Homework # 1

PHY-905-005
Computational Astrophysics and Astrostatistics
Fall 2023

This assignment is due by 11:59 p.m. on Friday February 24, 2023.

Instructions:

1. Turn in all materials via the GitHub Classroom. Create a separate subdirectory for each part of the homework, named `part_1`, `part_2`, etc., which should contain the code, plots, and answers to questions that are asked in that part (with sensible and descriptive file names for everything).

2. Your code must adhere to the class coding and commenting standards, as documented in the included file `class_coding_standards.txt`, and must pass `Pylint` to the greatest extent possible. Note that adhering to the coding standards, having useful comments, and passing `Pylint` will constitute a significant part of your homework grade.

3. While it's fine to prototype your programs using Jupyter notebooks, **you must turn the homework in as standalone programs in one or more text files** – Jupyter notebooks do not lend themselves to reusable and modular code! That code should run without crashing on the Linux command line using a standard Python distribution (e.g., the most recent Anaconda distribution), and produce the results that are part of your homework assignment. Please include in your writeup instructions on how the code should be executed!

4. Plots should be named sensibly, should have easily readable and logical axis labels and titles, and the source code and data used to generate the plots should be included in the subdirectories.

Part 1: The file `particle_pos_vel.txt` contains the positions and velocities of a rigid body made up of identical, discrete point particles. You are going to calculate the instantaneous spin period of this rigid body. To do this, use the fact that angular momentum for a system of N point particles is:

$$\mathbf{L} = \sum_{i=0}^{N} m_i(\mathbf{r}_i \times \mathbf{v}_i) = \mathbf{I}\omega \tag{1}$$

where m_i is the mass of the particle i, $\mathbf{r}_i$ and $\mathbf{v}_i$ are its position and velocity vectors with regard to the center of mass of the system, ω is the spin vector, and $\mathbf{I}$ is the moment of inertia tensor, which is defined as:

$$\mathbf{I} = \sum_{i=0}^{N} m_i(\mathbf{r}_i \cdot \mathbf{r}_i\mathbf{1} - \mathbf{r}_i \otimes \mathbf{r}_i) \tag{2}$$

where $\mathbf{1}$ is the identity matrix and $\otimes$ is the outer product. Write a program to solve Equation 1 for ω. Don't reinvent the wheel - use numpy for all vector and array functions, and use the SciPy linear algebra module for any linear algebra that needs to be done. Also, note that the spin period about each axis i is defined as $\tau_i = 2\pi/|\omega_\mathbf{i}|$.

1. For the given file, what is the spin period in hours? About which axis is it rotating the **most quickly**?

2. Make an image of this object using the Matplotlib mplot3d toolkit as well as more standard 2D scatter plots. Be sure to include enough viewing angles to get a complete picture (you can do this using the axis `view_init` method in mplot3d). What does this object look like?

Note: Numpy has routines for dot, cross, and outer products, matrix-matrix multiplication, matrix-vector multiplication, and other matrix and vector operations. Similarly, it has routines to create matrices/vectors full of zeros/ones/user-specified numbers and an identity matrix. You should use these to the greatest extent possible in your code! Similarly, look at numpy's `genfromtxt()` method – might that be useful in reading in the data file? (Extra hint: look at the `unpack` argument and think about what that might do...)

Part 2: Write the code for this section of the homwork so that it takes command line arguments using Python's `argparse` module (tutorial here), and uses those to choose between different solvers, set integration tolerances, define different behaviors of the code, etc. Make sure to document this in your comments and in the command line help string!

Imagine a simple solar system composed of a star with a mass of 1 $M_\odot$, a planet on a nearly circular orbit (with ellipticity $e = 0.02$ and semimajor axis $a = 1.0$ au), and a comet on a highly eccentric orbit ($e = 0.95$, $a = 3.0$ au). Assume for simplicity that the star is fixed in place at $x = y = 0$, the masses of the planet and comet are negligible, and at $t = 0$ in this model the planets start their orbits at periastron at:

$$x = 0, y = a(1 - e) \tag{3}$$

and

$$v_x = \sqrt{\frac{GM_*}{a}\frac{1+e}{1-e}}, v_y = 0 \tag{4}$$

You will evolve the system for 5 orbits of the comet ($t_{end} \simeq 26$ years) using your own implementation of four different integrators: Euler's method, the Euler-Cromer method, the midpoint method (i.e., the 2^{nd} order Runge-Kutta method), and the 4^{th} order Runge-Kutta method (implemented in a single function, which switches between them based on a function argument). Do the following:

1. Make a plot that shows the position of the planet and comet from $t = 0$ to $t = 26$ years for a timestep of $\Delta t = 0.01$ years. Do you see any significant differences between the behavior of the planet's orbit and the comet's orbit? Does this behavior differ from what you would expect from an actual physical system, and if so, in what way?

2. Compare conservation of **total energy** and **angular momentum** for the planet and the comet for each of these methods as you vary the timestep from $\Delta t = 10^{-4} - 10^{-1}$ years in powers of 10, plotting the **fractional error** of these quantities at $t = 26$ years on two separate plots – one for energy, one for angular momentum. Recall that fractional error in a quantity Q is $\epsilon = \frac{|Q(t=t_{end}) - Q(t=0)|}{|Q(t=0)|}$. Describe the behavior that you see. Is this what you expected, based on the integration schemes you are using? Why or why not? (Suggestion: think carefully about the type of graph that makes sense for visualizing this information – what should the axes look like?)

3. Finally, implement the adaptive time-stepping algorithm that we discussed in class and in the reading, and allow the planet and comet to take separate time steps (in effect, evolve them separately). If we limit the fractional error per timestep to $\epsilon_{rel} = 10^{-6}$, how many time steps does it take for the four numerical methods described above to advance the system 26 years for each object? Is there a significant difference in this answer between the planet and the comet? If you measure your code's execution time using the Python `time` module, what is the difference in cost between the different integration methods? (Note: you may wish to implement a maximum number of iterations in your code, just in case one or more of the methods cannot reach that level of precision in an acceptable time!) For this problem, what seems to be the best choice for the fastest solution at the desired level of accuracy? Is this different than what we saw in class for the "mass on a spring" problem?

Part 3: The goal for this problem is to write a test suite for the code you wrote in Part 2 that includes unit tests using `pytest`, integration tests that verifies the entire code works as expected, and a performance test that verifies that the code completes in an acceptable amount of time. This test suite should run using a single Python script that calls sub-programs as needed, and then report the outcome of the tests. Note that this might require making some changes to your code from

The unit test suite should test the following:

1. Each numerical integrator in Part 2, if given a fixed timestep and known set of equations with an analytical solution, should correctly advance those equations to within an error tolerance that is appropriate for that algorithm.

2. The numerical integration method should return an error with a useful error message if given an unknown integrator, if $dt \leq 0$, or if dt is small enough that it is likely to cause issues relating to floating-point precision (a number that is likely to vary depending on the integration method).

The integration tests should test the following. If the earth and the comet are integrated for a single orbit using a fixed time step and, separately, an adaptive time step, energy and angular momentum should conserve energy for each integration algorithm to within an acceptable (and algorithm-specific) tolerance of your choice. Demonstrate that if you make the tolerance too small for a given algorithm, this test fails!

Finally, the performance tests should demonstrate that the integration test completes within a user-defined amount of time for each integrator and for the adaptive step size code + RK4. Demonstrate that if you make the fixed time step too small (so it takes too many time steps), this test fails!

Make sure that your test runner presents the outcome of these test suites – in other words, what tests are run, whether they pass or fail, and then any quantitative information (e.g., for solver A test B was passed to within a fractional error of C, with D as the user-defined tolerance). Make sure to print out the total number of tests and the total number of passed tests! (And remember that you are verifying the code works as expected, so if you build a test that is supposed to break the code and it does, this is a "pass").

Part 4: We're going to build on the pre-class assignment you did for Day 6 of class (which focused on open source software development) by working through GitHub's process for dealing with the creation of issues and pull requests, as well as branching, and forking. This is going to require some group work, and so **I have assigned you to groups of three:**

- Group 1: Anita, Sierra, Steven V.
- Group 2: Zilin, Jack, Stephen W.
- Group 3: Elias, Emily, Carolyn

Get together with your group, and do the following things.

1. One member of the group should create a new (empty) git repository on GitHub, and give the other members access to it. Also add your instructor (GitHub username `bwoshea`) to the repository. Put the **broken code** from the Day 5 in-class assignment (on debugging – the file is named `crashing_program.py`) in the repository and push it to GitHub.

2. Create two GitHub issues associated with this repository: one describing the horrific formatting of the repository, and one detailing the two bugs that you found in class. In general it's best to be as informative and factual as possible in an issue as a courtesy to the developer, even pointing out exact lines of code, suggesting solutions, etc.

3. The other group member should clone the repository to their own machines. One of you should make a branch called `cleaned-code`, check out that branch (which you can verify with "`git status`"), run `pylint` on the code in that branch, commit your changes, and push them back to github (don't forget to push the branch – see the GitHub docs for instructions). Another group member should make a

second branch called `fixed-code`, check out that branch, fix the two code bugs from last time, and push those changes and new branch to GitHub as well. Coordinate with your group members to make sure that you've edited at least one of the same lines of code in a different way in each branch – **you want to deliberately create a merge conflict for a later step in this assignment!**

4. Now, each of the two people who created a new branch should issue a pull request (PR) **from their branch to the main branch** (so there will be two pull requests). Make sure to describe what you've done, and **refer to the issue describing the problem by its number** (if you do it correctly it will automatically create a link). Don't do anything with this PR yet!

5. Then, the person/people who did NOT create that pull request should comment on the PR with one or more suggested changes (you can make general comments or comments associated with specific lines of code – try doing both!). The creator of the PR should update the branch on their own machine and push to GitHub, which should update the GitHub PR. The original author of the PR should respond to the comments. If the commentors feel that you have done things correctly, then they can approve the pull request on GitHub (look under the "Files Changed" tab for the "Review changes" button to approve). Once both commentors approve, **merge only one of the branches!**

6. Assuming your second (unmerged) branch also has a line of code edited that was also modified in the merged PR, you will now have a merge conflict. If it's small you can resolve it via the GitHub web interface, but if it's complicated you will have to do it via the command line using a text editor. **Fix the merge conflict, get two approvals on the PR from the other two members of your group, and merge!**

7. Now that your two issues have been address, make sure to close the issues. It's good practice to make a final comment saying something to the effect of "Fixed in PR #1234. Closing issue." This provides a historical record that the issue was actually fixed, and where!

8. You no longer need the two branches that you've created, so delete them both locally on your laptop and on GitHub.

After you are done with this process, create a brief writeup that explains your experiences with this process. What was conceptually the most challenging? What was technically the most challenging?