PHYS358: Homework 1 (due Sep 04, 9:30am)

Ordinary Differential Equations I

Note: Please read the complete homework instructions before you start.

This homework consist of three parts. In part (A), the goal is to refamiliarize yourself with Python, and to refresh your knowledge of ordinary differential equations. In Part (B), we discuss the accuracy of fixed-step size integrators, and in Part (C), you'll solve a set of stiff coupled ODEs.

Part (A)

Downloading the homework package: Please find the assignment on Sakai and download the homework package `phys358-homework01.zip`. If you unpack this in your `phys358` directory (which should have been set up during class), you will get a subdirectory `homework/01/post`. Within `post`, you'll find the homework assignment.

(1a) Developing the single-step routines [6 pts]: Within `post`, locate the three programs `lunarlander.py, kepler.py, h2formation.py`, and the file `ode_step.py`. The first three contain the problem-specific initialization and plotting routines, and they call the actual integrators (which you can find in `ode_integrators.py`). `ode_step.py` contains the shells for the stepping functions `euler, rk2, rk4`. Follow the instructions for each function, implementing the corresponding integration update between the lines marked by "?????". For `rk2`, you can use the modified Euler method or the midpoint method.

Solution: See posted solutions. [2 pts] for each correct stepping function.

At this point, you'll have the machinery (i.e. the integrators). What's missing is the RHS, or the derivative functions, for the systems to be modeled. These need to be implemented in the functions `get_dydx()` within the three files `lunarlander.py, kepler.py, h2formation.py`. We'll start with the lunar lander.

(1b) Lunar Lander [10 pts]: A lunar landing module of mass $M_{tot} = M_{ship} + M_{fuel}$ is falling at an initial velocity of $v_z = -5$ m s^{-1} along the vertical (no lateral motions) from a height of 500 m (note that the initial conditions are already set in `ode_init()`). For the moment we assume that the descent engine works with a constant throttle (this will be changed in later homeworks). Thus, the force on the lunar lander is

$$F = T_{max}k - M_{tot}g, \tag{1}$$

where T_{max} is the maximum thrust, $k \in [0, 1]$ is the constant throttle value, and g is the lunar gravitational acceleration. Given this information, **write down** the three 1st order ODEs describing the motion of the lunar lander [3 pts], **and implement** them in `get_dydx()`. As before, the location is marked by "?????". **Add** a fourth ODE for the constant throttle fraction. We'll need that later, when we allow the throttle fraction k to change with time. Don't forget that running the engine will reduce the fuel reservoir. The exhaust velocity at the nozzle is given by V_{nozz} and is assumed to be constant.

Run `lunarlander.py` for the three stepping functions `euler, rk2, rk4` to test your implementation. The program expects an argument for the stepper function. For example,

```
python lunarlander.py euler
```

will run the lunar lander using the Euler step.

Solution: The ODEs are given by

$$\begin{aligned}\dot{z} &= v_z \text{ [1 pt]}\\ \dot{v_z} &= \frac{T_{max}k}{M_{tot}} - g \text{ [2 pts]}\\ \dot{m}_{fuel} &= -\frac{T_{max}k}{V_{nozz}} \text{ [2 pts]},\end{aligned}$$

where the last equation gives the source of the thrust $T_{max}k$ under the assumption of a constant exhaust velocity. [1 pt] per correctly implemented ODE in `get_dydx()`. It is a good idea to set $k = 0$ for $M_{fuel} \leq 0$ [1 pt].

(1c) Kepler Problem [10 pts]: Implement the Kepler problem via direct summation. As for the lunar lander, **write down** the 1st order ODEs. Start out by determining how many ODEs per body you will need, assuming that the motions are restricted to two dimensions. Calculate the gravitational accelerations via direct summation (this is ok for up to a few 10 objects – beyond that, more sophisticated techniques will be required. These will be discussed later in class). I strongly recommend to use a cartesian coordinate system (x, y). Run `kepler.py [euler,rk2,rk4]` to test your implementation. Discuss how the the results (i.e. the orbits) change with different stepper functions. For which one(s) can you get a closed orbit? *Hint: Remember that you can write the updates in* `[euler,rk2,rk4]` *as array operations.*

Solution: The ODEs for body i are given by

$$\begin{aligned}\dot{x}_i &= v_{x,i} \text{ [1 pt]}\\ \dot{y}_i &= v_{y,i} \text{ [1 pt]}\\ \dot{v}_{x,i} &= -Gm_i \sum_{j\neq i} m_j \frac{x_i - x_j}{|\mathbf{r}_i - \mathbf{r}_j|^3} \text{ [2 pt]}\\ \dot{v}_{y,i} &= -Gm_i \sum_{j\neq i} m_j \frac{y_i - y_j}{|\mathbf{r}_i - \mathbf{r}_j|^3} \text{ [2 pt]},\end{aligned}$$

where $\mathbf{r}_i$ is the coordinate vector of body i. [1 pt] per correctly implemented ODE in `get_dydx()`.

(1d) Chemical Reaction Network [12 pts]: The formation of molecular hydrogen H_2 in interstellar gas can be described by a simplified reaction network,

$$\begin{aligned}A + A &\rightarrow B \text{ H}_2 \text{ formation on grains, rate } k_1\\ B + \gamma &\rightarrow 2A \text{ photo-dissociation of H}_2\text{, rate } k_2\\ A + \gamma &\rightarrow C \text{ photo-ionization of H, rate } k_3\\ C + e &\rightarrow A + \gamma \text{ recombination of H}^+ \text{ with free electron, rate } k_4,\end{aligned}$$

where the letters A, B, C stand for atomic hydrogen H, molecular hydrogen H_2, and protons H^+. Assuming charge neutrality, the abundance of electrons e is that of protons H^+. The goal is to find the **reaction rate equations** describing the above system. An elegant way to do this is via the stoichiometry matrix[1]. Assume you have n species S_i, and m chemical reactions. For our case, $n = 3$ and $m = 4$. For each species S_i, **write down** the reaction rate equation. Remember that "addition" of reactants means multiplication, since these are collisional processes. Implement your equations in `get_dydx()`, and run `h2formation.py [euler,rk2,rk4]` to check the results.

Solution: The ODEs are given by

$$\begin{aligned}\dot{A} &= -2k_1A^2 + 2k_2B - k_3A + k_4C^2 \text{ [3 pts]}\\ \dot{B} &= k_1A^2 - k_2B \text{ [3 pts]}\\ \dot{C} &= k_3A - k_4C^2 \text{ [3 pts]}\end{aligned}$$

[1 pt] per correctly implemented ODE in `get_dydx()`.

Part (B)

The goal of Part (B) is to visualize the dependence of the cumulative integration error for the Euler, RK2 and RK4 methods.

(1e) The cumulative integration error: analytical solution [5 pts]: To determine the cumulative integration error, we need a "true" solution. The simplest way is to use an ODE whose exact solution for a given endpoint x_1 is known. We pick

$$y'(x) = \exp(y(x) - 2x) + 2,$$

integrated over the interval $0 \leq x \leq 1$, and with the initial condition $y(0) = -\ln 2$. Find (with pen and paper) the solution to the ODE, and give the value for $y(1)$.

Solution: Define $z(x) \equiv y(x) - 2x$ [1 pt]. Thus, $z'(x) = y'(x) - 2$, and the ODE becomes,

$$z' = \exp(z)$$

The ODE is separable [1 pt], so

$$\exp(-z)dz = dx,$$

yielding

$$\exp(z) = c - x$$

or [1 pt]

$$z(x) = -\ln(c - x).$$

With the above definition of z, and the initial condition $y(0) = -\ln 2$, we get [1 pt]

$$y(x) = -\ln(2 - x) + 2,$$

therefore, $y(1) = 2$ [1 pt]

[1] `http://users.abo.fi/ipetre/advcompmod/Lecture_4.pdf`

(1f) The cumulative integration error: test implementation [10 pts]: The script you'll need to run this homework problem with is called `error_test.py`. It contains the function `get_dydx()` – this is the RHS for our ODE. `error_test.py` is just a shell, importing the necessary libraries. Your task can be split in three steps (see instructions in `error_test.py`):
(i) Define the start and end x, and set the initial condition. Also, decide on a sequence of step numbers N you want to test. Since the stepsize $h \propto 1/N$, the error should drop with increasing N. I recommend a logarithmic scale for N, e.g. $N = 10, 100, 1000, 10000, 100000$.
(ii) Loop over the list of the three stepper functions `fORD = [euler,rk2,rk4]` and over the step numbers, call `fINT` for each combination, and calculate the error. You can check `ode_integrators.py` for the calling sequence of `fINT`. It will return `x,y,it`: the independent variable x, the solution y, and the number of iterations used (which for now is irrelevant, but will be used at a later step). Since we will compare the analytical solution at $y(1)$ to the integration result, the last element of the solution array `y` needs to be compared to $y(1)$. Store the absolute value of the error for each combination of `fORD` and N.
(iii) Plot the logarithm of the error against the logarithm of the step number, for all three stepper functions. Print out the (logarithmic) slopes. Which slopes do you expect for `euler, rk2, rk4`? What do you notice for the slope of `rk4`?

Solution: See solution code [10 pts]. The slopes should be $-1, -2, -4$ [6 pts]. For `rk4`, the error drops down to machine accuracy, hence the oscillations for large step numbers [2 pts]. Plot with axis labels [2 pts].

Part (C)

The goal of part (C) is to develop and test the backward Euler method for integrating a set of coupled stiff ODEs.

(1g) The backward Euler method [21 pts]: Here, we're exploring a little further the solution strategies for stiff ODEs, i.e. coupled ODEs whose components evolve on highly different scales. Typical examples are chemical reaction networks. As an example, we will consider the following pair of ODEs:

$$\begin{aligned} u' &= 998u + 1998v \\ v' &= -999u - 1999v, \end{aligned}$$

with the initial conditions $u(0) = 1$, $v(0) = 0$. These look inoccuous enough, but if you run `coupled_ode.py rk4`, you'll realize that the solution is unstable. To understand the issue with these ODEs, we'll first find the analytical solution. I recommend looking at Numerical Recipes in C, 1992, chapter 16.6 (see 8/30 on Sakai calendar, `04literature.pdf`).
(i) Analytical solution: Show that

$$\begin{aligned} u &= 2e^{-x} - e^{-1000x} \\ v &= -e^{-x} + e^{-1000x} \end{aligned}$$

is a solution to the above ODEs [3 pts], and show that a stepsize $h < 2/1000$ is required for stability [3 pts].

Solution: The simplest way to answer the first part is to plug u and v in the ODEs. Another possibility is to define $u \equiv 2y - z$ and $v \equiv -y + z$ and solve these [3 pts]. The second part about the step size is answered by realizing that the explicit (forward) Euler method for an exponential results in an update $y_{n+1} = y_n(1 - ch)$. Thus, if $h > 2/c$, the solution will diverge [3 pts].

(ii) The backward Euler method for general ODEs: First, implement the backward (implicit) Euler method in the function `backeuler` in file `ode_step.py` [5 pts]. Second, have `coupled_ode.py` plot three plots in one window, namely the solutions $u(x), v(x)$, their residuals (i.e. the difference between the integrated and analytical solutions, and finally, the iteration number [6 pts]. We'll need that in a moment.

Upon completion, you should be able to call `coupled_ode.py backeuler`. *Hint: I recommend to implement the update for non-linear coefficients.* Rerun the problem with `coupled_ode.py rk45` and write down the number of iterations used [1 pts]. How many does `backeuler` use [1 pts]? You can find the information in `ode_init`. What is the advantage of the implicit solution [2 pts]?

Solution: See solution files. `rk45` uses $\approx$ 100 iterations for the very first few steps, while `backeuler` uses 100 steps for the whole solution.