Problem Set 2

- *Submission format*—Submit your solution via NTULearn, with **one Python source file per problem (`.py` not `.ipynb`)**. Each file should begin with the lines

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

  You may also import other Scipy modules or standard Python modules, but do not use non-standard modules (e.g., stuff from PyPI).

- *Coding assistants*—AI coding assistants may be used if you want, though all assignments can be completed without them. If you use an AI assistant, you must carefully scrutinize, test, and fix up the code it generates. But do not perform verbatim copying of other people's work, including other students taking this course; that is treated as plagiarism.

- *Code quality*—For full marks, the submitted programs must meet these criteria:

  1. Keep all other top-level code, aside from import statements and function/class definitions, to a minimum. Most of your code should live inside functions. Your program should work if we `import` it as a module and call its functions.
  2. If the assignment asks you to write a function, follow the specification exactly. Do not modify the order or definitions of the inputs/outputs to suit yourself. You are free to define additional helper functions, classes, or data structures.
  3. Use comments to document important code blocks. In particular, if you define helper functions or class methods, document their inputs/outputs clearly.
  4. Avoid cryptic function or variable names like `abc`.
  5. Nontrivial numerical constants should be grouped appropriately (e.g., the start of a function), not "hard-coded" deep inside code blocks.
  6. Functions should run appropriate "sanity checks" (e.g., using `assert`) on inputs.
  7. Every generated plot should be properly formatted, with appropriate axis ranges, a proper figure title and axis labels, a legend if there are multiple curves per graph, etc.

0. THREE-BODY PROBLEM (15 MARKS)

In the so-called three-body problem, three objects whose masses are m_0, m_1, and m_2, and whose positions are $\mathbf{r}_0$, $\mathbf{r}_1$, and $\mathbf{r}_2$, interact gravitationally. The equations of motion are

$$\ddot{\mathbf{r}}_0 = \frac{m_1(\mathbf{r}_1 - \mathbf{r}_0)}{|\mathbf{r}_1 - \mathbf{r}_0|^3} + \frac{m_2(\mathbf{r}_2 - \mathbf{r}_0)}{|\mathbf{r}_2 - \mathbf{r}_0|^3} \tag{0}$$

$$\ddot{\mathbf{r}}_1 = \frac{m_0(\mathbf{r}_0 - \mathbf{r}_1)}{|\mathbf{r}_0 - \mathbf{r}_1|^3} + \frac{m_2(\mathbf{r}_2 - \mathbf{r}_1)}{|\mathbf{r}_2 - \mathbf{r}_1|^3} \tag{1}$$

$$\ddot{\mathbf{r}}_2 = \frac{m_0(\mathbf{r}_0 - \mathbf{r}_2)}{|\mathbf{r}_0 - \mathbf{r}_2|^3} + \frac{m_1(\mathbf{r}_1 - \mathbf{r}_2)}{|\mathbf{r}_1 - \mathbf{r}_2|^3}. \tag{2}$$

Here, the gravitational constant is normalized to unity. For simplicity, we will restrict all positions to the 2D plane:

$$\mathbf{r}_n = \begin{pmatrix} x_n \\ y_n \end{pmatrix}, \quad n = 0, 1, 2. \tag{3}$$

The equations of motion constitute six second-order coupled differential equations. These can be re-expressed in first-order form by defining the velocity variables $\mathbf{v}_n = \dot{\mathbf{r}}_n$, which gives twelve first-order coupled differential equations.

(a) (5 marks) Write a function to solve the three-body problem numerically:

`def three_body_integrate(m, r_init, v_init, t):`	
Inputs	
`m`	1D array of masses (m_0, m_1, m_2).
`r_init`	2D array of initial positions, where `r_init[n,:]` $\equiv \big[x_n(t_0), y_n(t_0)\big]$.
`v_init`	2D array of initial velocities, where `v_init[n,:]` $\equiv \big[\dot{x}_n(t_0), \dot{y}_n(t_0)\big]$.
`t`	1D array of times at which to report the solution. The first element, t_0, corresponds to the initial conditions.
Return values	
`r`	3D array of positions, where `r[i,n,:]` $\equiv \big[x_n(t_i), y_n(t_i)\big]$.
`v`	3D array of velocities, where `v[i,n,:]` $\equiv \big[\dot{x}_n(t_i), \dot{y}_n(t_i)\big]$.

To do the integration, use `scipy.integrate.ode` (see the documentation for usage examples). Note that `ode` offers a choice of various integration algorithms; please experiment and see what works best. Discuss your results in code comments.

Optional (ungraded) task: work out and supply the Jacobian to speed up the integrator.

(b) (5 marks) Write a function `three_body_demo()`, showing various three-body trajectories:

- A "hierarchical" system with two orbiting bodies and a smaller faraway third body. Let $m_0 = m_1 = 1$, $m_2 = 0.01$, and for the first two bodies, use

$$x_0 = -x_1 = 1, \quad y_0 = y_1 = 0$$
$$\dot{y}_0 = -\dot{y}_1 = 0.5, \quad \dot{x}_0 = \dot{x}_1 = 0.$$

  Place the last body further away; experiment with different settings and see what gives interesting results.

- A periodic orbit of three equal-mass bodies[1]: let $m_0 = m_1 = m_2$, with

$$x_0 = -x_1 = 0.97000436, \quad y_0 = -x_1 = -0.24308753$$
$$x_2 = y_2 = 0$$
$$\dot{x}_0 = \dot{x}_1 = -\dot{x}_2/2 = 0.46620368$$
$$\dot{y}_0 = \dot{y}_1 = -\dot{y}_2/2 = 0.43236573$$

- One other configuration you deem notable (document it clearly).

Label all plots clearly. In code comments, give some interpretation of the behavior you see.

(c) (5 marks) Fourier analysis is a standard way to analyze the results of numerical simulations. For a particular coordinate, say $x_n(t)$, its Fourier transform $X_n(\omega)$ is

$$X_n(\omega) = \int_{-\infty}^{\infty} dt\, e^{i\omega t}\, x_n(t). \tag{4}$$

Given $x_n(t)$ over a finite interval $t \in [0, T,]$, we can estimate $X_n(\omega)$ via the discrete Fourier transform (DFT), implemented in Scipy by the function `fft`. Note that you may have to transform the inputs/outputs of this function to fit the physics-based definition (4).

Write a function `three_body_spectrum_demo()`, which plots the Fourier power spectra, $|X_n(\omega)|^2$ versus ω, for one of the bodies in the three-body problem. You may choose the exact configurations to study, either from part (b) or something new. Show results for (i) a periodic or almost-periodic orbit, and (ii) a complicated or "chaotic" orbit. Discuss your results in code comments.

1. THE SPLIT-STEP FOURIER METHOD (15 MARKS)

In this problem, you will solve the time-dependent Schrödinger equation using the split-step Fourier method (to be discussed in class). The equation is

$$i\frac{\partial \psi}{\partial t} = \left[-\frac{1}{2}\frac{\partial^2}{\partial x^2} + V(x)\right]\psi(x,t), \tag{5}$$

where we have taken $\hbar = m = 1$. For simplicity, you may assume that V is time-independent.

(a) (4 marks) Write a 1D Schrödinger equation solver, using the split-step Fourier method:

`def schrodinger_1d_integrate(psifun, Vfun, nt, dt, N, L, output_step):`	
Inputs	
`psifun`	Function specifying the initial wavefunction. When called with an array x, it returns an equal-sized array containing the initial wavefunction, $\psi(x, 0)$.
`Vfun`	Function specifying the potential. When called with an array x, it returns an equal-sized array containing $V(x)$.
`nt`	Number of time steps to take (an integer).
`dt`	Length of each time step (a number).
`N`	Number of spatial discretization points, N (an integer).
`L`	Total length of the computational domain (a number). The computational domain will occupy the range $-L/2 \lesssim x \lesssim L/2$.
`output_step`	Number of time steps between each output record (an integer). At every `output_step` time steps, the wavefunction is recorded to `psi` (see below).
Return values	
`x`	1D array of size N, containing the spatial discretization points.
`t`	1D array specifying the times at which the wavefunction was recorded into `psi`. The first element of this array is 0 (the initial time).
`psi`	2D array recording the wavefunction, where `psi[i,j]` $\equiv \psi(x_j, t_i)$. The first row, `psi[0,:]`, is the initial wavefunction specified by `psifun`.

(b) (3 marks) Write a function `schrodinger_1d_demo()`, which plots solutions to the time-dependent 1D Schrödinger equation. The plots should consist of `pcolormesh` pseudocolor plots, with x as the horizontal axis and t as the vertical axis, and the colors showing the probability density $|\psi(x,t)|^2$. Take $L = 20$, and use the harmonic potential

$$V(x) = 20x^2. \tag{6}$$

Show results (in separate subplots) for the following initial conditions: (i) the ground state of the potential (look it up or compute it), and (ii) a "coherent state"

$$\psi(x,0) = (2/\pi)^{1/4}\, e^{-(x-2)^2}. \tag{7}$$

You are free to choose the other parameters appropriately. Discuss your findings in code comments.

(c) (4 marks) Consider the Schrödinger equation in 2D:

$$i\frac{\partial\psi}{\partial t} = \left[-\frac{1}{2}\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) + V(x,y)\right]\psi(x,y,t). \tag{8}$$

Write a split-step Fourier solver for this 2D problem:

`def schrodinger_2d_integrate(psifun, Vfun, nt, dt, N, L, output_step):`	
Inputs	
`psifun`	Function specifying the initial wavefunction. Called with equal-sized arrays x and y, `psifun(x,y)` returns an equal-sized array containing $\psi(x,y,0)$.
`Vfun`	Function specifying the potential. When called with equal-sized arrays x and y, `Vfun(x,y)` returns an equal-sized array containing $V(x,y)$.
`nt`	Number of time steps to take (an integer).
`dt`	Length of each time step (a number).
`N`	Number of spatial discretization points, N, in *each* spatial direction (an integer). Hence, there will be N^2 discretization points in total.
`L`	Total length of the computational domain (a number), taken to be the same in both directions. The computational domain will occupy the range $-L/2 \lesssim x \lesssim L/2$ and $-L/2 \lesssim y \lesssim L/2$.
`output_step`	Number of time steps between each output record (an integer). At every `output_step` time steps of the split-step Fourier algorithm, the wavefunction is saved to the array `psi` (see below).

Return values	
`x`, `y`	2D arrays of size $N \times N$, containing the x and y coordinates of the spatial discretization points.
`t`	1D array specifying the times at which the wavefunction was recorded into `psi`. The first element of this array is 0 (the initial time).
`psi`	3D array recording the values of the wavefunction, such that `psi[i,j,k]` $= \psi(x_k, y_j, t_i)$. Note that `psi[0,:,:]` corresponds to the initial wavefunction as specified by `psifun`.

Hint: in 2D space, the split-step Fourier method's kinetic step consists of two pieces,

$$U_{\mathcal{K}} = U_{\mathcal{K}_x} U_{\mathcal{K}_y}, \tag{9}$$

where $U_{\mathcal{K}_x}$ and $U_{\mathcal{K}_y}$ involve Fourier transforms along x and y respectively. You can use the optional `axis` argument of `fft` and `ifft` to take Fourier transforms along specific axes of a 2D array.

(d) (4 marks) Write a function `schrodinger_2d_demo()`, which shows an animated solution to the time-dependent 2D Schrödinger equation. The animation should consist of `pcolormesh`-generated pseudocolor plots, with x as the horizontal axis, y as the vertical axis, the colors showing the probability density $|\psi(x,t)|^2$, and animated in time t.

You are free to choose the potential and intial conditions. One possibility is a circular well:

$$V(x,y) = \begin{cases} 0, & x^2 + y^2 < 16 \\ 100, & \text{otherwise.} \end{cases} \tag{10}$$

Experiment with the grid size and find a balance between accuracy and speed; with split-step Fourier simulations, coarse grids like 50×50 often work surprisingly well. Discuss your results in code comments.

---

[1] A. Chenciner and R. Montgomery, A remarkable periodic solution of the three-body problem in the case of equal masses, Ann. Math. **152**, 881 (2000).