AST 303 Fall 2022 **Homework #4**

Due Wednesday, 16 Nov 2022

Please indicate on your solutions with whom you worked, what resources you consulted (if any), and what were the contributions of each team member. Your code should be well-documented; include it with your solutions. All plots should be labeled clearly.

1. **What on erf?!** By this time in the semester, you've seen a Gaussian a few times. This one should look no different:

$$f(t;\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{t^2}{2\sigma^2}\right). \tag{1}$$

And you might even remember integrating that Gaussian, at least over the entire real line:

$$\int_{-\infty}^{\infty} \mathrm{d}t\, f(t;\sigma) = 2\int_{0}^{\infty} \mathrm{d}t\, f(t;\sigma) = 1. \tag{2}$$

But have you ever tried to integrate a Gaussian over a *finite* portion of the real line? No? Well actually, you have (if only indirectly)! Remember Assignment 2, Problem 3? If you do, then this beast should look familiar:

$$\int_{-x}^{x} \mathrm{d}t\, f(t;\sigma) = 2\int_{0}^{x} \mathrm{d}t\, f(t;\sigma) \equiv \mathrm{erf}\left(\frac{x}{\sqrt{2\sigma^2}}\right). \tag{3}$$

Let's have some fun with it.

(a) Python knows about erf (the "error function"):

```
from scipy import special
print("erf(1) =",special.erf(1))
```

Matlab knows about erf too: `erf(x)`. Set $\sigma = 1/\sqrt{2}$ in (3) and make a plot of $\mathrm{erf}(x)$ for $x \in [0, 2\sqrt{2}]$. (Handy fact: for a Gaussian, the values less than n standard deviations away from the mean account for a fraction $\mathrm{erf}(n/\sqrt{2})$ of the set.) **[0.25 pt]**

(b) Suppose you weren't so fortunate as to have erf pre-programmed into python and matlab. You'd have to calculate (3) numerically. And that's just what you'll do. Use Monte-Carlo integration with uniform sampling to calculate $\mathrm{erf}(\sqrt{2})$.[1] Plot the error vs. the number of random samples $N \in [1, 10^6]$ used in the integration and show that it converges as $\sim 1/\sqrt{N}$. A reminder of the algorithm: for N samples $x_i \in [a, b]$,

$$\mathcal{I} \equiv \int_a^b \mathrm{d}x\, f(x) \approx \mathcal{I}(N) \equiv (b-a)\langle f\rangle \pm (b-a)\sqrt{\frac{\langle f^2\rangle - \langle f\rangle^2}{N}}, \tag{4}$$

$$\text{where} \quad \langle f\rangle \equiv \frac{1}{N}\sum_{i=0}^{N-1} f(x_i) \quad \text{and} \quad \langle f^2\rangle \equiv \frac{1}{N}\sum_{i=0}^{N-1} f^2(x_i).$$

[1] You need a random number generator; see https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.uniform.html. Also, note that both python and matlab are much faster if you first declare the arrays you're filling before you fill them. For example, in python, doing `x = np.array([0.])` and then `x = np.append(x,1.)` to append the float 1 to the end of the array `([0.])` is slower than if you do `x = np.zeros(2)` and then `x[1] = 1.` But you'll only notice a difference between these two approaches if you're filling an array, say, 10^6 times, as you must do for this part.

Note that the $\pm$ error term here is only an estimate—there is no guarantee that the error is distributed as a Gaussian. Indeed, plot $|\mathcal{I}-\mathcal{I}(N)|/\mathcal{I}$ (i.e., the percent difference between the "true" answer from `special.erf(x)` and your numerical answer) versus N and you'll see that the error term in (4) gets accurate only once $N \gg 1$. (You can only do this comparison if you know the exact answer; but if you did, then you wouldn't be doing numerical integration in the first place! That means that the error estimate in (4) is usually all you have on which to rely.) **[2 pts]**

(c) Doing uniform sampling of a peaked distribution (such as a Gaussian) is incredibly inefficient. Most of the time, you're taking samples from pieces of the integrand that hardly contribute to the integral. There must be a better way, and there is: "importance sampling".

The idea is to write

$$\mathcal{I} = \int_a^b \mathrm{d}x\, g(x)\,\frac{f(x)}{g(x)} = \int_a^b \mathrm{d}G(x)\,\frac{f(x)}{g(x)} = \int_{G(a)}^{G(b)} \mathrm{d}r\,\frac{f(G^{-1}(r))}{g(G^{-1}(r))}$$

with

$$G(x) \equiv \int_a^x \mathrm{d}x'\, g(x')$$

for some wisely chosen function $g(x) > 0$, and then sample not f uniformly but rather f/g nonuniformly, the function g dictating how do to the nonuniform sampling. (Essentially, you're asking what is the expectation value of f/g given the probability distribution g.) How does one choose g? Well, you'd want to sample the integrand with a higher density of points in the region(s) where the integrand is large. One can prove that setting $g = |f|/\int \mathrm{d}x\,|f|$ minimizes the variance (and thus the error for a given N). The issue is that drawing random samples from a $|f|$-distributed probability distribution function is basically what we were trying to do in the first place.

So, instead try the following. We just want something that is close to being proportional to $|f|$ and can be analytically integrated. Set $g(x) \propto \mathrm{e}^{-\sqrt{2}x}$, which decreases by a factor of e^2 from $x = 0$ to $x = \sqrt{2}$ (just as e^{-x^2} does). Normalize $g(x)$:

$$\int_0^{\sqrt{2}} \mathrm{d}x\,\mathrm{e}^{-\sqrt{2}x} = \frac{1}{\sqrt{2}}\bigl(1-\mathrm{e}^{-2}\bigr) \quad\Longrightarrow\quad g(x) = \frac{\sqrt{2}\,\mathrm{e}^{-\sqrt{2}x}}{1-\mathrm{e}^{-2}}.$$

Then

$$G(x) = \int_0^x \mathrm{d}x'\,\frac{\sqrt{2}\,\mathrm{e}^{-\sqrt{2}x'}}{1-\mathrm{e}^{-2}} = \frac{1-\mathrm{e}^{-\sqrt{2}x}}{1-\mathrm{e}^{-2}}.$$

The inverse is readily computed:

$$G^{-1}(r) = -\frac{1}{\sqrt{2}}\ln\bigl[1-r\bigl(1-\mathrm{e}^{-2}\bigr)\bigr]. \tag{5}$$

Now you just draw uniform random numbers $r_i \in [G(0), G(\sqrt{2})] = [0,1]$ from (5) and

use these in

$$\mathcal{I} = \int_{G(a)}^{G(b)} \mathrm{d}r\, \frac{f(G^{-1}(r))}{g(G^{-1}(r))} \approx \mathcal{I}(N) \equiv \left\langle \frac{f}{g} \right\rangle \pm \sqrt{\frac{\langle f^2/g^2\rangle - \langle f/g\rangle^2}{N}}, \tag{6}$$

$$\text{where} \quad \left\langle \frac{f}{g} \right\rangle \equiv \frac{1}{N}\sum_{i=0}^{N-1} \frac{f(r_i)}{g(r_i)} \quad \text{and} \quad \left\langle \frac{f^2}{g^2} \right\rangle \equiv \frac{1}{N}\sum_{i=0}^{N-1} \frac{f^2(r_i)}{g^2(r_i)}.$$

Write a program to do so, and compare with your results from part (b). In particular, how many fewer iterations were required using importance sampling with $g(x) \propto \exp(-\sqrt{2}x)$ to achieve the same error found in part (b)? Neat, huh? **[1.5 pts]**

(d) Next, write a program to compute $\mathrm{erf}(\sqrt{2})$ using both the trapezoidal rule,

$$\int_{x_0}^{x_{N-1}} \mathrm{d}x\, f(x) = \frac{h}{2}\left[f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{N-2}) + f(x_{N-1})\right] + \mathcal{O}(h^3 f''),$$

which uses a piecewise-linear approximation for the function, and the Simpson's rule,

$$\int_{x_0}^{x_{N-1}} \mathrm{d}x\, f(x) = \frac{h}{3}\left[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \ldots \right.$$
$$\left. + 2f(x_{N-3}) + 4f(x_{N-2}) + f(x_{N-1})\right] + \mathcal{O}(h^5 f''''),$$

which uses a piecewise-parabolic approximation for the function. In both formulas, N is the number of points sampled and $h \equiv (b-a)/(N-1)$ is the uniform step size taken between the integration limits a and b. (Note that $N-1$ is the number of divisions.) For Simpson's rule to work in the above form, N must be odd and ≥ 5.

For each approximation, plot the error as a function of $N = 5 \ldots 99$ and compare to the expected convergence rates. What step size h gives you roughly the same accuracy as the Monte-Carlo integration from part (b) after 10^6 uniformly distributed random samples? **[2 pts]**

(e) Now the easy way out. In python, type the following:

```
import numpy as np
import scipy.integrate as integ

def f(x):
    return 2/np.sqrt(np.pi)*np.exp(-x*x)

integ.quad(f,0,np.sqrt(2))
```

Voila. This should return the result plus an estimate of the error. Write down this answer so that we know you did this part (free points!), and then visit https://het.as.utexas.edu/HET/Software/Scipy/generated/scipy.integrate.quad.html. (You can even do `integ.quad(f,0,Inf)` with this function!) **[0.25 pts]**

2. **Simulating a planetary system.** Something that astronomers sometimes do to sanity check their fit of exoplanet data is use an N-body integrator to solve

$$\frac{\mathrm{d}\boldsymbol{r}_i}{\mathrm{d}t} = \boldsymbol{v}_i \quad \text{and} \quad \frac{\mathrm{d}\boldsymbol{v}_i}{\mathrm{d}t} = \sum_{i \neq j}^{N} \frac{G m_j \boldsymbol{r}_{ij}}{r_{ij}^3}, \text{ where } \boldsymbol{r}_{ij} \equiv \boldsymbol{r}_j - \boldsymbol{r}_i, \tag{7}$$

and check whether or not their fit parameters result in a dynamically stable system. In this problem, I'll teach you how to write your own two-body integrator, which we'll use to continue HW03's study of the exoplanetary system HAT-P-21. Start by visiting

http://openexoplanetcatalogue.com/planet/HAT-P-21%20b/

and fetching the planet mass m_p, the orbital eccentricity e, the semi-major axis a, and the host star's mass m_s. Recalling footnote 3 from HW03, you'd be best served by measuring time in yr, length in au, and mass in $\mathrm{M}_\odot = 1047.592421\ \mathrm{M}_\mathrm{J}$, where M_J is the mass of Jupiter. If you do so, then Kepler's third law gives a rather simple gravitational constant $G = (2\pi)^2$.

(a) Pretty much the worst thing you can do is use forward-Euler to solve the two-body problem. You can find some python code that I wrote at

https://www.astro.princeton.edu/~kunz/Site/AST303/euler-Nbody.py

that does forward-Euler for an arbitrary number of bodies.[2] Download the code and insert the values for m_p, e, a, and m_s where indicated. Then run the code and make plots of (i) the orbit in the x-y plane of the center-of-mass frame and (ii) the error in the total energy versus time. (The latter should be a `semilogy` plot. Don't forget to label your axes and specify their units.) Describe these plots in words. **[1.25 pts]**

(b) Perhaps the next best thing to use is second-order Runge-Kutta (RK2). You can find some python code that I wrote at

https://www.astro.princeton.edu/~kunz/Site/AST303/rk2-Nbody.py

that does RK2 for an arbitrary number of bodies. Download the code and insert your values for m_p, e, a, and m_s where indicated. Then run the code and make plots of (i) the orbit in the x-y plane of the center-of-mass frame and (ii) the error in the total energy versus time. (The latter should be a `semilogy` plot. Don't forget to label your axes and specify their units.) Compare your results to those from part (a). **[1.25 pts]**

(c) This is a problem set. So then why am I writing your code for you?! Well, because you are now going to use that code as a template to write your own. Upgrade the RK2 code to a fourth-order Runge-Kutta (RK4) code. Use it to integrate the same two-body problem as in parts (a) and (b)—same timestep h, same total integration time t_f, same masses and initial conditions—and make labelled plots of (i) the orbit in

[2] There are actually more "sophisticated" ways of writing such python code that avoid using for loops (something that python is particularly bad at doing). But I found those methods to actually be slower than using for loops when N is small (like 2 or 3), likely because the overhead of initializing vector operations isn't worth the cost. And there may be ways of writing this code more efficiently, but the way I wrote it is certainly more transparent. (Incidentally, matlab seems to be ≈3 times faster when performing the same exact integration routine... pretty sad for python.)

the x-y plane of the center-of-mass frame and (ii) the error in the total energy versus time. This should look much better than the RK2 solution... in fact, it should look $\sim\mathcal{O}(h^2)$ better. Show this explicitly by comparing $|\mathcal{E}(t_\mathrm{f}) - \mathcal{E}(0)|$ for RK2 and RK4. **[4 pts]**

(d) Write a python routine to integrate the two-body system using the leapfrog method. As explained in class, you need to first obtain one of the variables at the half-step to begin the leapfrog integration. It's your choice which one to offset. If you choose to offset $\boldsymbol{x}_n$ to $\boldsymbol{x}_{n+1/2}$, then the velocity is updated via $\boldsymbol{v}_{n+1} = \boldsymbol{v}_n + \Delta t\, \boldsymbol{a}(\boldsymbol{x}_{n+1/2})$. If you instead choose to offset $\boldsymbol{v}_n$ to $\boldsymbol{v}_{n+1/2}$, then the position is updated via $\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \Delta t\, \boldsymbol{v}_{n+1/2}$. Whichever version you choose, use my RK2 code to obtain the initial value of the offset variable by integrating (7) for a time $h/2$. Once you have this, leapfrog takes over...

Once finished, plot (i) the orbit in the x-y plane of the center-of-mass frame and (ii) the error in the total energy versus time. If you've done it correctly, the error in the energy of the system will not grow in time! But recall the discussion in class: To compute $\mathcal{E}(t)$, the velocities and positions are required at the same time, which is not a natural state of affairs during a leapfrog integration. One way to do this is to update the positions over a time interval $h/2$ first to make it cotemporal with the velocity, then compute $\mathcal{E}$, and then finish the leapfrog by updating the positions over the remaining $h/2$. It's like a leapfrog where one of the people leapfrogging ($\boldsymbol{r}$) pauses mid-air over the back of the other ($\boldsymbol{v}$), performs a calculation ($\mathcal{E}$), and then keeps going. You can achieve this via the drift-kick-drift or kick-drift-kick formulations of the leapfrog method. **[4 pts]**

(e) Re-run your leapfrog integrator but with $h = 10^{-4}$ until $t_\mathrm{f} = 2^{15}h$ (instead of the original $h = 10^{-5}$ and $t_\mathrm{f} = 2^{17}h$). Examine the orbit and the total energy. You'll see that, while the total energy is still conserved, the orbit is precessing. Two-body Keplerian orbits should not precess! What's going on? (Hint: Re-run your RK4 integrator with $h = 10^{-4}$ and $t_\mathrm{f} = 2^{15}h$ and examine the orbit. What's different?) **[0.5 pt]**

Because the code template I gave you works for an arbitrary number of bodies (as should your code), you could play a bit with ($N > 2$)-body systems if you'd like. Here are the initial conditions for a fun system only 15 light-years away:[3]

https://www.astro.princeton.edu/~kunz/Site/AST303/GJ876_initial_conditions.py

Enjoy!

3. **FFT'ing a planetary system.** Now that you have an orbit integrated with constant timestep, you can analyze its time series using an FFT and check that it has the same period P as declared on the exoplanet catalogue website.

(a) Take the time series $v_x(t)$ from Problem 2(d) and FFT it. Use this to plot the power spectrum of the signal versus frequency f. Does it peak near the period $P = 1/f$ that you found on the website? Identify any numerical artifacts or higher-order peaks that

[3] See https://en.wikipedia.org/wiki/Gliese_876 for information about this system. You can also visit https://exoplanetarchive.ipac.caltech.edu/index.html and enter "GJ 876" into the "Explore the Archive" field to see more details and have access to the actual data.

you see and comment on their origin. Also calculate the Nyquist frequency of the data. **[3 pts]**

(b) At the following url you will find heliocentric positions and velocities of the entire solar system, generated from a leapfrog integration run for more than 1000 years:

https://www.astro.princeton.edu/~kunz/Site/AST303/solarsystem/

Load the data into python using the following code:

```
mercury = np.load('solarsystem1.npy')
venus   = np.load('solarsystem2.npy')
earth   = np.load('solarsystem3.npy')
mars    = np.load('solarsystem4.npy')
jupiter = np.load('solarsystem5.npy')
saturn  = np.load('solarsystem6.npy')
uranus  = np.load('solarsystem7.npy')
neptune = np.load('solarsystem8.npy')
```

The data is stored in the order $(t, x, y, z, v_x, v_y, v_z)$, so that `earth[4,:]` contains $v_x(t)$ for Earth. Make separate plots of the orbits of the inner planets and the outer planets, each in the x-y and x-z planes of the heliocentric reference frame. Aren't they beautiful? (If you plot all the orbits together, the plot size required will obscure the inner planets' orbits.) *NOTE!!!* These are data recorded at 262,144 times! I did this to make your FFTs (see below) look nice, but you really **should NOT** turn in a plot of all 262,144 points or else your plots will be several MB each and your printers will gag on the bits.

Take the FFT of each time series $v_x(t)$ for each planet. (Note that 262,144 is a power of 2, which optimizes the FFT.) Then compute the total power spectrum and plot it on a log-log plot as a function of the period `1/f` (label your axes!). If you want, you can improve the looks of things on the low-frequency end by using windowing: e.g.,

```
w = np.hamming(dsize) ; v_earth *= w
```

To ensure you're doing it right, you should see that the Earth has a period equal to 1 yr. Record the peak periods in your power spectrum and see if you got them correct by visiting

https://en.wikipedia.org/wiki/Orbital_period#Examples_of_sidereal_and_synodic_periods

(If you want to have fun and pretend that you're Ptolemy, plot the orbits of the inner solar system in a coordinate system centered instead on the Earth. Then check out https://en.wikipedia.org/wiki/Deferent_and_epicycle if you don't know about epicycles.) **[2 pts]**