Homework 2 (Due 11:59 pm, Friday, October 14, 2022)

This assignment will have two due dates for different components. The full assignment must be submitted by **11:59 pm, Friday, 10/14/2022**. This includes the report submission to Canvas, and your code pushed to your course repository.

You will also provide a submission by **11:59 pm, Wednesday, 10/12/2022**. This is two days prior to the final deadline. Simply `add`, `commit`, and `push` all of your work up to that point. You will submit just the latest commit hash to Canvas. It does not matter how much of the assignment you have completed. This will purely be graded on completion.

Overview and set-up

In this homework, you will numerically evolve a two-body problem forward in time. The particles in this system evolve according to Newton's law of gravitation, and their motion is governed by a pair of coupled ordinary differential equations (ODEs):

$$\begin{aligned} m_1 \frac{d^2 \mathbf{x}_1}{dt^2} &= \mathbf{F}_{1,2} \\ m_2 \frac{d^2 \mathbf{x}_2}{dt^2} &= \mathbf{F}_{2,1} \end{aligned} \tag{18}$$

Here the masses are given by m_1 and m_2, the position vectors are given by $\mathbf{x_1}$ and $\mathbf{x_2}$, and the force vectors are given by $\mathbf{F}_{1,2}$ and $\mathbf{F}_{2,1}$. Note that the forces are equal and opposite, $\mathbf{F}_{1,2} = -\mathbf{F}_{2,1}$. We will simplify our notation by dropping the force subscripts and setting $\mathbf{F} = \mathbf{F}_{1,2}$. We can write the gravitational force as follows:

$$\mathbf{F} = \mathbf{F}_{1,2} = \frac{G m_1 m_2}{||\mathbf{x}_2 - \mathbf{x}_1||^2} \frac{(\mathbf{x}_2 - \mathbf{x}_1)}{||\mathbf{x}_2 - \mathbf{x}_1||} = \frac{G m_1 m_2 (\mathbf{x}_2 - \mathbf{x}_1)}{||\mathbf{x}_2 - \mathbf{x}_1||^3} \tag{19}$$

We can also simplify our lives by taking the gravitational constant to be one, $G = 1$. This is equivalent to making a change of variables, and hence doesn't effect the dynamics in a meaningful way.

It turns out that this system is much easier to solve if we split it into a set of first order equations. We can do this by defining the momentum, $\mathbf{p}_i$, of the i^{th} particle as:

$$\mathbf{p}_i = m_i \frac{d\mathbf{x}_i}{dt} = m_i \mathbf{v}_i \tag{20}$$

With this we can split our previous second order ODE system into this equivalent first order system:

$$\begin{aligned} \frac{d\mathbf{x}_1}{dt} &= \frac{\mathbf{p}_1}{m_1} \\ \frac{d\mathbf{x}_2}{dt} &= \frac{\mathbf{p}_2}{m_2} \\ \frac{d\mathbf{p}_1}{dt} &= \mathbf{F} \\ \frac{d\mathbf{p}_2}{dt} &= -\mathbf{F} \end{aligned} \tag{21}$$

which *could* be solved by hand, but it would not be very fun. We will solve these equations for motion in the plane. This means that each position and momentum vector has 2 components, and this is really a system of **8** equations!

This may seem a little daunting, and might seem like jargon soup. Don't worry though! We'll go through all of the relevant details here.

Discretizing the problem

At present, this is a *continuous* problem which the computer can not solve for us. To make this tractable we will consider only a discrete set of times. The details of this discretization process are important, but we'll delay them for now. We'll consider the discretization of ODEs in more detail in the final projects at the end of the quarter. In this discrete setting we will not consider $t \in \mathbb{R}$, but rather a discrete set of times given by:

$$t^n = n\Delta t \tag{22}$$

where the superscript is just a label (not an exponent), and Δt is the time step size. As a result the positions and momenta of the particles will only be known at these discrete times.

If you are interested in where the update formulas that we'll use come from you can read more on the Verlet integration wikipedia page. We are using the Velocity Verlet method which is particularly clean for this problem. This method is part of a larger class of methods called *symplectic integrators*. These methods are intimately tied to the underlying dynamics of Hamiltonian systems, and get used very frequently.

Solving the problem computationally

Ultimately, our goal is to evolve equation (7), paired with some initial conditions, forward to some final time T. To accomplish this task you will fill in two Fortran 90 source files: `orbits.f90`, and `timestep.f90`. Before proceeding, go to your git repository and create a `hw2` directory with a `code` subdirectory. Then download the following 5 files to the newly created code directory:

- **`Makefile`**
- **`utility.f90`**
- **`timestep.f90`**
- **`orbits.f90`**
- **`plotter.py`**

The `orbits.f90` file will contain your main program. This program will define the problem data, call the time stepper to walk forward in time, and write the numerical solution out to a file.

To aid this process and to keep everything organized, the `timestep.f90` file will hold a subroutine for producing the next state of the system given the current one. It will also hold an internal private function to help calculate intermediate quantities, which helps avoid duplicating code.

The `utility.f90` file currently holds only the `kind` parameter for double precision reals. Later we'll have more items to put in there. The `Makefile` will help you compile everything without needing to call `gfortran` repeatedly. Finally, the file `plotter.py` contains a small plotting routine to visualize the data generated by the Fortran code. You may use this if you already have Python 3 installed, along with NumPy and Matplotlib. If you don't have those then don't worry. You can generate plots using excel easily enough for now.

The `orbits.f90` file

This is the source file that defines the main program. This file is responsible for the following items:

- Defining number of time steps and the final time
  - From these it will set the time step size
- Declaring arrays to hold the solution for all time
- Set the initial conditions for the system
- Evolve the system by calling into the `timestep` module
- Write the solution out to a file

A few of these have already been done for you. The parts you need to fill in are indicated by comments using `!!!`, with numbers matching the steps that follow.

1. **Declaring variables**

   We need variables to store the final time and size of the time step to use, as well as a few arrays to hold the solution to the system. To this end declare the following variables:

   - `tFinal` and `dt` as scalars of type `real`
   - `mass` as a rank 1 `real` array with 2 elements
   - `pos` and `mom` as rank 3 `real` arrays of shape $2 \times 2 \times nSteps$

   After that, set the final time to be $tFinal = 50$ and the time step size to be $dt = tFinal/nSteps$. Make sure that everything is using the `fp` kind parameter imported from the `utility` module.

   We should also discuss the storage convention for the `pos` and `mom` arrays. These are rank 3 arrays, and the index for each rank indicates a different thing.

   Consider the position array. The first index corresponds to the x and y coordinates of the particles, and since we are working in the plane this rank must have a length of 2. The second index labels which particle is which, and since our system consists of 2

particles this rank must also have length 2. The third and final index corresponds to the time step where position data is known. We could read the following indices as such:

- `pos(1,2,10)` is the x position of the second particle at the tenth time step
- `pos(:,1,1)` is the position vector for the first particle at the first time step (note the slicing)

The momentum array is stored similarly. The only difference is that the first index refers to the x and y components of each momentum vector.

2. **Setting initial conditions**

   Navigate to the bottom of the file to find the `set_ics` subroutine. Inside here you will set up the initial configuration of the system. Let's set the first particle to have mass $m_1 = 1$ and the second to have mass $m_2 = 0.01$. We could do this in one line by writing:

   - `mass = (/1.0_fp,0.01_fp/)`

   Next set the first particle to lie at the origin with zero momentum. From our discussion about storage order above, you could set these as:

   - `pos(:,1,1) = (/0.0_fp,0.0_fp/)`
   - `mom(:,1,1) = mass(1)*(/0.0_fp,0.0_fp/)`

   Note: you will later change the momentum of this first particle, so you should go ahead and write it this way even though it's really just a fancy way to multiply by zero for now.

   Finally, set the position of the second particle to be $\mathbf{x_2} = (0, -1)$. Give it an initial velocity of $\mathbf{v_2} = (1, 0)$ recalling that the momentum is $\mathbf{p_2} = m_2\mathbf{v_2}$.

3. **Compile the code and try it out**

   You can now compile and run the code, though it won't do very much yet. Call `make` inside your `code` directory. This should compile without errors and produce `orbits.ex`. Try running the executable by calling `./orbits.ex`. Inspect `sol.dat` and verify that the first line holds your initial conditions.

   Question: What order are these columns in? Can you figure it out from the `write_data` subroutine?

The `timestep.f90` file

This file declares the public subroutine `take_step` and the private function `compute_acceleration`. Your tasks here are to fill in the bodies of each of these.

1. **Computing acceleration**

   The force that each particle exerts on the other depends only on their positions, and not on their momenta. You can see the definition for it in equation (5) above.

Navigate to the bottom of the file and start filling in the `compute_acceleration` function. Note that the input arguments and the output have already been declared. We will break this into three sub-steps:

a. **Declare local variables**

In addition to the input/output variables that have already been defined, we will need a couple local variables to make the calculation cleaner. Declare a `real` scalar called `dist` and a rank 1 `real` array called `force` with a length of 2.

b. **Find distance between particles**

Take note of the shape of the input `pos` variable. This time it is only a rank 2 array, where each rank has a length of 2. Here we are only taking in the position at a fixed time, so the final rank is no longer present.

The first particle has position `pos(:,1)` and the second has position `pos(:,2)`. Calculate the distance between these positions and store it in the `dist` local variable that you declared previously.

c. **Convert force to acceleration**

Go ahead and uncomment the line that has the force calculation on it.

The `take_step` subroutine that we are going to write later needs the acceleration of each particle. Recall that Newton's second law says that the acceleration is related to the force as $\mathbf{F} = m\mathbf{a}$. Since our particles have different masses we need to be a little careful how we fill the `acc` variable.

Fill `acc(:,1)` with the acceleration of the first particle as given by its mass and the force that was found previously. Fill `acc(:,2)` with the acceleration of the second particle. **Be careful** with the direction of the second acceleration!

2. **Updating position and momentum**

This is the moment we've all been waiting for. Navigate to the `take_step` subroutine. Here is where we will figure out what the next state of the system will be given its current state. This time all input and output variables have already been declared, as have all of the necessary local variables. Again we will split this into 3 sub-steps.

a. **Get velocity from momentum**

The position update is simpler if we already know the velocity of the particles. Set `vel(:,1)` to the velocity of the first particle by using `mass(1)` and `mom_old(:,1)`. Do the same for the second particle, but note that because of the different mass you'll need a second line to accomplish this.

b. **Update the position**

We now have the velocity and acceleration of all particles at the current time step. We have also been careful to pull off all of the mass-dependent terms, so we can use the nice array arithmetic of Fortran to accomplish this in one line.

The new position is set by assuming that the velocity and acceleration are constant up to the next time step. Then from regular calculus we find that the new positions are given by:

$$\mathbf{x}_i^{(new)} = \mathbf{x}_i^{(old)} + \Delta t \mathbf{v}_i^{(old)} + \frac{\Delta t^2}{2}\mathbf{a}_i^{(old)} \tag{23}$$

Use the `pos_old`, `vel` and `acc_old` variables to set the new position, and store it in `pos_new`.

c. **Recalculate the acceleration and velocity**

We are halfway done with the time update. So far we have set new positions using the old positions and the old particle velocities. We now need to find updated velocities. We could try to do this trivially by using just the old acceleration value and the time step size, but this turns out to be a bad idea.

Instead, start by calculating `acc_new` by calling the `compute_acceleration` function again, but this time passing in `pos_new`. Then update the velocity like so:

$$\mathbf{v}_i^{(new)} = \mathbf{v}_i^{(old)} + \frac{\Delta t}{2}\left(\mathbf{a}_i^{(old)} + \mathbf{a}_i^{(new)}\right) \tag{24}$$

This is slightly different in that we are using the *average* acceleration between the old and new time steps to update the velocity. This seemingly simple change actually has a pretty drastic impact on the final result.

d. **Compile and run the code**

You have now finished all the hard parts of the code! At this point you should be able to compile without errors (or warnings :0) and run the code to produce more interesting solution files. The next section will help you verify that you are getting the right outputs

How do you know your code is working?

There are several interacting pieces present here. How can you know that your implementation is correct?

For the given initial conditions above you should be able to produce the following figure:

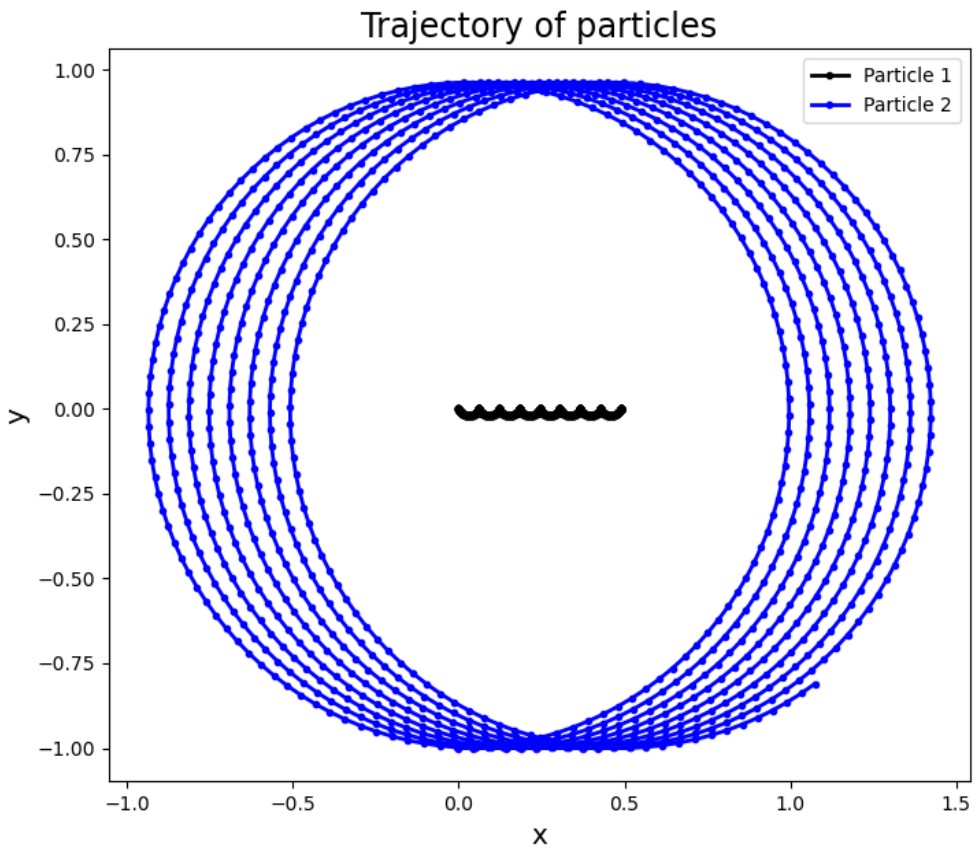

Fig. 22 Particle trajectory for first initial condition

You can create this plot by running the included Python script. This requires that you have Python 3 installed along with NumPy and Matplotlib. If you have those try typing `python plotter.py` in your `code` directory.

If you don't have Python installed there is no need to worry about it right now. You can also generate this plot by importing the `sol.dat` file into a tool like *Microsoft Excel*. The solution is stored column-wise. The first two columns give the x and y positions of the first particle. The third and fourth columns give the x and y positions of the second particle. The black line in the plot above is generated by the first two columns, and the blue line is generated by the third and fourth columns.

Take some time to debug your code until you can produce the above figure.

Commit and push the code

You are not done with the code quite yet, but now is a very good time to bundle up all of your work and do an `add`, `commit`, `push` cycle.

Find the solution for other initial conditions

Now that we have a functioning code let's see what other dynamics we can get out of this system. We will try three new configurations here. Make sure that you generate a plot for each one, and be careful to make any copies of data that you don't want to overwrite.

Also, we are just going to change the initial conditions directly in the code. This means that we need to re-compile each time we change it. This is a little inelegant, but it is simple. We'll remedy this in a future homework.

1. **A more massive second particle**

   As a first test let's try raising the mass of the second particle from $m_2 = 0.01$ to $m_2 = 0.1$. What do you observe about the system?

2. **A zero total momentum system**

   The *total* momentum of the system is conserved. Right now the first particle has zero velocity initially, and the second has non-zero velocity initially. This means that the system as a whole has *total* momentum that is not zero.

   Find a new initial velocity for the first particle so that it has equal and opposite momentum to the second one. Now the system will have zero *total* momentum for all time. How does the solution differ from the previous one?

3. **A system of your choosing**

   Finally, you should experiment with some initial conditions of your own. Find one configuration you find interesting and document it. Perhaps you could look for a binary system that follows a nice pair of trajectories, or you could look for an initial condition that makes the particles escape from each others influence. You can also try adjusting the final time and the number of time steps taken.

   Explore and find something cool!

Deliverables for the report

Your report should discuss briefly your actions in implementing the code. Were there any unexpected issues? What was your debugging strategy when an issue arose?

Your report should comment on the following items:

1. Include the commit hash for your final code submission.
2. Comment on why `real (fp)` can be used throughout the files `orbits.f90` and `timestep.f90`.
3. Comment on the separation of duties between `orbits.f90` and `timestep.f90`. Does the distinction make sense?
4. Right now the code is limited to handling two particles. What would you need to do to increase this to three particles? What about N particles?
   - Don't actually do this! Just comment on what you would do.
5. The code also evolves all of the dynamics in the plane. What would you need to do to raise this to dynamics in full 3D space? Would this be easier or harder than increasing the number of particles?

6. Present each of the three configurations discussed above. Present the initial conditions that give that solution and comment on what you see.
    - For the first and third configurations find the total momentum of the system. Comment on how this relates to the trajectories you observe.
    - If you adjusted the final time and/or number of time steps in the third configuration, be sure to comment on that as well.
7. Inside the `Makefile` flags are specified that control the compilation. What are these flags and what do they do?
8. Write a conclusion to wrap up the whole report.

Note

The report **must** be submitted in PDF format. Non-PDF submissions will not be graded. You should name your report something that clearly identifies you, e.g., `lastFirst_hw2.pdf`.

Your report should be no longer than 4 pages. Please ensure that this limit is maintained upon conversion to PDF from Microsoft Word, or compilation from LaTeX source. The PDF will be turned in on Canvas. In your report, you **must** have a conclusion section where you summarize your findings, and provide mathematical reasoning/justifications relevant to your key findings.