Damped and Undamped Harmonic Oscillators.

Start your system from rest at the equilibrium point (i.e., $x = 0$, $v = 0$), and let the driving frequency be the same as the natural frequency (i.e., $\omega = \omega_0$). Let $m = 2$ kg, $k = 8$ N/m, $\beta = 0.1$, and the size of the driver force, F_0, equal 0.2 N.

Run your integrator for 60 seconds (recalling that $\beta = 0.1$ corresponds to $\tau = 10$ seconds, so you will see the long-term behavior by going out beyond 5τ, where more than 99% of the transient term has died away).

Graph $x(t)$ and overlay a copy of the driver, via

```
ax.plot(r_soln.t, f_0*np.cos(omega*r_soln.t))
```

You should find that the amplitude of the long-term motion is about 0.25 m if $\omega = 2$ s^{-1}.

Exercise 3

Answer the following questions about the motion of the damped driven oscillator ***below***. Make adjustments to your code to explore the different situations. When you are done, reset your code to the parameters outlined in **Exercise 2**.

- What happens to the long-term and short-term motion if you change the initial position, x, and/or initial velocity, v?

- Looking at your figure, what do you observe about the relationship between the driver waveform and the position waveform? (We call this the *relative phase* of the motion with respect to the driver.)

- If you change the frequency to be 0.75 ω_0, what happens to the relative phase? What happens to the long-term amplitude of the motion?

- If you change the frequency to 1.25 ω_0, what happens to the relative phase? What happens to the long-term amplitude?

14.3 Amplitude of Driven Damped Oscillations at Resonance

14.3.1 Maximum steady-state amplitude, determined analytically

You should have found that this driven oscillator reaches a steady-state amplitude that is largest when driven at its natural frequency $\omega = \omega_0$. We call this the *resonance frequency*. Let us find out how large this amplitude becomes (you can find a version of the following derivation in Section 5.5 of *Classical Mechanics*).

Your equation of motion is

$$x'' + 2\beta x' + \omega_0^2 x = \frac{F_0}{m}\cos(\omega t).$$

We look for solutions that have the same frequency as the driver (because if the solutions were at some other frequency, there is no way we could balance this equation). So let us assume $x(t) = Ae^{i\omega t}$, with A a real constant. With no loss in generality, we can consider the driver to look like $F_0 e^{i\omega t}$.

Then the equation of motion is

$$(-\omega^2 A + 2i\beta\omega A + \omega_0^2 A)e^{i\omega t} = \frac{F_0}{m}e^{i\omega t},$$

or

$$(\omega_0^2 - \omega^2 + 2i\beta\omega)A = \frac{F_0}{m}.$$

At resonance ($\omega = \omega_0$), considering only the amplitudes of the real and imaginary parts, we see that

$$2\beta\omega A = \frac{F_0}{m}.$$

So we expect in the steady state that the amplitude

$$A = \frac{F_0}{2\beta m\omega}.$$

Alternatively, we could note that when the driving force matches the damping force (i.e., $F_0 = bv$), we must be in equilibrium, where the average work done by the driver exactly matches the average energy loss due to the damping force. Recalling that $|2\beta x'| = b(\omega A)/m = F_0/m$, we get $A = F_0/(b\omega) = \frac{F_0}{2\beta m\omega}$.

Exercise 4

Compare your numerical result at resonance ($\omega = \omega_0$) with the predicted steady-state amplitude for the system described in **Exercise 2**. Then see what happens if you 1) triple the force term or 2) double the damping parameter. Include your analysis ***here***. When you are done, reset the force term and damping parameter to the values from **Exercise 2**.

Exercise 5

Change your driver frequency to $1.08\omega_0$ and include the plot of $x(t)$ for the mass and the driver below. Then answer the following questions ***here***.

- Approximately at what time does the oscillator amplitude hit its peak value?

- If you want the steady-state value of the oscillator amplitude, when should you start looking at it?

- What's the value of e^{-5}?

- Why should we wait at least 50 seconds in this example?

```
# < Exercise 5, omega = 1.08 omega_0 >

# INCLUDE code for integration and plots
```

14.4 Phase of Driven Damped Oscillations

14.4.1 Relative phase of $x(t)$ and $f(t)$, determined analytically

What's with the phase of the oscillations? The easy way to see the relative phase of the $x(t)$ solution above and below resonance is to ignore damping altogether (or make sure we are very far from resonance).

Let us ignore damping for a moment and look at the driven oscillator equation of motion: $(\omega_0^2 - \omega^2)A(\omega) = \frac{F_0}{m}$, where $A(\omega)$ represents the steady-state amplitude of the oscillation for a given driver frequency ω. Solving for $A(\omega) = \frac{F_0/m}{(\omega_0^2-\omega^2)}$ shows that if $\omega < \omega_0$, the amplitude has the same sign as F_0, and so the motion of the mass is *in phase* with the driver. If $\omega > \omega_0$, then the amplitude has the opposite sign as F_0, and so the motion is *out of phase* with the driver.

Exercise 6

Check the relative phases of $x(t)$ and $F(t)$ for your $\omega = 0.75\omega_0$ and $\omega = 1.25\omega_0$ curves. Explain ***here*** whether your curves match the above prediction.

So, below resonance the mass is in phase with the driver, and above resonance the mass is out of phase with the driver. At resonance, what happens?

That question is hard to answer in the absence of damping, because the amplitude goes to ∞. But you can make a pretty bold guess that it lies between 0 degrees and 180 degrees relative phase. Does your numerical solution for $\omega = \omega_0$ provide a clue? Explain ***here***.

14.5 The Phase Dependence

By including damping in our equation of motion, we can come up with an expression for the difference in phase between $x(t)$ and $F(t)$, which we will call $\delta(\omega)$. The derivation follows (and can also be found in Section 5.5 of *Classical Mechanics*).

Starting with our general equation of motion

$$(\omega_0^2 - \omega^2 + 2i\beta\omega)A = \frac{F_0}{m}$$

and solving for $A(\omega)$ yields:

$$A(\omega) = \frac{\frac{F_0}{m}}{(\omega_0^2 - \omega^2 + 2i\beta\omega)}.$$

This amplitude can be represented as a vector in the complex plane, with a real part along the x-axis and an imaginary part along the y-axis. The magnitude of the vector is given by

$A \cdot A^*$, and the angle between the x-axis and the A vector can be found from

$$\tan(\delta) = \frac{Im(A)}{Re(A)} = -\frac{2\beta\omega}{(\omega_0^2 - \omega^2)}$$

It is traditional to multiply the right side by -1 and then interpret this quantity as the tangent of the phase *lag* of the oscillator with respect to the driver:

$$\tan(\delta) = \frac{Im(A)}{Re(A)} = \frac{2\beta\omega}{(\omega_0^2 - \omega^2)}$$

Exercise 7*

Plot this analytically derived phase lag δ as a function of frequency. Assume our usual $\omega_0 = 2, \beta = 0.1$. Look over a range $\omega = [0, 2\omega_0]$. You will need a lot of points (try 100,000) to make this a symmetrical plot.

```
# < Exercise 7, Plot phase lag as a function of frequency near resonance >
```

If you used `np.arctan()` to calculate δ, your plot of phase lag vs. frequency probably looks a bit funky. That is because we would like the angle to lie in the range [0,π], rather than [-π/2, π/2]. We need to fix our plot to make it look like Fig. 5.19 from *Classical Mechanics*, p. 192. You could adjust all negative values of δ by adding π. Even easier, though, is to replace `np.arctan()` with numpy.arctan2, which maintains the correct quadrant for the solution. The syntax to call np.arctan2 is `np.arctan2(y, x)`.

Exercise 8*

Replot δ vs. ω below and answer the following questions ***here***:

- What's the phase lag when the oscillator is driven at resonance?

- What's the phase lag well below resonance?

- What's the phase lag well above resonance?

```
# < Exercise 8, Plot of phase lag, plotted [0 - pi] >
```

14.6 Amplitude as a Function of Driver Frequency

14.6.1 Analytical solution to the amplitude of a damped, driven harmonic oscillator

We solved for the steady-state solutions analytically, but it is a little inconvenient to plot the complex amplitude:

$$A(\omega) = \frac{\frac{F_0}{m}}{(\omega_0^2 - \omega^2 + 2i\beta\omega)}.$$

Instead, multiply by the complex conjugate and then take the square root to get the magnitude:

$$|A(\omega)| = \frac{\frac{F_0}{m}}{\sqrt{(\omega_0^2 - \omega^2)^2 + (2\beta\omega)^2}}.$$

Exercise 9*

Use the above analytical expression to plot the amplitude as a function of ω to observe the resonance. (This is probably most neatly done by creating an array of frequencies over which you want to plot, then making an analytical function that is evaluated at these frequencies. Pay particular attention to all the squares and square roots.) If you would like to compare your curve with Figure 5.17 on p. 190 of *Classical Mechanics*, plot the amplitude squared versus ω, as well.

```
# < Exercise 9 Analytical plot of amplitude vs. frequency >

# Outline of code to make a resonance curve from analytical expression.

# 201 different values of omega from 0.9 to 1.1
omega_arr = np.linspace(0.90*omega0, 1.10*omega0, 201)

def calculate_ampl_analyt(omega, f_0, mass, omega0, beta_damp):
    """ Calculate the amplitude analytically for a particular value of omega"""
    ampl_analyt = 1.0*omega  ### FIX this expression

    return ampl_analyt

# Use the function above to calculate the amplitudes for an array of omega
#    values
model = calculate_ampl_analyt(omega_arr, f_0, mass, omega0, beta_damp)

# INCLUDE code to plot amplitude versus omega below
```

14.7 Creating a Numerical Resonance Curve by Plotting the Maximum Amplitude vs. Frequency

The goal for this section is to replicate the resonance curve you just plotted analytically by solving the driven oscillator numerically. Doing so will provide proof that the analytical expression for the amplitude of a driven damped harmonic oscillator is correct. To do this, run your integrator for each of the 201 values in `omega_arr`. At each ω step you will need to run the integrator out until the amplitude reaches a steady state. Find and store this final maximum amplitude each time. Remember, all we save from each run is the maximum steady-state amplitude. You should end up with an array of 201 points, each of which represents the steady-state amplitude A(ω).

Exercise 10

Create a loop in order to find $|A(\omega)|$ for $0.9\omega_0 < \omega < 1.1\omega_0$ and examine the resonance by plotting $|A(\omega)|$ versus ω. Use these parameters: $m = 2$ kg, $k = 8$ N/m, $\beta = 0.1$, $F_0 = 0.2$.

You may recall from previous units how to make an empty array, then append solutions to it using numpy.append. Remember that `np.append()` makes a new array and so must be assigned back to itself, like so:

```
max_amp_arr = np.array([]) # create empty array from an empty list
```

then in your loop, after you have found `latest_max_amp_value`

```
max_amp_arr = np.append(max_amp_arr, latest_max_amp_value)
```

In this case, you will make an empty array to store the different amplitudes. Then, loop through various frequencies (say, from 0.9 ω_0 to 1.1 ω_0), and numerically solve for the trajectory. Finally, you should extract the highest features after the integration has run for a long time (say, after the time has encompassed at least five time constants). That means you need to go at least $\frac{5}{\beta}$ before you look for the steady-state amplitude. To do so, you will need to run your integration for at least 50 seconds to let all the transient behavior die away and then let it run sufficient extra time to capture the steady-state motion. Hence 60 seconds is about right.

- *Hint 1:* Recall that `a_arr.max()` identifies the maximum value in `a_arr`. In the cell below is a reminder of how you might find the largest value in the *second half* of an array.
- *Hint 2:* When debugging code, it is often useful to print values from within an array. However, in long loops, it is easy to fill up the screen with output. The second cell block below shows a way to format output so that each print statement is separated by a space, rather than a new line. It also includes a command, `time.sleep()`, that makes the program pause for a set amount of time. This pause makes it easier to visualize the loop but should be removed when running production code.

```
# Hint 1: An example to illustrate how to find the largest element in the last
#    third of an array:
```

```
a_arr = np.array([0, 1, 20, 3, 4, 3, 6, 4])
print("The whole array is:  ", a_arr)

# Find the largest element in the entire array
max_a = a_arr.max()
print("The value of the largest item in the entire array: ", max_a)

# Find the largest element in the last third of the array
last_third_index = len(a_arr)*2//3  # the double // slash indicates an integer
                                    #    division, which truncates the
                                    #    result to allow for easy indexing of
                                    #    the array.  Note that we do the
                                    #    multiplication by 2, then integer
                                    #    divide by 3.  If you were to do
                                    #    2//3 first, you'd get 0
print("Start looking at index {:}".format(last_third_index))
# Restrict the search to the last third of the array
max_b = a_arr[last_third_index:].max()
print("The largest element in the last third of the array:  ", max_b)
```

```
# Hint 2: A handy thing to know:  if you want to print the current value of
#    omega_arr without filling up the screen, you can modify the print
#    command to give a couple of spaces as a separator (this replaces the \n
#    newline character).

# Import the sleep method to allow you to pause calculations for a fixed time
#    (in seconds)
from time import sleep
arr = np.linspace(0, 10, 201) # Example array
for item in arr:
    # Pause for 0.02 seconds before continuing on.
    #    Doing so makes it easier to visualize the loop.  Remove for real
    #    calculations!
    sleep(0.02)
    print ("{:5.2f}".format(item), end = " ")  # Keep track of current omega,
                                               #    so you know you're not
                                               #    stuck. The end = " " avoids
                                               #    a newline character at the
                                               #    end
```

```
# < Exercise 10, A(omega) for beta = 0.1 >

# This calculation takes a bit of time to run. One of the challenge problems
#    will show you a way to speed it up. In preparation for that challenge
#    problem, you will time how long it takes for the calculation to run.
#    To do this, we will use the library "time", which returns the current time
#    clock time. By comparing the clock time at the start and the end of the
#    calculation, the time for calculation can be determined.
```

```
import time

time_start = time.time()  # initial time

### ADD your code to loop through values of omega, use solve_ivp to find x(t),
#    and store the maximum amplitude after transients have died off



# Time for the calculation is the final time minus the initial time.
run_time = time.time() - time_start
print("\nDone! Calculation took {:3.2f} seconds".format(run_time))
```

```
# < Exercise 10, A(omega) for beta = 0.1, Plots >

# ADD your code to plot amplitude versus omega
```

14.7.1 How are the widths of the peaks and their heights related for small damping?

The analytical solution showed us that the height of each peak falls a factor of 2 for each factor of 2 increase in the damping. What does the width of each peak do?

The easy way to answer this question is to look at the amplitude squared:

$$A^2(\omega) = \frac{\left(\frac{F_0}{m}\right)^2}{(\omega_0^2 - \omega^2)^2 + (2\beta\omega)^2}.$$

At resonance the denominator becomes $(2\beta\omega)^2$ and A^2 is at its peak value. A^2 reaches half of that peak value when ω is such that

$$(\omega_0^2 - \omega^2)^2 = (2\beta\omega)^2,$$

i.e., the two terms in the denominator are equal.

Exercise 11*

On a separate piece of paper, work through the following derivation for width of the resonant curve.

Let us look very close to the resonance by defining a new frequency $\Delta \equiv \omega - \omega_0$, so we can write $\omega = \Delta + \omega_0$.

First, take the square root of each side, so

$$(\omega_0^2 - \omega^2) = \pm 2\beta\omega$$

Substitute for ω and *expand* the square. Then look at your result: anything multiplied by ω_0 is much larger than a term like $\beta\Delta$. Simplify, and show that:

$$\Delta \approx \pm\beta.$$

14.7.2 Now for the Geometric Interpretation

The maximum A^2 grows inversely with β, whereas the width of the peak grows proportionally to β.

Consider: if you were to plot A^2 vs. ω for differing β, what quantity stays approximately the same?

The ***Q*** of a damped driven oscillator is a figure of merit (*quality factor*) defined as

$$Q = \frac{\omega_0}{2\beta} = \frac{\omega_0}{2\Delta}.$$

Q can be thought of as an *amplification factor*, whereby small motions in the driver get turned into large excursions of the mass at resonance. For example, at resonance in the steady state, the driver supplies a force of magnitude $F_0 = -kx_0 = -m\omega_0^2 x_0$, so $|A(\omega = \omega_0)| = \frac{\omega_0}{2\beta} x_0 = Qx_0$. Therefore, the greater the value of Q, the greater the amplitude at resonance.

Experimentally, if you plot the amplitude squared vs. ω, the FWHM (full width at half maximum) of the peak is 2β, and the Q of the resonance is $Q = \frac{\sqrt{A^2}}{FWHM}$.

Exercise 12*

If desired, you can see how the resonance curve changes with β by attempting the second challenge problem. For now, though, simply recreate your resonance curve from **Exercise 10** for $\beta = 0.05$ and verify by eye that the full width at half maximum and the peak amplitude change by the amount you would expect. Explain ***here***.

```
# < Exercise 12, A(omega) for beta = 0.05 >

### ADD your code to loop through values of omega, use solve_ivp to find x(t),
#    and store the maximum amplitude after transients have died off
```

```
# < Exercise 12, A(omega) for beta = 0.05, Plots >

# ADD your code to plot amplitude versus omega
```

14.8 Check-out

Exercise 13

Briefly summarize in the cell below the ideas in this unit.

14.9 Challenge Problems

If you have time, consider working through the following challenge problems. They will introduce you to multiprocessing computing (1 and 3) and the effect of damping on the

resonance curve (2 and 3). Challenge Problem 2 may be completed on its own, but challenge Problem 3 must be preceded by Problem 1 and, ideally, Problem 2.

14.9.1 Challenge Problem 1: Multiprocessing

You probably noticed that creating a numerical resonance curve took a bit of time. One way to speed up calculations such as this is to use multiple computer cores at once so that different parts of the problem can be run at the same time, a.k.a., in parallel. Luckily, Python makes this method fairly straightforward. This challenge problem will walk you through the process. You will find the associated code in the cell below.

By using the multiple cores (CPUs) in the PC, calculations can be sped up considerably, depending on the type of program (this one is "CPU bound," because the rate-limiting step is calculations; another program may take time to print or plot, or get data from the Internet — these would be input/output, "I/O bound"). Nowadays, CPUs use "hyperthreading", which means that the core count will be the number of virtual cores. Often, the optimal number of cores to run a CPU-bound process is roughly half those reported. (If the process is I/O bound you can actually ask for many more cores than available.)

To use multiple processors on this problem, we need to have an external file called `resonance_for_multi_processingSolvers.py` that contains a function `solver()` of one variable (a two-element tuple containing `omega_arr` and `beta_damp`; notice that this is similar to how you send parameters to `solve_ivp()`) and returns the final amplitude for that integration. The external file needs to be self-contained, so it has to import `numpy` and `solve_ivp()`, and also the `deriv()` function. Then, instead of a loop structure to loop over the omegas, we map them onto the function. This allows the pool of processors to divide up the job. When finished, the call will return the `max_amp_arr` (array of steady-state maximum values) as `output`.

In order to generate the list of tuples of β and ω values, a "generator expression" is used. A generator expression is a concise way to create a list of values, similar to what could be accomplished through a loop but using only a single line of code. The syntax is

```
(expression for item in iterable)
```

where `expression` is the value that will be returned for each `item` contained in `iterable`. See if you can identify the generator expression in the example code.

Start by locating `resonance_for_multi_processingSolvers.py` and moving it to the same directory as this Jupyter Notebook (this would require uploading it to CoCalc or to your "Colab Notebooks" directory on Google Drive if you are using Google Colab). Next, open `resonance_for_multi_processingSolvers.py` and read through it. It should contain code that looks very similar to your work in **Exercise 2** and **Exercise 10**. Then use the code provided in the cell below to calculate the resonance curve for $\beta = 0.1$ and plot $A(\omega)^2$ versus ω.

How much faster was your calculation than in Exercise 10? (You may also wish to explore the speed-up when you use different numbers of cores. Do not expect the run time to be divided by the number of cores employed; the CPU typically shuffles a computationally intensive job among the various cores to keep the chip from overheating.)

```
# FOR GOOGLE COLAB ONLY

# < Challenge problem 1, Use multiprocessing to determine a resonance curve >

### UNCOMMENT the following lines if you are using Google Colab

#from google.colab import drive ## Mount your Google drive
#drive.mount('/content/drive/')

#import sys  # Append your Google drive "Colab Notebooks" directory to your
             #    path so that files contained in it may be found
#sys.path.append('/content/drive/My Drive/Colab Notebooks/')
```

```
# < Challenge problem 1, Use multiprocessing to determine a resonance curve >

# Use multiprocessing to plot a resonance curve for a single value of beta_damp
#    (0.1).

import multiprocessing  # import library to allow for multiprocessing
from multiprocessing import Pool
# import file containing the solver (look inside
#    resonance_for_multi_processing_solvers.py)
import resonance_for_multi_processing_solvers
import time  # import library that will allow us to time the process

num_processors = multiprocessing.cpu_count()
print("This computer has {:} processors".format(num_processors))
n_cores = int(np.ceil(num_processors/2))  # Use half the available cores,
                                          #    rounded up
print("This program will use {:} processors".format(n_cores))

# All parameters for solve_ivp() except for the driving frequency (the variable
#    to be looped over) and beta_damp are placed in the external file and can
#    be global or contained within solver(). We need these parameters here to
#    set up the range of omega_arr driving frequencies
mass = 2.0 # [kg] mass
k_spring = 8.0 # [N/m] spring constant
omega0 = np.sqrt(k_spring/mass)

omega_arr = np.linspace(0.90*omega0, 1.10*omega0, 201)  # an array that samples
                                                        #    every 0.002 in
                                                        #    frequency
beta_damp = 0.1

# Define a list of tuples, each one containing a value for beta_damp and omega
### DO THIS. Make sure you understand the syntax of the line below. Then print
#    the variable seq so that you can see the list of tuples.
seq = [(beta_damp, omega_item) for omega_item in omega_arr]
```

```
time_start = time.time()  # initial time
if __name__ ==  '__main__':
    # Ask the computer to select a number of processers
    p = Pool(processes = n_cores)
    # Tell the pool of processors, p, to call the function solver in
    #    resonance_for_multi_processing_solvers.py for each (beta_damp, omega)
    #    tuple contained in the list, seq.
    output = p.map(resonance_for_multi_processing_solvers.solver, seq)

run_time = time.time() - time_start
print("Done! Calculation took {:3.2f} seconds".format(run_time))
```

```
# < Challenge problem 1, Plots >

### EXAMINE the returned value from
#    p.map(resonance_for_multi_processing_solvers.solver, seq), "output".
#    Then plot the amplitude versus omega.
```

14.9.2 Challenge Problem 2: Effect of β on the resonance curve

Plot on the same axes a series of five resonance curves in which you vary β from $\beta = 0.16$ (heavy damping) all the way down to $\beta = 0.01$, with a factor of two difference in β for each curve. Use the usual parameters, with a fixed driver of force amplitude $F_0 = 0.2$, so illustrating the effect of damping on the width of the resonance.

Rather than manually running your $|A(\omega)|$ solution five times separately, make an outer `for` loop that changes β and an inner loop that changes ω. In computer science terminology, you'll be making a *nested for loop* structure. Each time you get a resonance curve, plot it (so the plot command should be placed just outside of the inner omega loop). The easiest way to do the nesting is to copy your working numerical resonance curve from above into a new outer loop that iterates over a β array. Make sure you indent the nested `for` loops properly.

Time-saving idea: Instead of running each simulation on the same time array, adjust the total time for each integration to be 6τ, allowing the transient to die off. (So you'll be redefining the time array within the outer loop, just after you set the latest value of β.) Look for the maximum value of $|A(\omega)|$ after 5τ has elapsed, i.e., from 5/6 of the way through the $x(t)$ array, all the way to the end. *Then each doubling of β ought to run twice as fast!*

➤ *Hint:* Because bigger values of β run faster, start your beta array from 0.16 and make your last value 0.01.

If you get bored waiting for the lowest values of β to finish, go ahead and start Challenge Problem 3, which will show you how to use multiprocessors to speed it up.

- ***Compare the peak values at resonance for different damping parameters to the analytic expression from** Exercise 4**. Do your results agree?***

- ***Compare the widths of the curves to the analytic expression for them from** Exercise 11**. Do your results agree?***

```
# < Challenge problem 2, Show a set of five damping curves, 2x apart >

r_init = np.array([0.0, 0.0])
omega_arr = np.arange(1.95, 2.050, 0.002)
beta_arr = np.array([0.16, 0.08, 0.04, 0.02, 0.01])
```

14.9.3 Challenge Problem 3: Multiprocessing for multiple resonance curves

Use multiprocessing computing to speed up your calculations for **Challenge Problem 2**. To do this, you will follow the same steps as in **Challenge Problem 1**, but use different values of β.

There is one difficulty to this Challenge Problem. We want to run over the `omega_arr` at each value of `beta_damp` so each time the solver starts with a different value of (`beta_damp`, `omega_item`), and then we keep track of the results. The trick is how to construct the list of tuples.

Here is our suggested solution. First construct a sequence of pairs (0.01, 1.950), (0.01, 1.952), and so on, by using the generator expression equivalent to a nested loop:

```
seq = [(x, y) for x in A for y in B]
```

where `x` is `beta_damp` from the array `beta_arr` and `y` is `omega_item` from the array `omega_arr`. Now pass each tuple (containing pairs of `beta_damp`, `omega_item`) into `solver()`. Inside `solver()`, the code unpacks the tuple via `beta_damp, omega_item = params`.

> *Note that you cannot map two independent values onto the function, because map will draw the first element from beta_arr and pair it with the first element from omega_arr, then the second element from beta_arr with the second element of omega_arr, all the way up to the fifth element in beta_arr, whereupon it quits. Not what we want. Turning them into paired tuples ensures that we get the desired parameters mapped together onto the solver function.*

Your call to `p.map` will return a list of amplitudes the same length as `seq`. To create plots of A^2 versus ω, you will need to first convert that list into an array and then use numpy.reshape to reshape it into a two-dimensional array where each row corresponds to a different value of β. A `for` loop can then be used to plot each set of amplitudes-squared versus ω.

➤ *Hint:* You will have to send `reshape()` a tuple indicating the dimensions of the new array, e.g., (n_row, n_col). If you don't know the value for one of those parameters, though, you can use "-1" for it and the computer will automatically determine the correct size.

```
# < Challenge Problem 3, Use multiprocessing to determine resonance curves for
#    different values of beta_damp >

### COPY the code from Challenge Problem 3 into this cell. If that cell has
#    already been run, you do not need to reimport the libraries or
#    redefine n_cores, mass, k_spring, or omega0.

### CHANGE the list of tuples "seq" to contain a set of values of omega for
#    each of beta = 0.16, 0.08, 0.04, 0.02 and 0.01.
```

```
# < Challenge Problem 3, Plots >

### RESHAPE the output list by first turning it into a 2-D array using
#    np.reshape where each row corresponds to a different value of beta



### PLOT A^2 versus omega for each value of beta
```

14.10 Contents of resonance_for_multi_processing_solvers.py

```
import numpy as np
from scipy.integrate import solve_ivp

def deriv_damp_drive(t_now, r_now, omega0 = 1, beta_damp = 0, omega = 1,
                     drive_amp = 0):
    """ Supplies the derivatives of position and velocity for a damped driven
    harmonic oscillator to be used by the differential equation integrator,
    solve_ivp.
    For the optional parameters, omega0 (natural frequency) is set to one by
    default, beta_damp is set to zero, omega (driver frequency) is set to one,
    and the amplitude of the driver acceleration (f_0/m) is set to zero."""

    # Unpack the variables
    pos_x = r_now[0]
    vel_x = r_now[1]

    # Perform the derivatives
    dx_dt = vel_x
    dv_dt = -(omega0**2)*pos_x -2*beta_damp*vel_x + drive_amp*np.cos(omega*t_now)

    return np.array([dx_dt,dv_dt])
```

```
def solver(params):
    """ This is the multiprocessor solver for creating a resonance curve for the
    damped oscillator.  It takes in a tuple, params, arranged to be beta_damp,
    omega_item, with beta_damp the damping parameter, and omega_item the driver
    frequency.  It returns the final amplitude of the oscillator"""
    beta_damp, omega_item = params  # unpack the tuple
    mass = 2.0 # [kg] mass
    k_spring = 8.0 # [N/m] spring constant
    f_0 = 0.2 # [N] driver amplitude
    drive_amp = f_0/mass
    omega0 = np.sqrt(k_spring/mass)

    tau = 1/beta_damp
    t_arr = np.linspace(0, 6*tau, 1000)  # Array of times. Be sure to let it run
                                         #    sufficiently long
    t_span = [0, t_arr[-1]]
    five_sixths_index = (len(t_arr)*5)//6  # Find the index of the latter 5/6 of
                                           #    the array.

    r_init = np.array([0.0, 0.0])  # Initial conditions (starting from rest at
                                   #    zero)

# Create an array of values for omega to sample
    r_soln =  r_soln = solve_ivp(deriv_damp_drive, t_span, r_init,
                                 t_eval = t_arr, rtol = 1.0e-11,
                                 atol = 1.0e-11, args = (omega0, beta_damp,
                                                         omega_item, drive_amp))
    final_amp = r_soln.y[0][five_sixths_index:].max()  # a_arr.max() gives the
                                                       #    largest element.
                                                       #    Restrict to the last
                                                       #    1/6 points to avoid
                                                       #    transient.
    return final_amp  # return single value to process
```

15 Brachistochrone Problem

The solution of the brachistochrone problem (Example 6.2 from *Classical Mechanics*) provides the shortest time path between two points for an object sliding down a hill. It is neither a straight path nor a circular path but rather, one connected by a "cycloid" of unique generating radius between the two points. A cycloid is the curve traced out by a point on the rim of a wheel of radius a rolling horizontally (Figure 6.5 from *Classical Mechanics*). The cycloid itself also has the interesting property that objects released from different initial positions along the curve will slide to the bottom in the same amount of time, i.e., they are "isochronous".

This unit will guide you through a more in-depth examination of cycloids. It culminates in an animation comparing the motion of an object sliding down a cycloidal path to one following a straight-line path.

15.1 Objectives

In this unit, you will:

- examine a set of cycloids with different generating radii (PHY);
- find the radius of the cycloid that connects two arbitary points with the shortest time path using numerical root finding (PRO);
- numerically calculate the time for an object to slide between two points on a cycloid (PHY);
- compare the motion of an object on a cycloidal and straight-line path (PHY);
- use an animation to observe the motion along a cycloidal path (PRO).

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp  # the differential equation integrator

plt.rcParams["figure.figsize"] = (12,8)  # make figures big for easier viewing
#    (width, height in inches)
# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

15.2 Introducing the Cycloid

The solution to the brachistochrone problem is a cycloid that connects two points. The parametric equation for a cycloid in θ is

$$x = a(\theta - \sin\theta)$$
$$y = a(1 - \cos\theta)$$

(Equation 6.26 from *Classical Mechanics*).

Exercise 1

Plot this parametric path for $0 < \theta < 2\pi$ using $a = 1.0$. The easiest way to do this is to write a function that returns x and y for a particular value of θ and a and then pass that function an array of θ values. In order to create a concave surface, multiply the y-values by -1 before plotting y versus x.

```
# < Exercise 1, Plot this parametric path >

### CREATE an array of theta values from 0 to 2 pi, use a function to
#    determine x and y for a particular value of theta, multiply the y value
#    by -1, and plot y vs. x

# Template for function
def cycloid(a_radius, theta):
    """ Function to return the x and y values of a cycloid of radius a for a
    particular theta """
    pos_x = 0 ### FIX
    pos_y = 0 ### FIX

    return np.array([pos_x, pos_y])
```

Exercise 2

A cycloid can be thought of as the path traced out by a point on a wheel of a particular radius a as the wheel rolls along a flat surface. To compare cycloids produced by different values of a, plot a series of cycloids for a = 1, 2, and 3 on a single set of axes, letting θ cover the range $[0, 2\pi]$.

- *Hint 1:* You can use the same function that you defined above to calculate x and y.
- *Hint 2:* This code is most elegantly written using a loop that iterates through different values of a.

```
# < Exercise 2, Plot the parametric paths for a = 1, 2, and 3 >
```

Exercise 3*

Find the unique cycloid that passes through the point $(x, y) = (6, -2)$. For solving this transcendental set of equations, first find the angle at which this would occur, solve for the

radius, then plot the curve. This problem follows naturally from Problem 6.14 c of *Classical Mechanics*.

To find the angle, you should employ the root-finding algorithm, scipy.optimize.root, introduced in Oscillations Around Stable Equilibria.

- Remember, to use `scipy.optimize.root()`, you must first define a function describing the equation for which you wish to find the root. In this case, use your set of equations for x and y to define a new equation that will equal zero for the correct value of θ. A template of this function is given below. To make it general, the function takes the values of x and y of the intersecting point.
- `scipy.optimize.root()` requires you to provide an initial guess at the value of the root. This guess needs to be somewhat close, but away from 0, the other root. You may find it helpful to look at the graph above to determine a value.
- The syntax to use `scipy.optimize.root()` will look like `root_soln = root(func, guess, args = (x, y))`, where "func" is the name of the function, "guess" is the initial guess, and "args" allows you to send additional arguments to `func()`.
- The solution will be stored in `root_soln.x[0]`

#/ < Derivation of the angle equation to be solved, Instructor solution >

$$x = a(\theta - \sin\theta)$$
$$y = -a(1 - \cos\theta))$$

will yield a concave-up cycloid (where the y-value has been multiplied by -1). Dividing these two gives (a is assumed to be non-zero):

$$\left(\frac{x}{y}\right) = \frac{-(\theta - \sin\theta)}{1 - \cos\theta}.$$

Rearrange this into a transcendental function for which we seek the zeros of the left side:

$$(\theta - \sin\theta) + \left(\frac{x}{y}\right)(1 - \cos\theta) = 0.$$

Substituting $(x, y) = (6, -2)$ gives:

$$(\theta - \sin\theta) + (-2)(1 - \cos\theta) = 0.$$

After finding the unique (non-zero) angle θ, substitute into either the x or y equation to get the unique radius a.

#/

In this example of $(x, y) = (6, -2)$, you should find

θ = 4.0516 radians, and a = 1.2394 m.

```
# < Exercise 3, Using root to find the cycloid that passes through (0, 0) and
#    (6, -2) >

# Find the radius of the unique cycloid that passes through (0, 0) and (6, -2).
#    Call it a_radius_62, with the corresponding theta_62, when it intersects
#    the point (6, -2). Print your results to 5 significant figures.
```

```
#  Hint:  You should find theta_62 to be about 4 radians, and a_radius_62 to
#    be about 1.2 m

# Import the library containing scipy.optimize.root
from scipy.optimize import root

# Define the function for which the root will be found
def func(theta, x, y):
    """Function for the transcendental equation to be solved by root."""
    soln = 0 ### FIX by setting soln equal to the equation for which you
             #    desire the root

    return soln

# Provide an initial guess for theta in radians
guess = 0 ### FIX

# Set the coordinates of the intersecting point
x = 6
y = -2

# Use scipy.optimize.root to solve for the value of theta
root_soln = root(func, guess, args = (x, y))
# root returns a lot of performance info.  We isolate the number via
#    root_soln.x[0]
theta_62 = root_soln.x[0]
print("Angle at which x and y intersect, theta_62 = {:5.4f} radians".
      format(theta_62))

# Use either x or y equation to solve for radius
a_radius_62 = 0 ### FIX

print("a_radius_62 = {:5.4f} m".format(a_radius_62))
```

```
# < Exercise 3, Using root to find the cycloid that passes through (0, 0) and
#    (6, -2), Plot >

### PLOT the resulting curve
```

15.3 Cycloidal Path Length

There are a few properties of cycloids that are worth examining before we analyze the motion in time. For instance, the next two exercises will show you how to find the total length of the path as θ goes from 0 to 2π.

Exercise 4

To convert between s and θ, consider infinitesimal displacements dx and dy as a puck slides down the cycloid. On a separate sheet of paper, use the Pythagorean theorem to write the displacement as $ds^2 = dx^2 + dy^2$, following Equation 6.28 from *Classical Mechanics*. With $u = \theta$, ***show that***

$$\frac{ds}{d\theta} = \sqrt{(dx/d\theta)^2 + (dy/d\theta)^2} = a\sqrt{2 - 2\cos\theta}.$$

This conversion between s and θ coordinates will be essential to solving the brachistochrone analytically and computationally.

Exercise 5

On a separate sheet of paper, write out the integral for the total path length S as θ goes from 0 to 2π. Solve the integral analytically and include your result below:

You may find this substitution handy:

$1 - \cos\theta = 2\sin^2(\theta/2)$.

Full page length of the cycloid is:

15.4 Travel Time

15.4.1 Analytical solution

Exercise 6

The usual way to solve for the time taken to go from the top of the cycloid $(x, y) = (0, 0)$ to the bottom $(x, y) = (a\pi, -2a)$ (with puck initially at rest) is to invoke conservation of energy to get the speed of the puck as a function of y.

$v(y)$=

The time increment taken to traverse any distance element is then ds/v, and the integral is disarmingly simple. On a separate sheet of paper, ***write the integral and solve it*** to show that the time taken to get from top to bottom is $\pi\sqrt{a/g}$.

What if you want to find the time taken to go from some other starting position, say, $\theta = \pi/2$?

You could try using $\theta = \pi/2$ as the starting angle following the work you did in **Exercise 6**, but you would need to solve a much more involved integral (see Problem 6.25 from *Classical Mechanics*). Instead, we will solve it numerically below.

15.4.2 Numerical solution

Rather than assuming conservation of energy to find the velocity at any height, we can instead solve the brachistochrone problem numerically with our usual `solve_ivp()` method, using only Newton's laws, as detailed in the next exercise.

Exercise 7

You might think of this problem as two-dimensional, but unlike for projectile motion, there is a spatially varying acceleration in both x and y components, subject to the constraint that they follow the cycloidal path. We would like to keep it as a one-dimensional problem, and we have a choice of integrating the s acceleration or θ acceleration. Here we keep everything in θ coordinates, for simplicity of interpreting the result. It does mean, however, that you will need to convert the familiar $g\sin(\alpha)$ linear acceleration for a ball on a plane inclined at angle α to $\ddot{\theta}$, the (parametric) angular acceleration.

Edit the `deriv()` function below to be used by `solve_ivp()` following these guidelines. First, find the linear acceleration along s (tangent to the curve) simply by

1. analytically finding the local slope, dy/dx, in terms of θ
2. taking the tangent to get the angle α above horizontal
3. then $\ddot{s} = g\sin(\alpha)$

Next we need to convert this linear acceleration $\frac{d^2s}{dt^2}$ into parametric angular acceleration $\frac{d^2\theta}{dt^2}$, beginning with the defining equations for the cycloid:

$$x = a(\theta - \sin\theta) \tag{15.1}$$
$$y = a(1 - \cos\theta), \tag{15.2}$$

and the definition of the path length

$$s^2 = x^2 + y^2.$$

The derivation is shown below:

1. Find the speed by differentiating with respect to time:

$$\frac{ds}{dt} = \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}$$

and substitute the defining equations for x and y to get the speed in terms of θ:

$$\frac{ds}{dt} = \sqrt{\left(a\frac{d\theta}{dt} - a\cos\theta\frac{d\theta}{dt}\right)^2 + \left(a\sin\theta\frac{d\theta}{dt}\right)^2} \tag{15.3}$$
$$= a\frac{d\theta}{dt}\sqrt{(1-\cos\theta)^2 + \sin^2\theta} \tag{15.4}$$
$$= a\frac{d\theta}{dt}\sqrt{2 - 2\cos\theta}. \tag{15.5}$$

2. Differentiate with respect to time:

$$\frac{d^2s}{dt^2} = \left(a\frac{d^2\theta}{dt^2}\right)\sqrt{2-2\cos\theta} + a\left(\frac{d\theta}{dt}\right)^2\left(\frac{\sin\theta}{\sqrt{2-2\cos\theta}}\right)$$

3. Solve for $\frac{d^2\theta}{dt^2}$:

$$\frac{d^2\theta}{dt^2} = \left(\frac{d^2s}{dt^2}\right)\frac{1}{a\sqrt{2-2\cos\theta}} - \left(\frac{d\theta}{dt}\right)^2\frac{\sin\theta}{(2-2\cos\theta)}.$$

Notice that $\frac{d^2\theta}{dt^2}$ is now given in terms of $\frac{d^2s}{dt^2}$ (the local linear acceleration, which you calculated above), and $\frac{d\theta}{dt}$. Use this result for $\frac{d^2\theta}{dt^2}$ and the local linear acceleration to update the angular acceleration within the `deriv()` function.

➤ *Hint:* In your `deriv()` function you will also need a line such as:

```
if (theta<0.00001): theta = 0.00001
```

to avoid the infinity in dy/dx at the origin where slope = ∞.

```
# < Exercise 7, Numerical calculation for cycloidal path, deriv function >

grav = 9.8 # [m/s**2] Gravitational acceleration near the surface of Earth

def deriv(t_now, r_now, a_radius = 1):
    """The deriv function for calculating the angular position and velocity
    for the brachistochrone problem"""
    # unpack the variables
    theta = r_now[0]
    dtheta_dt = r_now[1]

    #  Strategy:  This is a one-dimensional problem, so solve for the
    #    parametric angle theta and its first derivative.
    #    The local slope dy/dx determines the acceleration of the object. To
    #    find it, do the following:
    #    1) Get the local slope by using the analytical equations for dy/dtheta
    #       and dx/dtheta and taking the ratio.
    #    2) Get the angle from the horizontal by taking the arctan of the
    #       slope.
    #    3) Finally, get the acceleration from the sine of that angle.
    #    Don't forget to limit theta to > 0.00001, following the hint above.

    # Local linear acceleration from gravity
    acc_s = 0 ### FIX following instructions above

    # Define dvel_theta/dt based on acc_s, theta, and dtheta_dt, using the
    #    conversion derived in the cell above.
    dvel_theta_dt = 0 ### FIX, following instructions above

    return np.array([dtheta_dt, dvel_theta_dt])
```

Exercise 8

Now, use `solve_ivp()` to calculate the motion of an object under the influence of gravity along a cycloidal path with radius $a = 1$ m. Assume the object starts from rest at $\theta = 0.001$ and integrate your solution from 0 to 3 seconds. In your call to `solve_ivp()`, let `max_step = 0.001`.

Plot the x and y positions as a function of time on the same set of axes. Then plot y versus x to verify that the path is that of a cycloid.

```
# < Exercise 8, Numerical calculation for cycloidal path, calling solve_ivp >

a_radius = 1.0  # [m] return to unit radius

t_start = 0
t_end = 1 ### FIX # [s], Final time
t_span = [0, 0] ### FIX  [s], Time span

# Set initial conditions
theta0 = 0.001 # start just past the origin
r_init = np.array([theta0, 0.0]) # start from rest

# Numerically solve
r_soln = solve_ivp(deriv, t_span, r_init, args=(a_radius,), max_step = 0.001)
```

```
# < Exercise 8, Numerical calculation for cycloidal path, plots >

### PLOT x(t) and y(t) on the same axes

### PLOT y vs. x for your numerical solution
```

Exercise 9

How well does your numerical integration do at calculating the time? Use your analytical calculation from **Exercise 6** to determine the travel time to final angle theta = π. Then compare this with the time at which the object in your numerical solution reaches π.

- *Hint 1:* numpy.where (first introduced in Simple Pendulum with Large-Angle Release) will be helpful for finding when the x-position crosses a certain threshold. Remember that given an array `arr`, the indices of all the elements greater than or equal to a fixed `value` is given by `np.where(arr >= value)[0]`.
- *Hint 2:* Can you increase the accuracy by either decreasing `max_step` or `atol` and `rtol` in your call to `solve_ivp()` above?

```
# < Exercise 9, Compare the analytical and numerical travel time >

### CALCULATE the travel time analytically

### FIND the numerical travel time
```

Exercise 10

How long does it take the object to reach the bottom if it is started from a different initial position? Experiment with recalculating the numerical solution and travel time using a different initial value of θ.

What do you observe about the travel time?

```
# < Exercise 10, Travel time for different initial theta >

### USE solve_ivp for a different set of initial conditions

### CALCULATE the travel time
```

Exercise 11*

For this exercise, return to the situation in **Exercise 3**. Find the time taken to go from (0,0) to (6, −2) using the correct radius cycloid `a_radius_62` and your numerical `solve_ivp()` solver. Plot $x(t)$ and $y(t)$ on the same axes.

➤ *Hint:* Approximate the initial (0,0) position as $\theta = 0.001$.

```
# < Exercise 11, Returning to (6, -2) example >

### USE solve_ivp for a different set of initial conditions

### PLOT x(t) and y(t)

### CALCULATE the travel time
```

Exercise 12*

While this is not proof that the cycloid provides the shortest time path, it is instructive to compare the total distance and time taken in going from the origin to (6, -2) via the cycloid to the time taken along the straight-line path. Analytically calculate this time and compare with the time taken along the cycloid.

How do the times compare?

```
# < Exercise 12, Time along straight-line path >

# USE this space to calculate the time taken for an object moving under gravity
#    to travel along a straight path from (0, 0) to (6, -2)
```

15.5 An Animation

Exercise 13*

Perhaps the most fun way to visualize the motion of an object along each of these two paths is through an animation. You will learn more about how to generate an animation in later units or in Appendix 3: Creating Animations with FuncAnimation, but in this one, most of the code to generate the animation has already been provided.

The first step is to calculate the trajectories you would like to animate along a cycloid and along a straight-line path. In addition to setting the x and y locations for the object sliding

down the cycloid, you will have to calculate the position (x and y) of the object sliding down the straight-line path for each time step from your cycloid path solution (`r_soln.t`). To do this, remember that you can calculate the acceleration from the slope and the distance traveled from a kinematic equation. Then convert from that distance to an x and y position using trigonometry.

After you have made the appropriate adjustments to the cell below, plot the trajectories for both paths. This step is to verify your trajectories.

```
# < Exercise 13, Animation prep >

# Numerically solve for the motion along a cycloid path to reach (6, -2)
a_radius = a_radius_62
period = 4*np.pi*np.sqrt(a_radius/grav)  # full period
t_span = (0, period)
theta0 = 0.001
r_init = np.array([theta0, 0.0])

t_arr = np.arange(0, period, 0.001)  # Use an evenly spaced array of times so
                                     #    that the frames in the movie will be
                                     #    evenly spaced
r_soln = solve_ivp(deriv, t_span, r_init, t_eval = t_arr, args=(a_radius,),
                   rtol = 1e-11, atol = 1e-11)

# Set the x and y positions of the object sliding down the cycloid and the
#    object sliding down the straight-line path

# Cycloid path
x_soln = r_soln.t*0 ### REPLACE this dummy array with the x-positons along
                    #    the cycloid
y_soln = r_soln.t*0 ### REPLACE this dummy array with the y-positons along
                    #    the cycloid

# Straight-line path
x2_soln = x_soln ### REPLACE with array of x-positions along the straight-line
                 #    for each time
y2_soln = y_soln ### REPLACE with array of y-positions along the straight-line
                 #    for each time

fig,ax = plt.subplots()
ax.set_aspect('equal')  # Make the tick spacing along the x and y axes equal
ax.set_xlabel("Distance")
ax.set_ylabel("Height")
ax.set_title("Brachistochrone\n")
ax.set_xlim(0, 2*np.pi*a_radius)
ax.set_ylim(2*np.pi*a_radius*(-2/6), 0)
### ADD code to plot y_soln vs. x_soln and y2_soln vs. x2_soln
```

You've plotted the trajectories, but to animate them you will need to show how the positions of the objects on them change with time. That means you will need to plot some representation of the mass at a particular position corresponding to the time step.

We will use filled circles to represent the two masses. These can be drawn using `matplotlib.patches.Circle()`. This function must be provided the coordinates of the center of the circle. It may also be passed the radius of the circle, along with standard plotting arguments such as color and linestyle. For example,

```
ball = patches.Circle((0,0), ball_radius, fc ='red',
                  ec = 'DarkSlateGray', lw = 2 , zorder = 3)
```

from the code below creates a circle of radius equal to the variable `ball_radius` at the origin point. The face color (`fc`) of the circle is red, the edgecolor (`ec`) is a dark gray, the width of the line outlining it (`lw`) is "2", and it is placed in front of other plot elements, such as lines, by setting `zorder = 3`.

Once the patch has been created, it must be placed on the figure using `add_patch()`. For example,

```
ax.add_patch(ball)
```

places the `circle` patch object on the `ax` object.

The patch can be moved by using `set_center`. For example,

```
ball.set_center([x, y])
```

would place the circle at the x, y coordinate location.

The code cell below shows you an example of how circular patches will be drawn in the animation. ***Play around with it. See what happens if you comment out the `add_patch()` commands. Try modifying the cell to move the circles to different locations.***

```
import matplotlib.patches as patches  # Imports code to draw a circle

fig,ax = plt.subplots()
ax.set_aspect('equal')  # Make the tick spacing along the x and y axes equal
ax.set_xlabel("Distance")
ax.set_ylabel("Height")
ax.set_title("Brachistochrone\n")
ax.set_xlim(0, 2*np.pi*a_radius)
ax.set_ylim(2*np.pi*a_radius*(-2/6), 0)

ball_radius = 0.05 # Plot size of the circle representing the mass

# Create a circle representing the mass on the cycloid path
ball = patches.Circle((0,0), ball_radius, fc ='red', ec = 'DarkSlateGray',
                      lw = 2 , zorder = 3)  # "zorder = 3" ensures that it will
                                            #    be plotted on top
# Create a circle representing the mass on the straight path
ball2 = patches.Circle((0,0), ball_radius, fc ='DodgerBlue',
                       ec = 'DarkSlateGray', lw = 2 , zorder = 3)

# Add the circles to the plot
ax.add_patch(ball)
ax.add_patch(ball2)
```

```
# Change the location of the circles
ball.set_center([3, -1])
ball2.set_center([6, -2])
```

Exercise 14*

You now have all the components you need for your animation, so it is just a matter of putting it together. The function to create an animation is `animation.FuncAnimation()` from `matplotlib`. This function creates an animation (a series of figures to be shown in a series to make a movie) by repeatedly calling a given function. The required arguments are the figure object (`fig`) and the function that will be called for each frame (`animate`). In this case, we also pass it the name of a function to be called once before the generation of the frames using `init_func = init`, the number of frames to be used (`frames`), and the time interval between the frames in milliseconds (`interval`):

```
anim = animation.FuncAnimation(fig, animate, init_func = init, frames = num_frames,
   interval = 50)
```

`FuncAnimation` will first call the `init_func` function. In this case, this function initializes the patches representing the masses. It then repeatedly calls the `animate()` function, passing it a variable representing the frame number. Once it reaches the number of frames indicated by the `frames` parameter, it will start over at zero.

After the animation is created by this code, it is saved as `anim`. It can then be displayed as a video by

```
display(HTML(anim.to_jshtml()))
```

Or, if you are using the CoCalc server, it can be displayed by

```
display(HTML(anim.to_html5_video()))
```

Fill in the code below to create your final animation. First, add the trajectories to the plot as you did in **Exercise 13**. Then modify `animate()` by adding calls to the `set_center()` method to place the circles at the appropriate location for the particular frame. For a hint as to how to do this, look at how the time of the frame is displayed in the plot title.

```
# < Exercise 14, Animation continued >

# Animation of mass on cycloid path
#    and comparison with straight-line trajectory

# Import libraries
import matplotlib.patches as patches # Imports code to draw a circle

from matplotlib import animation    # animation package (for later)
from IPython.display import HTML    # for displaying animation inline

# Create plot and draw trajectories
fig, ax = plt.subplots()
ax.set_aspect('equal') # Make the tick spacing along the x and y axes equal
```

```
ax.set_xlim(0, 2*np.pi*a_radius)
ax.set_ylim(2*np.pi*a_radius*(-2/6), 0)
ax.set_xlabel("Distance")
ax.set_ylabel("Height")
ax.set_title("Brachistochrone\n")
### ADD code to plot y_soln vs. x_soln and y2_soln vs. x2_soln

# Create the patches representing the masses
ball_radius = 0.05  # Plot size of the circle representing the mass

# Create a circle representing the mass on the cycloid path
ball = patches.Circle((0,0), ball_radius, fc ='red', ec = 'DarkSlateGray',
                      lw = 2 , zorder = 3)  # "zorder = 3" ensures that it will
                                            #    be plotted on top
# Create a circle representing the mass on the straight path
ball2 = patches.Circle((0,0), ball_radius, fc ='DodgerBlue',
                       ec = 'DarkSlateGray', lw = 2 , zorder = 3)

def init():
    ax.add_patch(ball)
    ax.add_patch(ball2)
    return ball, ball2

# In order to produce a more manageable animation, only use every 20th point
mod_skip = 20
num_frames = int(len(x_soln)/mod_skip)  # number of frames is the floor of the
                                        #    number of integrator time steps
                                        #    divided by mod_skip

def animate(i):
    ax.set_title("a = {:5.4f} cycloid and straight path\ntime  = {:3.2f} s".
                 format(a_radius_62, r_soln.t[i*mod_skip]))  # Multiplying i
                                                             #    by mod_skip
                                                             #    ensures that
                                                             #    only every
                                                             #    20th step is
                                                             #    shown
    ### USE set_center() to place ball and ball2 at the appropriate locations
    #    for the frame

    return ball, ball2

### UNCOMMENT the following two lines of code to see the animation.
#anim = animation.FuncAnimation(fig, animate, init_func = init,
#   frames = num_frames, interval = 50, blit = True)

#display(HTML(anim.to_jshtml()))  # Display as JSHTML

### IF using CoCalc, comment out the above line and comment in the following
#    two
#plt.rcParams['animation.html'] = 'html5'
#HTML(anim.to_html5_video())
```

15.6 Check-out

Exercise 15

Briefly summarize the ideas in this unit.

15.7 Challenge Problem

If you have extra time, consider the following question:

- How far does the mass travel from (0,0) to (6, -2) along the cycloidal path? It is probably easiest to calculate this by computing a numerical integral via a Riemann summation. How does this distance compare to the straight-line distance?

16 Spherical Pendulum

In this unit, you will consider a spherical pendulum, as described in Problem 7.40 of *Classical Mechanics*. You will use Lagrangian mechanics to determine the equations of motion, analyze the equations of motion to approximate the trajectories under specific circumstances, and then calculate and compare the numerical solutions to the analytical approximations.

16.1 Objectives

In this unit, you will:

- use Lagrangian mechanics to determine the equations of motion for a spherical pendulum (PHY);
- numerically determine the trajectory for two special cases (PHY);
- show that the generalized momentum in ϕ is conserved (PHY);
- learn how to make polar plots (PRO);
- explore the motion of the spherical pendulum with different initial velocities in ϕ (PHY).

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp  # the differential equation integrator

plt.rcParams["figure.figsize"] = (12,8) # make figures big for easier viewing
#    (width, height in inches)

# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

16.2 Equations of Motion for a Spherical Pendulum

When a pendulum is not confined to a single vertical plane but free to move in both ϕ (angle around the vertical axes) and θ (angle from the vertical axes), it is referred to as a *spherical pendulum*. Here, you will analyze a spherical pendulum with a range of different release angular velocities.

If you have not already completed Problem 7.40 from *Classical Mechanics*, work through the following exercise. Otherwise, you may begin at **Exercise 2**.

Exercise 1

On a separate sheet of paper, write down the Lagrangian for a spherical pendulum using generalized coordinates θ and ϕ. Then, determine the equations of motion for θ and ϕ.

Exercise 2

Use the equations of motion determined from **Exercise 1** to write a `deriv()` function that returns the derivatives of ϕ, $\dot{\phi}$, θ, and $\dot{\theta}$.

```
# < Exercise 2, Computational solution for theta(t) and phi(t) of a spherical
#    pendulum, the deriv function >

grav = 9.8 # [m/s^2] gravitational acceleration near the surface of Earth

# Create a function to evaluate the derivatives
def deriv_sphere_pend(t_now, r_now, length = 1):
    """ System is a spherical pendulum.
    Supplies the derivatives of phi, theta, dphi/dt, and dtheta/dt
    to be used by the differential equation integrator, solve_ivp.
    The optional parameter, length, is the length of the pendulum.
    It is set to a default value of 1 for the system. """

    phi = r_now[0]     # unpack the r-array for the angle
    phiDot = r_now[1]  # unpack the r-array for the angular velocity

    theta = r_now[2]
    thetaDot = r_now[3]

    dphi_dt = 0     ### FIX
    dphiDot_dt = 0  ### replace 0 with the appropriate expression from
                    #    Exercise 1

    dtheta_dt = 0     ### FIX
    dthetaDot_dt = 0  ### replace 0 with the appropriate expression from
                      #    Exercise 1

    return np.array([dphi_dt, dphiDot_dt, dtheta_dt, dthetaDot_dt])
```

16.3 Validating Your Code with Special Cases

Comparing analytical and numerical results for "special cases" is an excellent method for testing your code. Analyzing special cases is also a good way to build a strong understanding of a system before increasing its complexity.

The following four exercises have you numerically solve for the motion of a spherical pendulum under two special cases (planar motion and circular motion).

Exercise 3*

Using `solve_ivp()` along with the `deriv()` function defined above, determine $\phi(t)$ and $\theta(t)$ from 0 to 10 seconds. Assume that the pendulum has a length of 1 m, mass of 1 kg, and is released from rest with $\theta(t) = 0.2$ and $\phi(t) = 0$. This set of initial conditions should result in a "planar" pendulum, i.e., one that never has any velocity in ϕ (see Problem 7.40 c from *Classical Mechanics*). A reasonable maximum step size might be 0.01 s.

```
# < Exercise 3, Computational solution for phi(t) and theta(t),
#    the integrator >

length = 1 # [m], length of the pendulum
mass = 1 # [kg], mass of the pendulum

# Create the time array for the solution going from 0 to 10 seconds in
#    steps of 0.05 s
t_start = 0.0
t_end = 0.0 ### FIX, [s] maximum time

t_span = (t_start, t_end) # Set the time span to run from t = t_start to t_end.

# Set the initial conditions by defining the starting angle
# Let this be a planar pendulum, i.e., give it no velocity in phi.
#    Let it start from rest at an angle theta = 0.2 radians from the vertical
start_phi = 0
start_phiDot = 0
start_theta = 0 ### FIX
start_thetaDot = 0
r_init = 0 ### FIX

# Use solve_ivp to determine the trajectory
r_soln = 0 ### FIX
```

Exercise 4*

Validate your solution by plotting $\theta(t)$ versus t along with the analytical approximation for a simple pendulum of the same length oscillating between small angles. Verify as well that $\phi(t)$ remains equal to zero.

```
# < Exercise 4, Computational solution for theta(t) and phi(t) >

### PLOT the theta vs. time for the computational solution

### PLOT theta vs. time according to analytic approx

### VERIFY that phi remains zero
```

Exercise 5*

Create a plot of the trajectory projected onto the $z = 0$ plane by using a polar plot and plotting ρ (distance from the vertical axis) versus ϕ.

To make a plot with a polar projection, either declare the figure and then the axis, like this:

```
fig = plt.figure()
ax = fig.add_subplot(projection = 'polar')
```

or declare the figure and axis together, like this (sending the keyword arguments as a dictionary to subplot using `subplot_kw` as the keyword argument for subplots):

```
fig, ax = plt.subplots(subplot_kw={'projection': 'polar'})
```

In your plot command, the first parameter in `ax.plot` is ϕ and the second parameter is the distance from the origin (in your case, ρ).

➤ *Hint:* The polar projection plot does not work well with negative values of ρ. You will therefore need to provide the absolute values of ρ (numpy.absolute, also available as `np.abs()` is useful for this) and adjust the value of ϕ to compensate for the change in ρ. When adjusting your values of ϕ, consider making a new array of angles using a command along the lines of `new_arr = old_arr.copy()`.

```
# < Exercise 5, Plot of the projected trajectory >

# Plot of the trajectory projected onto the z = 0 plane using a
#     polar projection plot
fig = plt.figure()
ax = fig.add_subplot(projection = 'polar') # Create a polar projection plot
### ADD the rest of the commands to plot below
```

Exercise 6

A second special case for the spherical pendulum is when the pendulum rotates in a circle around the vertical. In this case, both θ and $\dot{\phi}$ remain constant.

Determine an expression for the angular velocity in ϕ that will result in this motion for a particular value of θ_0 (see Problem 7.40 d of *Classical Mechanics*).

Calculate this value of $\dot{\phi}_0$ for $\theta_0 = 0.6$ radians.

$\dot{\phi}_0 =$

```
# < Exercise 6, Computational solution for theta(t) and phi(t) >

# INCLUDE your calculations in this cell
```

Exercise 7

Numerically solve for $\phi(t)$ and $\theta(t)$, assuming that the initial value of θ is 0.6 and that the initial velocity in ϕ is such as to yield a circular trajectory, as defined above. Set your time span to allow for several complete circles.

Plot ϕ and θ versus t and a polar plot showing the motion projected onto the $z = 0$ plane following the instructions from **Exercise 5**.

- *Hint 1:* How could you calculate how long it will take for the pendulum to circle once?
- *Hint 2:* To set the radial extent of your polar plot, you can use `ax.set_rmax(radial_extent)`, where `radial_extent` is a variable defined by the user.

```
# < Exercise 7, Computational solution for theta(t) and phi(t) resulting in a
#    circular trajectory >

### SET the initial conditions by defining the starting angles and angular
#    velocities
start_phi = 0
start_phiDot = 0  ### FIX
start_theta = 0   ### FIX
start_thetaDot = 0
r_init = 0 ### FIX

t_span = (0, 0) ### FIX

### USE solve_ivp to determine the trajectory
r_soln = 0 ### FIX
```

```
# < Exercise 7, Computational solution for phi(t) and theta(t), the plots >

### PLOT phi vs. time and theta vs. time for the computational solutions

# Plot of the trajectory projected onto the z = 0 plane using a
#    polar projection plot
fig = plt.figure()
ax = fig.add_subplot(projection = 'polar') # Create a polar projection plot
### ADD the rest of the commands to plot below
```

16.4 Varying Initial Velocity

Exercise 8

Now explore the behavior of the spherical pendulum as the initial velocity in ϕ changes. Let $\theta(t = 0) = 0.6, \dot{\theta}(t = 0) = 0$, and $\phi(t = 0) = 0$, while $\dot{\phi}(t = 0)$ ranges from 0 to the value resulting in the circular trajectory. Use `solve_ivp()` to find the numerical solution and create the same plots as in **Exercise 7**. You will likely want to increase the integration time to see the trajectory.

Describe your qualitative findings ***here***.

Retain your initial conditions, solution, and plots for one of your favorite cases below.

```
# < Exercise 8, Varying the initial velocity in phi >

### USE this space for your calculations and plots
```

16.5 Conservation of Generalized Momentum

Exercise 9

When determining your equations of motion in **Exercise 1**, you may have noticed that the generalized momentum in ϕ is conserved. Explicitly ***show this*** on a separate sheet of paper now. This quantity is the z-component of the angular momentum, ℓ_z.

Exercise 10

As a final test of your code, check that ℓ_z is conserved for the case you used in **Exercise 8**. To do this, calculate the initial, minimum, and maximum values of ℓ_z. They should all be very similar.

```
# < Exercise 10, Conservation of l_z >

### CALCULATE the initial, minimum, and maximum values of l_z
```

16.6 Near-Conical Motion

Exercise 11*

Pick a value of $\dot{\phi}(t = 0)$ that is close to the value that produces a circular trajectory at $\theta = 0.6$. Numerically calculate and plot the trajectory, as in **Exercises 6** and **7** and include your plots below.

What do you notice about the trajectory, especially $\theta(t)$? Describe it ***here***.

```
# < Exercise 11, An initial velocity in phi resulting in
#    close to conical motion >

### USE this space for your calculations and plots
```

16.7 Check-out

Exercise 12

Briefly summarize the ideas in this unit.

16.8 Challenge Problem

To explore the near-conical motion of the pendulum further, complete the following two challenge problems.

- You should have noticed in **Exercise 11** that $\phi(t)$ resembles simple harmonic motion centered on some non-zero angle. Another way to consider this situation is to let $\dot{\phi}_0$ be the angular velocity in ϕ that produces conical motion for a particular value of θ_0. Analytically show that if $\dot{\phi}(t=0) = \dot{\phi}_0$, $\dot{\theta}(t=0) = 0$, and $\theta = \theta_0 + \epsilon$ where ϵ is small, then θ oscillates about θ_0 in harmonic motion (Problem 7.40 d from *Classical Mechanics*).
- Using $\theta(t=0) = 0.6 + \epsilon, \dot{\theta} = 0$, and the value of $\dot{\phi}_0$ from **Exercise 5**, numerically calculate the trajectory of the pendulum and plot $\theta(t)$, $\phi(t)$, and the motion projected onto the $z = 0$ plane. On top of your plot of $\theta(t)$, show your solution from the previous challenge problem for these particular initial conditions.

  ➤ *Hint:* To determine the correct frequency and amplitude of oscillations for your analytical prediction, you will have to be careful about how you set the initial conditions. In particular, your initial value for $\dot{\phi}$ will *not* equal the value of $\dot{\phi}$ that produces conical motion for $\theta = 0.6$, because your initial value of $\theta = 0.6 + \epsilon$. To determine the correct value of $\dot{\phi}(t=0)$, recognize that ℓ_z is conserved, so increasing θ away from θ_0 will necessarily reduce $\dot{\phi}$. Calculate ℓ_z for the case where $\theta = 0.6$ and $\dot{\phi}$ produces conical motion. Then, use this calculated value of ℓ_z to determine the appropriate value of $\dot{\phi}$ at $\theta = 0.6 + \epsilon$.

17 The Three-Body Problem

In this unit, you will examine a problem that cannot generally be solved analytically, the three-body problem. Chapter 8 of *Classical Mechanics* covers the "Two-Body" problem, two massive objects moving within their mutually described Newtonian gravitational potential. These masses will undergo orbits shaped like conic sections — i.e., circles, ellipses, parabolas, and hyperbolas. But what are the shapes of orbits in a three-body system? The answer here is not straightforward and requires a computer to solve precisely, which is why a system of three bodies interacting gravitationally is one of the classic examples of chaos.

17.1 Objectives

In this unit, you will:

- create an integrator to find the motion of two- and three-body gravitational systems (PHY);
- use arrays to ensure code can flexibly handle different dimensionalities (PRO);
- test your integrator by applying it to stable situations where you can predict the orbit (PHY);
- transform the trajectories to the center-of-mass frame using two-dimensional arrays (PRO);
- create a movie of the motion (PRO);
- plot the trajectories in three dimensions (PHY);
- and observe chaotic three-body trajectories (PHY).

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp  # the differential equation integrator

plt.rcParams["figure.figsize"] = (12,8)  # make figures big for easier viewing
                                         #    (width, height in inches)
# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

17.2 Solving the Two-Body Problem Numerically for a Circular Orbit

Let us ease our way into the more complex three-body problem by first generating the orbital trajectory for a two-body system undergoing a *circular* orbit. Start with a large mass (say, a star) with $M_0 = 2000$ and a smaller mass (say, a planet) with $M_1 = 1$. We will work in a unit system where $G = 1.0$.

We will confine the motion of both bodies to the x-y plane for now. Later, if desired, the solution can be expanded into three-dimensional space.

Exercise 1

Determine the appropriate initial conditions such that the lower mass body, M_1, orbits in a circle 100 distance units away from the larger body, M_0. Determine, as well, a time span that will encompass at least two orbits.

➤ *Coding style tip:* Rather than doing any necessary calculations to find the initial velocity of M_1 elsewhere (say, a piece of scratch paper), include those calculations in the code cell. That way you will have a record of what you did and it will be easy to reproduce the calculations for different values, for instance, a larger radius circle.

```
# < Exercise 1, Setting ICs >

# Defining the gravitational constant, G
g_grav = 1

# Masses/
m0 = 1 ### FIX
m1 = 1 ### FIX

m_arr = np.array([m0, m1])  # Array of masses to be sent to solve_ivp

# Initial Conditions

# Mass 0
r0_0 = np.array([0, 0])  # Position of mass 0
drdt0_0 = np.array([0, 0])  # Velocity of mass 0

# Mass 1
r1_0 = np.array([0, 0])  ### FIX Position of mass 1
drdt1_0 = np.array([0, 0])  ### FIX Velocity of mass 1

# Define the array of initial conditions
#    Concatenates the arrays of ICs
r_init = np.concatenate((r0_0, drdt0_0, r1_0, drdt1_0), axis = None)

# Define the time span for the integration
t_span = (0, 0) ### FIX
```

Exercise 2

In the following three cells follow the prompts to fill in the appropriate initial conditions, the integration call, and plots of your output. Choose initial conditions such that the lower mass body, M_1, orbits in a circle 100 distance units away from the larger body, M_0. Integrate the trajectory for at least two orbits with maximum time steps of 0.1.

In creating the rules for your integrator, it may help to think of the gravitational force on M_0 by M_1 as:

$$\vec{F}_{G,0-1} = -\frac{GM_0M_1\vec{r}}{(r+\epsilon)^3}$$

where $\vec{r}$ is the *relative position* (i.e., the position of M_0 minus the position of M_1 or $\vec{r} = \vec{r}_0 - \vec{r}_1$) and ϵ (a.k.a. the gravitational softening) is a small number (0.1 is a good value) that will ensure that the masses never experience any extremely close encounters. (However, in calculating your initial conditions, you should not include ϵ.)

Create plots of the trajectories (y vs. x) of both objects on the same set of axes. To ensure a square plot, you can set the size of both the x and y axis to the same value when calling your `plt.subplots` command:

```
fig, ax = plt.subplots(figsize=(10, 10))
```

Also create plots of x vs. t and y vs. t for the planet (M_1).

```
# < Exercise 2, 2-Body Deriv Function >

def deriv_2body(t_now, r_now, m_arr = np.array([1, 1]), epsilon = 0):
    """ Derivative function for two gravitationally interacting masses.
    r_now is an array ordered (pos_1, vel_1, pos_2, vel_2) where the position
    and velocities may be in 2 or 3 dimensions.
    The optional parameter m_arr contains the masses, while epsilon is the
    gravitational softening"""

    # Determine if this is a two- or three-dimensional problem
    if np.size(r_now) == 8:
        dimension = 2
    elif np.size(r_now) == 12:
        dimension = 3
    else:
        return r_now*0  # If the positions are not in 2 or 3 dimensions,
                        #    return a derivative array just filled with zeros

    # Unpack the positions and velocities of the two masses from r_now
    pos_0 = r_now[          0 :   dimension]  # Position of mass 1
    vel_0 = r_now[  dimension : 2*dimension]  # Velocity of mass 1
    pos_1 = r_now[2*dimension : 3*dimension]  # Position of mass 2
    vel_1 = r_now[3*dimension :            ]  # Velocity of mass 2

    # Derivatives of the positions of the two masses
    dpos_dt_0 = 0  ### FIX
    dpos_dt_1 = 0  ### FIX
```

```
    # Define the relative position
    pos_rel = np.array([0, 0, 0]) ### FIX
    # Calculate the distance between the two masses
    dist = np.linalg.norm(pos_rel)  # np.linalg.norm returns the magnitude of a
                                    #    vector

    # Calculate the magnitude of the two forces/|r|: G m1 m2/(|r| + epsilon)^3
    k = g_grav*m_arr[0]*m_arr[1]/(dist + epsilon)**3

    # Calculate the acceleration of each mass
    acc_0 = 0  ### FIX
    acc_1 = 0  ### FIX

    # Create the array that will hold the derivatives for all elements in r_now
    r_deriv_now = np.concatenate((dpos_dt_0, acc_0, dpos_dt_1, acc_1),
                                 axis = None)  # creates a 1-D array by joining
                                               #   the four listed arrays

    return r_deriv_now
```

```
# < Exercise 2, 2-Body Integration >

# Defining the gravitational softening length
epsilon = 0 ### FIX

# Use solve_ivp to integrate for the solution
r_soln = solve_ivp(deriv_2body, t_span, r_init, args = (m_arr, epsilon),
                   max_step = 0.1)
```

```
# < Exercise 2, 2-Body Plots >

### PLOT the trajectory for both masses

### PLOT x and y vs. t for m1
```

17.3 Numerical Integration of the Three-Body Problem

Now you will extend your integrator to solve simultaneously for the orbits of three gravitationally interacting bodies.

17.3.1 Two planets orbiting a central star

As you build your integrator, you will want to run it on a solvable problem so you can tell whether it is working correctly. Therefore, the first case you will look at is that of two planets orbiting a central star. The star ($M_0 = 2000$) is initially located at the origin. The two planets ($M_1 = M_2 = 1$) should be set up to undergo circular (or approximately circular) orbits with radii of 100 and 200, respectively.

Exercise 3

Define your initial conditions for two planets orbiting a star in the cell below. Once again, set the time span to allow for the outer planet to complete at least two orbits.

```
# < Exercise 3, Two planets orbiting a star, setting ICs >

# Masses
m0 = 1 ### FIX
m1 = 1 ### FIX
m2 = 1 ### FIX

m_arr = np.array([m0, m1, m2])  # Array of masses to be sent to solve_ivp

# Setting the initial conditions
# Mass 0
r0_0 = np.array([0, 0])  # Position of mass 0
drdt0_0 = np.array([0, 0])  # Velocity of mass 0

# Mass 1
r1_0 = np.array([0, 0])  ### FIX Position of mass 1
drdt1_0 = np.array([0, 0])  ### FIX Velocity of mass 1

# Mass 2
r2_0 = np.array([0, 0])  ### FIX Position of mass 2
drdt2_0 = np.array([0, 0])  ### FIX Velocity of mass 2

r_init = np.concatenate((r0_0, drdt0_0, r1_0, drdt1_0, r2_0, drdt2_0),
                        axis=None)

# Define the time span for the integration
t_span = (0, 0) ### FIX
```

Exercise 4

Follow the steps outlined below to build your three-body integrator. For now, continue to confine all motion to the x-y plane. Most of your calculations will look the same as for a two-body system. However, this time you will have to consider the gravitational interactions between all three pairs of bodies.

- Paste your two-body integration from above (`deriv_2body_2D`) into the cell below.
- Rename it as `deriv_3body_2D`.
- Update it to provide the correct position and velocity vector of a third body of mass M_2, as well.

➤ *Jupyter Notebook Tip:* Oftentimes in coding, you will want to make the same edits to multiple lines of code. You can make these edits simultaneously in Jupyter! This is called "multicursor editing", and to enable it all you have to do is hold down the Ctrl key (Command on a Mac) while using your mouse to select the locations at which you want to make edits. Then just start typing. Give it a try below.

➤ *Note:* There are clever ways one can use arrays and loops in the deriv() function to limit the amount of written code (making it more elegant) and allow it to be generalized to more masses. For this unit, however, it is sufficient to explicitly calculate the acceleration of each of the three masses individually.

```
# < Exercise 4, Two planets orbiting a star, Deriv function >

### COPY deriv_2body to here

### EDIT it to work for three bodies
```

Exercise 5

Use `solve_ivp` to find the solution for the system of two planets orbiting a star. In preparation for making a movie later on, set the optional parameter, `t_eval`, so that the trajectory at regularly spaced time steps is returned.

As in **Exercise 2**, plot the trajectories (y vs. x) of all objects on the same set of axes. Also create plots of x vs. t and y vs. t for the planets (M_1 and M_2).

```
# < Exercise 5, Two planets orbiting a star, Integration >

### USE solve_ivp to integrate for the solution
```

```
# < Exercise 5, Two planets orbiting a star, Plots >

### PLOT the trajectory for both masses

### PLOT x and y vs. t for m1
```

17.3.2 Mystery system

Exercise 6

Apply your new three-body integrator to a more physically interesting problem. Try running your integrator again with the following initial conditions:

- $M_0 = M_1 = 1000$
- $M_2 = 1$
- $\vec{r}_0 = (0.5, 0)$
- $\vec{v}_0 = (0, \sqrt{500})$
- $\vec{r}_1 = (-0.5, 0)$
- $\vec{v}_1 = (0, -\sqrt{500})$
- $\vec{r}_2 = (10, 0)$
- $\vec{v}_2 = (0, \sqrt{200})$

Integrate the equations of motion for a long enough period of time to see at least *ten* orbits of each object. As in **Exercise 5**, use `t_eval` to ensure `solve_ivp` returns the solutions at regularly spaced time steps.

Plot the trajectories of all three objects on the same set of axes. For your second plot, show $x(t)$ and $y(t)$ for all objects. Then make an additional plot just showing $x(t)$ and $y(t)$ for M_0 and M_1 with the x-axis range set to be small enough that you can see their motion.

Write a description of the physical system these initial conditions correspond to.

```
# < Exercise 6, Mystery System, setting ICs >

### SET masses, initial conditions, and t_span
```

```
# < Exercise 6, Mystery System, Integration >

### USE solve_ivp to find the solution
```

```
# < Exercise 6, Mystery System, Plots >

### PLOT the trajectory for all masses

### PLOT x and y vs. t for all masses

### PLOT x and y vs. t for m0 and m1
```

17.3.3 Plotting in the center-of-mass frame of reference

You may have noticed an interesting precession effect. While we start the three masses in orbits that should take place (approximately) centered on the origin, their mutual gravitational interactions slowly carry them off in one direction over time — the exact direction will depend on exactly how you chose your initial conditions. Essentially, because of the initial conditions, the system has some non-zero momentum, which causes the center of mass to move. If this effect was not clear to you earlier, take another look at your data and plots from **Exercise 6** to convince yourself it is happening.

We can make our plots look much nicer by transforming to a coordinate system where the center of mass remains fixed at the origin.

Exercise 7

Define two new arrays, `x_com` and `y_com`, to track the location of the center of mass of the three-body system, and subtract these coordinates from the positions of the masses to find their locations in the center-of-mass frame. Plot 1) the trajectories of the three masses on the same set of axes and 2) just the trajectories of M_0 and M_1 for a shorter length of time so that you can see their orbits. If you have done your calculations correctly, you should find that the precession effect has disappeared.

The best way to accomplish this task is through a clever use of NumPy arrays. Your first goal will be to create arrays, `x_com` and `y_com`, each with a length equal to the number of time steps (`r_soln.t`). Here is some guidance on how to do this:

- Define a two-dimensional array `x_arr` of three columns (one for each mass) and the number of rows equal to the number of time steps. You may use numpy.column_stack to make this array, or simply use `np.array()` to create a new array by combining the x-positions for each of the masses and then numpy.transpose to exchange the columns and rows of your new `x_arr` array. You can check that you have the right configuration by printing the shape of your `x_arr` array (see numpy.shape; the number rows are listed first, then columns).
- Your desired `x_com` can be found by using matrix multiplication to multiply `x_arr` by your previously defined array of masses, `m_arr` (a one-dimensional array, three-element array), and dividing the resulting array by the sum of the masses. This step is equivalent to finding

$$\frac{\sum_{i=1}^{3} m_i x_i}{\sum_{i=1}^{3} m_i}$$

  at each time step. The command you will need for matrix multiplication is numpy.matmul, and the command for summing all elements in an array is numpy.sum.
- Finally, repeat the above two steps to find the y-coordinate of the center of mass.

```
# < Exercise 7, Finding center of mass >

### USE the guidance above to create arrays of the center of mass of the system
#    at each time step

x_com = np.empty(len(r_soln.y[0])) ### FIX
y_com = np.empty(len(r_soln.y[0])) ### FIX
```

```
# < Exercise 7, Mystery System, Plot in center of mass (COM) frame >

### PLOT the trajectories of all masses in COM frame

### PLOT the trajectories of m0 and m1 in COM frame
```

17.4 Making a Movie

Exercise 8*

Of course, one of the most enjoyable ways to visualize the motion of the three masses is with a movie. As in Brachistochrone Problem, most of the code to create the movie has already been provided for you. However, you are asked to provide the arrays of the x and y coordinates in the center of mass frame for each of the masses, and you will have to add two of the masses to the plot. Follow the prompts to fill in the code below to create a movie of the orbital motion. More instruction on animation can be found in Appendix 3: Creating Animations with FuncAnimation.

Once again, you will use `animation.FuncAnimation()` from `matplotlib` to create the animation. This function generates the series of figures (a.k.a. frames) that will constitute the animation by repeatedly calling a given function. The required arguments are the figure object (`fig`) and the function that will be called for each frame (`animate`). In this case, we also pass it the number of frames to be used (`frames`) and the time interval between the frames in milliseconds (`interval`):

```
anim = animation.FuncAnimation(fig, animate, frames = num_frames, interval = 50,
 blit = True, init_func = init,)
```

The `blit` keyword tells the animator not to redraw unchanged elements. The `init_func` keyword provides the name of the function that draws the base frame. This is necessary for the blitting algorithm to know what elements change.

The function argument "animate" then repeatedly calls the `animate()` function, passing it a variable representing the frame number. Once it reaches the number of frames indicated by the `frames` parameter, it will start over at zero.

After the animation is created by this code, it is saved as `anim`. It can then be displayed as a video by

```
display(HTML(anim.to_jshtml()))
```

(or `HTML(anim.to_html5_video())`, if you are using the CoCalc server).

The two stars and the planet are represented by "artists"[1] defined using `ax.plot()`:

```
m0_position, = ax.plot([],[], 'g*', ms = 10, markeredgecolor = 'k')
m1_position, = ax.plot([],[], 'r*', ms = 10, markeredgecolor = 'k')
m2_position, = ax.plot([],[], 'bo', ms = 6, markeredgecolor = 'k')
```

After these `artists` have been defined, their positions can be changed using `set_data()`. For example,

```
m0_position.set_data([[x0,x0], [y0,y0]])
```

places the `m0_position` artist (a green star outlined in black) at location $(x0, y0)$ (technically, it plots the star twice at that location). These position changes should happen inside the `animate()` function, so that each frame shows the position of each object at the corresponding time. An example has been shown for M_0. Make the appropriate changes in the code to add M_1 and M_2.

```
from matplotlib import animation  # animation package
from IPython.display import HTML  # To help view movie inline

# Create a figure for plotting
fig, ax = plt.subplots(figsize=(10, 10))
# Set appropriate x and y limits for the axes.
ax.set_xlim(-15, 15)
ax.set_ylim(-15, 15)
ax.grid()
```

[1] `artists` are objects that are rendered on the axes, such as lines or patches.

```
### FILL in expressions for the coordinates of each object in the center of
#    mass frame
# m0
x0 = np.empty(len(r_soln.t))
y0 = np.empty(len(r_soln.t))

# m1
x1 = np.empty(len(r_soln.t))
y1 = np.empty(len(r_soln.t))

# m2
x2 = np.empty(len(r_soln.t))
y2 = np.empty(len(r_soln.t))

# Plot orbital trajectories in the x-y plane
ax.plot(x0, y0, color = 'g')
ax.plot(x1, y1, color = 'r')
ax.plot(x2, y2, color = 'b')

# In order to produce a more manageable animation, only use every 20th point
mod_skip = 2

# Number of frames to include
max_frames = int(len(x0)/mod_skip)  # Max # of possible frames is the floor of
                                    #    the number of integrator time steps
                                    #    divided by mod_skip
num_frames = min([max_frames, 600])  # at 50 milliseconds a frame (the interval
                                     #   in the FuncAnimation call), 600 frames
                                     #   ->30 s, which is sufficient for a good
                                     #    animation
                                     # Set the number of frames to either 600
                                     #    or max_frames, whichever is smaller

m0_position, = ax.plot([],[], 'g*', ms = 10, markeredgecolor = 'k')
m1_position, = ax.plot([],[], 'r*', ms = 10, markeredgecolor = 'k')
m2_position, = ax.plot([],[], 'bo', ms = 6,  markeredgecolor = 'k')

# Function to draw the base frame
def init():
    m0_position.set_data([x0[0],x0[0]], [y0[0],y0[0]])
    m1_position.set_data([x1[0],x1[0]], [y1[0],y1[0]])
    m2_position.set_data([x2[0],x2[0]], [y2[0],y2[0]])

    return m0_position, m1_position, m2_position

# Function used in the FuncAnimation; loops through each of the time steps
def animate(i):
    j = i*mod_skip  # only plot every mod_skip-th element
    # Next line of code make the plot, by moving data into line's
    #    set_data method
```

```
    # Set the data twice to give animate the expected sequence
    m0_position.set_data([x0[j],x0[j]], [y0[j],y0[j]])
    ### ADD m1 and m2 to the plot, following the guidance for m0

    return m0_position, m1_position, m2_position

# Produce the animation
anim = animation.FuncAnimation(fig, animate, frames = num_frames,
                               init_func = init, interval = 50, blit = True)

# Display the animation
display(HTML(anim.to_jshtml()))

### IF using CoCalc, comment out the above line and comment in the following
#    two lines
#plt.rcParams['animation.html'] = 'html5'
#HTML(anim.to_html5_video())
```

17.5 Three Dimensions

Finally, extend your code to work in three dimensions. If you followed the provided template for the `deriv()` function correctly, it should extend to three dimensions without any changes needed. That is because the array manipulations you used (e.g., subtraction and `np.linalg.norm()`) are already defined for arrays of any length.

What you will need to change are the initial conditions you provide and how you access the different coordinates from `r_soln.y`. You will also need to calculate the z-coordinate of the center of mass.

Exercise 9*

Set up the initial conditions and solve the mystery system from **Exercise 6** with the following adjustment: $\vec{v}_2 = (0, 0, \sqrt{200})$, i.e., M_2 should now orbit in the x-z plane.

Calculate the z-coordinate for the center of mass and then plot the trajectories of all three masses in the center of mass frame. For a three-dimensional projection plot, you can use

```
fig = plt.figure()
ax = fig.add_subplot(projection = '3d')  # Add a 3-D subplot
```

Alternatively, the figure and axes can be defined together by sending a dictionary of keyword arguments to subplots using "subplot_kw":

```
fig, ax = plt.subplots(subplot_kw = {'projection': '3d'})
```

(Notice that this is similar to how you set `projection = 'polar'` in Spherical Pendulum.)

To plot three-dimensional data on such axes, all you need to do is provide the z-coordinate after the x and y coordinates:

```
ax.plot(x, y, z)
```

➤ *Hint:* Your plot will look best if you set limits for the x, y, and z axes to all be equal.

```
# < Exercise 9, 3-D Mystery System, ICs and Integration >

### SET masses, initial conditions, and t_span

### CALCULATE the solution with solve_ivp
```

```
# < Exercise 9, 3-D Mystery System, Plots >

### CREATE arrays of the center of mass of the system at each time step

x_com = np.empty(len(r_soln.y[0])) ### FIX
y_com = np.empty(len(r_soln.y[0])) ### FIX
z_com = np.empty(len(r_soln.y[0])) ### FIX

# Plot of trajectory
fig = plt.figure()
ax = fig.add_subplot(projection = '3d')  # Add a 3-D subplot
# Set the aspect ratio of the plot box to be equal
ax.set_box_aspect(aspect = (1,1,1))

### PLOT the trajectories of each of the masses

plt.show()
```

Exercise 10*

Now that you can calculate and plot the trajectories of three gravitationally interacting bodies, play around with your initial conditions to find some interesting trajectories. For example, can you find a configuration that results in unstable two-planet orbits?

Include the calculations and a plot of the trajectories for one of your favorite systems below.

```
# < Exercise 10, Exploration >

### USE this space for setting ICs, calculations, and plots
```

17.6 Check-out

Exercise 11

Briefly summarize in the cell below the ideas from this unit.

17.7 Challenge Problem

Lagrange points are special locations where a small test mass will orbit the Sun with exactly the same orbital period as Earth. Look up the location of these points, and generate a movie of the Earth/Sun system with a third low-mass object orbiting at a Lagrange point.

➤ *Hint:* Some of the Lagrange points are stable and others are unstable. To best see predictable long-term behavior, use one of the stable points.

```
# < Lagrange point challenge problem >
```

18 Orbits, Keplerian and Not

In this unit, you will determine the numerical solutions to the two-body problem, as covered in Chapter 8 of *Classical Mechanics*. You will work these exercises in the center-of-mass frame of reference. As one often does in computing, you will first attempt a calculation to which you know the answer. You will then attack a second problem that is more complex and cannot be solved analytically.

To be precise, you will start out by numerically solving for the two-body problem under Newtonian gravity. Similarly to the process for arriving at an analytical solution, you will reduce the two-body problem to a one-body problem by solving for and plotting the motion of the *reduced mass* about the fixed center of mass of the system. Then you will switch to plotting the motion of *both* bodies with a simple coordinate transformation. You will use your solver to approximate the orbit of Pluto and compare your results to data from NASA.

Finally, you will adapt your code to solve for non-Keplerian orbits that obey different force laws and for which there is not an analytic solution.

18.1 Objectives

In this unit, you will:

- numerically model circular, elliptical, and hyperbolic Keplerian orbits (PHY);
- visualize the motion of the two bodies with multipanel plots, polar projections, and a movie (PRO);
- read in a data file of the actual orbit of Pluto (PRO);
- compare the two-body numerical calculations to the actual orbit of Pluto (PHY);
- and model a non-Keplerian orbit (PHY).

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp  # the differential equation integrator

plt.rcParams["figure.figsize"] = (12,8)  # make figures big for easier viewing
                                         #    (width, height in inches)

# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

```
from matplotlib import animation    # animation package (for later)
from IPython.display import HTML    # for displaying animation inline
```

18.2 Circular Keplerian Orbit ($\epsilon = 0$)

18.2.1 Setting the initial conditions and other physical parameters

Exercise 1

Let us initially work in a unit system where $G = 1.0$. This unit system is frequently used in cosmological simulations as it results in easier analytical calculations.

Assume the two masses are $m_1 = 5$ and $m_2 = 1$ and that they start at a separation of $r_0 = 2.0$. Then fill in the rest of the constants and initial conditions such that the reduced mass will undergo a *circular* (Keplerian) orbit of radius 2.0 around the center of mass.

➤ *Coding style tip:* Rather than doing any necessary calculations elsewhere (say, a piece of scratch paper), include those calculations in the code cell. This way, you have a record of what you did and it will be easy to reproduce the calculations for different values.

```
# < Exercise 1, Setting ICs and constants >

# Initial conditions and constants
G_grav = 1.
r0 = 1. ### FIX
phi0 = 1. ### FIX
m1 = 1. ### FIX
m2 = 1. ### FIX

mu = 1. ### FIX
gamma_g = 1. ### FIX
ang_mom = 1. ### FIX
dr_dt0 = 1. ### FIX



r_init = [r0, dr_dt0, phi0]  # initial conditions
```

18.3 Numerical Solution

Exercise 2

First, solve the equations of motion numerically. Define a function that will return the derivatives for your three variables (r, $\dot{r}$, and ϕ). Then, use `solve_ivp()` to solve for $0 \leq t \leq 20$. Include `mu`, `gamma_g`, and `ang_mom` as parameters sent to `solve_ivp()`. In preparation for making a movie later on, set the optional parameter, `t_eval`, so that the trajectory at regularly spaced time steps is returned. Using a step size of 0.1 is a good place to start. Additionally, set `rtol` and `atol` to 1e-9 in order to achieve appropriate accuracy.

```
# < Exercise 2, Defining the deriv function >

# Derivative function
def deriv(t_now, r_now):  ### ADD parameters
    """ Document your function here """
    # Function values:
    r = 0        ### FIX
    drdt = 0     ### FIX
    phi = 0      ### FIX

    # Derivative definitions:
    dr_dt = 0.    ### FIX
    d2r_dt2 = 0.  ### FIX
    dphi_dt = 0.  ### FIX

    return [dr_dt, d2r_dt2, dphi_dt]
```

```
# < Exercise 2, Integration of circular Keplerian orbit, Integration >

### ADD your call to solve_ivp here
```

Exercise 3

Plot your results for r, $\dot{r}$, and ϕ as a function of time. Be sure to label your plots and your axes.

In the past, you have made figures that consisted of a single panel (`ax` object). The code template below will show you how to use subplots from matplotlib.pyplot to make a figure with multiple panels. The `subplots()` routine can take as arguments the number of rows and the number of columns of panels:

```
fig, ax = plt.subplots(nrows, ncols)
```

(When these arguments are not included, as you have typically done in previous units, a single panel plot is created.)

Each panel will then be stored as one element in the `ax` array. For example, to plot on the first panel, use

```
ax[0].plot(x, y)
```

Similarly, as you have used commands like `ax.set_xlabel` to label the x-axis of a plot, you can use `ax[0].set_xlabel` to label the first panel of the plot.

Explain below why each of your plots displays the expected behavior.

```
# < Exercise 3, Plots >

# Define a plot with three different subplots stacked vertically. The first
#    parameter in plt.subplots lists the number of rows (3)
#    and the second lists the number of columns (1)
```

```
fig, ax = plt.subplots(3, 1)

ax[0].set_ylabel('r')  # label the y-axis of the first panel
### USE ax[0].plot() to plot the radius versus time

ax[1].set_ylabel('dr/dt')
### USE ax[1].plot() to plot the radial velocity versus time

ax[2].set_ylabel('$\phi$')
ax[2].set_xlabel('time')
### USE ax[2].plot() to plot the angle versus time

# Set a title for the entire set of panels
fig.suptitle("Circular Keplerian Orbit\nReduced mass")
plt.show()
```

Exercise 4

Make a polar plot to better show the trajectory. Remember, to make a plot with a polar projection, you must set the "projection" keyword argument to "polar". In your plot command, the first input array will then be the angles, and the second input array will be the radial values.

➤ *Hint:* See Spherical Pendulum for an example of how you have made a polar projection plot in the past.

```
# < Exercise 4, Polar plot of the trajectory of reduced mass >

fig = plt.figure()
ax = fig.add_subplot(projection = 'polar')

# Alternatively, you may define fig and ax simultaneously, as in the
#    commented-out line below
#fig, ax = plt.subplots(subplot_kw={'projection': 'polar'})

### ADD the rest of the commands to plot below

plt.show()
```

18.4 An Elliptical Orbit

Exercise 5

Go through the same set of steps as above in **Exercises 1–4** but with a set of parameters that will produce an *elliptical* orbit with an eccentricity of 0.8 and a pericentric distance (r_{min}) of 2.0. (You should use the same masses and gravitational constant from **Exercise 1.**)

As before, set the parameter `t_eval` so that the solutions at evenly spaced time steps are returned. Additionally, set `rtol` and `atol` to 1e-9 in order to achieve appropriate precision.

- *Hint 1:* You might need to integrate over a larger time interval than in **Exercise 2** to obtain a closed orbit.
- *Hint 2:* Remember, there is no need to copy over your `deriv()` function again.

Explain here why each of your plots above has the "expected" behavior. Be as quantitative as possible:

```
# < Exercise 5, Set parameters for elliptical orbit >

### SET parameters and initial conditions

r_init = [r0, dr_dt0, phi0]  # initial values
```

```
# < Exercise 5, Integration of elliptical Keplerian orbit >

### ADD your call to solve_ivp here
```

```
# < Exercise 5, Plots for elliptical Keplerian orbit >

### ADD your code for plots of r, dtdt, and phi vs. time

### ADD your code for the trajectory
```

18.4.1 Visualizing two-body orbits

Visualizing the motion of the reduced mass is physically meaningful, but your plots will be even easier to interpret if you plot both masses together. This can be accomplished by transforming the coordinates of the reduced mass to the coordinates for each of the two masses.

Exercise 6

By performing an appropriate coordinate transformation, plot the orbits of **both** masses on the same set of polar axes.

- *Hint:* The Matplotlib polar plot projection cannot handle negative values for *r*, so you will have to come up with some other way to represent these values.

```
# < Exercise 6,  Plot trajectory of both masses >

### CALCULATE r and phi coordinates for both masses

### PLOT the trajectory of both masses
```

Exercise 7

It is even more revealing to watch the **motion** of the two objects rather than just plotting their trajectories. To do this, you will need to create an animation. The cells below provide a framework for the process. Note that there are several points where you will need to make edits to calculate the number of frames, draw the orbits, and place the masses in their orbits at each frame.

Ideally, your animation will show one complete orbit. Therefore, you will need to determine the period of the orbit, in preparation for making the animation. Once you know the length of the period and you have decided how much time you want each frame to represent, a simple calculation will determine which of the integrator time steps should be used in the movie (e.g., every other? every third? every twentieth?). ***Add the code to calculate the length of the orbit to the cell below, and then run it to determine how many integrator time steps per movie frame will be used.***

As in Brachistochrone Problem and Three-Body Problem, you will use `animation.FuncAnimation()` from `matplotlib` to create the animation. You will pass it a figure object (`fig`) and the function that will be called for each frame (`animate`).

The two masses are represented by "artists", defined using `ax.plot()`:

```
m1_position, = ax.plot([],[], 'ro', ms = 6)
m2_position, = ax.plot([],[], 'bo', ms = 6)
```

After these `artists` have been defined, their positions can be changed using `set_data()`. For example,

```
m1_position.set_data([[x0,x0], [y0,y0]])
```

places the `m1_position` artist at location $(x0, y0)$ (technically, it plots the star twice at that location). These position changes should happen inside the `animate()` function, so that each frame shows the position of each object at the corresponding time. More instruction on animation can be found in Appendix 3: Creating Animations with FuncAnimation.

> ***Describe the motion of the two masses in the animation, including their relative positions and velocities. Is this motion what you would expect? Why or why not?***

```
# < Exercise 7, Animation prep >

### DO THE FOLLOWING
# Define r_1, r_2, phi_1, and phi_2, if you haven't done so already.
# If you have previously defined them, you can leave these rows commented out
```

```
# r_1 =
# r_2 =
# phi_1 =
# phi_2 =

# We need to determine how long to run the movie for (the number of frames,
#    a.k.a. num_frames) in order to show one complete orbit.
#    To complete this calculation for the number of frames, you'll need to
#    know the period and the time per frame.
#    While the time_per_frame is currently set to 1, you are welcome to change
#    this variable to achieve increased or decreased time resolution in the
#    animation.

### DO the following
#    First find the period from Kepler's third law for masses m1 and m2 with an
#    eccentricity of ecc = 0.8 and an angular momentum of ang_mom.
period = 0 ### SOLVE

print("Period = ", period)

# Determine the number of frames to model one complete orbit
time_per_frame = 1.0  # each frame corresponds to a time step of 1 in units
                      #    where G = 1; ### MODIFY, if desired!
# Find the number of frames needed by dividing the period by the time per frame
num_frames = int(period/time_per_frame)
print("Number of frames = ", num_frames)

# Only a subset of the positions from the integrator solution will be used to
#    create a frame in the animation.
#    This calculation determines how many integrator time steps there are
#    between each frame.
dstep = np.max(t_arr)/len(t_arr)  # the integrator time step size
# Determine how many integrator time steps per frame. If, for example,
#    steps_per_frame = 2, only every other data point from the integrator will
#    be used in the animation
steps_per_frame = int(time_per_frame/dstep)  # round down to an integer value
print("Integrator time step (dt) = ", dstep)
print("Steps per frame (~{0:.2}/{1:.5}) = {2}".
      format(time_per_frame, dstep, steps_per_frame))
print("\nAssuming an interval of 50 milliseconds between frames,\n"
      "\tthe movie will be {:.2} seconds".format(0.05*num_frames))
```

```
# < Exercise 7, Animating the orbits >

# Recreate the same orbit plot as above in Exercise 6
#    In the animation, points representing the masses will move along the
#    orbit.
fig, ax = plt.subplots(subplot_kw={'projection': 'polar'})
ax.set_title("Elliptical Keplerian Orbit\nTrajectories of two masses")
### PLOT the orbits of m1 and m2, as in Exercise 6
```

```
# Plot the center of mass as a green star
com_position = ax.plot(0,0, 'g*', ms = 10)

# Initialize plotting of the positions of m1 and m2 for the animation
m1_position, = ax.plot([],[], 'ro', ms = 6)
m2_position, = ax.plot([],[], 'bo', ms = 6)

# Function used in the FuncAnimation to draw frames
def animate(i):
    # This makes the plot by moving data into line's set_data method
    j = int(i*steps_per_frame)  # only plot every steps_per_frame-th step
    ### ADD the positions of m1 and m2 at frame j using set_data

    return m1_position, m2_position

# Function to draw the base frame
def init():
    m1_position.set_data([phi_1[0],phi_1[0]], [r_1[0],r_1[0]])
    m2_position.set_data([phi_2[0],phi_1[0]], [r_2[0],r_1[0]])
    return m1_position, m2_position

anim = animation.FuncAnimation(fig, animate, frames = num_frames,
                               init_func = init, blit = True,
                               interval = 50)  # init_func sets the name of
                                               #    the function that makes the
                                               #    base frame
                                               # blit = True tells the animator
                                               #    not to redraw unchanged
                                               #    elements.
                                               # interval sets the time between
                                               #    frames to 50 milliseconds.
display(HTML(anim.to_jshtml()))  # Display as JSHTML

### IF using CoCalc, comment out the above line and comment in the following
#    two lines
#plt.rcParams['animation.html'] = 'html5'
#HTML(anim.to_html5_video())
```

18.5 Hyperbolic Orbits

Exercise 8*

Repeat your work from **Exercises 5–6** to solve for and plot a hyperbolic orbit with an eccentricity of 1.2 (and all the other parameters the same) for $0 \leq t \leq 10$.

```
# < Exercise 8, Hyperbolic orbits >

### SET parameters and initial conditions for a hyperbolic orbit
```

```
# Initial conditions
r_init = [r0, dr_dt0, phi0]

### SOLVE with solve_ivp
```

```
# < Exercise 8, Hyperbolic orbit plots >

### PLOT r vs. t, v_r vs. t, phi vs. t for the reduced-mass orbit
#    (in a polar plot)

### TRANSFORM from the reduced-mass coordinate system and
#    plot the trajectories of m1 and m2
```

18.6 Comparing with Pluto Ephemeris

Because the Sun dominates the mass of the solar system, the orbits of planets and smaller solar system bodies around it may be approximated as solutions to the two-body problem. In this section, you will use your orbital solver to calculate the orbit of Pluto if it and the Sun were the only masses in the solar system. You will compare your results to more precise data on the orbit of Pluto from the NASA Jet Propulsion Laboratory.

18.6.1 Reading Pluto ephemeris

The NASA Jet Propulsion Laboratory makes available *ephemeris* (the term for tables of the location and sometimes velocity of celestial bodies over time) for many of the bodies in the solar system. These data may be downloaded from the NASA Jet Propulsion Laboratory website in their particular format. To simplify the process, though, we have formatted the ephemeris for Pluto (technically, the ephemeris for the center of mass of Pluto and its moons) as a downloadable CSV file (a common spreadsheet format) listing the Cartesian positions and velocities of Pluto over time.

Exercise 9*

Start by obtaining a copy of "pluto_ephemeris.csv" and moving it to the same location as this notebook (for example, moving it to the correct directory on your computer or uploading it to CoCalc or your Google Drive server).

As you do this, take the opportunity to peek inside the file. Some options for looking at it include opening it in a text file reader, opening it from within Jupyter Notebook or JupyterLab (just click on it within the file browser window), or importing it to Excel. You will find that the data are organized into rows, each containing seven pieces of data separated by commas. Those seven pieces of data are as follows: (1) the Julian date in seconds (defined as seconds since noon Universal Time on January 1, 4713 BC), (2) x-coordinate of position, (3) y-coordinate of position, (4) z-coordinate of position, (5) x-component of velocity, (6) y-component of velocity, and (7) z-component of velocity. All the positions are given in kilometers, and the velocities in kilometers per second. The coordinate system is aligned

such that Earth's orbit around the Sun is in the x–y plane and the origin is at the center of mass of the solar system (known as the "Solar System Barycenter"). The first two rows of the file are headers describing the data.

Read in data from pluto_ephemeris.csv and store it as seven arrays, each corresponding to one of the columns of data using numpy.loadtxt (introduced in Simple Pendulum with Large-Angle Release). Because the data are separated by commas, set the `delimiter` argument to `','`, and because the first two rows of data contain header information, skip reading them using the `skiprows` argument. Finally, set the `unpack` argument to `True` so that the returned data are transposed and arguments may be unpacked using `x, y, z = np.loadtxt(...)`. Altogether, your line of code will look like the following:

```
t_pluto, x_pluto, y_pluto, z_pluto, vx_pluto, vy_pluto, vz_pluto = \
    np.loadtxt('pluto_ephemeris.csv', delimiter = ',', skiprows = 2, unpack = True)
```

Once you have read the data in, calculate the distance between Pluto and the solar system barycenter, r, for each of the times (remember, NumPy makes it easy to do array calculations like this). This array of data is what you will compare against.

➤ *Note:* If you are using Google Colab, you will have to upload pluto_ephemeris.csv to your Google Drive (we suggest you store it in the subdirectory containing your Colab notebooks). Then before you can access the data, you will need to mount your Google Drive directory by running the following lines of code

```
from google.colab import drive
drive.mount('/content/drive')
```

Once your Google Drive directory is mounted, you can access it as above with the only difference being that you must provide the full file path ("/content/drive/My Drive/Colab Notebooks/pluto_ephemeris.csv" if you saved the file in the directory called "Colab Notebooks").

```
# < Exercise 9, Pluto Ephemeris >

### READ the data file and calculate the distance between Pluto and the
#    Solar System Barycenter
```

Exercise 10*

Use your integrator to solve for Pluto's trajectory over the same period of time as the ephemeris.

To do this calculation, you will need to set the parameters and initial conditions of the system. Look up the mass of Pluto and the Sun, as well as the perihelion and aphelion distances for Pluto. Then use the latter two to calculate the eccentricity, and from that find the angular momentum. Redefine G to be in standard units (e.g., mks), and make sure that all your other units are also consistent. Use the first entry in the data file as the initial conditions for your calculation. Remember that while the data file lists all of the components of velocity in a Cartesian coordinate system, your initial conditions are for the radial component of the velocity.

➤ *Hint:* Dot products may be useful for finding the radial component.

To ease your comparison, you should ask `solve_ivp()` to generate solutions for the same set of times as in the data file. However, `solve_ivp()` assumes the initial conditions correspond to time zero, while the data file starts counting from 208657771210.0 seconds. The easy fix is to set `t_arr = t_pluto - t_pluto[0]`. You will have to account for this change in making the comparison plots, though.

Finally, compare your results using the following plots:

1) r as calculated from the ephemeris and for a two-body problem using `solve_ivp()` as a function of time,
2) the difference between the values of r as a function of time, and
3) the percent difference between the values of r as a function of time.

You should find that your two-body approximation of Pluto's distance from the center of the solar system is accurate to within about 0.6%.

```
# < Exercise 10, Solve for Pluto's orbit >

### DEFINE the system parameters and initial conditions
```

```
# < Exercise 10, Solve for Pluto's orbit cont. >

### SOLVE for Pluto's orbit over the same time period as the data file
```

```
# < Exercise 10, Compare Pluto's orbit >

### PLOT the desired comparisons
```

18.7 Non-Keplerian Orbits

A gravitational force law that deviates from $1/r^2$ will not produce Keplerian orbits (defined here as orbits that trace the shape of a conic section). It is also impossible to solve analytically, so it must be approached computationally. This problem is explored in Problem 8.25 of *Classical Mechanics*.

Exercise 11

To explore the shapes of a non-Keplerian orbit, you will complete the same set of steps that you have already completed in this unit but for a force law that scales as $1/r^{5/2}$.

Use the set of initial conditions given in the cell below, and then:

- Define the correct derivative function for this force law. Call it something different, like `deriv_nonkep()`.
- Plot $r(t)$, $dr/dt(t)$, and $\phi(t)$.
- Plot the orbital trajectory of the reduced mass on a set of polar axes.

Perform your integration from $0 \leq t \leq 40$ with a (maximum) time step size of 0.001.

```
# Initial conditions and constants for a non-Keplerian orbit
G_grav = 1.  # Resetting the gravitational constant

mu = 1.  # reduced mass
gamma_g = 1.  # constant before gravitational force
ang_mom = 1.  # angular momentum
phi0 = 0.  # initial phi
r0 = 0.6671  # initial distance of the reduced mass
dr_dt0 = 0.  # initial radial velocity of the reduced mass

r_init = [r0, dr_dt0, phi0] # initial values
```

```
# < Exercise 11, Deriv function for non-Keplerian motion >

# Derivative function
def deriv_nonkep(t_now, r_now): ### ADD parameters
    """ Document your function here """
    # Unpack variables
    r =  0       ### FIX
    drdt = 0     ### FIX
    phi = 0      ### FIX

    # Derivative definitions:
    dr_dt = 0    ### FIX
    d2r_dt2 = 0  ### FIX
    dphi_dt = 0  ### FIX

    return [dr_dt, d2r_dt2, dphi_dt]
```

```
# < Exercise 11, Integration of non-Keplerian orbit >

### ADD your call to solve_ivp here
```

```
# < Exercise 11, Non-Keplerian orbit plots >

### PLOT r vs. t, v_r vs. t, phi vs. t for the reduced-mass orbit
#    (in a polar plot)
```

18.8 Check-out

Exercise 12

Briefly summarize in the cell below the ideas from this unit.

18.9 Challenge Problems

For a further exploration of orbits, you may wish to attempt the following:

1. Play around with some other configurations of Keplerian orbits. Try systematically changing both the masses and the orbital eccentricity.
2. Confirm conservation of mechanical energy for your Keplerian orbits.
3. Confirm Kepler's third law for your elliptical orbit.

19 Motion on a Turntable

In this unit, you will use Python to visualize motion in both a non-inertial (rotating) reference frame and an inertial (fixed) reference frame. In this case, you will consider the trajectory of a frictionless puck released on a horizontal rotating turntable. It will give you practice mentally shifting between inertial and non-inertial reference frames and illustrate both the centrifugal and Coriolis forces.

First, you will predict the behavior of the puck in two special cases. Then, you will determine the equations of motion numerically and analyze the behavior of the puck under different initial conditions. For your final output, you will produce a movie of the puck in both the inertial and non-inertial reference frames.

19.1 Objectives

In this unit, you will:

- integrate the equation of motion to find the trajectory of the puck in a non-inertial reference frame (PHY);
- compare the behavior of the puck in both the inertial and non-inertial reference frames (PHY);
- explore the behavior of the puck with several different initial conditions (PHY);
- use "events" in `solve_ivp()` to cause the integration to end when a condition is met (PRO);
- expand your plotting ability by practicing using subplots and patches (PRO);
- and use movies to visualize the motion (PRO).

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp  # the differential equation integrator
from matplotlib import animation    # animation package (for later)
from IPython.display import HTML    # for displaying animation inline

plt.rcParams["figure.figsize"] = (12,8)  # make figures big for easier viewing
                                         #    (width, height in inches)
# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

19.2 Preliminary Questions

Exercise 1

To get you thinking about motion in a non-inertial reference frame, answer Problem 9.19 from *Classical Mechanics*, transcribed below.

> I am standing (wearing crampons) on a perfectly frictionless, flat merry-go-round, which is rotating counterclockwise with angular velocity Ω about its vertical axis.
>
> - I am holding a puck at rest just above the floor (of the merry-go-round) and release it. Describe the puck's path as seen from above by an observer who is looking down from a nearby tower (fixed to the ground) and also as seen by me on the merry-go-round. In the latter case, explain what I see in terms of the centrifugal and Coriolis forces.
>
> - Answer the same questions for a puck that is released from rest by a long-armed spectator who is standing on the ground leaning over the merry-go-round.

19.3 Framing the Problem

In order to solve this problem either numerically or analytically, the first steps are to find the equations of motion and set the initial conditions.

19.3.1 Equations of motion

Exercise 2

On a piece of paper, ***write down Newton's second law for the coordinates (x) and (y) as seen by someone standing on the turntable.*** Assume that the turntable is rotating counterclockwise with angular velocity Ω. (Be sure to include the centrifugal and Coriolis forces, but ignore Earth's rotation.)

Next, use your equations of motion to write your `deriv()` function in the cell below.

```
# < Exercise 2, Deriv function >

# Define the deriv function for use in numerical integration to find the
#    trajectory of a puck on a rotating turntable

def deriv(t_now, r_now):   ### ADD parameter for the angular velocity
    """ Document your function here """

    # Derivative definitions:
    dx_dt = 0  ### FIX
    d2x_dt2 = 0  ### FIX
```

```
    dy_dt = 0  ### FIX
    d2y_dt2 = 0  ### FIX

    return np.array([dx_dt, d2x_dt2, dy_dt, d2y_dt2])
```

19.3.2 Setting the initial conditions and other physical parameters

Exercise 3

In the cell below, define a set of variables for the initial position and velocity in the non-inertial reference frame and set $\Omega = 1\ \mathrm{s}^{-1}$. To begin, let the initial position be $\mathbf{r_0} = (\mathbf{1}, \mathbf{0})$ and the initial velocity in the rotating reference frame be $\mathbf{v_0} = (\mathbf{0}, \mathbf{1})$.

```
# < Exercise 3, Parameters and initial conditions >

### DEFINE omega, the angular velocity of the turntable's rotation

### DEFINE the initial position and velocity of the puck

# Create a 4-element array of initial values
r_init = np.array([0, 0, 0, 0])  ### FIX # initial values of x, v_x, y, and v_y
                                 #    in the rotating reference frame
```

19.3.3 Numerically solving the equations of motion

Exercise 4

Use `solve_ivp()` to solve for $\mathbf{r}(t)$ in the non-inertial reference frame for $0 \leq t \leq 2\pi$. In preparation for making a movie, set the optional parameter, `t_eval`, so that the trajectory at regularly spaced time steps is returned. Using 100 steps is a good place to start. Additionally, set `rtol` and `atol` to `1e-9` in order to achieve appropriate accuracy.

```
# < Exercise 4, Integrate for the trajectory >

### SOLVE for the trajectory of the puck from 0 <= t <= 2pi
```

Exercise 5

Plot the position of the puck below. The existing code will create a gray circle representing the turntable. All that is needed is to plot the trajectory over it.

➤ *Note:* The plot below makes use of the `patches` Matplotlib library, as previously done in **Exercise 14** of The Brachistochrone Problem. This library can be used to draw shapes on a figure. Here we use `matplotlib.patches.Circle()` to draw a circle representing the turntable. Other available shapes include ellipses (`matplotlib.patches.Ellipse()`), rectangles (`matplotlib.patches.Rectangle()`), and polygons (`matplotlib.patches.Polygon()`). For arguments, `matplotlib.patches.Circle()` takes the position of the center of the circle and (optionally) the radius, along with standard plotting arguments such as color and linestyle.

Once the patch has been created, it must be placed on the figure using `add_patch()`. For example, in the code below you will create and place the `circle` patch object on the `ax` object using the following lines of code:

```
circle = patches.Circle((0,0), radius = radius, color = '0.75')
ax.add_patch(circle)
```

```
# < Exercise 5, Plot trajectory >

# Import the matplotlib library patches, which contains code for drawing
#    various shapes, including circles
import matplotlib.patches as patches

### UNCOMMENT the following line of code to read the documentation on
#    matplotlib.patches.Circle
#help(patches.Circle)
```

```
# < Exercise 5, Plot trajectory continued >

# Plotting the trajectory of the puck

fig, ax = plt.subplots()

# Draw a circle representing the turntable
radius = 15 #  defining the radius of the circle
# Creates a circle centered at (0, 0) with radius set by "radius".
#    The parameter "facecolor" sets the color
circle = patches.Circle((0,0), radius = radius, color = '0.75')
# Add the circle to the plot
ax.add_patch(circle)

### FILL in the rest of the code to plot the trajectory

plt.axis('scaled')  # Scale the axes so that they are equal size (resulting in
                    #    a square plot frame)
#ax.axis('equal')   # Alternatively, keep the figure size constant, but make the
                    #    x and y have the same scale
plt.show()
```

19.3.4 Comparing the trajectory in the rotating reference frame to that in a fixed inertial reference frame

You will gain a better understanding of what is happening if you plot the trajectory of the puck in a fixed reference frame as well.

Exercise 6

First, fill in the code below to create a function that will return the x- and y-coordinates of the trajectory in the fixed reference frame. Your function should take a time (either as a

single value or in the form of an array) and a list of the initial conditions (similar to `r_init` but in the inertial reference frame) and return a list of the x and y positions at that time. If the function is properly set up, calling it with an array of time values should cause it to return a two-dimensional array in which one column is the x-coordinate and one column is the y-coordinate.

Define your initial conditions in the inertial reference frame, and use your function to calculate the x and y coordinates at each point in time for your `r_soln.t` array.

```
# < Exercise 6, Trajectory in inertial reference frame >

# x and y positions of the puck in the inertial reference frame
def pos_prime(t_arr, r_arr):

    # Function values:
    x = 0 ### FIX
    dx_dt = 0 ### FIX
    y = 0 ### FIX
    dy_dt = 0 ### FIX

    # Combine the arrays of the x and y positions into a single 2-d array using
    #    the function vstack
    pos_arr = np.vstack((0*t_arr , 0*t_arr)) ### FIX


    return pos_arr

# Initial conditions in inertial reference frame
r_init_prime =  np.array([0, 0, 0, 0]) ### FIX
# Calculate the trajectory in inertial reference frame
r_soln_prime = pos_prime(r_soln.t, r_init_prime)
```

Exercise 7

Plot the trajectories in both the rotating and non-rotating reference frames (including the turntable). Your plots will be easier to compare if they are lined up horizontally next to each other. As in Orbits, Keplerian and Not use matplotlib.pyplot.subplots to define a multipanel plot. Then plot the non-inertial and inertial solutions side by side.

Remember that the syntax to define a multipanel plot using `plt.subplots()` and then to plot data in the first panel is:

```
fig, ax = plt.subplots(nrows, ncols)
ax[0].plot(x, y)
```

```
# < Exercise 7, Plots of trajectories >

fig, ax = plt.subplots(1, 2) # one row of panels, two columns

# Rotating reference frame
circle = patches.Circle((0, 0), radius = radius, facecolor = '0.75')
```

```
ax[0].add_patch(circle)
ax[0].axis('scaled')
ax[0].set_xlabel('X')
ax[0].set_ylabel('Y')
ax[0].set_title('Non-inertial Reference Frame')
### ADD a line of code to plot the trajectory in the rotating reference frame
#    here


# Fixed reference frame
circle = patches.Circle((0, 0), radius = radius, facecolor = '0.75')
ax[1].add_patch(circle)
ax[1].axis('scaled')
ax[1].set_xlabel('X')
ax[1].set_title('Inertial Reference Frame')
### ADD a line of code to plot the trajectory in the fixed reference frame here


plt.show()
```

Exercise 8

In the space below, ***justify that your plots of the trajectories in both reference frames are consistent with each other.***

19.3.5 Exploring different initial conditions

Exercise 9

Change the initial velocity of the puck to reflect both situations asked about in **Exercise 1**, and include your plots below for both the inertial and non-inertial reference frames. Be sure to avoid any unnecessary duplication of code.

> ***In the space below, compare your predictions to the numerically derived solutions. Be sure to explain any discrepancies.***

```
# < Exercise 9, Person at rest relative to turntable >

### FILL in the code for the person at rest relative to the turntable dropping
#    the puck.
```

```
# < Exercise 9, Person at rest relative to inertial reference frame >

### FILL in the code for the person at rest relative to inertial reference
#    frame dropping the puck.
```

Exercise 10*

Explore two or three other initial velocities, including the plots below. Some ideas for $\mathbf{v_0}$ are $(-0.5, -0.5)$, $(-0.7, -0.7)$, and $(0, -0.1)$. Include the plots for one of your favorite situations below. (This exercise is very similar to Problem 9.24 of *Classical Mechanics*.)

➤ *Hint:* You may need to change the radius of the gray circle to see the details of the puck's motion.

One minor annoyance when playing with different initial conditions is that if the radius is small or the velocity is high, the puck might fly off the turntable. One possible way to approach this is to set the integration time span equal to the time it takes for the puck to travel from the initial location to the edge of the turntable. This time can be calculated using the velocity in the inertial reference frame.

Alternatively, you can tell `solve_ivp()` to end the integration when the puck reaches the edge of the turntable. This technique is analogous to what you did with your Euler integrator in Numerical Integration Applied to Projectile Motion, in which you ended the integration when the projectile hit the ground. To do something similar with `solve_ivp()` it is possible to define one or more "events". `solve_ivp()` will then track when the event(s) equals zero and, if asked, end the integration at that point. The time(s) at which the event condition is met is stored in `r_soln.t_events`. The following cell illustrates how to define such an event for when the puck crosses the edge of the turntable.

To call `solve_ivp()` with the event, your code will look like

```
r_soln = solve_ivp(deriv, t_span, r_init, args = (omega,),
                   t_eval = t_eval, events = reach_edge)
```

In this exercise you will practice using events with `solve_ivp()`, so make the necessary change to the following code cell to define an event that will cause the integration to halt when the puck crosses the edge of the turntable. Then for each set of initial conditions ***print the length of time the puck is on the turntable*** by accessing `r_soln.t_events`.

Note that when you used events to calculate the period of a simple pendulum in Simple Pendulum with Large Angle Release, the integration continued after the first zero-crossing. The difference is that here you will set

```
reach_edge.terminal = True
```

to tell the computer to stop the calculation the first time the event is triggered.

```
# < Exercise 10, Other ICs >

# Define an event to end the integration, in this case, when the puck reaches
#    the edge of the turntable

# Set the radius of the turntable
radius = 2

# Define the function that returns the value. When the value crosses zero,
#    the event will have been reached
def reach_edge(t_now, r_now, omega): # parameters must match those in deriv
```

```
    """ Function to be used to set an event for the turntable solve_ivp
    integrator. It will return zero when the puck crosses the edge of the
    turntable."""
    # event_ret should equal zero when the puck crosses the edge of the
    #    turntable
    event_ret = 1  ### FIX

    return event_ret

# Setting terminal to True causes the integration to end when the condition has
#    been met
reach_edge.terminal = True
```

```
# < Exercise 10, Other ICs >

### FILL IN the code to solve and plot other initial conditions
```

Exercise 11*

As you may have noticed in the previous exercise, when a puck slides on a rotating turntable, it can come instantaneously to rest in the non-inertial frame. ***Explain in the space below how this happens in terms of the inertial forces.*** (This exercise is based on Problem 9.21 of *Classical Mechanics*.)

19.4 Animating the Trajectory

Finally, we can make a movie of the puck. Given the right coordinates for the position in the non-inertial reference frame, these last blocks of code will create an animation of the puck with the trajectories in both the inertial and rotating reference frame shown.

Exercise 12

In order to show the turntable turning in the inertial reference frame, your movie should include a marker on the edge of the turntable that moves with the turntable. To follow the position of this marker, fill in the function below, `pos_marker`, to make it return the position of the marker in the inertial reference frame as a function of time. This function should take a time and a radius and return an array of positions. For ease, you can assume the edge marker starts at (r,0) (in Cartesian coordinates).

```
# < Exercise 12, Making an edge marker >

### DEFINE a function to return the position of a marker on the edge of the
#    turntable.
def pos_marker(t_arr, radius, omega):
```

```
    """ Document your function here"""
    return np.array([0,0]) ### FIX
```

This animation is more complicated than previous examples seen in Brachistochrone Problem, Three-Body Problem, and Orbits, Keplerian and Not. Here, you will plot not just the motion of a mass (or two), but also the paths the mass sweeps out in different reference frames. The basic structure of creating the animation, though, remains the same.

Again, we use `animation.FuncAnimation()` from `matplotlib` to create the animation. This function makes the animation by repeatedly calling a function, here called `animate()`.

`animate()` updates the positions of the four different artists: a patch representing the puck, a patch representing the marker on the edge, a line showing the path in the rotating reference frame, and a line showing the path in the inertial reference frame. The positions of these elements are set by `set_center()` for the patches and `set_data()` for the lines.

The `init()` function is called by `animation.FuncAnimation()` once before `animate()` to add the patches for the edge marker and puck to the axes.

After the animation has been created, it is displayed using

```
display(HTML(anim.to_jshtml()))
```

(or `HTML(anim.to_html5_video())`, if you are using the CoCalc server).

Exercise 13

In the cell immediately below, fill in code to select $x(t)$ and $y(t)$ for the puck from `r_soln`.

Then, try running the following blocks of code below to produce a movie. After you have successfully run it once, try playing around with the code by changing the amount of time for which it is run, the length of time between frames, the initial conditions, and so forth.

```
# < Exercise 13, Information for the movie >

# Calculate the trajectory of a puck in an inertial reference frame to be used
#     in the animation

# NOT NECESSARY to redefine reach_edge, if you did Exercise 10
# Define an event for when the puck crosses the edge of the turntable
def reach_edge(t_now, r_now, omega):  # parameters must match those in deriv
    """ Function to be used to set an event for the turntable solve_ivp
    integrator. It will return zero when the puck crosses the edge of the
    turntable."""
    return (r_now[0]**2 + r_now[2]**2) - radius**2  # (radial position)^2
                                                    #    minus (radius of
                                                    #    turntable)^2.
reach_edge.terminal = True  # integration will end when the puck reaches the
                            #    edge of the turntable

# Define system parameters
radius = 15  # Define the radius of the turntable
omega = 1  # Define the angular velocity of the turntable
```

```
# Define the range of times (here from 0 to max_time) over which to calculate
#    the solution.
dt = 0.02  # time interval for integrator in seconds
max_time = 3*np.pi #  maximum time the solution will be calculated to
t_span = [0, max_time]
# Define an array of evenly spaced times
t_eval = np.linspace(t_span[0], t_span[1], num = int(max_time/dt))

# Define the initial conditions
x0 = 1   # initial x position
y0 = 0   # initial y position
vx0 = 0  # initial x velocity in rotating reference frame
vy0 = 1  # initial y velocity in rotating reference frame
r_init = np.array([x0, vx0, y0, vy0])  # initial values of x, v_x, y, and v_y
                                       #    in rotating frame
# Initial values in inertial reference frame
r_init_prime = np.array([x0, vx0 - y0*omega, y0, vy0 + x0*omega])

# Call the integrator to numerically calculate the trajectory
r_soln = solve_ivp(deriv, t_span, r_init, args = (omega,), t_eval = t_eval,
                   events = reach_edge)
x_pos = t_eval*0 ### FIX
y_pos = t_eval*0 ### FIX
```

```
# < Exercise 13, Information for the movie >

# Define the parameters for the animation

# From the array of times, calculate information about the animation frames
frame_spacing = 1 # every value will be plotted
num_frames = int(len(r_soln.t)/frame_spacing) # total number of frames the
                                              #    movie will plot
```

```
# < Exercise 13, Function to update the figure for each frame of the movie >

# The function called by FuncAnimation to create the frames
def animate(i):
    # This makes the plot by moving data into line's set_data method

    j = frame_spacing*i  # To keep the movie running fast, we can choose to not
                         #    plot every frame

    t = r_soln.t[j]  # time step of the frame. Needed to calculate the position
                     #    of the puck and edge marker

    # Marker moving on the edge of the turntable
    edgemarker.set_center([pos_marker(t, radius, omega)[0],
                           pos_marker(t, radius, omega)[1]])
```

```
    # Marker representing the puck
    puckmarker.set_center([pos_prime(t, r_init_prime)[0],
                           pos_prime(t, r_init_prime)[1]])

    # Since we want to plot the track and not just the position of the puck,
    #    we need to define an array of times from zero to the current time of
    #    the figure.
    t_array = r_soln.t[:j+1]

    # Track in the inertial reference frame
    puck_track.set_data(pos_prime(t_array, r_init_prime)[0, :],
                        pos_prime(t_array, r_init_prime)[1, :])

    # Track in the rotating reference frame
    #    radius in rotating reference frame
    r_track = np.sqrt(x_pos[:j+1]**2 + y_pos[:j+1]**2)
    #    angle in rotating reference frame
    theta_array = np.arctan2(y_pos[:j+1], x_pos[:j+1])
    # Adjust the angle of the non-inertial track by the amount the turntable
    #    has rotated
    x_track = r_track*np.cos(theta_array + omega*t)  # recalculating the
                                                     #    x-coordinate
    y_track = r_track*np.sin(theta_array + omega*t)  # same as above for y
    puck_track_rot.set_data(x_track, y_track)

    return puck_track, puck_track_rot, edgemarker, puckmarker
```

```
# < Exercise 13, Making the movie >

# Define and customize the figure, including plotting the turntable.
fig, ax = plt.subplots()
ax.set_xlim(-radius, radius)
ax.set_ylim(-radius, radius)
ax.grid()
circle = patches.Circle((0, 0), radius = radius, fc = '0.75', zorder = 1)
ax.add_patch(circle)
ax.set_aspect('equal', adjustable = 'box', anchor = 'C')

# Define and customize the markings that will be animated.
puck_radius = 0.2  # Plot size of the circle representing the mass
#  Create the marker at the edge of the turntable
edgemarker = patches.Circle((0,0), puck_radius, fc ='black', ec = 'black',
                            lw = 2 , zorder = 3)  # "zorder = 3" ensures that
                                                  #    it will be plotted on top
#  Create a circle representing the puck
puckmarker = patches.Circle((0,0), puck_radius, fc ='DarkSlateGray',
                            ec = 'black', lw = 2 , zorder = 3)
# The path of the puck in the inertial reference frame
puck_track, = ax.plot([],[], 'b', lw = 2)
# The path of the puck in the rotating reference frame
puck_track_rot, = ax.plot([],[], 'r', lw = 2)
```

```
def init():
    ax.add_patch(edgemarker)
    ax.add_patch(puckmarker)
    return edgemarker, puckmarker

anim = animation.FuncAnimation(fig, animate, init_func = init,
                               frames = num_frames,
                               interval = 20)

display(HTML(anim.to_jshtml()))  # Display as JSHTML

### IF using CoCalc, comment out the above line and comment in the following two
#plt.rcParams['animation.html'] = 'html5'
#HTML(anim.to_html5_video())
```

19.5 Check-out

Exercise 14

Briefly summarize in the cell below the ideas from this unit.

19.6 Challenge Problems

For a further exploration of motion in a noninertial reference frame, you may wish to attempt the following. These exercises are all centered on solving for the analytical solution and comparing it to the numerical solution.

1) Solve for the analytical solution. To do this, solve the two equations by the trick of writing $\eta = x + iy$ and guessing a solution of the form $\eta = e^{-i\alpha t}$. As in the case of the critically damped simple harmonic motion (SHM), you will only get one solution. The other solution has the same form, $x(t) = te^{-\beta t}$ (Eq. 5.43 from *Classical Mechanics*), as we found for the second solution in the damped SHM. Leave your general solution in terms of η and appropriate constants. (Adapted from Problem 9.20 b of *Classical Mechanics*)
2) Solve for the constants of integration. To do this, assume that at time $t = 0$, the position was $\mathbf{r}_0 = (x_0, 0)$ and the velocity was $\mathbf{v}_0 = (v_{x0}, v_{y0})$. Show that

$$x(t) = (x_0 + v_{x0}t)\cos\Omega t + (v_{y0} + \Omega x_0)t\sin\Omega t$$
$$y(t) = -(x_0 + v_{x0}t)\sin\Omega t + (v_{y0} + \Omega x_0)t\cos\Omega t$$

   (Problem 9.20 c from *Classical Mechanics*)
3) Plot your analytical solution and compare it against the numerical solution.

20 Coriolis Force on Earth

In this unit, you will be examining the effect of the Coriolis force near the surface of Earth. Generally, we don't notice the non-inertial forces acting on us as Earth rotates. However, at sufficiently high velocities and over large enough distances, the Coriolis force can have real-world effects. For instance, it drives weather patterns, and urban legend has it that it threw off British aiming in a World War I battle near the Falkland Islands.

For the sake of simplicity, we will ignore the centrifugal force in this unit even though it frequently also has a component tangent to the surface of Earth.

20.1 Objectives

In this unit, you will:

- determine how to express the Coriolis force in spherical coordinates (PHY);
- numerically calculate the trajectory of an object constrained to the surface of a sphere under the influence of the Coriolis force (PHY);
- plot the trajectory on the surface of a sphere (PRO);
- test your code by examining special cases and conserved properties (PHY);
- and apply your code to real-world situations (PHY).

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp  # the differential equation integrator

plt.rcParams["figure.figsize"] = (12,8)  # make figures big for easier viewing
                                         #    (width, height in inches)
# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

20.2 Equations of Motion

Let us assume an object is fixed to the surface of a spherical Earth. In this case, the normal force due to the ground will perfectly balance the radial component of the net force, and there will be zero velocity or acceleration in the radial direction.

To take advantage of this lovely symmetry, we will work the problem in spherical coordinates. As happens with polar coordinates, the unit vectors change direction with position. In spherical coordinates:

$$\begin{aligned} d\hat{r}/dt &= \dot{\theta}\hat{\theta} + \sin\theta\dot{\phi}\hat{\phi} \\ d\hat{\phi}/dt &= -\sin\theta\dot{\phi}\hat{r} - \cos\theta\dot{\phi}\hat{\theta} \\ d\hat{\theta}/dt &= -\dot{\theta}\hat{r} + \cos\theta\dot{\phi}\hat{\phi} \end{aligned}$$

Therefore, when taking the derivatives of the position vector we find

$$\begin{aligned} \vec{r} &= r\hat{r}, \\ \vec{v} &= \dot{r}\hat{r} + r\sin\theta\dot{\phi}\hat{\phi} + r\dot{\theta}\hat{\theta}, \text{ and} \\ \vec{a} &= (\ddot{r} - r\sin^2\theta\dot{\phi}^2 - r\dot{\theta}^2)\hat{r} + (2\sin\theta\dot{r}\dot{\phi} + r\sin\theta\ddot{\phi} + 2r\cos\theta\dot{\phi}\dot{\theta})\hat{\phi} \\ &\quad + (2\dot{r}\dot{\theta} - r\sin\theta\cos\theta\dot{\phi}^2 + r\ddot{\theta})\hat{\theta} \end{aligned}$$

Using these equations for acceleration, we can write down the equations of motion. For example, because we know that the object won't be accelerating in $\hat{r}$, it is trivial to write the equation of motion in that direction:

$$F_r = ma_r = m(\ddot{r} - r\dot{\phi}^2\sin^2\theta - r\dot{\theta}^2) = 0$$

To find the equations of motion in $\hat{\phi}$ and $\hat{\theta}$, we need to be able to express the forces (in this case, the Coriolis force) that can have non-zero components in those directions. As you know, $\vec{F}\_\{cor\} = 2m\vec{v} \times \vec{\Omega} = -2m\vec{\Omega} \times \vec{v}$. For an object on the surface of Earth, $\vec{\Omega} = \Omega\hat{z}$ and $\vec{v} = v_\theta\hat{\theta} + v_\phi\hat{\phi}$.

Exercise 1

On a separate sheet of paper, ***work out*** $\vec{F}_{cor}$ ***for an object on the surface of the Earth in spherical coordinates.*** You will need to use the following relations (use your right hand to verify the directions to yourself before solving for the Coriolis force).

$$\begin{aligned} \hat{z} \times \hat{r} &= -\hat{r} \times \hat{z} = \hat{\phi}\sin\theta \\ \hat{z} \times \hat{\theta} &= -\hat{\theta} \times \hat{z} = \hat{\phi}\cos\theta \\ \hat{z} \times \hat{\phi} &= -\hat{\phi} \times \hat{z} = -\hat{r}\sin\theta - \hat{\theta}\cos\theta \end{aligned}$$

Exercise 2

Now use your equations for the Coriolis force and acceleration in a spherical coordinate system to write the equations of motion in ϕ and θ. Substitute in for v_θ and v_ϕ (if you haven't done so already), simplify as much as possible, and ***solve for*** $\ddot{\phi}$ ***and*** $\ddot{\theta}$.

➤ *Hint:* What is $\dot{r}$?

20.3 Solving the Equations of Motion

Now that you have the equations of motion, it is time to compute the numerical solution.

20.3.1 Writing the `deriv()` function

Exercise 3

Define a function that will return the derivatives for your four variables, $\phi, \dot{\phi}, \theta$, and $\dot{\theta}$. Let Ω be an optional parameter for the `deriv()` function.

```
# < Exercise 3, Deriv function >

# Define the deriv function for use in numerical integration to find the
#    trajectory of an object fixed to the surface of a frictionless,
#    rotating sphere

def deriv(t_now, r_now):   ### ADD parameter for the angular velocity
    """ Document your function here """

    # Derivative definitions:
    dphi_dt = 0  ### FIX
    d2phi_dt2 = 0  ### FIX
    dtheta_dt = 0  ### FIX
    d2theta_dt2 = 0  ### FIX

    return np.array([dphidt, d2phidt2, dthetadt, d2thetadt2])
```

20.3.2 Solving the equations of motion for a set of initial conditions

Exercise 4

Define the parameters of the problem and the initial conditions in the cell below. For now, set the radius of Earth to 1 and the angular speed of Earth, Ω, to 0.1. Let your initial conditions be $\phi_0 = \pi/4$, $\dot{\phi}_0 = 0$, $\theta_0 = \pi/4$, and $\dot{\theta}_0 = 0.1$.

Then use `solve_ivp()` to solve for $0 \leq t \leq 2\pi$. Set the integration time to a reasonable amount (about the time it would take for the object to circumnavigate the world at its initial velocity is a good first estimate). In order to ensure an accurate calculation set `rtol = 1.0e-11` and `atol = 1.0e-11` in your call to `solve_ivp()`. Set the `max_step` parameter so that your integration encompasses at least 100 steps.

```
# < Exercise 4, Defining constants and ICs >

# Define constants
radius = 0  ### FIX, radius of Earth
omega = 0  ### FIX, angular speed

# Define initial conditions
phi0 = 0  ### FIX, initial longitude
theta0 = 0  ### FIX, initial colatitude
dphi_dt0 = 0  ### FIX, initial velocity in phi
dtheta_dt0 = 0  ### FIX, initial velocity in theta
```

```
# Initial values of phi, dphi/dt, theta, and theta/dt in the rotating reference
#    frame
r_init = np.array([phi0, dphi_dt0, theta0, dtheta_dt0])
```

```
# < Exercise 4, Integrate for the trajectory >

### SOLVE for the trajectory of the object
```

Exercise 5

Check your integration by first plotting θ (colatitude) in degrees vs. ϕ (longitude) in degrees. Set the axes of your plot so that the entire possible range of ϕ and θ are visible.

Adjust your plot so that the values of ϕ wrap around. In other words, ϕ should be limited to the range $-\pi < \phi \leq \pi$. To do this, check out the modulus operator, %.

```
# < Exercise 5, Plot of theta vs. phi >

### INCLUDE your plot of theta vs. phi
```

20.3.3 Plotting on the surface of a sphere

You can make a potentially more meaningful plot by plotting in 3-D on the surface of a sphere. The following section of code will walk you through how to do this.

Exercise 6

First, write formulas to calculate the Cartesian coordinates of the object based on ϕ and θ:

```
# < Exercise 6, Transformation to Cartesian coordinates >

x = 0 ### FIX
y = 0 ### FIX
z = 0 ### FIX
```

Exercise 7

The following code plots a wire-frame sphere in 3-D, to which you will add the trajectory.

As you saw in Three-Body Problem, three-dimensional plots may be created very similarly to two-dimensional plots with the following key differences:

1) when creating the axis object, set the keyword argument `projection` to `'3d'`, similarly to how you set `projection = 'polar'` in Spherical Pendulum and Orbits, Keplerian and Not;
2) set data for the z-coordinate along with the x and y data like `ax.plot(x, y, z)`.

Once the axis object has been defined with a 3-D projection, `ax.view_init()` allows you to change the viewing angle. The arguments are the elevation and azimuth of the axes in degrees, respectively.

The wire-frame sphere is plotted using `ax.plot_wireframe(x, y, z)`, where x, y, and z are Cartesian coordinates defining the surface. The keyword arguments `rstride` and `cstride` are short for "row stride" and "column stride" and define the number of lines shown along each direction. In the block of code below, the sphere is defined by creating arrays of values of ϕ and θ and converting them to Cartesian coordinates, assuming a given radius of the sphere.

Add a command to plot the x, y, and z coordinates you calculated above on the sphere using the command `ax.plot(x, y, z)`. If you would like to see the sphere from a different viewing angle, change the values of `elev` and `azim` in `ax.view_init()`.

```
# < Exercise 7, Plotting on the surface of a sphere >

# Define a 3-D plot
fig = plt.figure()
ax = fig.add_subplot(projection = '3d')  # Add a 3-D subplot

# Alternatively, the figure and axes can be defined together, as in the
#    commented-out line below
#fig, ax = plt.subplots(figsize=plt.figaspect(1)*4,
#    subplot_kw={'projection': '3d'})

ax.set_box_aspect(aspect = (1,1,1))  # Set the aspect ratio of the plot to be
                                     #    the same in x, y, and z
ax.view_init(elev = 30, azim = 40) # change the orientation of the plot

# Define and plot a spherical shape
# Set of all spherical angles:
u = np.linspace(0, 2*np.pi, 100)
v = np.linspace(0, np.pi, 100)

# Cartesian coordinates that correspond to the spherical angles:
#    (this is the equation of a sphere):
x_sphere = radius*np.outer(np.cos(u), np.sin(v)) # np.outer computes the outer
                                                 #    product of two vectors,
                                                 #    e.g.,
                                                 #    out[i, j] = a[i] * b[j]
y_sphere = radius*np.outer(np.sin(u), np.sin(v))
z_sphere = radius*np.outer(np.ones_like(u), np.cos(v))  # np.ones_like returns
                                                        #    an array of ones
                                                        #    with the same
                                                        #    shape and type as
                                                        #    a given array
ax.plot_wireframe(x_sphere, y_sphere, z_sphere,
                  rstride = 4, cstride = 4, color = '0.5')  # plot the sphere

### ADD call to plot the trajectory below
```

20.4 Verifying Your Code

Exercise 8

Verify your integration by changing the initial conditions to check at least ***two*** special cases. Explain whether the results are what you predicted. Include your plots below.

```
# < Exercise 8, Special cases >

### INCLUDE solution and plots for two special cases here
```

Another way to verify your code is to examine the speed of the mass over time.

Exercise 9

How do you expect the Coriolis force to affect the speed of the mass?

➤ *Hint:* How much work does the Coriolis force do?

Exercise 10

Revert your initial conditions to those of **Exercise 4**. Include the following two plots below:

1) the speed of the puck, the component of the velocity in ϕ, and the component of the velocity in θ versus time and
2) the fractional amount the speed changes over time $((v - v_{init})/v_{init}$ versus $t)$.

Make any necessary changes to your integrator, and use your velocity plots to ***justify the accuracy of your code below***.

```
# < Exercise 10, Speed >

### INCLUDE solution and plots for speed
```

20.5 Exploring Different Initial Conditions

Exercise 11

Run your code again using additional initial conditions to answer the following questions. You may wish to change the `max_time` for your integration as you explore these.

1) ***What happens to the trajectory as the initial value of ϕ increases? Is this what you would have expected? Why?***

2) ***What happens to the trajectory as the initial value of θ increases? Is this what you would have expected? Why?***

3) ***What happens to the trajectory if the initial velocity is in $\hat{\phi}$ instead of $\hat{\theta}$?***

4) ***What happens to the trajectory as the initial velocity decreases?***

20.6 Integrating with Realistic Parameters

Now try running your code with some realistic values.

Exercise 12*

Change the parameters to reflect Earth's actual radius and angular velocity in SI units.

```
# < Exercise 12, Realistic parameters >

### SET actual values of Earth's angular velocity (omega) and
#    radius (radius)
```

Exercise 13*

Calculate and plot (in colatitude vs. longitude space *and* in physical distance) what the trajectory of a bullet fired east at 350 m/s from Grinnell, IA (92.7247 degrees N, 41.7436 degrees W), will be. For this, you will have to translate the velocity into spherical coordinates. Use `solve_ivp()` to solve for the trajectory of the bullet for $0 \leq t \leq 0.5$ seconds.

(Notice that here we are neglecting how the Coriolis [or centrifugal], force would affect the altitude of the bullet.)

```
# < Exercise 13, Trajectory of bullet fired east at 350 m/s from Grinnell, IA >

### FILL IN code to set initial conditions, integrate, and plot below.
```

Exercise 14*

How far north-south has the bullet traveled after 0.5 seconds? Compare this to the approximate distance traveled east.

```
# < Exercise 14, Calculation of distance traveled north-south and east >

### INCLUDE calculations here
```

Exercise 15*

Now let us assume you are a British gunman in 1914 sailing near the Falkland Islands (latitude of 52 degrees south, longitude 59 degrees west) on the battleship *Inflexible*. You are firing directly at the German battleship *Dresden*, which is ten miles to the south of you. Assuming a muzzle speed of 1,800 feet per second, calculate and plot the trajectory of the shell.

By reading off from a plot, determine how much and in what direction the shell will be deflected after it has traveled ten miles south.

Given the other uncertainties in this situation (the rolling motion of the boat, movement of the target, crosswinds, poor sighting, and so forth), how likely do you think it is that the Coriolis force was primarily responsible for the British missing the* Dresden*? Justify your answer.

```
# < Exercise 15, Falkland Battle >

### FILL IN code to set initial conditions, integrate, and plot below.
```

20.7 Check-out

Exercise 16

Briefly summarize the ideas in this unit.

20.8 Challenge Problems

If you have extra time, consider the following questions. These questions are all centered on adding the centrifugal force.

- Define a new `deriv()` function, and repeat your integration now including the tangential component of the centrifugal force.
- Redo your plots from **Exercises 5–7** (with the same parameters and initial conditions), and compare the trajectories when the centrifugal force is included and not.
- Try starting your mass with zero velocity and calculating its trajectory when the centrifugal force is included.

21 Principal Axes of a Cuboid

Analyzing rigid body motion involves lots of linear algebra. Luckily, computers are very good at solving matrices, which means that not every determinant needs to be determined by you. In this unit, you will find the principal axes of different shaped rectangular cuboids by computationally calculating the eigenvalues and eigenvectors of the inertia tensor. This unit reinforces the idea of principal axes by allowing you to visualize them for different shapes. It also introduces NumPy's linear algebra capabilities and makes use of Python `classes`.

This unit covers material from Ch. 10.2–10.8 of *Classical Mechanics* and references Problem 10.25.

21.1 Objectives

In this unit, you will:

- define an inertia tensor for a solid cuboid (PHY);
- use Python to calculate the moments and principal axes of the inertia tensor (PRO);
- create a class to represent a solid cuboid capable of rotation (PRO).

```
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = (12,8)  # make figures big for easier viewing
                                         #    (width, height in inches)
# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

21.2 Coding Techniques Preamble

21.2.1 Linear algebra with NumPy

Thus far, you have primarily used NumPy for one-dimensional arrays. But you have also seen that its capabilities easily extend to two- and higher-dimension matrices, for instance in Three-Body Problem. One powerful extension of this capability is that it is easy to work

with tensors using the NumPy library. Therefore, we will start this unit with a brief reminder of two-dimensional NumPy arrays.

Defining two-dimensional NumPy arrays

To define a two-dimensional NumPy array with a specific set of elements, you can nest lists together and then convert them to a NumPy array, as shown:

```
nest_list = [[1, 2], [3, 4], [5, 6]]
M_arr = np.array(nest_list)
```

or, more simply,

```
M_arr = np.array([[1, 2], [3, 4], [5, 6]])
```

The result is a matrix of three rows and two columns, as can be checked using the numpy.shape command (in contrast, `len()` only returns the total number of elements). As you have seen before, specific elements can be accessed by providing the index of first the row, then the column.

```
# Examples of two-dimensional NumPy arrays

M_arr = np.array([[1, 2], [3, 4], [5, 6]])

print('Matrix M:\n', M_arr)
print('Shape of Matrix M (rows, columns): ', np.shape(M_arr))
print('Zeroth element of row 1: ', M_arr[1, 0])
print('Row 2: ', M_arr[2,:])
print('Column 1: ', M_arr[:,1])
```

Other ways of creating two-dimensional NumPy arrays include using numpy.zeros or numpy.ones to request an array of a particular shape filled with either zeros or ones and using numpy.reshape to reshape a one-dimensional array into a multidimensional one.

The reverse of `np.reshape` is numpy.flatten, which will turn a multidimensional array into a single-dimensional one. By default, all elements of the first row will appear before all elements of the second row and so on (a.k.a. "row-major order").

```
# Examples of different ways to create and reshape arrays

print("A 3x3 array of zeros:\n", np.zeros((3, 3)))

a_arr = np.linspace(1, 12, num = 12)
a_arr_43 = a_arr.reshape(4, 3)
a_arr_34 = a_arr.reshape(3, 4)
a_arr_26 = a_arr.reshape(2, 6)
print("Using np.linspace to create a one-dimensional array and reshape to make"
      " it two-dimensional\n(4x3):\n {}\n(2x6): \n{}".
      format(a_arr_43, a_arr_26))
print("Flattening a multidimensional array:\n", a_arr_26.flatten())
```

Basic matrix operations

To transpose a NumPy array, you can either use the function `np.transpose(ndarray)` or access the transpose property of the NumPy array object with either `ndarray.transpose()` or `ndarray.T`. Note that none of these methods actually changes the underlying array. To modify the array itself, you will need to assign it the value of the transposed array, as shown below.

```
# Examples of three ways to transpose a NumPy array

M_arr = np.array([[1, 2], [3, 4], [5, 6]])
print("M_arr = \n", M_arr)

print("np.transpose(M_arr) = \n", np.transpose(M_arr))

print("M_arr.transpose() = \n", M_arr.transpose())

print("M_arr.T = \n", M_arr.T)

print("M_arr: \n", M_arr)

M_arr = M_arr.T

print("M_arr now, after reassigning it the value of its transpose: \n", M_arr)
```

Technically, you cannot take the transpose of a one-dimensional NumPy array (it will just return the same array). To transpose a vector, you need to use a double set of square brackets to define it as a two-dimensional array that only has one row or column.

```
# Example of trying to transpose a vector, first defined as a one- and then
#    two-dimensional array

vec = np.array([1, 2, 3])

print("vector: {}\nvector.T: {}".format(vec, vec.T))

vec = np.array([[1, 2, 3]])

print("\nvector: {}\nvector.T: \n{}".format(vec, vec.T))
```

Another basic matrix operation you will surely encounter is matrix multiplication. This can be done by providing numpy.matmul with the two matrices as the arguments of the function. The same task can be done using the `@` operator. Examples of both follow. Be careful about the shape of the arrays you are trying to multiply, as the number of columns of the first must match the number of rows of the second.

```
# Example of multiplying two-dimensional arrays with matrix multiplication

print("a_arr_43 ({}x{}):\n{}".format(np.shape(a_arr_43)[0],
                                     np.shape(a_arr_43)[1], a_arr_43))
```

```
M_arr = np.array([[1, 2], [3, 4], [5, 6]])

print("M_arr ({}x{}):\n{}".format(np.shape(M_arr)[0],
                                  np.shape(M_arr)[1], M_arr))

# using np.matmul
print("\na_arr_43 x M_arr:\n", np.matmul(a_arr_43, M_arr))

# using the @ operator
print("Or equivalently, using the @ operator:\n", a_arr_43 @ M_arr)

### UNCOMMENT the following line to see the error message
# print("\nM_arr x a_arr_43:\n", np.matmul(M_arr, a_arr_43))
```

Finally, many of the more advanced commands, including those to calculate the eigenvectors and eigenvalues of a matrix, are stored in the `numpy.linalg` library. (You have also used `np.linalg.norm` to find the magnitude of a vector in The Three-Body Problem.)

Examples of how to find the determinant, and eigenvector and eigenvalues for a particular array using numpy.linalg.det, and numpy.linalg.eig follow.

```
# In preparation for using NumPy to find eigenvectors, load the
#    numpy.linalg library
from numpy import linalg
```

```
# Examples of finding the determinant, eigenvectors, and eigenvalues of a
#    numpy array

b_arr = np.array([[1, 0],[0, 2]])

print("b_arr:\n", b_arr)

print("Determinant of b_arr: {}".format(linalg.det(b_arr)))

# Eigenvector and (real) eigenvalues are calculated simultaneously with
#    linalg.eig
w_arr, v_arr = linalg.eig(b_arr)
print("\nEigenvalues:")
print(w_arr)
print("\nEigenvectors:")
print(v_arr)
```

21.2.2 Classes

Throughout these units you have written functions and strung them together to accomplish your goals. As such, you have primarily been working in a "functional programming" style, in which the functions themselves were the basic building blocks of your program.

Python, however, is also highly supportive of a different style of programming, known as "object-oriented" programming. In this style of programming, the basic units are "objects" that contain some combination of variables (a.k.a. "attributes") and/or functions (a.k.a. "methods"). A "class" is a template that contains descriptions for what information is to be stored and how it can be modified. An "object" is a particular instance of a class. For example, you could define a class, `car`, that allowed for information about gas mileage, amount of fuel, and present location to be stored along with methods for changing its location. Then you could create multiple instances of that class (e.g., `car_1`, `car_2`), each with its own unique mileage, amount of fuel, and location. In this example, `car_1` and `car_2` are both objects of the type class `car`.

You have actually been using objects throughout these units. For example, the NumPy arrays you have defined are all objects of class `ndarray`. Likewise, when you have used the familiar `fig, ax = plt.subplots()` command, you created an object `ax` of the `matplotlib.axes.Axes` class. You then modified that object using commands such as `ax.plot(x, y)`.

One way you can recognize objects in your code is that a period is used to separate the name of the object (e.g., `ax`) from the reference to the attribute or method for that object (e.g., `plot()`).

```
# An illustration of objects using NumPy arrays

# Define an object of class ndarray
a_arr = np.array([1, 2, 3])

# Define another object of the same ndarray class
b_arr = np.array([4, 7, -1, 5])

# Verify the types of these objects
print("a_arr is an instance of ", type(a_arr))
print("b_arr is an instance of ", type(a_arr))

# Determine the shape of the array, either by using a function or by accessing
#    the shape information stored as an attribute of the object
print("\nThe shape of a_arr can be found by using np.shape(a_arr)"
      " [{0}]\n\tor a_arr.shape [{1}]".format(np.shape(a_arr), a_arr.shape))

# Methods can be used to modify the object, as well as return specific
#    information about it.
print("\nThe method ndarray.max() returns the maximum value of the array.\n"
      "\tTherefore, b_arr.max() = {}".format(b_arr.max()))

b_arr.sort() # sort b_arr
print("\nThe method ndarray.sort() sorts the array, actually modifying the "
      "array as opposed to returning a copy.\n"
      "Therefore, after applying b_arr.sort(), b_arr now equals {}".
      format(b_arr))
```

While you have used objects extensively, you have not yet created your own classes. That will change in this unit.

21.3 Moment of Inertia Tensor and Angular Momentum

In order to describe the motion of a solid rotating body, one must first be able to describe the body itself. Therefore, you will start by calculating the inertia tensor for a solid body rotating around different pivot points. Once the inertia tensor has been defined, you will be able to use Python's linear algebra capabilities to determine the angular momentum given a rotational vector.

21.3.1 Defining the inertia tensor

For these exercises, you will be working with a uniform cuboid (a rectangular brick shape — perhaps your textbook?) centered on the origin. Let the lengths of the sides be $2a$, $2b$, and $2c$ in the x, y, and z directions, respectively, and the mass be m.

These variables are defined globally below. Later, you will learn how to use them to define a cuboid class.

```
# Defining variables for the rectangular cuboid

a_side = 2 # Half the length of the side parallel to the x-axis
b_side = 1 # Half the length of the side parallel to the y-axis.
c_side = 3 # Half the length of the side parallel to the z-axis.

mass = 3 # Mass of the rectanglar cuboid
```

Exercise 1

Determine all nine elements of the moment of inertia tensor for the cuboid with respect to its center of mass. You should be able to calculate the off-diagonal elements without doing any integration (this task corresponds to Problem 10.25 a from *Classical Mechanics*).

Then fix the code below to define the inertia tensor, keeping all elements in terms of the variables `a_side`, `b_side`, `c_side`, and `mass`.

```
# < Exercise 1, Inertia tensor around the center of mass >

### FIX the inertia tensor
inert_CM = mass*np.array([[0, 0, 0],
                [0, 0, 0],
                [0, 0, 0]])
```

Exercise 2

Now calculate the angular momentum of a cuboid with $m = 3$ kg, $a = 2$, $b = 1$, and $c = 3$ m spun around its center of mass with angular velocity $\vec{\omega} = 5\ \mathrm{s}^{-1}\hat{x}$. Justify to yourself that the answer makes sense.

You will want to make use of NumPy's matrix capabilities. Define an angular momentum vector using NumPy arrays, and then try multiplying (`matmul`) to find $\vec{L} = I\vec{\omega}$.

```
# < Exercise 2, Calculating the angular momentum >

### DEFINE the omega vector as a numpy array

### CALCULATE the angular momentum
```

Exercise 3

Suppose the cuboid were instead to be rotated around the x-axis from the far corner A at (a, b, c). Find the new moment of inertia tensor. (This exercise is equivalent to Problem 10.25 b from *Classical Mechanics*.)

➤ *Hint:* The "generalized parallel axis theorem" (Eq. 10.117 from *Classical Mechanics*) claims that if $\mathbf{I}^{cm}$ denotes the moment of inertia tensor of a rigid body (mass M) about its center of mass, then the corresponding moment of inertia tensor $\mathbf{I}$ about a pivot point P displaced from the center of mass by $\Delta = (\xi, \eta, \zeta)$ is:

$$\mathbf{I}_{xx} = \mathbf{I}^{cm}_{xx} + M(\eta^2 + \zeta^2); \quad \mathbf{I}_{yz} = \mathbf{I}^{cm}_{yz} - M\eta\zeta$$

and so on.

```
# < Exercise 3, Inertia tensor around the corner >

### Fix the definition of the origin
origin = np.array([0,0,0])

### Fix the inertia tensor
#     To ensure code flexibility, reference the origin array and
#     the inertia tensor around the center of mass
inert_P = mass*np.array([[0, 0, 0],
                [0, 0, 0],
                [0, 0, 0]])
```

Exercise 4

Now calculate the angular momentum of a cuboid with $m = 3\,\text{kg}$, $a = 2$, $b = 1$, and $c = 3\,\text{m}$ spun around point P with:

1. angular velocity $\vec{\omega} = 5\ \text{s}^{-1}\hat{x}$
2. angular velocity $\vec{\omega} = 5\ \text{s}^{-1}\hat{y}$.

Again, make use of NumPy's matrix capabilities.

```
# < Exercise 4, Calculating the angular momentum >

### DEFINE the omega vector as a NumPy array

### CALCULATE the angular momentum
```

21.3.2 Creating a class

It would be nice to be able to define different cuboids of different masses and dimensions and then manipulate them in the same way (e.g., find the moment of inertia or change the pivot point). Defining a `class` is a natural way to accomplish this goal and allows us to make use of some of Python's powerful object-oriented features.

In this situation, the object is an instance of a class representing uniformly solid cuboids capable of rotating with their center of mass at the origin. Once this class has been created, you can use it to define different individual cuboids and compare their properties. In particular, this class should allow the user to define the mass and side-lengths of the cuboid. It will then determine the moment of inertia for rotation around the center of mass or a user-defined pivot point. Later in this unit you will modify it to calculate the principal axes and moments.

Exercise 5

The code below defines such a class. Make the requested modifications using the code you wrote above for calculating the moments of inertia.

```
# < Exercise 5, Defining the inertia tensor in a class >

### FIX the code below to define the inertia tensor for both rotation around
#    the center of mass and around a user-defined pivot point

class SolidCuboid():
    """This is a class to represent a solid cuboid of uniform density with its
    center of mass at the origin"""

    # __init__() initializes an instance of the SolidCuboid class.
    #     It is the code that is run when a user defines a particular
    #    SolidCuboid
    def __init__(self, mass = 1, a_side = 1, b_side = 1, c_side = 1):
        # Default mass and half-side lengths are of unit 1

        # Define attributes of the class, self.cm (center of mass) and
        #    self.origin (location of the pivot point)
        self.cm = np.array([0, 0, 0])  # Location of center of mass
        self.origin = self.cm  # Initial location of the pivot point

        # The next block of code defines attributes of the class using the
        #    information the user supplied when initiating it
        self.mass = mass # Mass
        self.a_side = a_side  # Half-side length along x-axis
        self.b_side = b_side  # Half-side length along y-axis
        self.c_side = c_side  # Half-side length along z-axis

        ### FIX
        # Calculate the inertia tensor around center of mass
        self.inert_CM = self.mass*np.array([[1, 0, 0],
                        [0, 1, 0],
                        [0, 0, 1]])
```

```
        # By default, the pivot point is assumed to be the center of mass, so the
        #    default moments of inertia are equal
        self.inert_P = self.inert_CM

    # This method shifts the pivot point to the desired location.
    #    The moment of inertia is then calculated around the new pivot location
    def setPivot(self, origin):
        self.origin = origin # Update the location of the pivot point

        # Calculate the inertia tensor around new pivot point
        ### FIX
        self.inert_P = np.array([[self.inert_CM[0, 0],
                                  self.inert_CM[0, 1],
                                  self.inert_CM[0, 2]],
                                 [self.inert_CM[1, 0],
                                  self.inert_CM[1, 1],
                                  self.inert_CM[1, 2]],
                                 [self.inert_CM[2, 0],
                                  self.inert_CM[2, 1],
                                  self.inert_CM[2, 2]]])
```

To use a class, you will want to define an object that is an instance of that class. The code block below shows how to define an instance of the `SolidCuboid` class "cube" that has the default side-length parameters ($a = b = c = 1$ m) and mass $m = 3$ kg. Once this object has been defined, the class can be used to calculate and access different attributes of this object, such as the center of mass and the moments of inertia around different pivot points.

```
print("Documentation:\n", SolidCuboid.__doc__)

# Define an object of the SolidCuboid class "cube" that has the default
#    side-length parameters and mass 3
cube = SolidCuboid(mass = 3)
# print the center of mass
print("\nDefault pivot point is the center of mass: ", cube.cm)
# moment of inertia for the cube around the center of mass:
print("Moment of inertia about center of mass:\n", cube.inert_CM)
# moment of inertia for the cube around the default pivot location,
#    the center of mass:
print("Moment of inertia about default pivot point:\n", cube.inert_P)
# Change the pivot location to the far corner of the cube
cube.setPivot(np.array([cube.a_side, cube.b_side, cube.c_side]))

print("Placing the pivot point at the far corner of the cube makes the"
      " moment of inertia: ")
print(cube.inert_P)
```

Exercise 6

Following the example above, define an object of the `SolidCuboid` class called `rectangularCuboid` with $m = 3$ kg, $a = 2$, $b = 1$, and $c = 3$ m. Print the moment of inertia about

- the center of mass
- the default pivot point
- a pivot point located at the far corner of the rectangular cuboid

```
# < Exercise 6, Creating and accessing an object >

### DEFINE a new object of the SolidCuboid class, "rectangularCuboid", that
#    has mass = 3, a = 2, b = 1, and c = 3

### PRINT the moment of inertia around different pivot points
```

21.4 Finding the Principal Axes

Once the moment of inertia tensor has been created, you can find the principal axes by computing the eigenvectors. Remember, you can use numpy.linalg.eig to calculate the eigenvalues and vectors.

For example:

```
w_arr, v_arr = linalg.eig(M_arr)
```

returns an array of eigenvalues (`w_arr`) and a two-dimensional array of the eigenvectors (`v_arr`). Each eigenvalue corresponds to a different column in `v_arr` (listed in order), so

```
M_arr @ v_arr[:, i] == w_arr[i] * v_arr[:,i]
```

Unfortunately, `linalg.eig` does not return the eigenvalues in any particular order. Therefore, you will also want to sort the eigenvalues and their corresponding eigenvectors for the sake of consistency. This sorting can be accomplished using the following lines of code after the eigenvalues and eigenvectors have been calculated:

```
idx = w_arr.argsort()
w_arr = w_arr[idx]
v_arr = v_arr[:, idx]
```

Here, `w_arr.argsort()` uses numpy.argsort to return an array of indices that would sort `w_arr` into ascending order. Once the sorting is known, `w_arr = w_arr[idx]` reorders the array of eigenvalues, and `v_arr = v_arr[:, idx]` reorders the eigenvectors in the same way.

Exercise 7

Use `linalg.eig` to find the principal axes (and corresponding eigenvalues) for a uniform rectangular cuboid of $m = 3$ kg, $a = 2$, $b = 1$, and $c = 3$ m rotated around its far corner. Sort the eigenvalues in ascending order and reorder the eigenvectors to match that sorting. Verify that the principal axes are eigenvectors, i.e., that

$$\mathbf{I}\omega_i = \lambda_i \omega_i$$

➤ *Hint:* Because of numerical round-off errors, it is unlikely that the above eigenvalue equation will hold *exactly*, which means using == in your comparison will likely return `False`. Instead, you can either check the values by eye or use numpy.isclose to check for equivalency within a certain precision.

```
# < Exercise 7, Calculating the eigenvalues and eigenvectors >
```

Exercise 8

Now that you have determined how to calculate the principal axes, you can add this attribute to your `SolidCuboid` class. Modify the class you wrote earlier to include a code to calculate the principal axes in both `__init__()` and `setPivot()`. Make sure that the eigenvalues and eigenvectors are sorted, as in **Exercise 7**. If your implementation is correct, the following block of code should run without error after you have recompiled the class definition.

```
# < Exercise 8, Adding the principal axes to SolidCuboid >

# Code for testing. Run after having modified your class.

# Define a rectangular cuboid to be pivoted around the origin
rectangularCuboid_centered = SolidCuboid(mass = 3, a_side = 2,
                                         b_side = 1, c_side = 3)
print("\nA rectangular cuboid of mass m = {:.1f} and with"
      "\nhalf-sides a = {:.1f}, b = {:.1f}, & c = {:.1f}"
      " pivoted around the origin".
      format(rectangularCuboid_centered.mass,
             rectangularCuboid_centered.a_side,
             rectangularCuboid_centered.b_side,
             rectangularCuboid_centered.c_side))
print("Principal moments: ", rectangularCuboid_centered.w_arr)
rectangularCuboid_centered.v_arr = np.flip(rectangularCuboid_centered.v_arr,1)
print("Principal axes: \n", rectangularCuboid_centered.v_arr)


# Define a rectangular cuboid to be pivoted around (a, b, c)
rectangularCuboid_offset = SolidCuboid(mass = 3, a_side = 2, b_side = 1,
                                       c_side = 3)
rectangularCuboid_offset.setPivot(np.array([rectangularCuboid_offset.a_side,
                                            rectangularCuboid_offset.b_side,
                                            rectangularCuboid_offset.c_side]))
```

```
print("\nA rectangular cuboid of mass m = {:.1f} and with"
      "\nhalf-sides a = {:.1f}, b = {:.1f}, & c = {:.1f}"
      " pivoted around (a, b, c)".
      format(rectangularCuboid_offset.mass, rectangularCuboid_offset.a_side,
             rectangularCuboid_offset.b_side, rectangularCuboid_offset.c_side))
print("Principal moments: ", rectangularCuboid_offset.w_arr)
print("Principal axes: \n", rectangularCuboid_offset.v_arr)
```

Exercise 9

Now that you have calculated the principal axes, we have the opportunity to use Python's graphics capability to help visualize them. The second block of code below creates a 3-D projection plot of the rectangular cuboid.

At the end of the block of code, ***add `ax.plot` commands to plot each of the three principal axes.*** The easiest way to do this is to provide a set of two 3-D coordinates that will draw a line extending from the pivot point to a location along the vector some distance from the pivot point.

Once you have plotted the principal axes for the rectangular cuboid pivoted around the origin (`rectangularCuboid_centered`), ***try uncommenting the second line of code and rerunning the cell*** to plot the principal axes for a rectangular cuboid pivoted around the far corner (`rectangularCuboid_offset`).

This block of code uses a couple of new procedures to plot the surfaces of the cuboid. The command `plot_surface()` from the Matplotlib library will draw a surface that intersects with the x, y, and z points provided. In order to calculate those points, the three-dimensional coordinates for each side must be provided. For example, to define the top of the cuboid, this code uses `np.linspace()` to create 1-D arrays of x and y locations. Then it uses numpy.meshgrid to return 2-D arrays of x and y values corresponding to the set of (x, y) coordinates (see the example immediately below). Finally, the corresponding z coordinates are defined to be a 2-D array with the same shape as the x and y coordinates, all set to the value of c.

```
# Example of numpy.meshgrid

# Define the 1-D arrays of x and y values
x = np.linspace(-2, 2, 5)
y = np.linspace(-2, 2, 5)

print('x: ', x)
print('y: ', y)

# Use the above arrays to define a grid of points with a coordinate
#    corresponding to each x and y value. The 2-D arrays defining each of the
#    x and y coordinates are returned.
X, Y = np.meshgrid(x, y)

print('X: \n', X)
print('Y: \n', Y)
```

```
# < Exercise 9, Plot the solid, along with the principal axes >

rectangularCuboid = rectangularCuboid_centered
### UNCOMMENT the following line to see the principal axes for pivot point
#    (a, b, c)

#rectangularCuboid = rectangularCuboid_offset

fig = plt.figure()
# Set the projection to make a 3-D plot
ax = plt.axes(projection = '3d')
# Set the aspect ratio of the plot box to be equal
ax.set_box_aspect(aspect = (1, 1, 1))

# Define the rectangles that will be three sides of the cuboid
#    Define the arrays that go from the minimum and maximum values of the
#    sides.
x = np.linspace(-rectangularCuboid.a_side, rectangularCuboid.a_side, 6)
y = np.linspace(-rectangularCuboid.b_side, rectangularCuboid.b_side, 6)
z = np.linspace(-rectangularCuboid.c_side, rectangularCuboid.c_side, 6)

# Use the plot_surface command to plot a 3-D surface
#    Set the color to blue, the edgecolor to black, and make semi-transparent by
#    setting alpha to 0.4
X, Y = np.meshgrid(x, y)  # define a grid of x and y coordinates for the top
Z = X*0 + rectangularCuboid.c_side  # the z coordinates of the top will all be
                                    #     equal to c
ax.plot_surface(X, Y,    Z,
                alpha = 0.2, color = 'b', edgecolor = 'b')  # top
ax.plot_surface(X, Y, -1*Z,
                alpha = 0.2, color = 'b', edgecolor = 'b')  # bottom

Y, Z = np.meshgrid(y, z) # define a grid of x and y coordinates for the top
X = X*0 + rectangularCuboid.a_side  # the z coordinates of the top will all be
                                     #    equal to a
ax.plot_surface(   X, Y, Z,
                alpha = 0.2, color = 'b', edgecolor = 'b')  # right
ax.plot_surface(-1*X, Y, Z,
                alpha = 0.2, color = 'b', edgecolor = 'b')  # left

X, Z = np.meshgrid(x, z) # define a grid of x and y coordinates for the top
Y = X*0 + rectangularCuboid.b_side  # the z coordinates of the top will all be
                                    #    equal to b
ax.plot_surface(X,    Y, Z, alpha = 0.2, color = 'b', edgecolor = 'b') # Front
ax.plot_surface(X, -1*Y, Z, alpha = 0.2, color = 'b', edgecolor = 'b') # Back

# Set the limits of the axes to be equal to the maximum side
lim = max(rectangularCuboid.a_side, rectangularCuboid.b_side,
          rectangularCuboid.c_side)
ax.set_xlim3d([-1.0*lim, lim])
ax.set_ylim3d([-1.0*lim, lim])
ax.set_zlim3d([-1.0*lim, lim])
```

```
# Draw the axes for ease of visulization
ax.plot([-lim, lim], [0, 0], [0, 0], 'k--')
ax.plot([0, 0], [-lim, lim], [0, 0], 'k--')
ax.plot([0, 0], [0, 0], [-lim, lim], 'k--')

# Label the axes
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("z")

### ADD code here to plot the three principal axes for the object.
#    Add an offset so that the principal axes start at the pivot point.
#    You will likely also find it useful to include a constant scalefactor that
#    can be used to increase (or decrease) the length of the principal axes.
```

21.5 Check-out

Exercise 10

Briefly summarize the ideas in this unit.

21.6 Challenge Problem

If you have extra time, consider:

- Systematically changing the location of the pivot point and examining how the principal axes shift.
- Changing the shape of the cuboid and examining how the principal axes shift.

22 Precession of a Cuboid

Previously in Principal Axes of a Cuboid you calculated the principal axes and moments of a rectangular cuboid rotated around different points. In this unit, you will see how those result in the rotation of the rectangular cuboid. Specifically, you will use the calculated principal moments in Euler's equations to computationally solve for the angular momentum over time.

This unit covers material from Ch. 10.2–10.8 of *Classical Mechanics* and references Problem 10.25.

22.1 Objectives

In this unit, you will:

- save your previously defined class representing a solid cuboid capable of rotation as a module and import it (PRO);
- model the evolution of the angular momentum vector over time for cuboids of different axis lengths spun around their centers of mass (PHY);
- analytically determine the frequency of precession if the cuboid has two equal-length sides (PHY);
- use events in `solve_ivp()` to determine the frequency of precession (PRO).

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg

from scipy.integrate import solve_ivp  # the differential equation integrator

plt.rcParams["figure.figsize"] = (12,8)  # make figures big for easier viewing
                                         #    (width, height in inches)
# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

22.2 Creating and Importing Modules

In this unit, you will be working with the SolidCuboid class created in **Exercise 5** of Principal Axes of a Cuboid. You can simply copy the code defining the class to a cell in this notebook. However, there is a more elegant solution, which is to copy the class into a Python file and

import it into this notebook. Writing "modules" (files with code available for import) allows you to easily reuse code for different purposes and generally results in cleaner code.

Exercise 1

Create a new file named `principal_axes.py` located in the same directory as this notebook. (This would require uploading it to CoCalc or to your "Colab Notebooks" directory on Google Drive if you are using Google Colab. If you are using Google Colab, you will also need to mount Google Drive by uncommenting the code in the cell below.) Then copy all of **Exercise 5** of Principal Axes of a Cuboid into the file and save it. At the top of the file, be sure to include

```
import numpy as np
from numpy import linalg
```

as these libraries are necessary for methods within the class.

The next step is to import everything contained in this file (a.k.a. module). You could do this by running the following statement:

```
import principal_axes
```

This line tells Python that you wish to import the code contained in "principal_axes.py".

Oftentimes, though, you will want to use a shortened nickname or "alias" for the module. For example, `numpy` is generally shortened to `np`. In that case, your code would look something like

```
import principal_axes as pa
```

Once imported, you should be able to access the class as you did in Principal Axes of a Cuboid, with one minor modification. Now when you call the SolidCuboid class, you need to add `pa.` as a prefix. For example,

```
cube = pa.SolidCuboid(mass = 3)
```

```
# FOR GOOGLE COLAB ONLY

# < Exercise 1, Importing a class >

### UNCOMMENT the following lines if you are using Google Colab

#from google.colab import drive ## Mount your Google Drive
#drive.mount('/content/drive/')

#import sys  # Append your Google Drive "Colab Notebooks" directory to your
             #    path so that files contained in it may be found
#sys.path.append('/content/drive/My Drive/Colab Notebooks/')
```

```
# < Exercise 1, Importing a class >

# Import the class "SolidCuboid" from principal_axes
import principal_axes as pa
```

```
# See the documentation about the class
help(pa.SolidCuboid)
```

```
# < Exercise 1, Importing a class, testing >

# Define an object of the SolidCuboid class "cube" that has the default
#    side-length parameters and mass 3
cube = pa.SolidCuboid(mass = 3)
# Print the center of mass
print("\nDefault pivot point is the center of mass: ", cube.cm)
# moment of inertia for the cube around the center of mass:
print("Moment of inertia about center of mass:\n", cube.inert_CM)
# moment of inertia for the cube around the default pivot location,
#    the center of mass:
print("Moment of inertia about default pivot point:\n", cube.inert_P)
# Change the pivot location to the far corner of the cube
cube.setPivot(np.array([cube.a_side, cube.b_side, cube.c_side]))

print("Placing the pivot point at the far corner of the cube makes the moment"
      " of inertia: ")
print(cube.inert_P)
```

Exercise 2

Now that you have imported your `SolidCuboid` class, use it to define the cuboid, `rectangularCuboid_centered`, which has mass = 3 kg, side lengths a = 2, b = 1, and c = 3 m, and pivots around the origin.

```
# < Exercise 2, Define an object >

# < Exercise 2, Define an object, Instructor solution >
# DEFINE a new instance of the SolidCuboid class, "rectangularCuboid_centered",
#    with mass = 3, a = 2, b = 1, and c = 3 and is pivoted around the origin
```

22.3 Euler's Equations

Given the principal axes and principal moments for a body, one can use Euler's equations to determine the rotation of the body over time. If the body is subject to zero torque, Euler's equations reduce to:

$$\lambda_1\dot{\omega}_1 = (\lambda_2 - \lambda_3)\omega_2\omega_3$$
$$\lambda_2\dot{\omega}_2 = (\lambda_3 - \lambda_1)\omega_3\omega_1$$
$$\lambda_3\dot{\omega}_3 = (\lambda_1 - \lambda_2)\omega_1\omega_2$$

(Equation 10.89 from *Classical Mechanics*)

Exercise 3

Write a `deriv()` function that will return an array of $\dot{\omega}_1$, $\dot{\omega}_2$, and $\dot{\omega}_3$. Provide the array of principal moments as one of the arguments in the `deriv()` function.

```
# < Exercise 3, Deriv function for a rotating solid body subject to zero
#    torque >

### DEFINE your deriv function
```

Exercise 4

Next, use your `deriv()` function and `solve_ivp()` to find the temporal evolution of ω_1, ω_2, and ω_3 for your solid rectangular cuboid ($m = 3$ kg, $a = 2$, $b = 1$, and $c = 3$ m) when rotated around the center of mass.

Let the initial rotation be primarily aligned with the $\vec{e}_2$ principal axes (the one with the intermediate moment and aligned with the x-axis). Because your `deriv()` function assumes that the values of $\vec{\omega}$ are relative to the different principal axes, your values of `omega_init` should be relative to the principal axes $\vec{\omega}_{e,0} = (\omega_{e_1}, \omega_{e_2}, \omega_{e_3})$, rather than in Cartesian coordinates $\vec{\omega}_0 = (\omega_x, \omega_y, \omega_z)$. Therefore, use $\vec{\omega}_0 = (0.01, 0.5, 0.0)$ and *not* $\vec{\omega}_0 = (0.5, 0.01, 0.0)$ when defining `omega_init`, even though the rotation is happening primarily along the x-axis.

Integrate your solution for 100 seconds and set the maximum time step to 0.1 seconds. Plot ω_1, ω_2, and ω_3 versus time on the same axes.

➤ *Hint:* You will need to pass an array of the principal moments as an argument in `solve_ivp()`. Make use of your previously defined object `rectangularCuboid_centered` by using a call like `args = (rectangularCuboid_centered.w_arr,)`.

Is rotation around this axis stable? How can you tell?

```
# < Exercise 4, Solving for rotation of rectangular cuboid over time >

### SET the initial angular velocity mainly along e2

### SOLVE with solve_ivp

### PLOT omega_1, omega_2, and omega_3 over time
```

Exercise 5*

Now try letting the initial rotation of the same rectangular cuboid be primarily along e_3: $\vec{\omega}_0 = (0, 0.01, 0.5)$ s^{-1}. Repeat the same steps as in **Exercise 4**.

Is rotation around this axis stable? How can you tell?

```
# < Exercise 5, Solving for rotation of rectangular cuboid over time >

### SET the initial angular velocity mainly along e1

### SOLVE with solve_ivp

### PLOT omega_1, omega_2, and omega_3 over time
```

22.3.1 Free precession for a body with two equal principal moments

The motion of a body subject to zero external torque is much simplified if two of the principal moments are equal. For example, if $\lambda_1 = \lambda_2$, the final equation of motion

$$\lambda_3 \dot{\omega}_3 = (\lambda_1 - \lambda_2)\omega_1 \omega_2$$

reduces to $\dot{\omega}_3 = 0$. In other words, ω_3 is constant.

Exercise 6

Use your `SolidCuboid` class to create a cuboid of mass $m = 3$ kg for which the length of the first and second (x and y) sides are equal: $a = b = 3$ m, $c = 1$ m. Print the principal moments corresponding to rotation around the center of mass, and confirm that the first two principal moments are equal.

```
# < Exercise 6, Rectangular cuboid with two equal principal moments >

### CREATE the object from the SolidCuboid class

### VERIFY that two of the principal moments are equal
```

Exercise 7

Use `solve_ivp()` to calculate the components of the angular velocity, ω, as a function of time, and plot them on a single graph for at least five cycles of the precession. Feel free to choose any value for the initial angular velocity provided there is at least some motion along either of the two principal axes with equal moments.

> ***How does your plot illustrate that the third component of the angular velocity (the one corresponding to the unequal principal moment) is constant, as measured in the body frame?***

```
# < Exercise 7, Solving for rotation of rectangular cuboid with two equal
#    principal moments >

### SET the initial angular velocity

### SOLVE with solve_ivp

### PLOT omega_1, omega_2, and omega_3 over time
```

Following the derivation in "Motion of a Body with Two Equal Moments: Free Precession" from Section 10.8 of *Classical Mechanics*, when $\lambda_1 = \lambda_2$, the first two Euler equations become

$$\dot{\omega}_1 = \Omega_b \omega_2$$
$$\dot{\omega}_2 = -\Omega_b \omega_1$$

where $\Omega_b = \frac{\lambda_1 - \lambda_3}{\lambda_1}\omega_3$. This constant angular frequency, Ω_b, is the angular frequency at which the magnitude of both ω_1 and ω_2 oscillate.

Exercise 8

Calculate Ω_b for your solid body using the same initial angular velocity as before.

Ω_b ***(analytically calculated)*** =

If $\lambda_1 > \lambda_3$, $\Omega_b < 0$. ***What does a negative value of*** Ω_b ***mean physically?***

```
# < Exercise 8, Calculate Omega_b >

### INCLUDE your calculations here
```

Exercise 9

How well do the angular frequencies of oscillations of ω_1 and ω_2 in your computational solution compare with the analytically derived angular frequency you calculated in **Exercise 8**? To answer this question, you will need to be able to measure the period of oscillations of ω_1 (or ω_2).

In Simple Pendulum with Large Angle Release, you calculated the period of a pendulum using `events` in `solve_ivp()` to record times at which it crossed zero degrees. In Motion on a Turntable, you also used events to terminate the integration when the puck left the edge of the turntable.

Recall, all you need to do to use events is define a function that goes to zero for the desired occurrence, then call it from the integrator with the keyword `events =`. The function call has to take the same arguments as for the `deriv()` function. For example, you could define an event like

```
def event_name(t_now, r_now, param1, param2, etc.):
```

that triggers when the function returns zero. Then to use the event, add the parameter `events = event_name` to your `solve_ivp()` call. The time of the event triggering is then recorded in `r_soln.t_events`, assuming `r_soln` is the name of the return from `solve_ivp()`. The contents of `r_soln.t_events` will actually be a list of arrays, with each element in the list corresponding to a different event definition. So if you wanted to access those times the first event was triggered, you would use `r_soln.t_events[0]`.

In this case, you will want events to be triggered only if the zero point is crossed from a particular direction. To do this, set `event_name.direction` to a positive float so that only negative-to-positive crossings are recorded.

➤ *Hint:* `np.diff()` will determine the difference between consecutive elements in an array.

Ω_b ***(numerically determined)*** =

```
# < Exercise 9, Determine Omega_b from numerical solution, define event >

### DEFINE an event by creating a function that returns zero when you want the
#    event triggered


### ADD code to set the direction of the event,
#    e.g., eventName.direction = 1.0 or eventName.direction = -1.0
```

```
# < Exercise 9, Determine Omega_b from numerical solution,
#    calculate frequency >

### NUMERICALLY INTEGRATE for the rotation of the cuboid using
#    "events = eventName" in the solve_ivp call


### CALCULATE the mean period from the times stored as t_events in the solution
#    *Hint* remember that 'np.diff' returns the difference between consecutive
#    values in an array.
### CALCULATE the angular frequency
```

Exercise 10

Argue for how the plot in **Exercise 7** describes conical motion of the angular momentum vector in the body frame at an angular frequency equal to the constant angular frequency, Ω_b.

22.4 Check-out

Exercise 11

Briefly summarize the ideas in this unit.

22.5 Template Code for principal_axes.py

```
# Define the Solid Cuboid class as in the solution to Exercise 5
#    of principalAxes.ipynb

import numpy as np
from numpy import linalg

class SolidCuboid():
    """This is a class to represent a solid cuboid of uniform
    density with its center of mass at the origin"""

    # __init__() initializes an instance of the SolidCuboid class.
    #    It is the code that is run when a user defines a
    #    particular SolidCuboid
    def __init__(self, mass = 1, a_side = 1, b_side = 1,
                 c_side = 1): # Default mass and half-side lengths
                              #   are of unit 1


    # This method shifts the pivot point to the desired location.
    #    The moment of inertia is then calculated around the new
    #    pivot location.
    def setPivot(self, origin):
```

Masses Connected with Springs

23

When two oscillators (masses and springs or pendula, for example) are brought into proximity, they can influence one another via the (coupling) force between them, causing a shift in their natural frequencies. Strong coupling leads to two distinct frequencies, wherein the two oscillators can move in phase or out of phase with each other. In the limit of weak coupling, we can observe slow energy flow from one oscillator to the other, causing a beat pattern.

Fourier analysis is introduced in the final section of this unit, and in **Exercises 15** and **16** a fast Fourier transform algorithm is used to identify the normal modes and frequencies from the trajectory of an oscillator. Fourier analysis was introduced in Chapter 5.7 of *Classical Mechanics* as an optional topic and may be skipped here, if not previously covered. The first two exercises may also be completed ahead of time.

23.1 Objectives

In this unit, you will:

- solve analytically for two identical coupled oscillators, along the lines of Chapter 11 of *Classical Mechanics*, using the characteristic equation and solving the quadratic equation, yielding the normal mode frequencies and amplitude ratios (PHY);
- use `solve_ivp()` to model the behavior of the oscillators and plot both normal modes and non-repeating patterns (PHY);
- use NumPy's linear algebra package to solve the eigenvalue problem (PRO);
- extend the general solution to three masses (PHY);
- examine the behavior of weakly coupled oscillators (PHY);
- create an animation showing the motion of the masses (PRO);
- use Fourier analysis to find the normal frequencies and reconstruct the trajectory of the oscillators (PRO).

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp  # the differential equation integrator
from numpy import linalg  # numpy's linear algebra package
```

```
# For the animation
import matplotlib.patches as patches # Imports code to draw a circle
from matplotlib import animation     # animation package (for later)
from IPython.display import HTML     # for displaying animation inline

plt.rcParams["figure.figsize"] = (12,8)  # make figures big for easier viewing
                                         #    (width, height in inches)
# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

23.2 Two Carts Connected and Attached between Two Walls by Three Springs

We will start our exploration of coupled oscillators with two masses attached by three springs. Consider two carts attached to each other and fixed walls by springs, as in Figure 11.1 of *Classical Mechanics*. Let the spring attaching cart 1 (of mass m_1) to the wall have spring constant k_1, the spring attaching both carts to each other have spring constant k_2, and the spring attaching cart 2 (of mass m_2) to its neighboring wall have spring constant k_3.

Exercise 1

On a separate piece of paper, draw a free-body diagram for cart 1 and use it to write Newton's second law for cart 1, noting that as spring k_1 stretches by x_1, spring k_2 stretches by $x_2 - x_1$.

Repeat for cart 2.

Write these two equations in the form $F_1(x_1, x_2)$ and $F_2(x_1, x_2)$ to get:

$$m_1\ddot{x}_1 = -(k_1 + k_2)x_1 + k_2x_2,$$
$$m_2\ddot{x}_2 = k_2x_1 - (k_2 + k_3)x_2,$$

which can be written as a matrix equation:

$$\mathbf{M}\ddot{\mathbf{x}} = -\mathbf{K}\mathbf{x}$$

with

$$\mathbf{M} = \begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix}$$

and

$$\mathbf{K} = \begin{bmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 + k_3 \end{bmatrix}.$$

The procedure followed by Chapter 11 of *Classical Mechanics* is to assume a solution wherein both carts oscillate simultaneously at some frequency ω and phase angle δ such that

$$x_1(t) = \alpha_1 \cos(\omega t - \delta_1) \text{ and } x_2(t) = \alpha_2 \cos(\omega t - \delta_2).$$

This could equally be written as a pair of sines:

$$y_1(t) = \alpha_1 \sin(\omega t - \delta_1) \text{ and } y_2(t) = \alpha_2 \sin(\omega t - \delta_2),$$

or more compactly combined via Euler's identity as:

$$z_1(t) = x_1(t) + iy_1(t) = \alpha_1 e^{i(\omega t - \delta_1)} = a_1 e^{i\omega t},$$

with

$$a_1 = \alpha_1 e^{-i\delta_1}.$$

Similarly,

$$z_2(t) = x_2(t) + iy_2(t) = \alpha_2 e^{i\omega t - \delta_2} = a_2 e^{i\omega t}.$$

Writing this solution as a matrix (Equation 11.10 from *Classical Mechanics*),

$$\mathbf{z}(\mathrm{t}) = \begin{bmatrix} z_1(t) \\ z_2(t) \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} e^{i\omega t} = \mathbf{a}\mathrm{e}^{i\omega t}. \tag{23.1}$$

We then follow the real-part convention wherein the result is the real part of the complex $\mathbf{z}(\mathrm{t})$ solution.

Note that the **K** matrix is tridiagonal and symmetric, in that spring k_2 (the coupling spring) appears as negative off-diagonal terms to either side of the diagonal. SciPy's linear algebra package includes optimized eigenvalue routines for solving these coupled equations and returning the sorted eigenvalues and their eigenvectors — a capability you will make use of soon.

Exercise 2

Complete the following steps to arrive at the solution algebraically.

On a separate piece of paper, find $\ddot{\mathbf{z}}$, then substitute $\mathbf{z}$ and $\ddot{\mathbf{z}}$ into the matrix equation instead of $\mathbf{x}$, and $\ddot{\mathbf{x}}$ and rearrange to get:

$$(\mathbf{K} - \omega^2\mathbf{M})\mathbf{a} = 0.$$

Substituting in for **M** and **K**, the matrix on the left side becomes

$$\begin{bmatrix} (k_1 + k_2) - m_1\omega^2 & -k_2 \\ -k_2 & (k_2 + k_3) - m_2\omega^2 \end{bmatrix}$$

which we will refer to as the "KM-matrix". To find non-trivial solutions, require that the determinant of the matrix = 0.

23.3 Identical Masses and Springs

Classical Mechanics shows solutions for two limiting cases with equal masses: 1) all springs are equal (Chapter 11.2), or 2) the two outer springs are equal and k_2 is much weaker (Chapter 11.3). Let us work with the completely symmetrical first case, setting all springs and masses equal: $k_1 = k_2 = k_3 \equiv k$, and $m_1 = m_2 \equiv m$.

23.3.1 Analytical solution

Write down the equation for the determinant. Put it in standard form so that ω^4 is the leading term. Solve the resulting quadratic equation for ω^2. You should find:

$$\omega^2 = \frac{4k/m}{2} \pm \frac{\sqrt{16(k/m)^2 - 12(k/m)^2}}{2},$$

$$\text{so } \omega_1 = \sqrt{\frac{k}{m}} \text{ and } \omega_2 = \sqrt{\frac{3k}{m}}$$

showing the two normal mode frequencies, one (ω_1) at the normal mode frequency of the two oscillators without the center coupling spring k_2, and the other (ω_2) pushed higher in frequency.

To complete the problem, substitute each solution, one at a time, into the matrix equation of motion to see that at the lower frequency the amplitudes of **x** have the same sign and hence are in phase, whereas the higher frequency mode requires that the two amplitudes are out of phase, as indicated by their opposite signs. We refer to these as *even* and *odd* solutions, respectively, just as $\cos(x)$ is an even function because $\cos(x) = \cos(-x)$, and $\sin(x)$ is an odd function because $\sin(x) = -\sin(-x)$.

Now that you have solved the symmetrical case by hand, we can plot the solutions by taking the real part (a cosine term) at the appropriate frequency.

Exercise 3

Plot the lower-frequency (even) solutions for both x_1 and x_2 from 0 to 100 seconds. Let $k = 1$ N/m and $m = 1$ kg. For best results, make your plots of x_1 and x_2 two different vertically stacked subplots of the same figure.

```
# < Exercise 3,  Plot the low mode solutions from t = 0 to 100 s >

k_const = 1  # spring constants
mass = 1  # mass of each cart

### INCLUDE code to calculate x1 and x2 for an array of times

### PLOT x1 and x2 versus time
```

Exercise 4

Next, plot the higher-frequency (odd) solutions for both x_1 and x_2 from 0 to 100 seconds. As in **Exercise 3**, let $k = 1$ N/m and $m = 1$ kg. Again, for best results, make your plots of x_1 and x_2 two different vertically stacked subplots of the same figure.

```
# < Exercise 4,  Plot the high mode solutions from t = 0 to 100 s >

### INCLUDE code to calculate x1 and x2 for an array of times

### PLOT x1 and x2 versus time
```

The general solution for this system is

$$\mathbf{x}(\mathrm{t}) = A_1 \begin{bmatrix} 1 \\ 1 \end{bmatrix} \cos(\omega_1 t - \delta_1) + A_2 \begin{bmatrix} 1 \\ -1 \end{bmatrix} \cos(\omega_2 t - \delta_2)$$

(Equation 11.21 from *Classical Mechanics*).

In this equation, $\begin{bmatrix}1\\1\end{bmatrix}$ and $\begin{bmatrix}1\\-1\end{bmatrix}$ are the two normal modes (eigenvectors) that correspond to the normal frequencies, ω_1 and ω_2 (the eigenvalues). The constants, A_1, δ_1, A_2, and δ_2, are determined from the initial conditions. When finding these constants, though, it is often more convenient to rewrite the general solution as a sum of sines and cosines:

$$\mathbf{x}(\mathrm{t}) = (B_1 \cos\omega_1 t + C_1 \sin\omega_1 t)\begin{bmatrix}1\\1\end{bmatrix} + (B_2 \cos\omega_2 t + C_2 \sin\omega_2 t)\begin{bmatrix}1\\-1\end{bmatrix}$$

(see Problem 11.7 from *Classical Mechanics*). In the following exercise, you will solve for these constants and plot the resulting motion.

Exercise 5

Suppose cart 1 starts from rest at $x_1 = 1$ and cart 2 starts from rest at $x_2 = 0$. This situation does not correspond to either of the two normal modes, so the resulting motion must be a linear combination of both modes.

Analytically solve for B_1, C_1, B_2, and C_2 for these initial conditions. Then plot the x_1 and x_2 versus time for $0 \le t \le 100$ s as two vertically stacked plots. On the x_1 versus time plot, also show the components of the two normal modes ($B_1 \cos\omega_1 t + C_1 \sin\omega_1 t$ and $B_2 \cos\omega_2 t + C_2 \sin\omega_2 t$ for this cart). Do the same for the x_2 versus time plot. By examining the curves, it should be clear that the motion of the carts results from summing the normal modes.

```
# < Exercise 5, non-normal mode motion >

### CALCULATE B1, C1, B2, and C2

### PLOT x1 and x2 versus time, along with the normal modes for each
```

23.3.2 Numerical solution

In **Exercises 3** through **5**, you manually plotted your analytical solutions. Here, you will instead numerically simulate the positions and velocities of each mass by integrating the equations of motion. (Later, that will allow you to add friction or other forces.)

To do this numerical integration, you will need a generalized derivative function (`derivN`) that applies Newton's second law to each of the masses. It takes in a square KM-matrix of any size N × N and returns the 2N derivatives arranged in the appropriate v, $\frac{dv}{dt}$ order for each mass. You will call this function when running `solve_ivp()`.

```
# Define generalized derivative function for N identical masses

def derivN (t_now, r_now, km_matrix):
    """This function takes the derivatives for N arbitrary equal-mass
    oscillators, given a square km-matrix size NxN.  Derivatives are returned
    in velocity, acceleration order for each mass."""
```

```
    # Extract the x_i values, every other element of the array starting at 0
    x_arr = r_now[::2]  # the final "2" tells it to return every second
                        #    element
    # Extract the v_i values, every other element of the array starting at 1
    v_arr = r_now[1::2]  # the initial "1" tells it to start at element 1,
                         #    the final "2" tells it to return every second
                         #    element

    dx_arr_dt = v_arr  # set the new x derivative equal to the old velocity
    dv_arr_dt = -np.dot(km_matrix, x_arr)  # set the v time derivatives by
                                           #    multiplying km-matrix times
                                           #    x-array

    # A little tricky bit:
    #    First, we set aside an empty array of the right size and shape to
    #    accommodate the 2N derivatives:
    s = np.empty((2*dx_arr_dt.size,), dtype=dx_arr_dt.dtype)
    #   then these two lines interleave the result:
    s[0::2] = dx_arr_dt  # put velocities into the output array (every other
                         #    index starting at 0)
    s[1::2] = dv_arr_dt  # put accelerations into the output array (every other
                         #    index starting at 1)

    return s  # returns the derivatives in proper order v1, a1, v2, a2, etc.,
              #    for each mass
```

Exercise 6

Return to the KM-matrix from the two equal-mass, three equal-spring case (**Exercise 2**). Use `solve_ivp()`, calling `derivN()` to simulate the positions of the masses from 0 to 100 seconds (let `max_step` equal 0.1). Start the masses from rest with each displaced in such a way as to produce pure sinusoidal motion in the lower mode, $\omega_1 = \sqrt{\frac{k}{m}}$. As before, plot both x_1 and x_2 in separate subplots in the same figure.

```
# < Exercise 6, Solve and plot low mode positions vs. time using solve_ivp() >

# Define the KM-matrix
km_matrix = np.array([[2,-1], [-1,2]])

### SET the initial positions and velocities

### SOLVE using solve_ivp

### PLOT x1 and x2 versus t
```

Exercise 7*

Repeat **Exercise 6** using initial conditions that will produce the higher-frequency mode, $\omega_2 = \sqrt{\frac{3k}{m}}$.

```
# < Exercise 7, Solve and plot the high mode positions vs. time using
#    solve_ivp() >

### SET the initial positions and velocities

### SOLVE using solve_ivp

### PLOT x1 and x2 versus t
```

Exercise 8

Now start the masses from rest with $x_1 = 1$ m, $x_2 = 0$ m. Because this is not an eigenmode, you should see the effect of an arbitrary displacement, with oscillations in both modes. Your graphs should match the motion charted in **Exercise 5** exactly.

```
# < Exercise 8, Plot of x1 = 1, x2 = 0, from rest >

### SET the initial positions and velocities

### SOLVE using solve_ivp

### PLOT x1 and x2 versus t
```

Exercise 9*

One can also use an animation to help visualize the motion. By representing the carts as circular patches, an animation can show how the location in x changes with time (the y position can always remain zero). Fill in the code below to create such an animation.

Because the values of x_1 and x_2 recorded in the solution are the distances from equilibrium, you will have to add an offset. For instance, you can assume that the equilibrium position of cart 1 is at -1.5 m and the equilibrium position of cart 2 is at 1.5 m.

➤ *Hint:* The mechanics of this animation are very similar to **Exercise 13** from Brachistochrone Problem, which also used circular patches to animate motion of masses. You can also read more about animation in Appendix 3: Creating Animations with FuncAnimation.

```
# < Exercise 9, Animation >

### DEFINE arrays of positions for cart 1 and positions for cart 2

# Set up the plot
ball_radius = 0.05 # Plot size of the circle representing the carts
fig, ax = plt.subplots()
ax.set_xlim(-3, 3)
ax.set_ylim(-2, 2)
ax.set_xlabel("x")
ax.set_title("Coupled Oscillators\n")
```

```
### USE patches.Circle to define circles representing carts 1 and 2;
#    call them ball1 and ball2 and let their radii be "ball_radius"
ball1 = 0  ### FIX
ball2 = 0  ### FIX

### DEFINE an init function that uses ax.add_patch to add each of the patches
#    to the axes
def init():

    return ball1, ball2

### SET the center of each patch at a particular instance of time = t[i] using
#    ball1.set_center and ball2.set_center
def animate(i):

    return ball1, ball2

### CALL animation.FuncAnimation and display the resulting animation
```

23.3.3 Solving the eigenvalue problem numerically

While the previous case was easily solved analytically, as the number of oscillators increases or damping is included, we may wish to turn to numerical solutions of the eigenvalue problem:

$$(\mathbf{K} - \omega^2 \mathbf{M})\mathbf{a} = 0.$$

Therefore, it is helpful to use one of Python's linear algebra packages to solve the eigenvalue problem numerically, yielding both the normal mode frequencies (from the square root of the eigenvalues) and the eigenvectors. There are many different solvers for eigenvalue problems in the SciPy and NumPy packages. In Principal Axes of a Cuboid, you used numpy.linalg.eig. In situations like this one, where the matrix is known to be symmetric, numpy.linalg.eigh is more efficient. It also has the additional advantage of returning the eigenvalues sorted in ascending order. Otherwise, it can be used the same way as `numpy.linalg.eig()`.

An illustration of how to numerically solve for the frequencies from **Exercise 2** follows. Remember $\sqrt{3} \sim 1.73205$.

```
# Example of using numpy.linalg.eigh

# Define the km_matrix:
km_matrix = np.array([[2,-1],[-1,2]])
print("Here is the KM-matrix: \n", km_matrix)

# Then we get eigenvalues and eigenvectors via
print("\nSolving for the eigenvalues and eigenvectors: \n")
eig_vals, eig_vecs = linalg.eigh(km_matrix)  # solve the problem
```

```
# Remember that we are actually solving for omega^2, so we need to take the
#    square root of the eigenvalues
omegas2 = np.sqrt(eig_vals)  # here we put a '2' to indicate the two-mass
                             #    oscillator
print("    Eigenfrequencies: \n", omegas2)  # frequencies
print("\n    Eigenvectors: \n", eig_vecs)   # normal modes are columns
print("First column of eigenvectors gives x1, x2 relative amplitudes: ",
      eig_vecs[:,0])
print("Second column of eigenvectors gives x1, x2 relative amplitudes:",
      eig_vecs[:,1])
```

Exercise 10

Create a new KM-matrix `km_matrix3` for three identical masses ($m = 1$ kg) connected by four identical springs ($k = 1$ N/m), then find the eigenvalues and eigenvectors using the procedure above and answer these questions:

- What are the three eigenfrequencies?

- For the lowest mode (frequency), use arrows to sketch the relative motion of each mass.

- What can you say about the motion of $m1$, $m2$, and $m3$ for the middle mode? Make a sketch.

- What can you say about the motion of $m1$, $m2$, and $m3$ for the highest mode? Make a sketch.

```
# < Exercise 10, Create a KM-matrix for 3 identical masses, k = 1, m = 1 >

### DEFINE your KM-matrix

### USE linalg.eigh to solve for the eigenvectors and eigenfrequencies

### PRINT the eigenfrequencies and eigenvectors
```

23.4 Weak Coupling, an Illustration of Beats

Now that you have examined the case with three identical springs, you will observe the behavior of the system when the middle spring is weakened. This situation is an example of two weakly coupled oscillators and, as you will see, produces the beat phenomenon.

Exercise 11*

Modify your 2×2 K-matrix to permit a weak coupling spring, so that $k_2 = k/10$, and redo the eigenvalue problem. Be sure to print your KM-matrix, along with the normal mode frequencies and eigenvectors.

```
# < Exercise 11, Weak coupling, 2 masses, eigenfrequencies >

### DEFINE your KM-matrix

### USE linalg.eigh to solve for the eigenvectors and eigenfrequencies

### PRINT the eigenfrequencies and eigenvectors
```

Exercise 12*

You should have found the same eigenvectors but different mode frequencies. Repeat **Exercises 6**, **7**, and **8** to numerically calculate and plot the motion in both normal modes and when masses are started from rest at $x_1 = 1$ m and $x_2 = 0$ m.

As before, for each set of initial conditions, plot both x_1 and x_2 in separate subplots of the same figure.

```
# < Exercise 12a, Lower mode, weak coupling, solve and plot >

### SET the initial positions and velocities

### SOLVE using solve_ivp

### PLOT x1 and x2 versus t
```

```
# < Exercise 12b, Higher mode, weak coupling, solve and plot >

### SET the initial positions and velocities

### SOLVE using solve_ivp

### PLOT x1 and x2 versus t
```

```
# < Exercise 12c, Masses started from rest at x1 = 1, x2 = 0 >

### SET the initial positions and velocities

### SOLVE using solve_ivp

### PLOT x1 and x2 versus t
```

Starting the masses from rest at $x_1 = 1$ m and $x_2 = 0$ was not an eigenmode. However, it did not produce a plot like that from **Exercise 7**, either. Instead, the weakness of the coupling produced a "beat" pattern in which the energy from the oscillations is passed back and forth between the two carts.

Exercise 13*

To witness this transfer of energy, reexamine the final situation from **Exercise 12** ($m = 1$ kg, $k = 1$ N/m, $k_2 = k/10$, $x_1 = 1$ m, and $x_2 = 0$). Compute the potential energies associated with each spring and the mechanical energy for each cart.

As separate lines on the same axes, plot the (a) kinetic energy of cart 1 + the potential energy of spring 1, (b) the kinetic energy of cart 2 + the potential energy of spring 3, and (c) the potential energy of the coupling spring.

Then, in a separate figure, plot the potential energy, kinetic energy, and total energy of the *entire system* versus time.

Describe how these plots illustrate conservation of energy within the entire system and energy transfer among the different components.

```
# < Exercise 13,  Compute PE, KE, and TE for the weak coupling case >

### COMPUTE kinetic, potential, and total energies

### PLOT energies versus time
```

Exercise 14*

Repeat the animation creation from **Exercise 9** for weak coupling.

```
# < Exercise 14, Animation >

### DEFINE arrays of times, positions for cart 1, and positions for cart 2

# Set up the plot
ball_radius = 0.05 # Plot size of the circle representing the carts
fig,ax = plt.subplots()
ax.set_xlim(-3, 3)
ax.set_ylim(-2, 2)
ax.set_xlabel("x")
ax.set_title("Coupled Oscillators\n")

### USE patches.Circle to define circles representing carts 1 and 2;
#    call them ball1 and ball2
ball1 = 0  ### FIX
ball2 = 0  ### FIX
```

```
### DEFINE an init function that uses ax.add_patch to add each of the patches
#    to the axes
def init():

    return ball1, ball2

### SET the center of each patch at a particular instance of time = t[i] using
#    ball1.set_center and ball2.set_center
def animate(i):

    return ball1, ball2

### CALL animation.FuncAnimation and display the resulting animation
```

23.5 Fourier Analysis

You have seen that the motion of the two carts is the linear combination of two normal modes of oscillation. The normal modes and their corresponding frequencies are determined by the system, while the amount of contribution from each of those modes is based on the initial conditions.

In many physical situations, you will be able to measure the trajectory of the oscillators but will *not* know what the normal modes are. In these cases, Fourier analysis provides a powerful tool for determining the normal modes and their contribution to a given signal. (This topic was first introduced in Chapter 5.7 of *Classical Mechanics* in the context of driven oscillators.)

At the heart of Fourier analysis is Fourier's theorem, which states that any periodic function can be written as a sum of sines and cosines of different frequencies:

$$f(t) = \sum_{n=0}^{\infty} (a_n \cos(n\omega t) + b_n \sin(n\omega t))$$

(Equation 5.82 from *Classical Mechanics*). The coefficients for each of the terms in the sum can be determined mathematically through

$$a_n = \frac{2}{\tau} \int_{-\tau/2}^{\tau/2} f(t) \cos(n\omega t) dt \text{ for } [n \geq 1]$$

$$b_n = \frac{2}{\tau} \int_{-\tau/2}^{\tau/2} f(t) \sin(n\omega t) dt \text{ for } [n \geq 1]$$

(Equations 5.83 and 5.84). (While the motion of carts in **Exercises 5** and **8** is aperiodic, it is also, as we have established, a linear combination of sinusoidal terms, and the coefficients may be determined in a similar manner.)

Rather than computing these integrals by hand, it is possible to use the fast Fourier transform algorithm to determine these coefficients. The implementation we will use is numpy.fft.fft. When provided with a periodic signal (in our case, x_1), this algorithm will return the coefficients. The corresponding frequencies for these coefficients are found through numpy.fft.fftfreq. The plot of the coefficients versus the frequencies is referred to

as the Fourier spectrum. Peaks in this spectrum represent frequencies that provide much of the power to the summed signal. *In the case of coupled oscillators, these peaks will correspond to the normal mode frequencies.*

The two code cells below provide an example of how `np.fft.fft()` may be used to determine the normal frequencies and the relative contributions of the different normal modes given a particular trajectory (e.g., $x_1(t)$). The peaks in the Fourier spectrum are identified using scipy.signal.peaks. The frequencies at which these peaks appear are the frequencies of the sinusoidal terms; their relative strength provides the relative contribution of the different terms. This information is then used to reconstruct the original trajectory by summing together the different sinusoidal terms.

Exercise 15

The next code cells numerically solve the case of two equal-mass carts attached by three equivalent springs wherein cart 1 is released from rest at $x_1 = 1$ and cart 2 is released from rest at $x_2 = 0$ (the same situation as in **Exercises 5** and **8**). The following code cell computes the Fourier transform for the position of cart 1 (x_1), identifies the peaks in the Fourier spectrum, and reconstructs the $x_1(t)$. Read through and run both blocks of code, making sure you understand what is going on. Explore the code by

1) changing the initial conditions, including trying at least one case that produces motion in entirely one normal mode, and
2) increasing the time span of the solution (`t_span`) so that the Fourier transform can be computed over a longer time period (Note: The frequency resolution for this fast Fourier transform is inversely proportional to the time span of the signal).

Then answer the following questions.

How do the angular frequencies of the peaks in the Fourier spectrum compare to the normal frequencies?

How do the relative heights of the different peaks in the Fourier spectrum compare to your calculations of B_1, C_1, B_2, and C_2 from Exercise 5?

How does motion reconstructed from the Fourier analysis compare to the original motion?

What is the impact of increasing the time span (t_end) over which the analysis is done ?

```
# < Exercise 15, Fourier analysis, determining the signal >

# This block of code numerically determines x_1(t) etc. for the case of 2 equal
#    masses and 3 equal springs.
```

```
# Define the KM-matrix
km_matrix = np.array([[2,-1], [-1,2]])

# Set the initial conditions
r_init = np.array([1, 0, 0, 0]) ### ADJUST to explore different ICs
print("r_init: {:}".format(r_init))

# Define a time array of evenly spaced times. Even temporal spacing simplifies
#    the Fast Fourier Transform
t_start = 0
t_end = 100  ### INCREASE the time span to increase the accuracy of the Fourier
             #    analysis
t_span = (0, t_end)
sample_spacing = .01  # spacing of time steps
t_eval = np.arange(t_span[0], t_span[1], sample_spacing)

# Calculate the numerical solution
r_soln = solve_ivp(derivN, t_span, r_init, args = (km_matrix,),
                   t_eval = t_eval)
```

```
# < Exercise 15, Computing the Fourier transform >

# This block of code runs the Fourier transform on x_1(t) and reconstructs
#    x_1(t)

# Import the library from SciPy that contains find_peaks
from scipy import signal

# Find the Fourier transform of the x1 position
ft = np.abs(np.fft.fft(r_soln.y[0]))
# Determine the frequencies that correspond to the different elements in the
#    Fourier transform (ft)
freq_domain = np.fft.fftfreq(len(ft), d = sample_spacing)

# Find the peaks of Fourier transform
#    "height" filters out all peaks below this height, here set to be 10% of
#    the maximum height
peaks, properties = signal.find_peaks(ft, height = np.max(ft)*0.1)

# Because of numerical noise, find_peaks may pick up on smaller, irrelevant
#    peaks. However, you know that the mass will be oscillating as the sum of
#    2 modes (the number of masses and normal modes) or, if the oscillation is
#    entirely in the normal mode, 1 mode. The following line of code sets the
#    number of sine waves to consider to be the maximum of either the number of
#    rows in the KM-matrix or half the number of identified peaks. The reason
#    for half the peaks, as opposed to all of them, is that np.fft.fft actually
#    returns pairs of frequencies (both positive and negative), and we only
#    care about the positive values.
num_sines = min(np.shape(km_matrix)[0], int(len(peaks)/2))
```

```
# Print the identified peaks as both frequencies and angular frequencies
print("Frequencies: ", freq_domain[peaks[:num_sines]])
print("Angular Frequencies: ", 2*np.pi*freq_domain[peaks[:num_sines]])

# Calculate the sum of the height of all peaks.
#    The fraction of this "total_height" will be the fraction each frequency
#    curve contributes to the total motion
total_height = np.sum(properties['peak_heights'][:num_sines])
print("Fractional amplitudes: ",
      properties['peak_heights'][:num_sines]/total_height)

# Plots
fig, axs = plt.subplots(2, 1)
# Plot the Fourier transform and peaks
axs[0].plot(freq_domain, ft)  # plotting the transform
for i in range(num_sines):
    axs[0].plot(freq_domain[peaks[i]], ft[peaks[i]], color = 'r', marker = 'x',
                linestyle = 'None')  # plotting the peaks
axs[0].set_xlim(0, 1)
axs[0].set_title("Fourier Spectrum w/ Peaks Plotted")
axs[0].set_xlabel('Frequency [Hz]')

# Sum the contribution from the different modes to find the reconstructed
#    signal
y = 0
for i in range(num_sines):
    a_n = properties['peak_heights'][i] / total_height
    y += a_n * np.cos(2*np.pi*freq_domain[peaks[i]]*t_eval)
# Plot original x_1
axs[1].plot(t_eval, y)
# Plot the reconstructed x_1 curve using Fourier-identified frequencies and
#    amplitudes
axs[1].plot(t_eval, r_soln.y[0])
axs[1].set_xlim(0, 100)
axs[1].set_ylabel('$x_1$')
axs[1].set_xlabel('Time [s]')

plt.show()
```

Exercise 16*

Complete the Fourier analysis of the situation from **Exercises 10** and **11**. To do this, copy the code from **Exercise 13** into the cells below but adjust the KM-matrix to reflect the weak-coupling case from **Exercise 10**. Once again, let both carts start from rest with $x_1 = 1$ m and $x_2 = 0$ m.

Because the normal frequencies in this situation are so similar, you will need to provide the Fourier transform with a longer temporal baseline to achieve sufficiently high frequency resolution. Therefore, make `t_span` extend to at least 500 seconds.

```
# < Exercise 16, Fourier analysis, determining the signal >

### COPY and modify code from above to numerically determine x_1(t) etc.
```

```
# < Exercise 16, Computing the Fourier transform >
```

23.6 Check-out

Exercise 17

Briefly summarize the ideas in this unit.

23.7 Challenge Problems

If you have extra time, consider the following questions.

1. Modify your `km_array` for the case in which there is no second wall on the right side, so the problem is one of two identical springs and two masses, and the third spring is missing. Find the normal mode frequencies and show plots of the oscillations of x_1 and x_2 in the two modes, and then with $x_1 = 1$ m, $x_2 = 0$ initial conditions. You should see that the amplitudes are no longer symmetrical, so you can plot both x_1 and x_2 on the same figure. This is Problem 11.5 in *Classical Mechanics*.
2. Solve the same problem as shown in the previous challenge problem, but this time analytically. Compare your analytical results with those above.
3. Make a function `km_matrix_maker(N, k_value)` whose input is the number `N` of identical oscillators and single-spring constant `k_value`, and whose output is a km_array that is suitable for solving the eigenvalue problem. You may want to experiment with `np.diag(np.ones(N))`, `np.diag(np.ones(N-1, k = 1)`, and `np.diag(N-1, k =-1)`. Test it on the N = 2 and N = 5 cases.

```
# < Challenge problem 1, 2 masses, 2 springs >
```

```
# < Challenge Problem 2, comparisons between numerical and analytical solutions >
```

```
# < Challenge problem 3, K-matrix creator, identical oscillators >

# Illustration of creating an N x N diagonal matrix, suitable for
#    nearest-neighbor coupling
N = 3
diagonal = np.diag(np.ones(N))
```

```
upper_diagonal = np.diag(2*np.ones(N-1),k = 1)
lower_diagonal = np.diag(-5*np.ones(N-1),k = -1)
print("Put ones down the diagonal: \n", diagonal)
print("Put (one fewer) twos down the upper off diagonal: \n", upper_diagonal)
print("Put (one fewer) minus fives down the lower off diagonal: \n",
      lower_diagonal)
matrix = diagonal+upper_diagonal+lower_diagonal
print("\nAssembled matrix: \n",matrix)
```

```
# < Eigensolution for N = 5 case >
```

24 Damped Driven Pendulum

Chaotic behavior is highly sensitive to initial conditions. In chaotic systems, small differences in initial conditions grow exponentially. Because we cannot know precisely the initial conditions of any real-world phenomena, chaotic systems are effectively impossible to predict even though they are deterministic.

For a system to exhibit chaos it must be nonlinear and sufficiently complex. Throughout this course, you have seen multiple examples of how to approach analytically solvable linear differential equations (so they must not be chaotic). Nonlinear ones frequently cannot be analytically solved, allowing them to be potentially chaotic. Not all nonlinear systems are chaotic, though. For instance, a simple pendulum is nonlinear but very predictable. For a pendulum to exhibit chaotic behavior complexity must be added — here in the form of driving and damping.

Across the next three units, you will be investigating the behavior of a damped driven pendulum (DDP). This system can, but does not always, exhibit chaotic behavior. In this unit, you will study the approach to chaos by modeling the DDP in the chaotic and non-chaotic regimes.

The DDP system is introduced in Ch. 12.2 of *Classical Mechanics*, and the topics in this unit extend through 12.5 of the text. Bifurcation Diagrams and State-Space Orbits and Poincaré Sections build on this unit with additional analysis of the system.

24.1 Objectives

In this unit, you will

- model a damped, driven pendulum (PHY);
- identify period doubling from the pendulum's motion (PHY);
- follow the approach to chaos (PHY);
- show that solutions generated from very similar initial conditions diverge for chaotic systems (PHY);
- fit the rates of convergence/divergence between solutions (PRO); and
- identify peaks in an array of data (PRO).

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp  # the differential equation integrator
```

```
from scipy.optimize import curve_fit  # for fitting the exponential decrease/
                                      #    increase between ICs

plt.rcParams["figure.figsize"] = (12,8)  # make figures big for easier viewing
                                         #    (width, height in inches)
# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

24.2 Equation of Motion

Exercise 1

You will need to find the equation of motion for a damped driven pendulum to model its behavior numerically. To do this, start by determining the equation of motion for a simple pendulum and then add in damping and driving. For this exercise, please fill in your answers to each step below.

1) Starting with Newton's second law in angular form, $\sum \vec{\tau} = I\vec{\alpha}$, with $\tau = \vec{r} \times \vec{F}$, determine the restoring torque for a simple pendulum (just a point mass m on a frozen massless string of length L).

2) Write down the moment of inertia of a point mass at radius equal to the length of the pendulum.

3) Hence the second derivative of the angle is some function of the angle. That is the equation of motion for a simple pendulum. Write it here (don't divide out any L and m terms yet).

4) Modify your equation of motion to include damping by adding a term to the force (torque) side of the equation to represent linear air resistance.

5) Further modify your equation of motion to include driving by adding a sinusoidal varying torque: $LF_0 \cos \omega t$.

6) Determine the natural angular frequency of a simple pendulum, ω_0.

7) Now simplify your equation of motion and rewrite it in terms of ω_0, a damping constant ($\beta = \frac{b}{2m}$), and a dimensionless parameter γ representing the drive strength ($\gamma = \frac{F_0}{mg}$).

When γ is small, the weight of the pendulum is more than the driving force, and the resulting force will be small. As we increase γ above 1, the pendulum will start to display chaotic motion.

24.3 Numerical Solution

24.3.1 Defining the constants

Let the angular frequency of the driving force (ω or `omega_drive`) equal 2π. Set the natural frequency (ω_0 or `omega_0`) equal to $1.5\omega = 3\pi$, and set the damping constant (β or `beta_damp`) to $\frac{\omega_0}{4}$.

We will be changing the drive strength (γ or `gamma_drive`).

Exercise 2

What do you expect the period of the DDP to be?

Exercise 3

In the cell below, set all the constants associated with the system *except* γ as global variables.

```
# < Exercise 3, Initializing the constants omega_drive, omega_0, and beta_damp >
```

24.3.2 Numerically solving the equation of motion

Exercise 4

Use the equation of motion from **Exercise 1** to write a function to return the derivatives of your two variables, ϕ and $\dot{\phi}$. You will be changing the value of γ throughout the unit, so rather than setting γ as a global variable, allow it to be passed as a parameter to the `deriv()` function.

To find solutions, you will also need a set of initial conditions. For the most part, you will have the pendulum start from rest in line with the vertical: $\phi(t = 0) = 0$ and $\dot{\phi}(t = 0) = 0$.

Use `solve_ivp()` to solve the equations of motion for $0 < t < 10$ s and ϕ versus time, assuming $\gamma = 0.2$. Let `rtol = 1e-8` and `atol = 1e-8`.

Finally, plot $\phi(t)$ versus t, and check to make sure that your graph makes sense. Your plot should match Figure 12.2 of *Classical Mechanics*.

```
# < Exercise 4, Deriv function >

### DEFINE your deriv function
```

```
# < Exercise 4, Numerical solution >

### COMPUTE the numerical solution using solve_ivp
```

```
# < Exercise 4, Plots >

### PLOT phi versus t
```

24.3.3 Coding style

Exercise 5

You are going to be solving the DDP many times for different initial conditions and values of γ. In order to expedite this, create a function called `solve_DDP()` to set up and solve the equation of motion for a set of initial conditions over a given period of time. Your function should take the variables `phi0`, `dphi0`, `gamma_drive`, and `max_time` and return an array containing `time`, `phi`, and `dphi`. You should use `t_eval` to fix your time step size of 0.01 s. Once again, let `rtol = 1e-8` and `atol = 1e-8`.

```
# < Exercise 5, Define a function to solve the DDP >

# Define the function solve_DDP to set up and solve the equation of motion for
#    a set of ICs over a given period of time.

def solve_DDP(): ### FIX
    """ For a given set of initial conditions, damping and maximum time solve
    the damped driven pendulum and return an array containing time, angle, and
    angular velocity"""

    ### ADD code

    return ### FIX
```

24.4 Behavior with Increasing Values of Driving

24.4.1 Long-term behavior with weak driving

Exercise 6

Use the function you wrote above to solve for $\phi(t)$ over $0 < t < 6$ s for at least three different values of γ between 0 and 1. Plot $\phi(t)$ versus t, either on the same set of axes or using a three-panel figure. Examine your graphs to determine the period of the long-term oscillations for each of the values of γ.

In what ways are your plots for different values of γ similar and different?

```
# < Exercise 6, Motion of a damped, driven pendulum for 0 < gamma < 1 >

### CALCULATE trajectories

### PLOT phi versus t
```

24.4.2 Intermediate and strong driving

As γ is increased slightly above 1 (the divide between weak and strong driving), strange long-term behavior starts to appear.

Exercise 7*

First, solve for $\gamma = 1.06$ and include the plot of $\phi(t)$ versus t for $0 < t < 12$ s (Figure 12.4 of *Classical Mechanics*).

In the space below, **describe the motion of the pendulum before $t = 6$**. *(Hint: What does it mean for ϕ to be greater than π?)* Also, describe the long-term behavior of the pendulum, including the period.

```
# < Exercise 7, DDP with gamma = 1.06 >

### CALCULATE trajectory

### PLOT phi versus t
```

As we increase γ, we begin to see the emergence of harmonics and subharmonics. For instance, $\gamma = 1.073$ produces long-term oscillations with alternating values of the maxima and minima. It now takes twice the period of the driving force for the cycle to repeat, so this motion has period *two* (alternatively, we could say that this has a subharmonic of half the driving frequency).

Exercise 8

Solve for $\gamma = 1.073$ from $0 < t < 30$ s and include the plot of $\phi(t)$ versus t below. Your plot should match Figure 12.5 of *Classical Mechanics*, so you will want to run your solver out to 30 seconds.

```
# < Exercise 8, DDP with gamma = 1.073 >

### CALCULATE trajectory

### PLOT phi versus t
```

Increasing γ further yet will produce longer period oscillations in a process called a "period-doubling cascade".

Exercise 9*

Try several more values of γ between 1.073 and 1.090 and identify a value of γ that produces longer-period oscillations. You may need to increase `max_time` (the time range) in order to reach the point after the transients have died off. Include a plot below.

```
# < Exercise 9, Motion of a DDP for  1.073 < gamma < 1.09 >

### CALCULATE trajectory

### PLOT phi versus t
```

Increasing γ still more produces ***chaos!*** To see it, all you need to do is numerically solve the DDP with $\gamma = 1.16$.

Exercise 10

Solve for $\gamma = 1.16$ and include the plot below.

```
# < Exercise 10, Motion of a DDP for gamma = 1.16 >

### CALCULATE trajectory

### PLOT phi versus t
```

24.5 Sensitivity to Initial Conditions

One of the most important features of chaos is how very slight differences in the initial conditions can lead to wildly different behavior, i.e., "sensitivity to initial conditions". This is why a deterministic system can be, in a practical sense, completely unpredictable. This chaotic behavior has a host of real-world implications, including the inability of forecasters to accurately predict weather days in advance.

Exercise 11

Let us start at the non-chaotic end. Verify that completely different initial conditions ($\phi(\text{t} = 0) \equiv 0$ and $\phi(\text{t} = 0) \equiv \pi/2$) lead to the same long-term behavior if the driving force is weak ($\gamma = 0.7$) by plotting the two curves (`soln1` and `soln2`) on the same graph for $0 < t < 30$ s. Additionally, plot the logarithm of the absolute value of the difference between the solutions as a function of time. For convergence, the difference between the solutions will decrease exponentially, thus falling in a straight line on this log-linear ("semi-log") plot.

```
# < Exercise 11, Insensitivity to ICs in non-chaotic regime >

### SOLVE for both trajectories
```

```
### PLOT phi versus t and
#    log|(phi_1 - phi_2)| versus t
```

Exercise 12*

One way to check whether the difference between the solutions is actually decreasing exponentially and to learn how fast that decrease is happening is to fit an exponential to it. You will do this by using scipy.optimize.curve_fit to fit a *linear* function to $\log(|\phi_1 - \phi_2|)$ vs. t (while you could also directly fit an exponential, the method isn't as robust).

As you saw when introduced to `curve_fit()` in **Exercise 9** from Taylor Series with Loops and Functions and in **Exercise 15** from Simple Pendulum Comparison to Data and Theory, the first step is to define the functional form of the fit. To do this, in the cell below, define a function that returns $m * x + b$ when passed the arguments x, b, and m.

Then, use scipy.optimize.curve_fit (already imported in the first code cell of this notebook as `curve_fit`) to find the best-fit values of m and b. The form of the function call is

```
popt, pcov = curve_fit(fit_func, x, y, p0)
```

where `fit_func` is the name of the fitting function you defined above, `x` is your independent variable (time), and `y` is your dependent variable ($\log(|\phi_1 - \phi_2|)$). The variable `p0` is an array of initial guesses for the parameters in your fitting function, in this case, a guess for b and a guess for m. The returned values are 1) the array of fitted parameters (`popt`) optimized such that the square of the residuals is minimized and 2) the estimated approximate covariance of the fitted parameters (`pcov`), which is a measurement of the quality of the fit.

After you have found the best-fit curve, plot it atop your graph of $\log(|\phi_1 - \phi_2|)$ versus t.

What is the exponential decay rate for the differences between your solutions?

```
# < Exercise 12, Fitting the exponential rate of convergence >

### DEFINE a fitting function: a function that takes x, the independent
#    variable, and two parameters (b, the y-intercept, and m, the slope)
#    as arguments and returns m*x + b
```

```
# < Exercise 12, Fitting the exponential rate of convergence, cont. >

### DEFINE an array of initial guesses for b and m

### CALL curve_fit for log(|phi_1 - phi_2|) vs. t, assuming a linear fit

### PLOT your best-fit line over the graph of log(|phi_1 - phi_2|) vs. t
```

Exercise 13*

A better fit can be obtained by fitting to the peaks (local maxima) of the curve, rather than the entirety of the data. To do this, though, you must first identify the peaks. Luckily, the SciPy package comes with a function, scipy.signal.find_peaks, which can identify peaks in data.

The function `find_peaks()` takes as an argument the data it should identify the peaks of (in this case, $\log(|\phi_1 - \phi_2|)$). It also allows for a number of optional keyword arguments, such as `height`, `width`, `threshold`, and `prominence`, that can be used to set the characteristics of the identified peaks. These keyword arguments are especially useful when dealing with noisy data. For this situation, setting `width = 10` is sufficient. Doing so ensures that each peak is the maximum within the surrounding 10 data points, equivalent to a $10\delta t = 10 \times 0.01\ \text{s} = 0.1$ span of time.

The return of `scipy.signal.find_peaks()` is a numpy array of peak indices and a dictionary of peak characteristics. To set a variable to only the peak indices, use a call like

```
pk_indices = find_peaks(data)[0]
```

to select the first return of the function.

Find a fit to only the peaks of the differences and plot your best-fit line on top of the data, as in **Exercise 12**.

```
# < Exercise 13, Better fit of the exponential rate of convergence >

# Import find_peaks from scipy.signal
from scipy.signal import find_peaks

### USE find_peaks to identify the indices corresponding to peaks in the data

### CALL curve_fit with only those data points that correspond to peaks

### PLOT your best-fit over the graph of log(|phi_1 - phi_2|) vs. t
#    On your plot, use a different symbol and color to mark the peaks,
#    e.g., '*r'
```

Exercise 14*

Redo the above plots for slightly stronger driving ($1.073 < \gamma < 1.09$).

How do your plots fit with the behavior you observed in Exercise 9?

```
# < Exercise 14, Response to ICs in the period-doubling regime >

### SOLVE for both trajectories

### PLOT phi versus t and
#    log|(phi_1 - phi_2)| versus t
```

Exercise 15

Now redo the above plots for the primarily chaotic regime ($\gamma = 1.105$). Because very slightly different initial conditions should produce very different results, change your initial angles so that they are very close together, e.g., $\phi(t = 0) \equiv 0$ and $\phi(t = 0) \equiv 0.0001$. Your figure should be similar to Figure 12.13 from *Classical Mechanics,* which we note is plotted using log10, not the natural log. Your solution will probably diverge more quickly because of taking coarser time steps.

How are your results indicative of chaos? Why does the difference between the angles not increase forever?

```
# < Exercise 15, Sensitivity to ICs in the chaotic regime >

### SOLVE for both trajectories

### PLOT phi versus t and
#    log|(phi_1 - phi_2)| versus t
```

Exercise 16*

Repeat the steps in **Exercise 12** for this new value of γ, this time fitting to the exponential rate of *increase* in the difference between the two solutions over time for the chaotic regime. If desired, you can fit only to the peaks, as in **Exercise 13**.

Only compute your fit for the first 7 seconds of the solutions in order to focus on the increasing portion of the graph.

- *Hint 1:* The syntax to select the elements in an array that fit some criteria, such as elements less than 2, is

  ```
  arr_select = arr[arr < 2]
  ```

- *Hint 2:* Because you are once again fitting a linear function to the log of your dependent variable, there is no need to redefine a fitting function for use in `curve_fit()`.

What is the (approximate) exponential rate increase for the differences between your solutions?

```
# < Exercise 16, Fitting the exponential rate of divergence >

### DEFINE an array of initial guesses for b and m

### CALL curve_fit for log(|phi_1 - phi_2|) vs. t, assuming a linear fit

### PLOT your best-fit over the graph of log(|phi_1 - phi_2|) vs. t
```

Unfortunately, this high degree of sensitivity to initial conditions means that it is also very difficult to model chaotic systems computationally. Essentially, small rounding or integration errors in code get hugely magnified. The result is that the same initial conditions simulated with different degrees of tolerance will produce very different behavior.

Exercise 17*

To see this sensitivity to simulation accuracy, plot ϕ versus t for $0 < t < 40$ s with `gamma = 1.16`, using different values of `rtol` and `atol` in your `solve_ivp()` call. Start with tolerance `rtol = 1e-8 and atol = 1e-8`, and decrease by factors of ~10 down to the minimum tolerance (`atol = rtol` ≈ 1e-13). In all cases, set `phi0` = 0 and `dphi0` = 0. You can remind yourself of the meaning of `atol` and `rtol` by reading the solve_ivp() documentation. Also, recognize that you will not be able to use your solve_DDP function as written. It will probably be easier just to use `solve_ivp()` here directly or add the tolerance to the parameter list.

The sad upshot is that because the trajectories diverge, we are unable to predict with any accuracy the state of the pendulum after some length of time. (This is the same reason why weather predictions a week out are so inaccurate.) Luckily, though, we can still use simulations to tell us about the statistical properties of a chaotic system. You will be examining this in the following units through bifurcation diagrams and Poincaré plots.

```
# < Exercise 17, Accuracy of integration in chaotic regime >

### SOLVE for all trajectories

### PLOT phi versus t
```

24.6 Check-out

Exercise 18

Briefly summarize the ideas in this unit.

24.7 Challenge Problem

24.7.1 A parametrically driven pendulum leading to instability

Rather than driving the DDP along the line of travel (i.e., perpendicular to the string), a simple modification leads to a surprising result. We periodically vary a parameter of the pendulum, perhaps its length or, in this case, the local value of gravity, by pulling up and down at a particular phase of the motion. To motivate this problem, imagine holding a simple pendulum in your hand and letting it oscillate normally at small amplitude.

At what part of the cycle is the tension in the string the greatest? (Remember, it has some velocity $L\dot{\phi}$.)

At what part of the cycle is the string tension the least?

Now consider what happens if you raise the center of mass precisely when the tension is the greatest, and lower it when the tension is the lowest. Notice that you will be doing (net) work on the system over the periodic cycle, and the system must respond by increasing its energy — its amplitude will increase.

Consider that for best effect, you could raise the pendulum bob on both halves of the cycle, and so applying a driver at twice the natural frequency, $2\omega_0$, is most advantageous, leading to the strongest growth rate. We then expect lower growth rate if we raise it and lower it only once per cycle, or once every 1.5 cycles, etc.:

$$\omega_n = (2\omega_0)/n, \quad \text{for } n = 1, 2, 3, ...,$$

an infinite series of solutions!

For more background on the theory and an experimental test of this, see: W. Case, "Parametric instability: An elementary demonstration and discussion," *Am.J.Phys.* **48**, 218-221 (1980). Famously, this parametric instability is exploited to impressive effect at the Santiago de Compostela Cathedral, in Spain; see: J. R. Sanmartin, "O Botafumeiro: Parametric pumping in the Middle Ages," *Am.J.Phys.* **52**, 937-945 (1984). See the video here: https://www.youtube.com/watch?v=rOoHyEEXxoA

Challenge Problem, Part 1

Following Case's Eq. (3), we write the driving term as:

$$\frac{\tau}{mL} = \frac{y_0}{L}\omega^2 \sin\phi \cos\omega t$$

where y_0 is the vertical displacement of the attachment point of the pendulum, ϕ is the angle from the vertical, and ω is the driving frequency. In brief, the vertical motion of the attachment point, $y = y_0 \cos\omega t$, adds a fictitious force in the reference frame of the pendulum equivalent to $m\ddot{y} = y_0\omega^2 \cos\omega t$. To select for the tangential component of that force, we multiply it by $\sin\phi$, just as is done with the gravitational acceleration. (Note that in Case [1980], the small-angle substitution is used to replace $\sin\phi$ with ϕ.)

This driving term will replace $\frac{\tau}{mL} = \frac{F_0}{mL}\cos\omega t$ from the previously analyzed damped driven pendulum system. One fundamental change is the introduction of ϕ to the driving term. Note that this introduction means that *nothing* happens if $\phi = \dot{\phi} = 0$; that is, if the pendulum starts at the equilibrium location with no velocity in ϕ, it will not oscillate. But if ϕ is initially some vanishingly small angle but not quite zero, the driving term can get bigger, because its size depends upon ϕ. What happens next depends entirely on the frequency ω of the driver.

Create a new deriv() function, deriv_para, that describes this equation of motion, including damping. Define a new variable

$$h_0 \equiv \frac{y_0}{L}$$

to characterize the strength of the driving. This variable will take the role of γ, and should be a parameter of `deriv_param`, along with `t_now` and `r_now`.

```
# < CP 1, Deriv function for parametric oscillator >
### MODIFY your derivative function to replace the (linear) driver with this
#    parametric driver

def deriv_para(t_now, r_now, h_drive, beta_damp_deriv):
    """ Return derivatives of the array (phi, dphi/dt) for the parametric
    oscillator. The parameter y_0 is the amplitude of the vertical sinusoidal
    displacement of the pendulum attachment point. """

    phi = r_now[0]
    dphi_dt = r_now[1]  # dphi/dt
    d2phi_dt2 = 0 ### FIX

    return np.array([dphi_dt, d2phi_dt2])
```

Challenge Problem, Part 2

Assume a pendulum of the same natural frequency as in the previous exercise ($\omega_0 = 3\pi$) is vertically driven at twice its natural frequency. Let your value of h correspond to a 30% change of the pendulum length and keep $\beta = \omega_0/4$. Solve the equation of motion using `solve_ivp()` and plot the trajectory of ϕ vs. time.

First verify to yourself that nothing happens if $\phi = \dot{\phi} = 0$. Then solve for $\phi = 0.001, \dot{\phi} = 0.0$. Plot the ϕ versus t.

```
# < CP 2, Parametrically driven pendulum at twice the natural frequency >

### DEFINE variables

### SOLVE with solve_ivp
```

```
# < CP 2, Parametrically driven pendulum at twice the natural frequency,
#    plots >

### PLOT phi vs. t
```

Challenge Problem, Part 3

As you can see, for this particular set of ω, y_0, and β, the amplitude grows. The shape of the envelope appears to be exponential. Go ahead and see if this is the case by fitting an exponential to the amplitudes. The procedure to do this is very similar to what you used in **Exercise 13**: Identify the peaks, then fit a linear function to the log of the peaks.

```
# < CP 3, Growth of amplitude for parametrically driven pendulum >

### IDENTIFY peaks using find_peaks
```

```
### FIT the log of the amplitudes with a linear function using curve_fit

### PLOT log amplitude vs. time with fit
```

Challenge Problem, Part 4

Case's definition of $h \equiv \frac{y_0}{L}\omega^2$ is not dimensionless but is ω^2 larger than h_0 above. At the peak of the instability in the small-angle regime, he predicts a growth rate $\lambda_{\max} = h/(2\omega) = h_0\omega^2/(2\omega) = h_0\omega_0$. Compare your value of the initial growth rate with $h_0\omega_0$ for these parameters:

`beta_damp` = 0.0, `h_drive` = 0.05

```
# < CP 4, Comparison >
```

25 Bifurcation Diagram

In the previous unit, Damped Driven Pendulum, the damped driven pendulum was shown to demonstrate period-doubling and chaotic behavior under certain conditions. Specifically, you were asked to look at ϕ as a function of time for different values of forcing, γ. By examining the trajectory of the pendulum over time, it was clear for some values of γ, the pendulum exhibited periodic behavior with the same period as the forcing. For other values of γ (i.e., within the period-doubling regime), the pendulum exhibited periodic behavior with longer periods, and for others it exhibited no periodic motion at all, otherwise known as chaos.

This observation, however, raises the questions: for what values of the forcing amplitude γ is the system periodic, and how does the system transition between these different states? An elegant way to display how the behavior of the system changes for different values of γ is the *bifurcation diagram*. In a bifurcation diagram, the value of ϕ after every period of the forcing is plotted against γ. In this unit, you will recreate Figure 12.17 from *Classical Mechanics* by generating the bifurcation diagram for the damped driven pendulum. Walking through the different steps should give you a better understanding of how the bifurcation diagram encodes information.

The material for this exercise is covered in Ch. 12.6 of *Classical Mechanics*.

25.1 Objectives

In this unit, you will:

- create and import a module that models the damped driven pendulum (DDP) (PRO);
- learn how to use a spline fit to interpolate between data points (PRO);
- generate the bifurcation diagram for the DDP (PHY);
- interpret the bifurcation diagram in the context of the approach to chaos (PHY);
- read and write data from files such that multiple groups can contribute to a shared bifurcation diagram (PRO).

```
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = (12,8)  # make figures big for easier viewing
                                         #    (width, height in inches)
```

```
# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

25.2 Setup

In order to understand how a bifurcation diagram is generated, you will start by looking at both plots of ϕ versus t and the bifurcation diagram for individual values of γ. To do that, though, you first need your code to solve the DDP.

Exercise 1

As you may have encountered in Precession of a Cuboid, creating and importing modules is an elegant way to share code between notebooks. For this notebook and State-Space Orbits and Poincaré Sections, you will want to use the code you wrote in Damped Driven Pendulum. Therefore, the first step is to copy the necessary code for modeling the damped driven pendulum system from Damped Driven Pendulum into a "module".

Create a new file named `damped_driven_pendulum.py` located in the same directory as this notebook (this would require uploading it to CoCalc or to your "Colab Notebooks" directory on Google Drive if you are using Google Colab). (If you are using Google Colab, you will also need to mount the Google Drive by uncommenting the code in the cell below.) At the top of the file, include

```
import numpy as np
from scipy.integrate import solve_ivp
```

as these libraries will be necessary for the functions (if you check the cell above, you'll see that `solve_ivp()` is not imported in this notebook as it is only used within functions contained in your module). Then copy over the following:

- the constants defining the system from that notebook as in **Exercise 3**. Specifically, set the angular frequency of the driving force (ω or `omega_drive`) to 2π, producing a drive period of $2\pi/\omega = 1$ s. Set the natural frequency (ω_0 or `omega_0`) to 1.5ω, and set the damping constant (β or `beta_damp`) to $\frac{\omega_0}{4}$.
- the entirety of `deriv()` from **Exercise 4**.
- the entirety of `solve_ddp()` from **Exercise 5**.

The next step is to import everything contained in this file (a.k.a. module). You could do this by running the following statement:

```
import damped_driven_pendulum as ddp
```

where "as ddp" defines an alias for "damped_driven_pendulum" and saves you some typing.

After importing this module, you should be able to access the constants and functions defined in it by affixing "ddp." to the start of it, e.g., `ddp.solve_ddp()`. Test this by printing `ddp.omega_0` after importing.

```
# FOR GOOGLE COLAB ONLY

# < Exercise 1, Importing a module >

### UNCOMMENT the following lines if you are using Google Colab

#from google.colab import drive ## Mount your Google Drive
#drive.mount('/content/drive/')

#import sys  # Append your Google Drive "Colab Notebooks" directory to your
             #    path so that files contained in it may be found
#sys.path.append('/content/drive/My Drive/Colab Notebooks/')
```

```
# < Exercise 1, Importing a module >

### IMPORT your module
```

Exercise 2

Now call `ddp.solve_ddp()` to model a damped driven pendulum with the following characteristics:

- $\gamma = 1.062$,
- $\phi(t = 0) \equiv -\pi/2$, and
- $\dot{\phi}(t = 0) \equiv 0$ for $0 < t < 600$.

Then, check your work by plotting ϕ versus t. In order to make your plot legible, set your x-axis range from 0 to 30 s.

```
# < Exercise 2, Solve and Plot DDP >

### PLOT phi versus t for gamma = 1.062
```

25.3 One-Point Bifurcation Diagram

The next step is to add a single value of the forcing amplitude parameter γ to the bifurcation diagram by plotting the angle after each period of the driver at that value of γ. To do this, though, you will need to find the values of ϕ for any given time. You will be sifting through your ϕ solution, picking out values every 1 second, the period of the driver. Unfortunately, your integrator only returns ϕ and $\dot{\phi}$ for the discrete times listed in your time array, which may not correspond to a multiple of the driver period. So, you will need to interpolate between those points if the time you want does not correspond to one of the values in your time array. A "spline" fit does exactly that. *(A third-order spline fit may be familiar to you from Excel's scatter plotting option that connects points with a smooth curve.)*

An example of how to use the spline fit function from the SciPy.interpolate library, scipy.interpolate.UnivariateSpline is included below.

```
# Import the UnivariateSpline from scipy
from scipy.interpolate import UnivariateSpline

# Define test data
x = np.linspace(0, 10, num = 11 , endpoint = True)
y = np.cos(-x**2/9.0)

# This call creates the spline fit to the data.
#    The first value in the function call is the x data, the second is the
#    y data;
#    The values of k = 5, s = 0 were chosen to make the fit:
#      k = 5 tells it to compute a fifth-degree polynomial fit, and s = 0 tells
#      it to go through all the data points.
spl = UnivariateSpline(x, y, k = 5, s = 0)

# The next two lines of code use the above spline fit to interpolate the value
#    of y at the given x values.
x_fit = [0.5, 3.7, 4.8]  # define the list of x values for which we want y
y_fit = spl(x_fit)  # use the spline function defined earlier to calculate the
                    #    corresponding y values
print("Interpolated y values: ", y_fit)

# To demonstrate the accuracy of the fit, this plot shows the data points given
#    the theoretical spline curve connecting them, and the interpolated values
fig, ax = plt.subplots()

data =  ax.plot(x, y, marker = 'x', ms = 8, linestyle = 'None') # Data points,
                                        #    marked by "x's";
                                        #    ms (a.k.a. "markersize") sets the
                                        #    size of the data point markers.
                                        #    linestyle = 'None' ensures that
                                        #    points are not connected
data_fit = ax.plot(x_fit, y_fit,
                   marker = 'o', ms = 8, linestyle = 'None')  # interpolated
                                                              #  values, marked
                                                              #  by a filled
                                                              #   circle ("o")
# Smooth curve connecting data points
ax.plot(np.linspace(0, 10, num=110, endpoint=True),
        np.cos(-np.linspace(0, 10, num=110, endpoint=True)**2/9.0))
ax.legend(["Data", "Interpolated Values", "Fit"])
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_title("Spline example")
plt.show()
```

To plot a given value of γ on the bifurcation diagram you will need to find the value of $\phi(t)$ at every delta t (dt, set to be equal to the period of the forcing) and plot them versus γ.

In order to let the initial transients die down, only the values of $\phi(t)$ after some minimum time (`tmin_plt`) should be included.

Exercise 3

Use a spline fit to find the values of $\phi(t)$ for 500 s $< t <$ 600 s in steps of $dt = 2\pi$ /omega_drive.

```
# < Exercise 3, Find phi for t using spline >

### FILL IN and uncomment the following lines of code

# t_ind = # fill in an array of time values (going from 500 to 600 s)
# spl = UnivariateSpline(    ) ### FILL IN the call to the UnivariateSpline
                                #    function
# phi_t_ind = spl(t_ind)
# print(phi_t_indices)
```

Exercise 4

Check the accuracy of your spline fit interpolation by plotting $\phi(t)$ versus t and `phi_t_ind` versus `t_ind` on the same plot. In order to see what is going on, you should make your x-axis range very small. In other words, only plot a small region around a particular value of `t_ind`, rather than across the entire range of values. You should find that for $\gamma = 1.062$, all of the ϕ values are the same.

```
# < Exercise 4, Verify accuracy of the spline fit >

# PLOT phi versus t and interpolated values, phi_t_ind versus t_ind
```

Exercise 5

Make your first point on the bifurcation diagram by plotting the phi values you calculated in the previous exercise (`phi_t_ind`) on the y-axis versus γ on the x-axis. Set your axes range from 1.06 to 1.085 for γ and -0.7 to 0 for ϕ.

- *Hint 1:* Even though you are using the same value of γ, you will have to make an array of γ values with the same length as your `phi_t_ind` array in order to plot them against each other. Later, when the behavior gets more interesting as γ increases, you will see that ϕ is no longer single-valued.
- *Hint 2:* Notice the very small range of the y-axis. Then adjust it to range from the possible values of ϕ, $-\pi$ to π.
- *Hint 3:* Setting the argument `marker` in your `plot` command allows you to use different symbols for the data. Using `marker = ','` will make the marker a pixel, `marker = '.'` will make it a point, and `marker = 'o'` will make a circle. You can change the size of the symbol using the

`markersize` argument. Setting `linestyle = 'None'` ensures the points are not connected on the plot.

```
# < Exercise 5, Add a single value of gamma_drive to the bifurcation diagram >

### PLOT phi_t_ind versus gamma
```

25.4 Creating the Full Bifurcation Diagram

Exercise 6

The above bifurcation diagram is very boring because all 100 ϕ values that occur in sync with the driver period have the same value. If we steadily increase γ, we can explore more interesting regions, mapping out a complete bifurcation diagram. To make a full one, create a `for` loop to cycle through different values of γ for $1.06 < \gamma < 1.087$. For each value of γ, plot the values of $\phi(t)$ at every Δt (period of the forcing) versus γ. At the end of the loop, call `plt.show()` to display all of the accumulated data sets. You should get something that looks like Figure 12.17 from *Classical Mechanics*. You will see that the phase is no longer single-valued as the forcing amplitude gets higher.

- *Hint 1:* When you are first writing your `for` loop, increase the value of γ by relatively large steps (say, 0.001). Then, once you are sure your code is working properly, decrease the step size to fill in the diagram.
- *Hint 2:* Try using the pixel marker for this plot, e.g., using `marker = ','` as a keyword argument in your plot command.

Be patient — these plots can take a long time to render.

```
# < Exercise 6, Create the full bifurcation diagram >

### SET up your plot using plt.subplots()

### USE a "for" loop to loop through different values of gamma,
#    calculate the trajectory with solve_DDP,
#    use a spline fit to determine values of phi after each period,
#    and plot
```

Exercise 7

What ranges of γ produce chaotic behavior? What ranges produce period doubling?

Exercise 8

The beauty of the bifurcation diagram is that it gives you a quick visual clue as to interesting values of γ. Using the diagram as a guide, select at least three values of γ (each from different regimes) and plot ϕ versus t.

➤ *Hint:* You will be able to see the long-term behavior best if you adjust your plot axes to show a later period of time, i.e., $500 < t < 600$ s.

Are the graphs what you would expect? Explain.

```
# < Exercise 8, Plot phi versus t for three interesting values of gamma_drive >

### FILL in code below
```

25.5 Features of the Bifurcation Diagram

Exercise 9*

You will be able to see another interesting property of the bifurcation diagram if you zoom in on some of the period-doubling portions of it. Rerun the bifurcation diagram for $1.0826 < \gamma < 1.083$, and show the plot for $-0.535 < \phi < -0.5$.

What evidence of fractal behavior do you see?

```
# < Exercise 9, Zoom in on a portion of the bifurcation diagram >

### INCLUDE code here
```

Exercise 10*

Your bifurcation diagram may look very different for different initial conditions. Create a bifurcation diagram for a different set of values of $\phi(t = 0)$ and $\dot{\phi}(t = 0)$, and include your plot below.

Describe the differences you see. What are the similarities?

```
# < Exercise 10, Create the bifurcation diagram for a different set of ICs >

### INCLUDE code here
```

25.6 Class Parallel Programming Project

One way to speed up the creation of your bifurcation diagram is to have multiple computers working at once. This type of problem is sometimes called an "embarrassingly parallel" (or "perfectly parallel") problem because the lack of interaction between the different calculations makes it exceedingly easy to compute in parallel. In other words, one computer could be working on solving the trajectory for one value of γ, and another computer working on a different value of γ with no need for them to share information beyond divvying up the values of γ. While one could use the Python multiprocessing capabilities to divide the problem across multiple CPUs contained in one workstation, as in Challenge Problems 1 and 3 of Driven Damped Harmonic Oscillator and Resonance, the embarrassingly parallel nature of this problem opens up a different possibility: a class project in which each team of students has their computer solve a unique set of values of γ. Once the solutions have been obtained, they can be combined together to create a more detailed bifurcation diagram.

Exercise 11*

Let's suppose you have access to a large number of computers. For `n_CPU` computers working between `gamma` = 1.060 and 1.087 (exclusive) with a step size `d_gamma` = 0.001 (a fairly short running time), what set of starting values would you want for each computer so that each has a unique set of γ values to solve?

```
# < Exercise 11, Starting assignments for non-overlapping calculations >

d_gamma = 0.001  # step size for gamma values for one computer to work on
n_CPU = 5  ### SET to the number of computers that will be working together

### Print out a list of CPU assignments and starting values for each CPU
```

Exercise 12*

Next, each team should have its computer find the values for a bifurcation diagram that uses the same 0.001 spacing but with different starting values, as calculated above. The results should then be written into a file so that they can be combined for the final plot.

In Simple Pendulum with Large Angle Release, you wrote NumPy arrays of data to a CSV file using numpy.savetxt. The organization of the data is more complex this time; rather than having two NumPy arrays to write, you will have different arrays of data for each value of γ. Therefore, we will exert more control over the file writing by following these steps.

1) Open a file for writing. The command is

```
fh = open(fname, "w")
```

where `fname` is the variable that stores the name of the file (e.g., `fname = "bifurcationDiagram1.csv"`) and `fh` is the file handle and how you will refer to the file throughout the program. The "w" tells the computer that you wish to write to the file.

Each time the file is opened for writing in this way, the existing contents will be erased. Alternatively, you can use "a" to tell the computer you would like to append to a file.

2) Write to the file using `fh.write("some string")`. In this case, the statement looks like

```
fh.write("{:9.6f},{:9.6f}\n".format(x,y))
```

where "x" and "y" are the two pieces of data you would like to write to the file, and the text within the quotation marks tells the computer that you would like the data formatted as two floats, separated by a comma, and ending with a new line. Notice that this is exactly the same formatting as you have been using with your print statements. Because you would actually like to write multiple pairs of data points, this statement is found within two nested loops.

3) Close the file using

```
fh.close()
```

```
# < Exercise 12, Calculations for shared bifurcation diagram >

# Here's how you might handle a given CPUx, showing how to write the results to
#    a file

CPU = "CPU2"  ### CHANGE to your group number
gamma_offset = 0.0002  ### FIX for your group

start_gamma = 1.060 + gamma_offset
end_gamma = 1.087
d_gamma = 0.001

# Open a file name unique for your CPU:
#    Name the new file with an extension such as .txt, .dat, or .csv so that
#    other programs can open it.
#    In this case, Comma Separated Value (.csv)
fname = "bifurcationDiagram"+CPU+".csv"
print("Storing (gamma, phi) data in "+fname+ "starting from {:}".
      format(start_gamma))
fh = open(fname, "w")  # Open the newly named file for "w" writing or
                       #    "a" appending.  Choose "w" for a new file.
                       #    Choose "a" to open an existing file
                       #    and continue writing at the end of it

# Open a plot that the data points will be added to
fig, ax = plt.subplots()
ax.axis([1.06, 1.087, -0.7, 0])
ax.set_xlabel(r'$\gamma$')
ax.set_ylabel(r'$\phi$(t)')

# Calculate the data for the bifurcation diagram
for gamma_drive_i in np.arange(start_gamma, end_gamma, d_gamma):
    print("{:9.6f}".format(gamma_drive_i)) # to show progress
```

```
    ### ADD code to solve the DDP for this value of gamma and find phi for
    #    each multiple of the driving period
    gamma_drive_arr = np.array([1])  ### FIX
    phi_values = np.array([1])  ### FIX

    # Plot as you go
    ax.plot(gamma_drive_arr, phi_values, marker = ',', color = 'b',
            linestyle = 'None')

    # Write each array to the file as you go (this will write 100 lines for each
    #    value of gamma_drive)
    #    Use a loop to format such that elements of the arrays are joined in
    #    (x,y) pairs, line-by-line
    for x, y in zip(gamma_drive_arr, phi_values):
        fh.write("{:9.6f},{:9.6f}\n".format(x,y))  # don't forget the "\n" to
                                                  #    get new lines after
                                                  #    each entry

#  Close the file
fh.close()

#  Show the plot
plt.show()
```

Exercise 13*

Once all the teams have had their computer finish generating the data, it is time to combine the data into a better, more detailed bifurcation diagram.

Put each group's results (the bifurcationDiagramxxx.csv files) into a common directory (the code below assumes this directory is the same as where you are running your code from). Then modify and run the next code block to open, read, and plot them together. Since the data was written in columns, separated by commas, you can use np.loadtxt to read it, similarly to what you did in Simple Pendulum with Large Angle Release and Orbits, Keplerian and Not. The `delimiter` argument should be set to `','` because you have commas separating the data, and you should set `unpack = True` so that the returned data is transposed and may be unpacked using `x, y, z = np.loadtxt(...)`.

In order to locate all the files, it is necessary to list the contents of the directory. This can be done using `os.listdir()`, as in the example below. The `os` module provides miscellaneous procedures for interfacing with the operating system. For example, the code below also uses it to return the current directory (`os.getcwd()`) and to change to a particular directory (`os.chdir()`).

```
#  < Exercise 13, Create the combined bifurcation diagram >

# Combine all the 2-column CSV data files into a single plot

import os  # operating system utilities
```

```
# Move to the directory containing the data files
cwd = os.getcwd()  # get the current directory
print("You are currently in directory: " + cwd)
pathSource = cwd  # you can change this to the source directory.  Here it
                  #    is assumed to be local directory
os.chdir(pathSource)
print("\nSource path: " + pathSource)

# Find the relevant data files
infiles = os.listdir()  # get the names of all files in the directory
bifurcationFiles = []  # empty list to hold relevant files
count = 0
for fn in infiles:
    #print("  ",fn)  # optionally, print all the files in the directory
    if ('bifurcation' in fn) and ('.csv' in fn[-4:]): # look only for relevant
                                                      #    files
        bifurcationFiles.append(fn)  # create a list of relevant files
        count +=1  # increment the counter
print("\nThere are {:} bifurcation data files: ".format(count))

# Set up the plot
fig, ax = plt.subplots()
ax.axis([1.06, 1.087, -0.7, 0])
ax.set_xlabel(r'$\gamma$')
ax.set_ylabel(r'$\phi$(t)')
ax.set_title("Combined Bifurcation Diagram")

# Loop through each of the files
for infile in bifurcationFiles:
    print("Reading " + infile)
    # Read the data from the file as two arrays, one of values of gamma and one
    #    of values of phi
    ### ADD code to read and plot data from "infile"

ax.legend(bifurcationFiles)
plt.show()  # after all files have been read in, show the combined plot
```

25.7 Check-out

Exercise 14

Briefly summarize the ideas in this unit.

25.8 Challenge Problem

You may be (rightly!) curious what happens with larger values of γ. We can easily extend the bifurcation diagram to examine this regime. The one modification it is useful to make

is to plot dϕ/dt rather than ϕ(t). The same physical locations can correspond to different values of ϕ(t) because the pendulum may have rolled over and increased its value by 2π. The angular velocity, dϕ/dt, however, does not share this degeneracy. Therefore, bifurcation diagrams using angular velocity rather than the angle will be cleaner.

If you have time, consider completing the following:

- Remake a bifurcation diagram plotting dϕ/dt versus γ for $1.06 < \gamma$ 1.53 (Figure 12.18 from *Classical Mechanics*). To what extent does increasing γ result in greater amounts of chaos?

25.9 Template Code for damped_driven_pendulum.py

```
import numpy as np
from scipy.integrate import solve_ivp  # the differential equation integrator

omega_drive = 0 # angular frequency of the driving force
omega_0 = 0  # natural frequency
beta_damp = 0  # damping constant

def deriv(t_now, r_now, gamma_drive):
    """ Return derivatives of the array (phi, dphi/dt) for the DDP
    The parameter, gamma_drive, is the drive strength. """

    return

def solve_DDP(phi0, dphi0, gamma_drive, max_time):
    """ For a given set of initial conditions, damping and maximum time solve the
    damped driven pendulum and return an array containing time, angle, and
    angular velocity"""

    return
```

State-Space Orbits and Poincaré Sections

26

In Damped Driven Pendulum, you were introduced to the damped driven pendulum (DDP) system and explored how its periodicity and sensitivity to initial conditions changed with the driving force. Then, in Bifurcation Diagrams you created and used a bifurcation diagram to further explore period doubling and the approach to chaos. In this unit, you will move from plotting $\phi(t)$ versus t to plotting $\dot{\phi}(t)$ versus $\phi(t)$, i.e., the state-space orbit. State-space orbits offer a powerful and elegant way to display information about a system. Taking a cross section of state-space orbits, known as a Poincaré section, offers a pared-down view of the system's motion and, in so doing, reveals a mysterious pattern within the chaotic motion known as a "strange attractor". After creating full state-space orbital plots, you will plot and interpret Poincaré sections of the DDP.

The material for this unit is covered in Ch. 12.7 and Ch. 12.8 of *Classical Mechanics*.

26.1 Objectives

In this unit, you will:

- construct state-space diagrams and Poincaré sections for the damped driven pendulum system (PHY);
- identify signatures of the non-chaotic, period doubling, and chaotic behavior in state-space diagrams and Poincaré sections (PHY);
- produce a Poincaré section showing a strange attractor, and determine what parameters affect it (PHY).

```
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = (12,8)  # make figures big for easier viewing
                                         #    (width, height in inches)
# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]

from scipy.interpolate import UnivariateSpline  # import the package to create
                                                #    a spline function
```

26.2 Setup

Exercise 1

Because we are working once again with a damped driven pendulum, you will need access to the `solve_DDP()` you wrote for Damped Driven Pendulum. The most elegant way to do this is to import a module that contains this code. If you have not already created such a module, follow the instructions in **Exercise 1** of Bifurcation Diagram.

Look over your `solve_DDP()` function in damped_driven_pendulum.py and modify it, if necessary, to use fixed time steps, e.g., $\Delta t = 0.01$ s. You may choose to use `np.arange()` rather than `np.linspace()` to form your time array to make it easier to choose Δt. Later, when we carry out the integration for longer times, we can keep the same resolution but increase the total number of steps.

Import the module by running the code below.

Finally, numerically find the solution to a damped driven pendulum with $\phi(t = 0) \equiv -\pi/2$ and $\mathrm{d}\phi/dt(t = 0) \equiv 0$ for $0 < t < 30$ using $\gamma = 0.6$.

```
# FOR GOOGLE COLAB ONLY

# < Exercise 1, Importing a module >

### UNCOMMENT the following lines if you are using Google Colab

#from google.colab import drive ## Mount your Google Drive
#drive.mount('/content/drive/')

#import sys  # Append your Google Drive "Colab Notebooks" directory to your
             #    path so that files contained in it may be found
#sys.path.append('/content/drive/My Drive/Colab Notebooks/')
```

```
import damped_driven_pendulum as ddp
```

```
# < Exercise 1, Solve DDP >
```

26.3 State-Space

Exercise 2

As your first introduction to state-space, plot the following for the system:

- $\phi(t)$ versus t (your familiar trajectory plot) and
- $\dot{\phi}(t)$ versus $\phi(t)$ (the state-space plot; Figure 12.21 from *Classical Mechanics*).

How is information about the trajectory encoded in the state-space plot? For example, how can the amplitude of the oscillation be read from the state-space plot? How is the oscillatory nature of the motion evident in the state-space plot? What direction does the system "move" in state-space over time?

```
# < Exercise 2, Plot trajectory and state-space orbit >
```

In Damped and Undamped Harmonic Oscillators, it was shown that under certain conditions, identical systems started with different initial conditions will exhibit the same oscillatory motion after a transient. This type of motion is called an attractor because the system gets "attracted" to that motion.

Exercise 3

Solve the damped driven pendulum a second time for a different set of initial conditions, saving the solution to a new variable. Then create the state-space plots for the previous system and the new system on the same axes.

How is the transient behavior and the attractor at $\gamma = 0.6$ apparent in state-space?

```
# < Exercise 3, Plot state-space orbit for different ICs >
```

Exercise 4

Remake both $\phi(t)$ versus t and $\dot{\phi}(t)$ versus $\phi(t)$ for γ = 1.078 and the original set of initial conditions ($\phi(t = 0) \equiv -\pi/2$ and $d\phi/dt(t = 0) \equiv 0$) for $0 < t < 30$ s (Figure 12.23 from *Classical Mechanics* with transients).

Using γ = 1.078 produces "period doubling". How does period doubling appear in the state-space plot?

```
# < Exercise 4, Plot trajectory and state-space orbit for gamma = 1.078 >
```

Exercise 5*

Complete the following example to get a feel for state-space orbits. Generate the two graphs from **Exercise 4** for $\gamma = 1.100$. If you only solve from $0 < t < 30$ s, transient behavior will overwhelm any pattern. By integrating for a longer period of time and eliminating the transient, can you find a version of the state-space orbit that reveals an identifiable structure?

```
# < Exercise 5, Plot trajectory and state-space orbit for gamma = 1.100 >
```

In general, simply plotting angular velocity versus angle will produce nice state-space diagrams for low values of γ. As we increase γ, though, we can run into problems.

Exercise 6

For example, try producing a state-space orbit for $\gamma = 1.400$ (using the same initial conditions as previously and only integrating from $0 < t < 6$ s; Figure 12.25 from *Classical Mechanics*).

Qualitatively describe the motion of the pendulum.

```
# < Exercise 6, Plot state-space orbit for gamma = 1.400 >
```

In order to get a more usable state-space diagram, we will want to limit ϕ to between $-\pi$ and $+\pi$. For example, if our numerical solution gives us a value of $\phi = \pi + 1$, we would like our plot to interpret that as $-\pi + 1$. Similarly, $\phi = 2\pi + 1$ should be interpreted as $\phi = 1$.

Exercise 7

Adjust your code to produce a state-space plot so that the values plotted on the ϕ-axis range between $-\pi$ and $+\pi$. Then verify your code by showing $\dot{\phi}$ versus ϕ for $\gamma = 1.400$ (Figure 12.26 from *Classical Mechanics*). Continue to use the same initial conditions and integrate from $0 < t < 30$ s.

➤ *Hint:* The modulo operator designated by "%", will be particularly helpful for modifying the angle. What the modulo operator does is return the remainder of a division problem. For example,

- 6%4 = 2
- 4%4 = 0
- 10%4 = 2

```
# < Exercise 7, Plot state-space for gamma = 1.400 eliminating rollover >
```

Exercise 8*

What is the source of the horizontal lines across your state-space plot?

If they bother you, you can plot the data points alone without the connecting lines using something like `ax.plot(x, y, marker = '.', linestyle = 'None')`.

It is also possible to eliminate the connecting lines by inserting `np.nan` into the ϕ and $\dot{\phi}$ arrays at each "rollover" using numpy.insert. That is because Matplotlib will automatically create a break in the plot when it encounters not-a-number values. The challenge is to find the indices where the jump occurs. For that, remember that the Boolean "True" is evaluated as one and "False" is evaluated as zero. Therefore, if you can write a logical statement that will evaluate True or False if there is a jump, the numpy.nonzero can be used to return the indices corresponding to True.

```
# < Exercise 8, Plot state-space orbit for gamma = 1.400 without horizontal
#    lines >
```

26.4 Poincaré Sections

So far, our state-space diagrams have been very simple. However, once we enter the chaotic regime, they can get complicated fast. Therefore, we are going to introduce another new tool: Poincaré sections. Rather than plotting the entire state-space orbit, Poincaré sections only plot the position in state-space *once every period*. This is analogous to a bifurcation diagram where the ϕ value every period is plotted against γ. The difference is that here we will plot angular velocity versus angle every period, rather than just the angle every period versus the forcing parameter.

Exercise 9

Based on your previous state-space plot for $\gamma = 0.600$ (**Exercise 2**), ***predict what the Poincaré section will look like.***

In order to plot the Poincaré section, you will need to know the values of the angle and angular velocity at specific times. Because the times you want may not have been included in your time array, you will need to interpolate across data points. As you found in Bifurcation, a spline function is the easiest way to do this.

As a reminder, to use scipy.interpolate.UnivariateSpline from `scipy.interpolate` library, the fit must first be created using a line of code similar to

```
spl = UnivariateSpline(x, y, k = 5, s = 0)
```

where `x` and `y` are the data arrays and the arguments `k` and `s` define qualities of the fit. Setting `k` to 5 tells it to compute a fifth-degree polynomial fit, while setting `s` to zero tells it to go through all the data points. Once the fit has been created,

```
y_fit = spl(x_fit)
```

will determine the y-values that correspond to an array of x locations contained in `x_fit`.

Exercise 10

Create a Poincaré section for $\gamma = 0.600$. To do this, use a spline fit to find the values of $\phi(t)$ and $\dot{\phi}(t)$ across time. Then create an array of discrete times from 20 to 60 s in steps of the drive period: $\Delta t = 2\pi/\omega_D$. Because the spline fits, `phi_fit` and `ang_vel_fit`, are smoothed functions over your original time array, you can use the number array as indices to create a new array of the values at every drive period from $t = 20$ to $t = 60$ s, e.g., `phi_fit(time_steps)`.

Starting at $t = 20$ rather than $t = 0$ means that transient behavior is eliminated. You still have to modify your `phi_fit(time_steps)` values to lie between $-\pi$ and π as you did in **Exercise 7**. Then plot (as points rather than a line) your angular velocity spline versus your modulo ϕ spline at those discrete time steps. Set your axis range to $-\pi < \phi < \pi$ and $-15 < \dot{\phi} < 15$ so that it will be easily comparable to your plot from **Exercise 2**.

➤ *Hint:* When using the plot command, the smallest marker is a single pixel, designated by a `','`. Slightly larger markers are designated with a `'.'` and a keyword argument marker size (`ms = 4`, for example). Some options:

```
plot(x, y, ',k')            # black dot (single pixel)
plot(x, y, '.b')            # blue dot, small
plot(x, y, '.r', ms = 4)    # red dot, medium
plot(x, y, ',g', ms = 100)  # green dot (single pixel; ms is ignored)
```

In the above example code, Python interprets the color and marker style arguments without your needing to write `marker = ','` and `color = 'k'` as the equivalent. As an added bonus, writing the arguments in this way eliminates the need to use `linestyle = 'None'` to ensure the lines are not requested.

```
# < Exercise 10, Poincare section for gamma = 0.600 >

### ADD code to create a Poincare section for gamma = 0.600
```

Exercise 11

Is your Poincaré section what you expected? Explain.

Exercise 12*

For each of $\gamma = 1.078$ (Figure 12.28 from *Classical Mechanics*), 1.081, and 1.105, generate a state-space plot and predict what the Poincaré section will look like. Then, generate the Poincaré section. Include all of your plots and predictions below. Adjust your axes ranges and the maximum integration times to produce good plots.

➤ *Hint:* For this specific set of values of γ, you may prefer to *not* modify ϕ to lie between $-\pi$ and π for your state-space diagrams. Please do make the modification before computing the Poincaré section, though.

Prediction for γ = 1.078

Prediction for γ = 1.081

Prediction for γ = 1.105

```
# < Exercise 12, State-space diagrams and Poincare sections for different
#    gamma >

### INCLUDE state-space diagrams and Poincare sections for the three different
#    values of gamma
```

26.5 Strange Attractors

As you can imagine, things get very exciting in the chaotic regime. You will start by looking at the system for $\beta = \frac{\omega_0}{8}$ and $\gamma = 1.500$.

➤ *Hint:* The variables for ω_0 and β in your damped_driven_pendulum module are accessible as `ddp.omega_0` and `ddp.beta_damp`, respectively.

Exercise 13

First plot ϕ versus t and the state-space diagram for the above values.

```
# < Exercise 13, Plot trajectory and state-space for gamma = 1.5 and
#    beta_damp = omega_0/8 >

### INCLUDE code to calculate and plot trajectory and state-space diagram
#    for gamma = 1.5 and beta_damp = omega_0/8
```

Exercise 14

Now plot the Poincaré section (Figure 12.29 from *Classical Mechanics*). Clip your transient behavior for $t < 20$ s. Keep on increasing the maximum time until you get a pattern you like and/or it starts taking too long to run.

➤ *Hint:* Recall that you can increase your figure size by using a command like this: `figure(figsize=(12, 8))`.

```
# < Exercise 14, Poincaré section for gamma = 1.5 and beta_damp = omega_0/8 >

### INCLUDE code to plot Poincaré section for gamma = 1.5 and
#    beta_damp = omega_0/8
```

Initially, it seemed like there was no underlying structure to the motion of a chaotic pendulum. Now, however, we see that there is. The pattern you have produced is called a "strange attractor," something typical of chaotic systems. Although not very evident here, strange attractors have a fractal structure.

Exercise 15*

Do you expect the Poincaré diagram to change if you try different initial conditions? How so?

Test your prediction.

```
# < Exercise 15, Poincaré section using different ICs >

### INCLUDE code to plot Poincaré section for gamma = 1.5 and
#    beta_damp = omega_0/8 with New ICS
```

Exercise 16*

Do you expect the Poincaré diagram to change for different values of the forcing and damping? How so?

Test your prediction.

```
# < Exercise 16, Poincaré section using different values of forcing and
#    damping >

### INCLUDE code to plot Poincaré section for new values of gamma and
#    beta_damp
```

Exercise 17*

Now, regenerate your plot $\beta = \frac{\omega_0}{8}$ and $\gamma = 1.500$. *However*, start your series of time instances at a slightly different moment in time. For instance, rather than plotting angular velocity versus angle at t = 20.0, 21.0, 22.0, 23.0 s, and so forth, plot them for $t = 20.05$, 21.05, 22.05 s, and so on. Therefore, your plot will still generate a point once per period of forcing but at a slightly later time. Slowly increase your starting time in steps of 0.05 s to see how the sections of the Poincaré plot evolve.

- *Hint 1:* You only need to find the numerical solution to the DDP once; it is only the spline fit to the new set of times that will need to be recalculated.
- *Hint 2:* You can compute each plot individually, but a better method is to loop through the offsets in time.

```
# < Exercise 17, Poincaré section at different times >

### CREATE Poincaré sections at different temporal offsets
```

26.6 Check-out

Exercise 18

Briefly summarize the ideas in this unit.

Appendix 1: Using solve_ivp() for Numerical Integration

27

The ordinary differential equation solver, scipy.integrate.solve_ivp (short for "solve initial value problem"), is part of the SciPy package. The `solve_ivp()` integrator takes a system of first-order differential equations and numerically integrates to find the solution across a range of provided times. As a result, any looping through time steps and appending of solutions takes place on the backend, hidden from the user.

One of the advantages of the `solve_ivp()` integrator is that it can run "adaptively" such that it automatically chooses the time steps for integration (including allowing for non-evenly spaced time steps). Alternatively, the user can ask that it be run using a fixed time array. `solve_ivp()` also supports a number of different integration methods. For instance, it includes Runge-Kutta 4/5 (the default), and the Fortran subroutine `lsoda`, which switches automatically between stiff and non-stiff methods, depending on the system of equations. All in all, `solve_ivp()` is a highly flexible and robust differential equation solver.

```
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = (12,8)  # make figures big for easier viewing
                                         #    (width, height in inches)
# To revert to the small default size, uncomment the following line
# plt.rcParams["figure.figsize"] = plt.rcParamsDefault["figure.figsize"]
```

```
# Importing solve_ivp. This must be done in any notebook that calls the
#    function

# The differential equation integrator
#    To reduce demand on memory, we import only the subset of the scipy
#    package that contains solve_ivp
from scipy.integrate import solve_ivp
```

27.1 Using solve_ivp()

To use the `solve_ivp()` solver, the user must provide the system of first-order ordinary differential equations in the form of a function (what will typically be called `deriv()`) that returns the derivative for each variable. For example, consider the mass-spring problem from

introductory mechanics:

$$F = ma = -kx$$
$$\frac{d^2x}{dt^2} = -\frac{k}{m}x,$$

where the x values are functions of time. The usual technique is to convert this second-order equation into two coupled first-order equations by defining a velocity function, which is itself a first derivative of the position function:

$$v = \frac{dx}{dt}$$

Then we seek simultaneous solutions to these two first-order equations:

$$v = \frac{dx}{dt}$$
$$\frac{dx}{dt} = -\frac{k}{m}x$$

The first parameter passed to the `deriv()` function is the point in time, `t_now`, at which derivatives are being calculated.

The second parameter is a small array, `r_now`, defining the state of the system at a particular point in time (here, `r_now` is a two-element array containing the position and velocity). The `deriv()` function returns an array containing the derivative for each of the elements. Notice that the lengths of `r_now` and the returned vector are the same as the number of differential equations.

Typically, a system will also depend upon other variables. For example, the mass-spring system depends on both the mass and spring constant. If it is driven, then the force of the driver needs to be known at the time `t_now`. The variables may be either defined globally or included in the `deriv()` function as additional parameters. If the second method is chosen, it is wise to supply default values for these parameters.

```
def deriv(t_now, r_now, mass = 1, spring_const = 1):
    """ Supplies the derivatives of x and v to be used by the differential
    equation integrator, solve_ivp.
    t_now is a parameter containing the time at which the derivatives are being
    calculated.
    r_now is an array containing the state of the system at a particular time:
    r[0] is position, r[1] is the velocity. The optional parameters, here mass
    and spring_const, are set to default values for the system. """

    x_pos = r_now[0]  # set x_pos to the first element of r_arr and vel to the
                      #    second
    vel = r_now[1]

    dx_dt = vel  # the derivative of the position is the velocity
    dv_dt = -(spring_const/mass)*x_pos  # the derivative of the velocity is the
                                        #    acceleration, here -k/m x
    return np.array([dx_dt, dv_dt])
```

The mechanics of the solution are then as follows:

1) Determine a range of times, `t_span`, over which the solution will be calculated. This range of times should be passed as a two-element list (or, often, a tuple): `t_span = [t_start, t_end]` to `solve_ivp()`. Unless directed otherwise, `solve_ivp()` will determine the time steps within that range at which to calculate a solution. However, if the user desires, an array of user-defined time steps can be passed to `solve_ivp()` through the `t_eval` parameter, as shown below. Alternatively, the parameter `max_step` may be set to define the maximum allowed time step.
2) Define an array of n initial conditions, one for each first-order differential equation (here called `r_init`).
3) Pass the `solve_ivp()` function the name of the `deriv()` function, the range of times, and the initial conditions. The return variable (here called `r_soln`) will contain the solution: `r_soln = solve_ivp(deriv, t_span, r_init)`

   - If any additional parameters are used in the `deriv()` function, they may be passed in a list to the `solve_ivp()` function using the `args` parameter: `r_soln = solve_ivp(deriv, t_span, r_init, args = (m, k))`.
   - If solutions are desired at specific user-defined times, those times may be passed as: `r_soln = solve_ivp(deriv, t_span, r_init, t_eval = t_arr)`.
   - If the user desires a higher level of accuracy for the solution, this may be set by including the parameters `atol` and `rtol` in the call to `solve_ivp()`. The optional parameters `atol` and `rtol` are absolute and relative tolerances in the calculations. Typically, setting both to `1e-11` will be sufficient for these exercises: `r_soln = solve_ivp(deriv, t_span, r_init, atol = 1e-11, rtol = 1e-11)`.

   On the backend, invisible to you, `solve_ivp()` will do something along the lines of the following:

   - Given a starting position x0[0] and velocity x0[1], calculate the new position by integrating the velocity function.
   - At the same time, find the new velocity by integrating the position function. The actual order of operations is important and dependent on the choice of algorithm but beyond the scope of this document.
   - Move to the next time step.

4) The result, `r_soln`, is an object containing the solution and information about it. Most importantly, `r_soln.t` is an array of N times and `r_soln.y` is an $n \times N$ array of values for the numerical solution, where n is the number of coupled first-order differential equations, and N is the number of times. In our example, this is a $2 \times N$ array with the position in the zeroth row and the velocity in row one, as illustrated below. If the keyword `t_eval = t_arr` is included in the function call, the length of these solutions is fixed at `len(t_arr)`. If `t_eval` is omitted, the solver is free to pick its own times consistent with any optional accuracy keywords (`atol = 1e-8, rtol = 1e-8`, for example). `t_arr` must lie within the range of `t_span`.

r_soln.t	t_0	t_1	t_2	...	t_N
r_soln.y[0]	x_0	x_1	x_2	...	x_N
r_soln.y[1]	v_0	v_1	v_2	...	v_N

The following cell illustrates how `solve_ivp()` may be called to solve the simple harmonic oscillator using the `deriv()` function defined above.

```
# Example of calling solve_ivp()

# Define the range of times (here from 0 to 100) over which to calculate the
#    solution.
#    This is shown in the more familiar list syntax, but it can also be defined
#    as a tuple, i.e., t_span = (0, 20)
t_span = [0, 20]

# Create a 2-element array of initial values
r_init = np.array([1.0, 0.0])  # initial values of x = 1, v = 0

# Define any additional variables
mass = 2 # mass
spring_const = 1 # spring constant

# Call the solver
r_soln = solve_ivp(deriv, t_span, r_init, args = (mass, spring_const))
```

The cell below will print the solution returned by `solve_ivp()` and show how to access different portions of that solution, such as the times, positions, and velocities. In this example, the first row of `r_soln.y` (accessible as `r_soln.y[0]`) contains the positions, and the second (accessible as `r_soln.y[1]`) contains the velocities.

```
print("Entire returned solution: ", r_soln)

print("\nFirst 10 times: ", r_soln.t[0:10])  # the first ten times
print("First 10 positions: ", r_soln.y[0, 0:10])   # the first ten positions
print("First 10 velocities: ", r_soln.y[1, 0:10])  # the first ten velocities
```

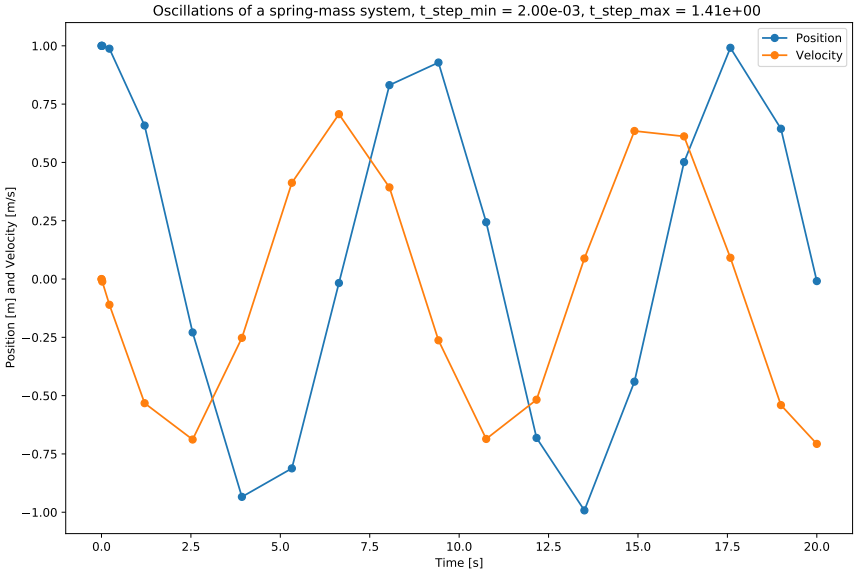
```
Entire returned solution:    message: 'The solver successfully reached the end
of the integration interval.'
     nfev: 110
     njev: 0
      nlu: 0
      sol: None
   status: 0
  success: True
```

```
        t: array([0.00000000e+00, 1.99800200e-03, 2.19780220e-02,
2.21778222e-01,
       1.20549753e+00, 2.54790102e+00, 3.92600314e+00, 5.32287887e+00,
       6.63841223e+00, 8.05052409e+00, 9.42173373e+00, 1.07562692e+01,
       1.21639869e+01, 1.35017451e+01, 1.49011372e+01, 1.62884134e+01,
       1.75837866e+01, 1.89969943e+01, 2.00000000e+01])
 t_events: None
        y: array([[ 1.00000000e+00,  9.99999002e-01,  9.99879244e-01,
         9.87728782e-01,  6.58131681e-01, -2.29072440e-01,
        -9.34122593e-01, -8.11989445e-01, -1.71663604e-02,
         8.31388959e-01,  9.28297972e-01,  2.43596339e-01,
        -6.81323820e-01, -9.92076931e-01, -4.40206391e-01,
         5.01482381e-01,  9.91512502e-01,  6.44420481e-01,
        -8.87838970e-03],
       [ 0.00000000e+00, -9.99000667e-04, -1.09885687e-02,
        -1.10435157e-01, -5.32368390e-01, -6.88280875e-01,
        -2.52320423e-01,  4.12662519e-01,  7.06955026e-01,
         3.92848059e-01, -2.62780513e-01, -6.85736231e-01,
        -5.17495737e-01,  8.81856862e-02,  6.34815448e-01,
         6.11663377e-01,  9.11191655e-02, -5.40567644e-01,
        -7.06962803e-01]])
 y_events: None

First 10 times:  [0.00000000e+00 1.99800200e-03 2.19780220e-02 2.21778222e-01
 1.20549753e+00 2.54790102e+00 3.92600314e+00 5.32287887e+00
 6.63841223e+00 8.05052409e+00]
First 10 positions:  [ 1.          0.999999    0.99987924  0.98772878
0.65813168 -0.22907244
 -0.93412259 -0.81198944 -0.01716636  0.83138896]
First 10 velocities:  [ 0.         -0.000999   -0.01098857 -0.11043516
-0.53236839 -0.68828088
 -0.25232042  0.41266252  0.70695503  0.39284806]
```

We can plot or otherwise analyze the results by accessing the appropriate rows in `r_soln.y` and the time steps stored in `r_soln.t`.

```
# Plot of position and velocity versus time, as determined by solve_ivp()

fig, ax = plt.subplots()
ax.plot(r_soln.t, r_soln.y[0], '-o')   # position
ax.plot(r_soln.t, r_soln.y[1], '-o')   # velocity
min_step = np.min(np.diff(r_soln.t))
max_step = np.max(np.diff(r_soln.t))
ax.legend(["Position", "Velocity"])
ax.set_xlabel("Time [s]")
ax.set_ylabel("Position [m] and Velocity [m/s]")
ax.set_title("Oscillations of a spring-mass system, t_step_min = {:.2e},"
             " t_step_max = {:.2e} ".format(min_step, max_step))


print("Minimum time spacing: {0:3.2e} s\nMaximum time spacing: {1:3.2e} s".
      format(min_step, max_step))
plt.show()
```

```
Minimum time spacing: 2.00e-03 s
Maximum time spacing: 1.41e+00 s
```



27.1.1 Time steps

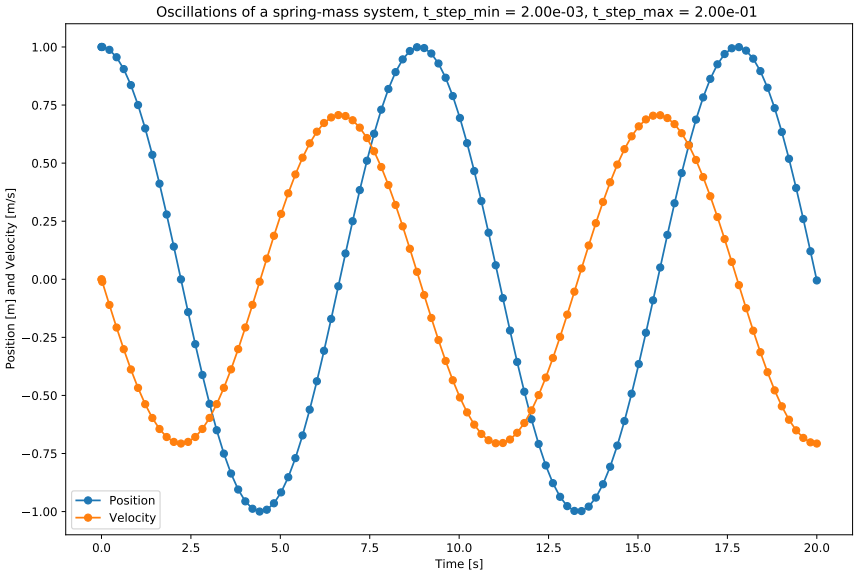
The solution above appears choppy because the integrator is picking relatively large time step sizes to solve for the final solution. This strategy allows the integrator to achieve the desired level of accuracy for the solution at the final time using the minimum number of time steps (and, therefore, the fastest computational speed). If you look closely at the graph or the list of the first 10 times from the output above, you may notice that the separation between time steps is not constant. The integrator is making the decision to take smaller steps when the system is quickly changing (i.e., the derivatives are large) than when it is more stable.

The integrator will use smaller time steps if the parameter `max_step` is used in `solve_ivp()` to set the maximum time step. An example of this is shown below.

```
# Solve and plot the solution with a maximum time step size of 0.2 seconds.

r_soln = solve_ivp(deriv, t_span, r_init, args = (mass, spring_const),
                   max_step = 0.2)
min_step = np.min(np.diff(r_soln.t))
max_step = np.max(np.diff(r_soln.t))
print("Minimum time spacing: {0:3.2e} s\nMaximum time spacing: {1:3.2e} s".
      format(min_step, max_step))
fig, ax = plt.subplots()
```

```
ax.plot(r_soln.t, r_soln.y[0], '-o') # Position
ax.plot(r_soln.t, r_soln.y[1], '-o') # Velocity
ax.legend(["Position", "Velocity"])
ax.set_xlabel("Time [s]")
ax.set_ylabel("Position [m] and Velocity [m/s]")
ax.set_title("Oscillations of a spring-mass system, t_step_min = {:.2e}, "
             "t_step_max = {:.2e} ".format(min_step, max_step))

plt.show()
```

```
Minimum time spacing: 2.00e-03 s
Maximum time spacing: 2.00e-01 s
```

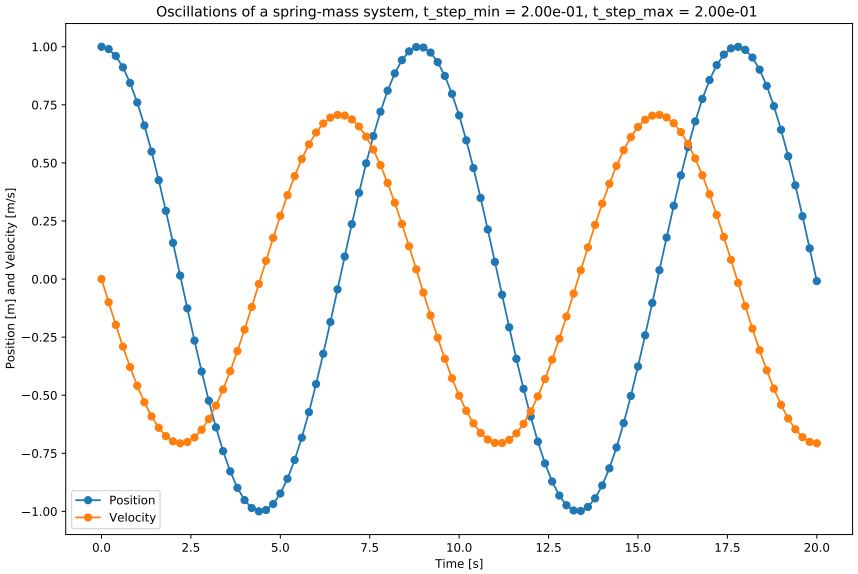


Alternatively, the user may define an array of times for which the solution should be calculated. An example is shown below.

```
# Define an array of 101 times going from the minimum to the maximum time in
#    t_span
t_arr = np.linspace(t_span[0], t_span[1], 101)

r_soln = solve_ivp(deriv, t_span, r_init, args = (mass, spring_const),
                   t_eval = t_arr)  # the optional parameter, t_eval, is used
                                    #    to define the array of times over
                                    #    which the integration should proceed
min_step = np.min(np.diff(r_soln.t))
max_step = np.max(np.diff(r_soln.t))
print("Minimum time spacing: {0:3.2e} s\nMaximum time spacing: {1:3.2e} s".
      format(min_step, max_step))
fig, ax = plt.subplots()
```

```
ax.plot(r_soln.t, r_soln.y[0],'-o')   # position
ax.plot(r_soln.t, r_soln.y[1],'-o')   # velocity
ax.legend(["Position", "Velocity"])
ax.set_xlabel("Time [s]")
ax.set_ylabel("Position [m] and Velocity [m/s]")
ax.set_title("Oscillations of a spring-mass system, t_step_min = {:.2e}, "
             "t_step_max = {:.2e} ".format(min_step, max_step))
plt.show()
```

```
Minimum time spacing: 2.00e-01 s
Maximum time spacing: 2.00e-01 s
```



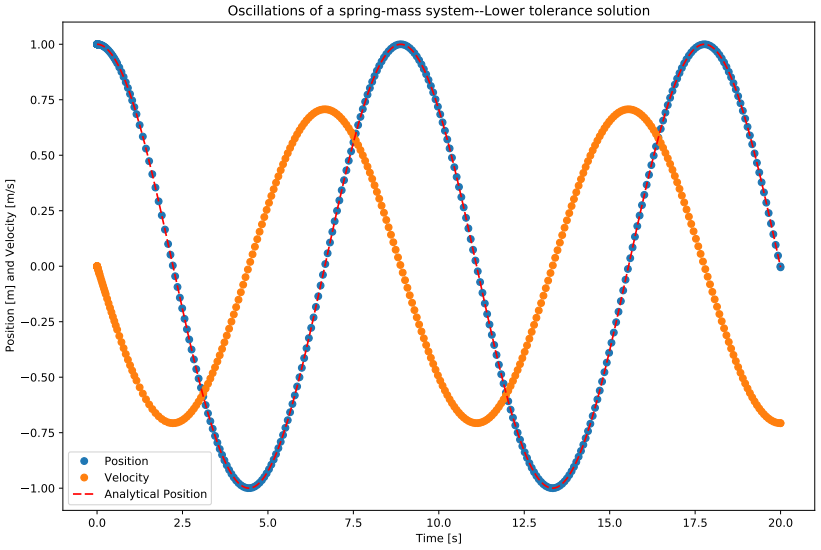
27.1.2 Accuracy

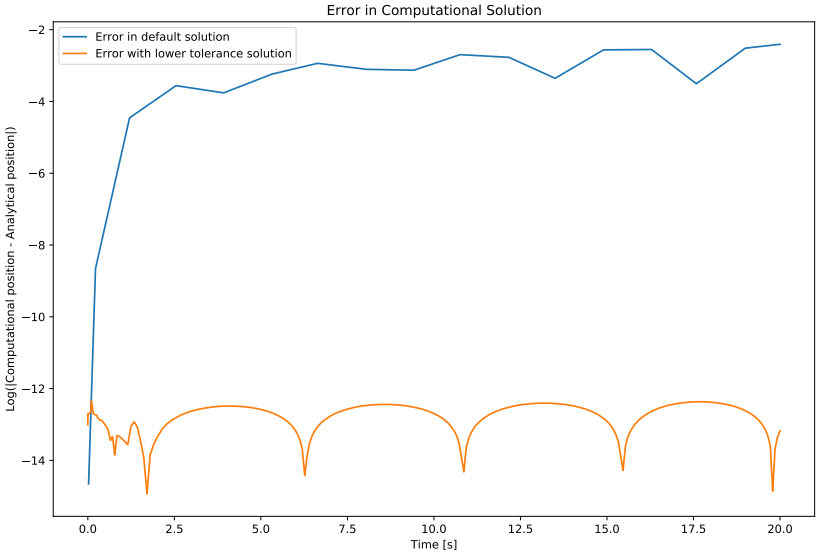
By default, `solve_ivp()` allows for a somewhat high level of inaccuracy in order to complete the calculations quickly. If you would like to require higher accuracy, the optional parameters `rtol` and `atol` in `solve_ivp()` may be set. These define the maximum relative and absolute tolerance, respectively. Specifically, local error estimates are required to be less than `atol + rtol * abs(y)`. The default values are `1e-3` for `rtol` and `1e-6` for `atol`.

Setting both to `1e-11` or `1e-13` will be more than sufficient for all of these exercises. When making this change, it is also helpful to set the integration method to LSODA, as that method is able to achieve high computational speeds in situations where high accuracy is desired.

Below is an example of how to call `solve_ivp()` with these changes, and it shows how the accuracy is increased.

```
# Example of calling the solver with greater requested accuracy

# Default solve_ivp call for comparison
r_soln_default = solve_ivp(deriv, t_span, r_init, args = (mass, spring_const))
# The analytical (exact) solution
r_analyt = np.cos(np.sqrt(spring_const/mass)*r_soln_default.t)
# Calculate the difference between the analytical solution and the default
#    computational solution
resid_default = r_analyt - r_soln_default.y[0]

# Calling solve_ivp to be computed using the "LSODA" integration method,
    # an absolute tolerance of 1e-13, and a relative tolerance of 1e-13
r_soln_better = solve_ivp(deriv, t_span, r_init, args = (mass, spring_const),
                          method='LSODA',
                          atol = 1e-13, rtol = 1e-13)  # Set parameters to
                                                       #    require high accuracy
# The analytical (exact) solution
r_analyt = np.cos(np.sqrt(spring_const/mass)*r_soln_better.t)
# Calculate the difference between the analytical solution and the more accurate
#    computational solution
resid_better = r_analyt - r_soln_better.y[0]

# Plot the better (lower-tolerance) solution
fig, ax = plt.subplots()
ax.plot(r_soln_better.t, r_soln_better.y[0],'o')   # position
ax.plot(r_soln_better.t, r_soln_better.y[1],'o')   # velocity
ax.plot(r_soln_better.t, r_analyt,
        color = 'red', linestyle = (0, (5, 2)))   # analytical position
ax.legend(["Position", "Velocity", "Analytical Position"])
ax.set_xlabel("Time [s]")
ax.set_ylabel("Position [m] and Velocity [m/s]")
ax.set_title("Oscillations of a spring-mass system--Lower tolerance solution")
plt.show()

# Plot the residuals on a logarithmic scale for the two different numerical
#    solutions compared to the analytical solution
np.seterr(divide = 'ignore')  # suppress error warning generated by taking
                              #    log(0)
fig, ax = plt.subplots()
ax.plot(r_soln_default.t, np.log10(np.abs(resid_default)))   # position residual
ax.plot(r_soln_better.t, np.log10(np.abs(resid_better)))   # position residual
ax.legend(["Error in default solution", "Error with lower tolerance solution"])
ax.set_xlabel("Time [s]")
ax.set_ylabel("Log(|Computational position - Analytical position|)")
ax.set_title("Error in Computational Solution")
plt.show()
```





Increasing the desired accuracy comes at the cost of increased computational time. The code below uses `timeit` to determine the amount of time it takes to find a solution at various desired levels of accuracy. Similar comparisons may be made between integration methods (i.e., LSODA or RK45) or time step criteria. Note that the differences in computational speeds will be unique to a particular problem.

```
import timeit
```

```
%%timeit
r_soln = solve_ivp(deriv, t_span, r_init, args = (mass, spring_const))
```

```
1.59 ms ± 101 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
%%timeit
r_soln = solve_ivp(deriv, t_span, r_init, args = (mass, spring_const),
                           method='LSODA', atol = 1e-13, rtol = 1e-13)
```

```
5.83 ms ± 246 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%%timeit
r_soln = solve_ivp(deriv, t_span, r_init, args = (mass, spring_const),
                           method='LSODA', atol = 1e-9, rtol = 1e-9)
```

```
2.87 ms ± 47.6 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

27.1.3 Checking for events

Sometimes you may wish to know the times at which certain events happen, such as "When did x equal 5?" or "When did the velocity exceed 0.1 m/s?" You can find these times by defining a function that will return zero when the event happens, and then passing that function to `solve_ivp()` using the `events` argument.

The following code cells illustrate this technique using the mass-on-a-spring system. Here, we identify the times that the mass crossed the zero point ($x = 0$) going from positive locations to negative locations.

Once `solve_ivp()` has been run and the events recorded, the times of the events may be accessed with

```
r_soln.t_events[0]
```

And the values of the solutions at the times of these events can be found using

```
r_soln.y_events[0][:,0]
```

for the first parameter in `deriv()` (here, the position) and

```
r_soln.y_events[1][:,0]
```

for the second parameter (here, the velocity), and so on.

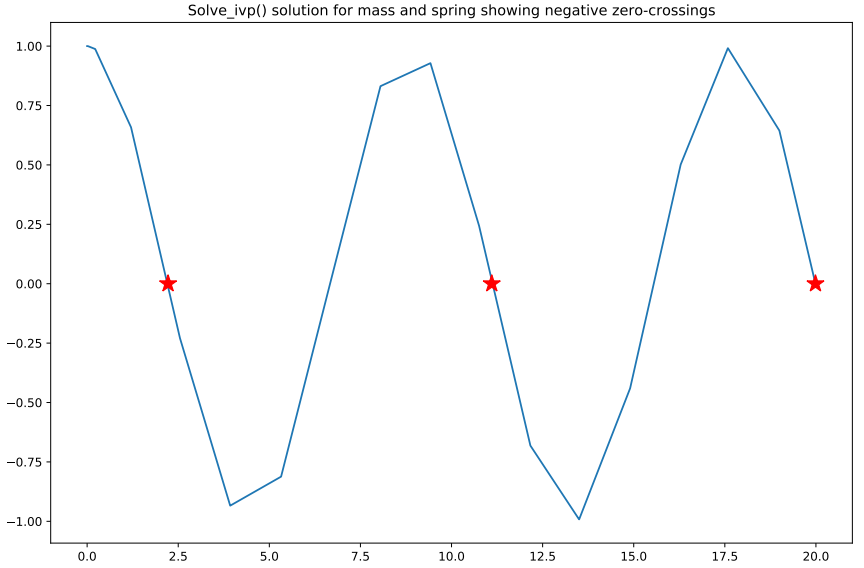
One word of caution: because `solve_ivp()` is adaptive, it may make large steps and be confused by multiple crossings.

```
# Example of defining an event for use in solve_ivp()

def t_zc(t_now, r_now, mass = 1, spring_const = 1):
    """ This function serves as the event called by solve_ivp().  It will
    record the time at which the returned variable goes to zero. In this
    example, that means when the position r_now crosses zero. The list of
    arguments passed to the event function must match those of deriv(), even
    if not all those arguments are used.  """
    return r_now[0]
```

```
# By default, both positive- and negative-going solutions will be recorded.
#    You can specify only negative-going crossings via a
#    t_zc.direction = -1 statement executed outside of this function, as below.
#    For positive crossings, you would use t_zc.direction = 1.
t_zc.direction = -1 # to select only negative-going zero-crossings
```

```
# Example of using events with solve_ivp()

# Modify the solver by adding 'events = event_function'
r_soln = solve_ivp(deriv, t_span, r_init, events = t_zc,
                   args = (mass, spring_const))

# Print information about the events.
print("Negative zero-crossings at times:", r_soln.t_events[0])
print("Periods:", np.diff(r_soln.t_events[0]))

# Plot illustrating the events
fig, ax = plt.subplots()
ax.plot(r_soln.t, r_soln.y[0])

# In the next step we overplot the events (note that r_soln.y_events[0][:,0]
#    is an array of zeros)
ax.plot(r_soln.t_events[0], r_soln.y_events[0][:,0], '*r', ms = 15)
ax.set_title("Solve_ivp() solution for mass and spring showing negative zero-"
             "crossings")
plt.show()
```

```
Negative zero-crossings at times: [ 2.22129673 11.10414833 19.98744207]
Periods: [8.8828516  8.88329373]
```



Appendix 2: Basic Input/Output of Files in Python

28

An important skill is to be able to read and write data from files. This skill enables you to store and retrieve data (rather than recalculating it), to share data between researchers, and to use it within different contexts.

Typically data are stored in human-readable form, consisting of columns representing time, distance, velocity, and so on, written out as floating-point numbers. The most common format is comma-separated values (CSV), though others such as tab-delimited or even binary can be used. This notebook illustrates various techniques for reading and writing data, beginning with NumPy methods, then introducing the more sophisticated (and memory-intensive) Pandas library, which can handle many different file formats, including Excel.

```
import numpy as np
import matplotlib.pyplot as plt
```

28.1 Read/Write to a File Handle

28.1.1 Write

The most basic and flexible approach to writing data is to identify a file using a "file handle" and then write to it. A file handle is an object upon which one can read or write. The file handle is assigned for a particular filename and with a particular intent: write (which overwrites files), append (which continues writing to a file), and read. The basic steps for using a file handle are

1) open the file handle
2) read or write (or append) to the file using the file handle
3) close the file handle

It is important not to leave a file handle open because it may lock the file from being edited or used in another application.

The cell below shows an example of creating a filename and file handle for writing (or appending). Here we write the data in CSV format, comma-separated values.

➤ *Caution: This overwrites files of the same name.*

(If you are using Google Colab, you will also need to mount the Google Drive by uncommenting the code in the cell below.)

```
path = './'

# FOR GOOGLE COLAB ONLY

### UNCOMMENT the following lines if you are using Google Colab

#from google.colab import drive ## Mount your Google Drive
#drive.mount('/content/drive/')

#import sys  # Append your Google Drive "Colab Notebooks" directory to your
             #    path so that files contained in it may be found
#sys.path.append('/content/drive/My Drive/Colab Notebooks/')
#path = '/content/drive/My Drive/Colab Notebooks/'
```

```
# Create some data
x_arr = np.arange(0, 10, 0.3)
y_arr = np.sin(x_arr)

# Name the new file with an extension such as .txt, .dat, or .csv so that other
#    programs can open it
fname = path + "myfile1.csv"

# Open the newly named file for "w" writing or "a" appending
fh = open(fname, "w")  # Choose "w" for a new file.  You could also choose "a"
                       #    to open an existing file and continue writing at
                       #    the end of it

# Use a loop to step through the elements in the array. "zip" allows the user
#    to step through both arrays together
for x_data, y_data in zip(x_arr, y_arr):
    # Write the formatted data file as comma-separated values (CSV)
    fh.write("{:5.3f},{:5.3f}\n".format(x_data, y_data))  # Include a newline
                                                         #    \n format
                                                         #    character
                                                         #    to mark the end
                                                         #    of each line
# Close the file so that another application can use it
fh.close()
```

If you open `myfile1.csv` in a text editor (look for it in your Google Drive "Colab Notebooks" folder, if you are using Google Colab), it will have the expected (x,y) format. Note that we achieved this by writing the file line-by-line with a `for` loop. If you had written the entire arrays (x_arr, y_arr) they would appear as all *x*-values, followed by all *y*-values. No one wants to read a file in that format.

28.1.2 Read

To read the file, open the file under the same name and create a new file handle for reading. Then use `read()`, `readline()`, or `readlines()` to read in the contents of the file. When you are done, close the file handle.

Your options for functions that read data in from a file are as follows:

- `read(n)` reads in the next `n` bytes from the file and returns them as a string. If `n` is not specified, the entire file is read in.
- `readline()` reads in the next one line of the file and returns it as a string.
- `readlines()` reads in all of the lines and returns them as a list of strings.

```
# Read the file written by the cell above

fh = open(fname, "r")  # Open the file handle
f_line_list = fh.readlines()  # Read all the lines of data as a list of strings
fh.close()  # Close the file handle

print("Line 1 of {0}: {1}".format(len(f_line_list), f_line_list[0]))
```

```
Line 1 of 34: 0.000,0.000
```

Alternatively, you can simplify your code by using a `with` statement. Doing so allows you to avoid explicitly closing the file handle, as shown below.

```
# Use "with" to read in data from a file.

with open(fname, "r") as fh:
    f_line_list = fh.readlines()  # Read all of its lines into a list

# Print the raw list, showing that it is a list of strings.  We'll need to
#   convert them into lists or arrays for plotting
print(f_line_list)
```

```
['0.000,0.000\n', '0.300,0.296\n', '0.600,0.565\n', '0.900,0.783\n',
'1.200,0.932\n', '1.500,0.997\n', '1.800,0.974\n', '2.100,0.863\n',
'2.400,0.675\n', '2.700,0.427\n', '3.000,0.141\n', '3.300,-0.158\n',
'3.600,-0.443\n', '3.900,-0.688\n', '4.200,-0.872\n', '4.500,-0.978\n',
'4.800,-0.996\n', '5.100,-0.926\n', '5.400,-0.773\n', '5.700,-0.551\n',
'6.000,-0.279\n', '6.300,0.017\n', '6.600,0.312\n', '6.900,0.578\n',
'7.200,0.794\n', '7.500,0.938\n', '7.800,0.999\n', '8.100,0.970\n',
'8.400,0.855\n', '8.700,0.663\n', '9.000,0.412\n', '9.300,0.124\n',
'9.600,-0.174\n', '9.900,-0.458\n']
```

As you saw in the output above, the data are not currently in a usable format. Each data pair is contained in a single string with a comma separating the values and a newline character, `\n`, at the end of it. What we would like instead are two arrays of data in the form of floats.

The cell below converts the list of strings (organized as *x*, *y* pairs) into two data elements by "parsing", or splitting, on the comma character. These data are then appended to their respective arrays.

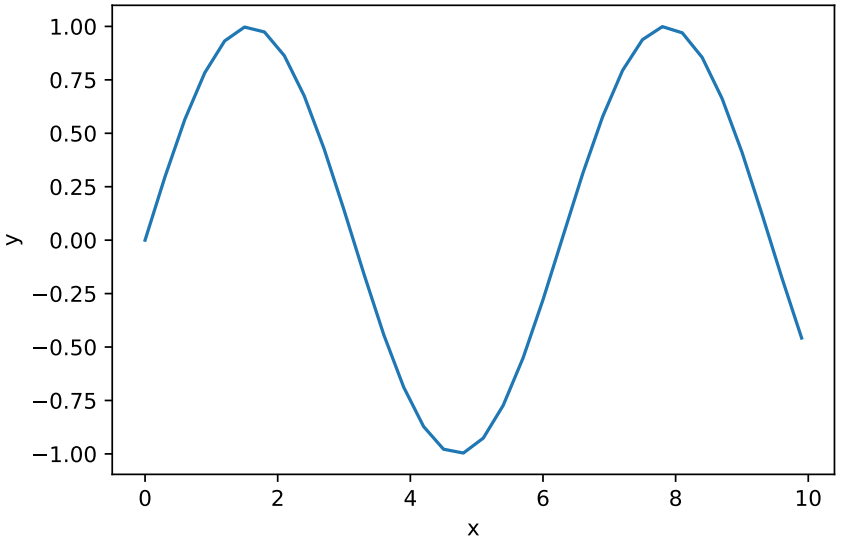
```
# Convert the raw contents of the file into arrays of data

x_new = np.array([])  # Create empty arrays to hold the x and y data
y_new = np.array([])

for element in f_line_list:  # Scan through the line list
    parsed = element.split(",")  # and split it into x and y pairs, using the
                                 #    comma character and split() method.
    print(parsed[0], parsed[1], end="")  # Notice now that the split method
                                         #    creates a two-element list for
                                         #    each line.
    x_new = np.append(x_new, float(parsed[0]))  # Convert each element into a
                                                #    floating-point number and
                                                #    append it to x_new.
    y_new = np.append(y_new, float(parsed[1]))  # Repeat for y_new.
```

```
0.000 0.000
0.300 0.296
0.600 0.565
0.900 0.783
1.200 0.932
1.500 0.997
1.800 0.974
2.100 0.863
2.400 0.675
2.700 0.427
3.000 0.141
3.300 -0.158
3.600 -0.443
3.900 -0.688
4.200 -0.872
4.500 -0.978
4.800 -0.996
5.100 -0.926
5.400 -0.773
5.700 -0.551
6.000 -0.279
6.300 0.017
6.600 0.312
6.900 0.578
7.200 0.794
7.500 0.938
7.800 0.999
8.100 0.970
8.400 0.855
8.700 0.663
9.000 0.412
```

```
9.300 0.124
9.600 -0.174
9.900 -0.458
```

Now that the lists are organized into (x,y) pairs, they can easily be plotted.

```
# Plot the pairs of x and y data

fig, ax = plt.subplots()
ax.plot(x_new, y_new)
ax.set_xlabel("x")
ax.set_ylabel("y")
plt.show()
```



28.2 Using NumPy for Reading and Writing Data Arrays

Using file handles with read/write commands is the most flexible way to manage input/output. However, frequently in physics you will want to read or write sets of data, for example the x and y position and times for ten different measurements. For such highly formatted data, there are simpler methods that allow you to avoid some of the loops and the string formatting and parsing from the examples above.

The function numpy.savetxt allows you to write NumPy arrays directly to a file. If you have multiple one-dimensional data arrays, all of the same length, you can first use numpy.column_stack to align the columns of the parallel arrays before calling `np.savetxt()`.

With `np.savetxt()`, you can format the data by sending the standard formatting commands to the `fmt` keyword argument. You can also set the delimiter (we recommend

commas) with the keyword argument of the same name. Finally, you can include a header that will appear at the top of the file with the `header` keyword argument. Such a header can be used to provide the names of the data columns or other metadata.

Once you have run the cell below, open up myfile2.csv to take a peek inside. Even better, try importing the contents into a spreadsheet program such as Excel.

```
# Here we illustrate with 4 columns

# Create 4 arrays of data
t_arr = np.arange(0,10,0.1)
x_arr = np.sin(t_arr)
y_arr = np.cos(t_arr)
z_arr = y_arr*x_arr

# Create a header
head = "{},{},{},{}".format("t","x","y","z")

# Write the data to a file
np.savetxt(path + "myfile2.csv", np.column_stack((t_arr, x_arr, y_arr, z_arr)),
           header = head, comments = "", delimiter = ",", fmt = "%5.3f")
# "header" allows you to write a first line to the file, typically containing
#    information about the different columns of data.
#    "comments" suppresses the # at the start of the header
#    "delimiter" sets the comma (alternatively, you could use tab via '\t')
#    fmt = '%5.3f' is a five-place floating-point format
```

Just as `np.savetxt` allows one to concisely write column-formatted data to a file, numpy.loadtxt enables one to read the contents into NumPy arrays.

In this example, the `delimiter` argument should once again be set to `','` because you have commas separating the data. The keyword argument `skiprows` is used to tell the computer to skip the (1) header row. This could alternatively be accomplished by writing the header with a starting "#" character (as would be the default by `np.savetxt()` if the `comments` keyword argument was not used).

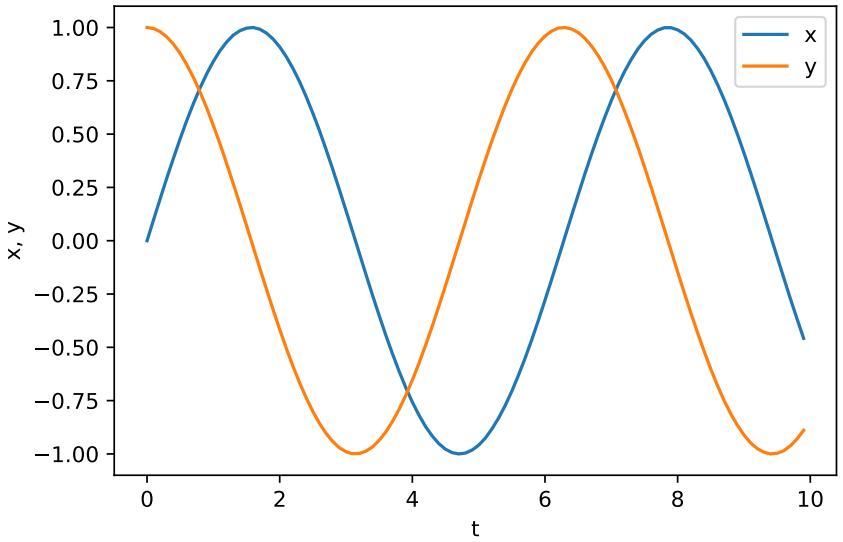
By default, the entire file will be read into one two-dimensional array. If you would prefer to read in each column of data into separate arrays, you may set `unpack = True` so that the returned data are transposed and may be unpacked using `x, y, z = np.loadtxt(...)`.

Examples of both are shown in the cell below.

```
# Read in data using loadtxt

data_f = np.loadtxt(path + "myfile2.csv", delimiter = ",", skiprows = 1)
print("First three rows of data:\n", data_f[:3])

t_arr_f, x_arr_f, y_arr_f, z_arr_f = np.loadtxt("myfile2.csv",
    delimiter = ",", skiprows = 1, unpack = True)
```

```
First three rows of data:
 [[0.    0.    1.    0.   ]
 [0.1   0.1   0.995 0.099]
 [0.2   0.199 0.98  0.195]]
```

```
# Plot the pairs of x and y data

fig, ax = plt.subplots()
ax.plot(t_arr_f, x_arr_f)
ax.plot(t_arr_f, y_arr_f)
ax.set_xlabel("t")
ax.set_ylabel("x, y")
ax.legend(["x", "y"])
plt.show()
```



28.3 Pandas

The powerful `Pandas` library provides tools for more extensive data analysis. Pandas is built on top of the NumPy library and provides many of the same functionalities. Unlike NumPy, however, the basic Pandas data structures, Series and DataFrames, allow one to store labeled data of different types. As a result, Pandas can accomplish many of the same tasks as Excel and, as you'll see, interfaces nicely with it. It can also be used to scrape data from web pages.

Pandas is an extensive library and we will only scratch the surface of it. We strongly recommend exploring it more fully if you want to do more complex data analysis than presented in this text.

```
import pandas as pd
```

```
# Read in a data file using Pandas

# Create an object called a 'DataFrame' by reading in a file
df = pd.read_csv(path + "myfile1.csv")  # myfile1.csv was created using
                                        #    "write()" in the first
                                        #    section: "Read/Write
                                        #    to a File Handle"
print(df)
```

```
    0.000  0.000.1
0     0.3    0.296
1     0.6    0.565
2     0.9    0.783
3     1.2    0.932
4     1.5    0.997
5     1.8    0.974
6     2.1    0.863
7     2.4    0.675
8     2.7    0.427
9     3.0    0.141
10    3.3   -0.158
11    3.6   -0.443
12    3.9   -0.688
13    4.2   -0.872
14    4.5   -0.978
15    4.8   -0.996
16    5.1   -0.926
17    5.4   -0.773
18    5.7   -0.551
19    6.0   -0.279
20    6.3    0.017
21    6.6    0.312
22    6.9    0.578
23    7.2    0.794
24    7.5    0.938
25    7.8    0.999
26    8.1    0.970
27    8.4    0.855
28    8.7    0.663
29    9.0    0.412
30    9.3    0.124
31    9.6   -0.174
32    9.9   -0.458
```

You will see that this is nice, but not perfect. Pandas assumes that the first line in the file contains header information, a label such as “x”, “y”, “time”, or “distance”. Note also that the index number 0 is shown on the second line. Subsequent lines give a row number (such as used in a spreadsheet) that is not part of the DataFrame itself.

An easy fix to the first line is to specify that there is no header using `header = None` and then specify the names of the different columns with the keyword argument `names`.

```
# Read in a data file without a header

df = pd.read_csv(path + "myfile1.csv",
                 header = None,      # tells read_csv the first line is data
                 names = ["x", "y"]) # "names" labels the data columns
print(df)
```

```
     x      y
0   0.0  0.000
1   0.3  0.296
2   0.6  0.565
3   0.9  0.783
4   1.2  0.932
5   1.5  0.997
6   1.8  0.974
7   2.1  0.863
8   2.4  0.675
9   2.7  0.427
10  3.0  0.141
11  3.3 -0.158
12  3.6 -0.443
13  3.9 -0.688
14  4.2 -0.872
15  4.5 -0.978
16  4.8 -0.996
17  5.1 -0.926
18  5.4 -0.773
19  5.7 -0.551
20  6.0 -0.279
21  6.3  0.017
22  6.6  0.312
23  6.9  0.578
24  7.2  0.794
25  7.5  0.938
26  7.8  0.999
27  8.1  0.970
28  8.4  0.855
29  8.7  0.663
30  9.0  0.412
31  9.3  0.124
32  9.6 -0.174
33  9.9 -0.458
```

Better yet, one can include a header when the data file is written to identify the columns. If you completed the section on **Using NumPy for Reading and Writing Data Arrays**, you created such a file, called "myfile2.csv," using `np.savetxt()`. If you have not yet, you may run the following cell to do so.

```
# Create four arrays of data
t_arr = np.arange(0,10,0.1)
x_arr = np.sin(t_arr)
y_arr = np.cos(t_arr)
z_arr = y_arr*x_arr

# Create a header
head = "{},{},{},{}".format("t","x","y","z")

# Write the data to a file
np.savetxt(path + "myfile2.csv", np.column_stack((t_arr, x_arr, y_arr, z_arr)),
           header = head, comments = "", delimiter = ",", fmt = "%5.3f")
```

Now read in the DataFrame, indicating that headers are present. Here we specify that only the zeroth, first, and second columns are used (handy if you had many columns).

```
df = pd.read_csv(path + "myfile2.csv", usecols = [0,1,2])
print(df)
```

```
     t      x      y
0   0.0  0.000  1.000
1   0.1  0.100  0.995
2   0.2  0.199  0.980
3   0.3  0.296  0.955
4   0.4  0.389  0.921
..  ...    ...    ...
95  9.5 -0.075 -0.997
96  9.6 -0.174 -0.985
97  9.7 -0.272 -0.962
98  9.8 -0.366 -0.930
99  9.9 -0.458 -0.889

[100 rows x 3 columns]
```

Once the columns of the DataFrame have been labeled, a single column of data may be selected using `frame[column_name]` or `frame.column_name`, as illustrated below. These columns can then be manipulated similarly to NumPy arrays.

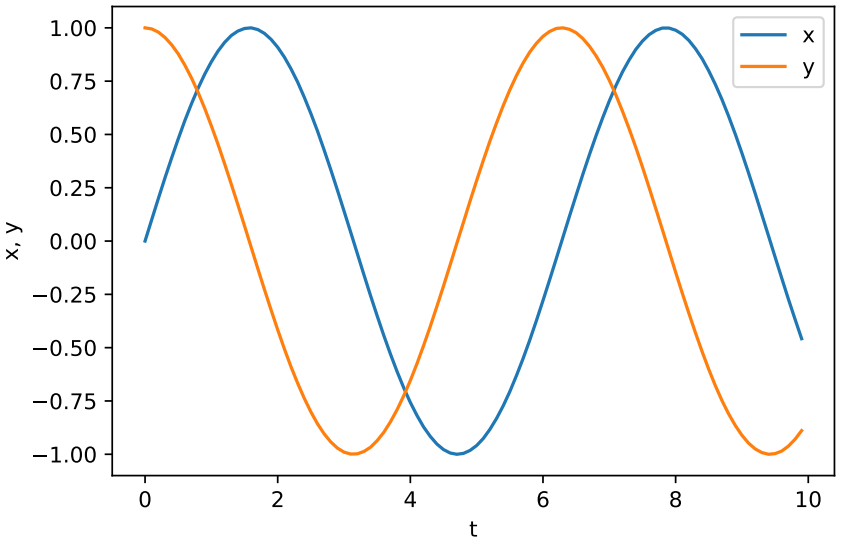
Rows of data may be selected by using `.loc[]` with the label of the row or `.iloc()` with the index of the row (these are actually the same thing in this example, as the row labels were not otherwise specified).

```
print("Column t of the DataFrame:\n", df.t)

print("\n\nThird row of the DataFrame:\n", df.iloc[2])

print("\n\nx - y:\n", df.x - df.y)
```

```
Column t of the DataFrame:
 0     0.0
1     0.1
2     0.2
3     0.3
4     0.4
     ...
95    9.5
96    9.6
97    9.7
98    9.8
99    9.9
Name: t, Length: 100, dtype: float64


Third row of the DataFrame:
 t    0.200
x    0.199
y    0.980
Name: 2, dtype: float64


x - y:
 0    -1.000
1    -0.895
2    -0.781
3    -0.659
4    -0.532
      ...
95    0.922
96    0.811
97    0.690
98    0.564
99    0.431
Length: 100, dtype: float64
```

Now we plot, referring to the x and y elements as `df.x` and `df.y`:

```
# Plot the data in the DataFrame

fig, ax = plt.subplots()
ax.plot(df['t'], df['x'])
ax.plot(df['t'], df['y'])
ax.set_xlabel("t")
ax.set_ylabel("x, y")
ax.legend(["x", "y"])
plt.show()
```



28.3.1 Creating an Excel workbook

If you already have a Pandas DataFrame such as the one above, you can turn it into a single-sheet Excel workspace:

```
# Write a DataFrame to Excel

df.to_excel(path + "ExcelFileFromDataFrame.xlsx",
            index = False)  # don't write the row names
#    Caution: this overwrites!
```

```
# Read in the just-created file
df_new = pd.read_excel(path + "ExcelFileFromDataFrame.xlsx")
df_new
```

```
      t      x      y
0   0.0  0.000  1.000
1   0.1  0.100  0.995
2   0.2  0.199  0.980
3   0.3  0.296  0.955
4   0.4  0.389  0.921
..  ...    ...    ...
95  9.5 -0.075 -0.997
96  9.6 -0.174 -0.985
97  9.7 -0.272 -0.962
98  9.8 -0.366 -0.930
99  9.9 -0.458 -0.889

[100 rows x 3 columns]
```

Alternatively, if you just have a few arrays, you can stack the columns, turn the resulting two-dimensional NumPy array into a DataFrame, and write it to an Excel file.

```
# Create a DataFrame from NumPy arrays and write to Excel

# Create arrays of data
x_arr = np.arange(0,20,0.1)
y_arr = 3*x_arr**2

# Create a DataFrame by stacking the columns together
column_values = ['x', 'y']  # the labels of the columns
df1 = pd.DataFrame(np.column_stack((x_arr, y_arr)),
                   columns = column_values)  # send a list of strings to
                                             #    "columns" to provide labels
                                             #    for the columns

df1.to_excel(path + "ExcelFileFromNumPyAndDataFrame.xlsx",
             index = False)  # don't write the row names
#    Caution: this overwrites!
```

```
df1_new = pd.read_excel(path + "ExcelFileFromNumPyAndDataFrame.xlsx")
df1_new
```

```
        x        y
0     0.0     0.00
1     0.1     0.03
2     0.2     0.12
3     0.3     0.27
4     0.4     0.48
..    ...      ...
195  19.5  1140.75
196  19.6  1152.48
197  19.7  1164.27
198  19.8  1176.12
199  19.9  1188.03

[200 rows x 2 columns]
```

Pandas can also write multiple sheets to an Excel file by using `pd.ExcelWriter` and then writing each DataFrame to a different sheet. The cells below illustrate writing and reading to Excel workbooks containing two sheets.

```
#  Writing two sheets to Excel

df2 = df1.copy()  # recall that this is a two-column sheet. Here we duplicate it
                  #    for simplicity
# This requires an ExcelWriter object like so:
with pd.ExcelWriter(path + "TwoSheetExcelFile.xlsx") as writer:
```

```
    df1.to_excel(writer, sheet_name='Sheet_name_1')
    df2.to_excel(writer, sheet_name='Sheet_name_2')
#  if you get a PermissionError, be sure that the file is not open in Excel
```

```
df = pd.read_excel("TwoSheetExcelFile.xlsx", header = 1, sheet_name = [0,1])

print(df[0])  # print "Sheet 1"
print(df[1])  # print "Sheet 2"
```

```
      0   0.1      0.2
0     1   0.1     0.03
1     2   0.2     0.12
2     3   0.3     0.27
3     4   0.4     0.48
4     5   0.5     0.75
..  ...   ...      ...
194  195  19.5  1140.75
195  196  19.6  1152.48
196  197  19.7  1164.27
197  198  19.8  1176.12
198  199  19.9  1188.03

[199 rows x 3 columns]
      0   0.1      0.2
0     1   0.1     0.03
1     2   0.2     0.12
2     3   0.3     0.27
3     4   0.4     0.48
4     5   0.5     0.75
..  ...   ...      ...
194  195  19.5  1140.75
195  196  19.6  1152.48
196  197  19.7  1164.27
197  198  19.8  1176.12
198  199  19.9  1188.03

[199 rows x 3 columns]
```

28.4 Binary Data

While beyond the scope of this text, you may eventually wish to manage huge data sets. This short introduction to reading and writing binary data will get you started on that path.

As a data set gets large, it becomes unwieldy to save it in human-readable integer or floating-point format. Each character, including commas, spaces, and new lines is encoded as an 8-bit (1 byte) character. The largest (unsigned) integer that can be stored in 64 bits is 18,446,744,073,709,551,615. That number takes up 20 characters, excluding the commas, and yet it can be represented in binary with 64/8 = 8 bytes, or 8 characters. If you are willing

to store floating-point numbers with limited precision, you can have a vastly greater storage compression. Briefly, here's how you store and retrieve binary data in Python.

28.4.1 Writing binary data

Python writes and reads binary files with the `.npy` extension. It stores (single-precision) integers as 4 bytes (32 bits), which is 8 hexadecimal digits, because a hexadecimal digit takes 4 bits (representing 0-9, A-F). Double-precision integers, or so-called long integers, are stored in 8 bytes.

Now try this code to create an array, display the contents in binary, and write the array to a file in binary format.

```
# Create sample data and save it in a binary file

x_arr = np.arange(30,40,1) # create an array of integers from 30 to 39
print(x_arr)

# Print the array in binary format to illustrate
print("x_arr in binary format:")
for index, x in enumerate(x_arr):
    print("{:}: {:064b}".format(index, x))

# Save the array as a binary file
np.save(path + "binary.npy", x_arr)
```

```
[30 31 32 33 34 35 36 37 38 39]
x_arr in binary format:
0: 0000000000000000000000000000000000000000000000000000000000011110
1: 0000000000000000000000000000000000000000000000000000000000011111
2: 0000000000000000000000000000000000000000000000000000000000100000
3: 0000000000000000000000000000000000000000000000000000000000100001
4: 0000000000000000000000000000000000000000000000000000000000100010
5: 0000000000000000000000000000000000000000000000000000000000100011
6: 0000000000000000000000000000000000000000000000000000000000100100
7: 0000000000000000000000000000000000000000000000000000000000100101
8: 0000000000000000000000000000000000000000000000000000000000100110
9: 0000000000000000000000000000000000000000000000000000000000100111
```

On a 64-bit computer, the native size of integers is 8 bytes. If you have Notepad++ installed, you can take a closer look at the file. If you do not have Notepad++ installed, you will not be able to complete all of the following subsection, but you may still find it informative to learn how data is stored in binary and hexadecimal.

Examining a binary file

Open up `binary.npy` in Notepad++ and take a peek. (The bytes won't encode correctly in Notepad, so don't even try.) It should show you a header line of reasonable text explaining the format of the file, and something about its byte ordering, followed by a line of more or less ASCII-code representation of the raw bytes. (Google "ASCII chart" if you want to

see a complete list of the 127 characters. ASCII stands for American Standard Code for Information Interchange. It is a 7-bit standard, where A = 65 decimal, B = 66 decimal, and so on. Anything below 32 decimal is unprintable, but meaningful to a printer, and includes such gems as 10 decimal = A hexadecimal = LF, "line feed", which in the Python or C world is designated as "\n", or newline, and 13 decimal = D hexadecimal = CR, "carriage return", which, don't laugh, comes from returning the typewriter-like carriage that holds the paper to its left-most position. Python understands this as "\r". All of the bytes consist of pairs of hexadecimal digits.)

Select all of the characters on the next line, and convert them from ASCII codes into hexadecimal via the Plugins/Converter ASCII->HEX menu in Notepad++.

You should see something like this (on a 64-bit Intel computer):

```
1E0000001F000000200000021000000
220000002300000024000000250000000
2600000027000000
```

You can verify that this is the same hexadecimal encoding that Python uses by printing the same data array in hexadecimal using the cell below.

```
# Print the data in hexadecimal

print("x_arr in hexadecimal:")
for index, x in enumerate(x_arr):
    print("{:}: {:x}".format(index, x))
```

```
x_arr in hexadecimal:
0: 1e
1: 1f
2: 20
3: 21
4: 22
5: 23
6: 24
7: 25
8: 26
9: 27
```

Now examine how the number 30 gets stored, looking only at the first 2 hexadecimal characters 1E, where each character requires 4 bits. A hexadecimal character represents one of $2^4 = 16$ values, ranging from 0 to 9, and A = 10, B = 11, C = 12, D = 13, E = 14, and F = 15. Because 30 decimal exceeds $2^4 = 16$, we need two hex digits to store it:

```
00011110
```

Starting from the right side, which is $2^0 = 1$, the presence of a 1 indicates whether that bit is needed to represent the number. Add up all the bits and you have your answer: 128x0 + 64x0 +32x0 + 16x1 + 8x1 + 4x1 + 2x1 + 1x0 = 30 decimal.

This gets encoded as two hexadecimal digits (each one takes 4 bits), 1 and E, where the 1 represents $2^4 = 16$, and E represents 14, hence the decimal result is 30.

Try storing a decimal number bigger than 255 (but less than 65,535) in a binary file, by modifying your code above to store some bigger numbers, which will now take two bytes (4 hex digits) each.

```
# Here we illustrate with the 10 integers from 300 to 309

x_arr = np.arange(300,310,1)
print('x_arr:\n', x_arr)
np.save(path + "binary2.npy", x_arr)

print("x_arr in binary format:")
for index, x in enumerate(x_arr):
    print("{:}: {:064b}".format(index, x))

print("x_arr in hexadecimal:")
for index, x in enumerate(x_arr):
    print("{:}: {:x}".format(index, x))
```

```
x_arr:
 [300 301 302 303 304 305 306 307 308 309]
x_arr in binary format:
0: 0000000000000000000000000000000000000000000000000000000100101100
1: 0000000000000000000000000000000000000000000000000000000100101101
2: 0000000000000000000000000000000000000000000000000000000100101110
3: 0000000000000000000000000000000000000000000000000000000100101111
4: 0000000000000000000000000000000000000000000000000000000100110000
5: 0000000000000000000000000000000000000000000000000000000100110001
6: 0000000000000000000000000000000000000000000000000000000100110010
7: 0000000000000000000000000000000000000000000000000000000100110011
8: 0000000000000000000000000000000000000000000000000000000100110100
9: 0000000000000000000000000000000000000000000000000000000100110101
x_arr in hexadecimal:
0: 12c
1: 12d
2: 12e
3: 12f
4: 130
5: 131
6: 132
7: 133
8: 134
9: 135
```

This time we get

```
2C0100002D0100002E0100002F010000
3001000031010000320100003301000 0
3401000035010000
```

Note that 300 decimal is 012C in hexadecimal, where the (less) significant byte is 2C and the (more) significant byte is 01. But which byte appears first in the file? The less significant byte.

How does the next value (301 decimal) get stored? It turns out that there is no clear standard as to how integers are stored. Intel chose Little-Endian byte order, where the least significant byte is stored first. IBM, Motorola, and the Internet Protocol all use Big-Endian byte order where the bytes appear as you'd expect to read them. You might want to Google "Little- and Big-Endian" to read more about this (and catch the literary allusion).

In either Big- or Little-Endian, the bits within the bytes are still stored in the familiar readable order. There are times when you need to get raw integer output from a device, and you may need to be able to reverse the byte order. This is so common that there are canned Python routines (see `np.byteswap()`) to accomplish it. The need to swap storage order often crops up in converting high-resolution camera images from one format to another.

To illustrate this byte-swapping, try running the following block of code, which displays a single number, 300, in binary, hexadecimal, and with the order of the bytes swapped.

```
x = np.array(300)
print("The number x = 300 decimal in normal order: {:}".format(x))
print("Binary representation of decimal 300 in normal order: {:b}".format(x))
print("Hexadecimal representation of decimal 300 in normal order: {:x}".
      format(x))
print("Now swap the bytes:")
swap_300 = x.byteswap()
print(" x with the bytes swapped as decimal: {:}\n".format(swap_300))
print(" x with the bytes swapped as binary: {:b}\n".format(swap_300))
print(" x with the bytes swapped as hex: {:x}\n".format(swap_300))
```

```
The number x = 300 decimal in normal order: 300
Binary representation of decimal 300 in normal order: 100101100
Hexadecimal representation of decimal 300 in normal order: 12c
Now swap the bytes:
 x with the bytes swapped as decimal: 3170815612645539840

 x with the bytes swapped as binary:
1011000000000100000000000000000000000000000000000000000000000000

 x with the bytes swapped as hex: 2c01000000000000
```

And we really can't leave this topic until you've played with storing *negative* integers in Python. Repeat your storage example with these lines:

```
x_arr = np.arange(-300,-310,-1)  # Note that we need to count down, hence the
                                 #    spacing between values is -1.
print('x_arr:\n', x_arr)
np.save(path + "binaryNeg.npy", x_arr)
```

```
x_arr:
 [-300 -301 -302 -303 -304 -305 -306 -307 -308 -309]
```

Before you open up binaryNeg.npy, let's predict how Python will store the first element of the array. A common storage method for negative numbers is called "two's complement".

1) Write out 300 decimal in binary, using a total of 32 binary digits (bits)
   - 0000 0000 0000 0000 0000 0001 0010 1100 (which is 256 + 32 + 8 + 4)
   - Pause to reflect that each 4 bits = 1 hex digit, so this is 0000 012C hex
2) Negate every one of the bits, replacing 0s with 1s and vice versa
   - 1111 1111 1111 1111 1111 1110 1101 0011
   - This is FFFF FED3 hex
3) Add 1 in the usual right-to-left order
   - 1111 1111 1111 1111 1111 1110 1101 0100
   - This is FFFF FED4 hex, which is how -300 decimal is represented.

What you're left with is a number that, when added to 300 decimal, will yield 0, as expected. You can try it, longhand, in binary (32 bits, or if you insist, 64 bits).

But if you are on a Windows computer in the Little-Endian world, the bytes (represented by pairs of hex digits) will be in small-to-large order, so what do you expect the file to contain?

Now, open the binaryNeg.npy file in Notepad++. You'll find there is a problem showing and converting these bytes into ASCII characters, because Notepad++ doesn't recognize 8-bit bytes as ASCII, only 7-bit bytes.

Go to the Encoding menu in Notepad++ and select one of the UTF-8 encodings. You should now see a display like so, where "x" is shorthand for hexadecimal, indicating that the trailing 2 digits are in hex, not decimal:

xD4xFExFFxFFxD3xFExFFxFF …..

28.4.2 Reading binary data

To read in the arrays from their binary files and print the results, we can use `np.load()`.

```
x1 = np.load(path + 'binary.npy')
x2 = np.load(path + 'binary2.npy')
x3 = np.load(path + 'binaryNeg.npy')
print(x1)
print(x2)
print(x3)
print(x2+x3)  # 2's complementary works!
```

```
[30 31 32 33 34 35 36 37 38 39]
[300 301 302 303 304 305 306 307 308 309]
[-300 -301 -302 -303 -304 -305 -306 -307 -308 -309]
[0 0 0 0 0 0 0 0 0 0]
```

Here's the payoff: once you get to writing large files, storing your data in binary can save lots of file space (by a factor of 2). We illustrate this by creating the following large file of 10,000 floats.

```
x_arr = np.arange(0,1000,0.1)  # 10,000 numbers
fname = "TenThousandFloats"
np.save(path+fname+".npy", x_arr)

np.savetxt(path+fname+".csv", x_arr, comments='', delimiter=',',
           fmt='%15.12f')
```

Now, look at information about these two files: TenThousandFloats.npy (binary) and TenThousandFloats.csv (regular text file). You can do this by looking at the file information in your computer's file browser. Alternatively, you can use the `os` package to display information about the files, including their size.

```
# The os package contains miscellaneous operating system interfaces
import os

file_stats = os.stat(path+fname+".npy")
print("Binary file size in bytes: ", file_stats.st_size)

file_stats = os.stat(path+fname+".csv")
print("Text file size in bytes: ", file_stats.st_size)
```

```
Binary file size in bytes:  80128
Text file size in bytes:  169000
```

```
# Finally, verify that the data contained in the binary file is what you want

x_arr_from_file = np.load(path+fname+".npy")

print(x_arr_from_file)
```

```
[0.000e+00 1.000e-01 2.000e-01 ... 9.997e+02 9.998e+02 9.999e+02]
```

Appendix 3: Creating Animations with FuncAnimation

29

Animations of physical systems are frequently informative and always entertaining. Animations allow one to actually see the time evolution of a system, providing a conceptual underpinning to the mathematical solution. Although not explored in this text, animations can also be useful when taking data, as they can provide real-time graphical feedback.

The Matplotlib method `FuncAnimation` provides a tool for producing an animation by repeatedly calling a user-defined function that modifies a figure. Here we provide a short guide to working with it.

```
# Read in the relevant libraries

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches  # imports code to draw a circle

from matplotlib import animation  # import animation library from matplotlib
from IPython.display import HTML  # for displaying animation inline

plt.rcParams["figure.figsize"] = (8,8)  # make a square figure
#    (width, height in inches)
```

```
path = './'

# FOR GOOGLE COLAB ONLY

### UNCOMMENT the following lines if you are using Google Colab

#from google.colab import drive ## Mount your Google Drive
#drive.mount('/content/drive/')

#import sys  # Append your Google Drive "Colab Notebooks" directory to your
             #    path so that files contained in it may be found
#sys.path.append('/content/drive/My Drive/Colab Notebooks/')
#path = '/content/drive/My Drive/Colab Notebooks/'
```

29.1 Overview

The basic steps to create an animation are:

1) Initialize the figure window and axes, as well as any objects (such as lines and patches) that will be modified by the animation.
2) Define a function that will be called to create each frame. This function should take the frame number, `i`, and update the plot objects according to the value of `i`.
   2a) Optionally, another function may be created that will be called at the start of the animation.
3) Create the animation object by calling `animation.FuncAnimation()` with the information about the figure window and the animation function. This function call may take additional keyword arguments specifying the number of frames, the temporal spacing between the frames, additional parameters, and so on.
4) The animation object should be shown and/or saved.

Step 1 may be done after step 2. However, it is easiest to think through this process by considering step 1 first, so that is what we will do here.

29.2 Walk-through Example

We will use a simple pendulum (in the small-angle limit) as the example for this tutorial. Before creating the animation, we must therefore determine the parameters for the system, including the trajectory.

```
# Define a simple pendulum and write a function for its trajectory in the
#    small-angle limit

grav = 9.8  # [m/s^2] gravitational acceleration near the surface of Earth
length = 1.0  # [m]

omega = np.sqrt(grav/length)  # angular frequency of the pendulum

class Pendulum():
    """This class represents a simple pendulum in the small-angle limit. All
    values are in mks SI units. The length (length), initial angle (theta0),
    and initial angular velocity (theta_dot0) may be set using keyword
    arguments. By default, a 1 meter pendulum starts from rest at pi/10
    radians."""

    # Set the properties of the pendulum
    def __init__(self, length = 1, theta0 = np.pi/10, theta_dot0 = 0):
        self.length = length  # length
        self.theta0 = theta0  # initial angle
        self.theta_dot0 = theta_dot0  # initial angular velocity

        self.omega = np.sqrt(grav/length)  # angular frequency of the pendulum
        self.period = 2*np.pi/self.omega
```

```
        # Calculate the amplitude of the pendulum for the initial conditions
        self.amplitude = np.arccos(np.cos(self.theta0)
                                   - 1/(2*grav)*self.length*self.theta_dot0**2)
        # Calculate the phase shift
        self.delta = np.arccos(self.theta0/self.amplitude)

    # Return the angle of the pendulum from the vertical at a given time
    def angle_time(self, t):
        self.angle = self.amplitude*np.cos(self.omega*t + self.delta)

    # Return the x coordinate of the pendulum mass at a given time
    #    Assume that the pivot is at (0,1)
    def x_pos(self, t):
        self.angle_time(t)
        return self.length*np.sin(self.angle)

    # Return the y coordinate of the pendulum mass at a given time
    #    Assume that the pivot is at (0,1)
    def y_pos(self, t):
        self.angle_time(t)
        return 1 - self.length*np.cos(self.angle)

# Make a pendulum object, pnd, that will be animated
pnd = Pendulum()
```

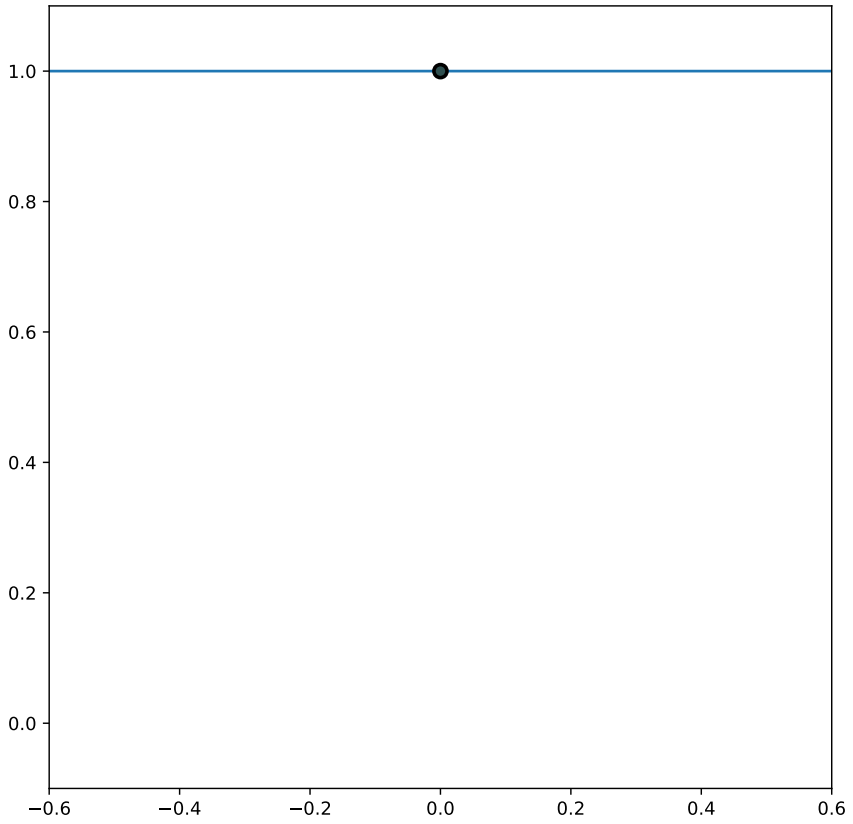
29.2.1 Initialization

Initializing a figure window and axes proceeds in the same manner as for creating a static plot. This time, though, we will also initialize the plot objects, a.k.a. artists. Here, we define both a line and a circular patch to make the string and the bob on the end of it. While we define these elements here, they will not actually appear on the figure until later on.

Code from this cell will have to be repeated later on. Here we introduce it just so that you can see how it works.

```
# Initialize the figure, axes, and artists

fig, ax = plt.subplots()
ax.set_aspect('equal')  # Make the tick spacing along the x and y axes equal
ax.set_xlim(-0.6, 0.6)
ax.set_ylim(-0.1, 1.1)

# Define the static plot elements:
#    a support, consisting of a horizontal line
ax.axhline(1)
#    a pivot point
pivot = patches.Circle((0,1), 0.01, fc ='DarkSlateGray', ec = 'black',
                       lw = 2 , zorder = 3)  # "zorder = 3" ensures that it will
                                             #    be plotted on top of the line
```

```
ax.add_patch(pivot)

# Define the plot elements that will change each frame:
#    a string
line, = ax.plot([], [], lw = 2, color = 'k')
#    a mass at the end of the string
bob = patches.Circle((pnd.x_pos(0), pnd.x_pos(0)), 0.02, fc ='red',
                     ec = 'black', lw = 2 , zorder = 3)

#    a title
title = ax.text(0.15, 0.9, '', horizontalalignment='left',
                verticalalignment='bottom')
plt.show()
```



29.2.2 Frame functions

Each frame, the position of the patch representing the pendulum mass and the line defining the pendulum string will change. Below, we write a function that will make these changes for the time associated with a particular frame.

This function takes as a required argument the value of the frame. The function may also take additional arguments. How to pass those arguments is described in **Call the Animator**.

You will notice that this function makes several changes to the artists. It relocates the bob using `bob.set_center()`, and it redraws the string using `line.set_data()`. Other properties of artists, such as their facecolor, their linewidth, and so on, can also be changed using `set`. For example `bob.set_color('red')` would change the color of the patch to red. For a list of properties, see the Matplotlib documentation for the specific artist class, for example by typing `bob??`.

```
def animate(i):
    """Plot the pendulum bob and string at a time corresponding to the frame
    number, i"""
    ftime = i*interval*1e-3  # conversion between frame number and time.
                             #    We use the interval between frames defined
                             #    for FuncAnimation (in units of milliseconds)
                             #    in order to make it run in real time

    # Update the plot title to show the time
    title.set_text("time  = {:3.2f} s".format(ftime))

    # Update the position of the bob at time = ftime
    bob.set_center([pnd.x_pos(ftime), pnd.y_pos(ftime)])
    # Redraw the string
    line.set_data([0, pnd.x_pos(ftime)], [1, pnd.y_pos(ftime)])

    # For blitting, return an iterable of artists to be redrawn
    return (bob, line, title)
```

We also define a function that will be called first, before any of the frames. This `init()` function creates the base frame upon which the animation takes place. Returning the line and the patch ensures that the animator knows which plot objects will be updated in each frame.

This step is important for "blitting", which can save time in generating the animation by only redrawing artists that change. For blitting to be used, both the `animate()` and `init()` function must return an iterable of artists to be redrawn.

If blitting is not being used (so the keyword "blit" in FuncAnimation is False or not set) it may be skipped. In that case, the first frame generated by the `animate()` function will serve as the base frame.

```
def init():
    """Create the base frame for the pendulum animation"""
    ax.add_patch(bob)  # add the bob
    line.set_data([0, pnd.x_pos(0)], [1, pnd.y_pos(0)])  # draw the string
    title.set_text('')  # initialize the title

    # For blitting, return an iterable of artists to be redrawn
    return (bob, line, title)
```

29.3 Call the Animator

Generating the animation requires repeatedly calling the `animate()` function and then determining how those generated frames will be looped through. The method to generate the animation is `animation.FuncAnimation()`. It takes as required arguments the figure object (`fig`) and the name of the function that generates the frames (`animate`).

```
anim = animation.FuncAnimation(fig, animate)
```

We also use the following keyword arguments in our example:

1) `init_func` takes the name of a function that creates the base frame. If you would like to pass arguments to this function beyond the frame count, this can be done using functools.partial. The syntax is to replace the name of the function in the argument with a call to `partial` that contains the function name and all the arguments except the frame count:

   ```
   anim = animation.FuncAnimation(fig, partial(animate, art=ln, y='foo'))
   ```

2) When `blit` is set to `True`, blitting is used to determine which parts of the figure change, and then only those parts are updated. For the blitting algorithm to work, the function called to generate the frames and the function set by `init_func` must return an iterable (e.g., a list or tuple) of the artists.
3) `frames` provides the source of the frame count to pass to the animate function to generate each frame. It may be an iterable, int, or generator function. If it is an iterable, then each of the values of it will be passed to the function to generate the complete set of frames. If it is an `int`, then the argument sets the number of frames to be generated; this is equivalent to passing `range(frames)`.
4) `interval` sets the time delay between each frame in milliseconds. By default, it is set to 200. Typically, you will need a smaller number to create a visually smooth animation. Frames per second, fps, is the inverse of `interval`. Therefore, a "cinematic" frame rate of 24 fps equates to an interval of about 42 milliseconds.

For a full list of keyword arguments, see the online documentation of animation.FuncAnimation.

```
# Generate the animation

### UNCOMMENT BELOW LINE if you are running Jupyter on your computer rather
#    than over a web server
#%matplotlib qt

# ------COPIED FROM EARLIER-------
# Initialize the figure, axes, and plot objects
fig, ax = plt.subplots()
ax.set_aspect('equal')  # Make the tick spacing along the x and y axes equal
ax.set_xlim(-0.6, 0.6)
ax.set_ylim(-0.1, 1.1)
```

```
# Define the static plot elements
#    a support, consisting of a horizontal line
ax.axhline(1)
#    a pivot point
pivot = patches.Circle((0,1), 0.01, fc ='DarkSlateGray', ec = 'black',
                       lw = 2 , zorder = 3)  # "zorder = 3" ensures that it will
                                             #    be plotted on top
ax.add_patch(pivot)

# Define the plot elements that will change each frame:
#    a string
line, = ax.plot([], [], lw = 2, color = 'k')
#    a mass at the end of the string
bob = patches.Circle((pnd.x_pos(0), pnd.y_pos(0)), 0.02, fc ='red',
                       ec = 'black', lw = 2 , zorder = 3)
#    a title
title = ax.text(0.15, 0.9, '', horizontalalignment='left',
                  verticalalignment='bottom')

# ------NEW CODE-------
interval = 40  # [milliseconds], interval of time between frames
# Calculate the number of frames needed to show five complete cycles of the
#    pendulum using 5*period/(the interval of times between frames). Since
#    num_frames needs to be an integer, we use int() to round down
num_frames = int(5*pnd.period/(interval*1e-3))
anim = animation.FuncAnimation(fig, animate, init_func = init, blit = True,
                               frames = num_frames, interval = interval)

plt.show()
```

When generating an animation, it is important to consider how long you would like the animation to run and what content you would like to display. For example, here we wanted to show five complete cycles of the pendulum, and we wanted it to run in real time (i.e., one second of the animation running should show one second of pendulum motion). To make the animation run in real time, the time we calculate the pendulum position for should correspond to the frame number times the interval between frames: `ftime = i*interval*1e-3`, as seen in `animate()` (the factor of 1e-3 is because the time interval is measured in milliseconds). We chose the value of `interval` to result in visually smooth motion.

In other situations, you may wish to animate a range of behavior within a certain amount of time. For example, what if you wanted to fit five periods of the pendulum into five seconds? In that case, the maximum time you would want to calculate the pendulum position at would be five times the period (about 10.03 seconds), and you would want to fit it into five seconds of animation, resulting in a speedup of about a factor of two. For a frame interval of 40 milliseconds, we would need 125 frames: (length of animation) = (interval)×(number of frames). Therefore, a frame count of 125 should correspond to 10.03 seconds for the pendulum calculation, requiring us to set `ftime = i*10.03/125` in `animate()`.

For many of the exercises in this text, you will have an array of values generated by your numerical integrator that you would like to animate. In these situations, you can use the frame count as the index of the array, e.g., `x_pos = x_arr[i]`. Frequently, though, you will only want to generate frames for a subsample of the data array. Multiplying the frame count by a constant to get the index enables you to sample the data using a longer cadence, e.g., `x_pos = x_arr[5*i]`. The same timing considerations outlined above must be taken into account here. So, if you want to use 125 frames to represent data from an array of 1000 elements, you would want to sample every eighth point, because 1000/125 = 8.

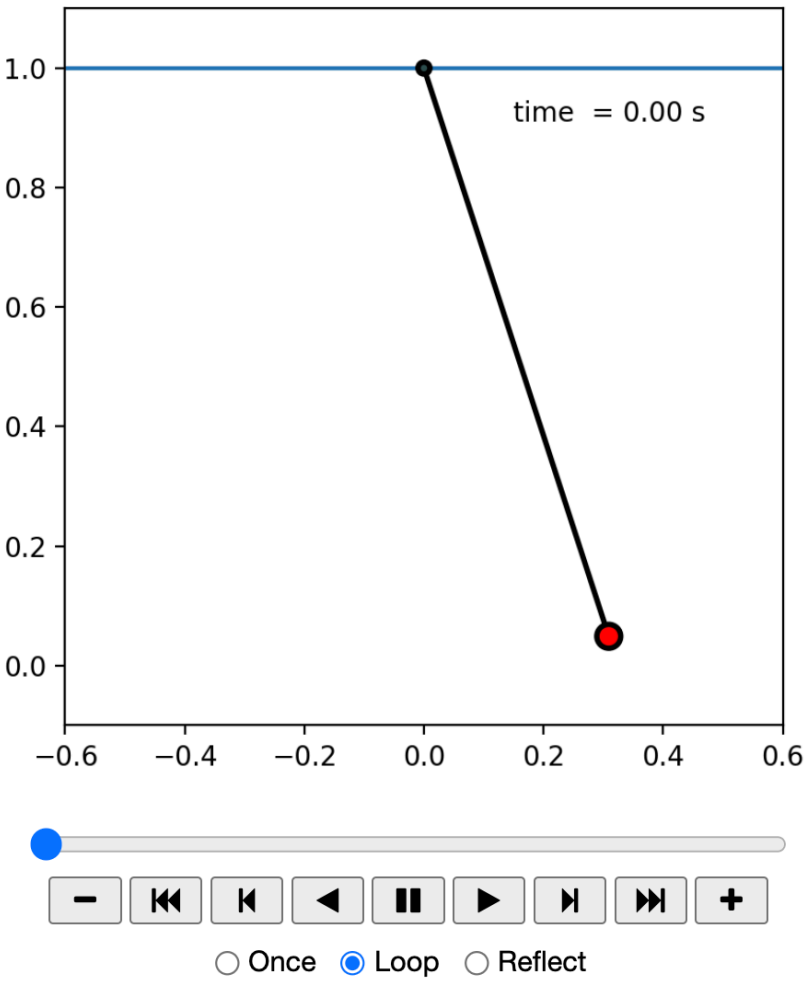
29.3.1 Display and save your animation

Now that we have created an animation, the final step is to display and/or save it. Unfortunately, displaying and saving animations is highly system-dependent. If you experiment, though, you should be able to find a process that works for you using the guidance below.

If you are running Jupyter Notebook on your own computer and if you uncommented `%matplotlib qt` in the code cell above, your animation may have "just worked". This line of code allows a separate window for interactive plotting to be opened, wherein your animation should have appeared. This interactive plotting window gives you the opportunity to save your animation by selecting the floppy disk icon.

In other situations, though, you will want to have the animation display within the notebook. And if you are running on a web server such as Google Colab or CoCalc, you will need another option just to get the animation to show at all. Our preferred option throughout this text is to use `to_jshtml` to generate an HTML representation of the animation and show it using `display`. A similar method is to convert it to an HTML5 <video> tag by using `to_html5_video()`. This method is recommended for users of CoCalc and will also work with Google Colab.

```
# Switch back to inline plotting
%matplotlib inline

# Display as JSHTML
display(HTML(anim.to_jshtml()))
```




```
# FOR GOOGLE COLAB ONLY

### UNCOMMENT the following lines if you are using Google Colab

# Display as html5
#plt.rcParams['animation.html'] = 'html5'
#display(HTML(anim.to_html5_video()))
```

To save your animations, you will need to designate a "writer" that will write the animation into the appropriate file format. Either PillowWriter or ImageMagick can be used to produce GIF files; however, the latter may need to be installed on your computer to be used.

To produce an MP4 formatted video file, the appropriate writer is FFMpegWriter. If FFmpeg is not installed on your computer, you will get an error. Once you have

installed FFmpeg, you may still need to designate the path to ffmpeg.exe by setting `plt.rcParams['animation.ffmpeg_path']`.

In all cases, the number of frames per second can be designated using the keyword argument `fps`.

```
# To write the animation as a GIF, use PillowWriter (shown here) or replace
#    PillowWriter with ImageMagick

writer_video = animation.PillowWriter(fps=1/(interval*1e-3))
ani_ext = '.gif'
```

```
# To write the animation as an MP4, use FFMpegWriter

# CAUTION: The final two commented lines will only work if you have FFmpeg
#    installed

# If you have installed FFmpeg and you are still getting an error, try
#    modifying the following lines to designate the path to the ffmpeg.exe
#plt.rcParams['animation.ffmpeg_path'] =
#    r'C:\\Users\\xx\\Desktop\\ffmpeg\\bin\\ffmpeg.exe'

#writer_video = animation.FFMpegWriter(fps=1/(interval*1e-3))
#ani_ext = '.mp4'
```

```
# Run this cell after designating the appropriate writer to save the file

anim.save(path + 'pendulum_animation' + ani_ext, writer = writer_video)
```

Index