CS 370 Winter 2025: Assignment 2

Instructors: Leili Rafiee Sevyeri (email: leili.rafiee.sevyeri@uwaterloo.ca)
Eugene Zima (email: ezima@uwaterloo.ca)

Due Friday, Feb 14, at 5:00pm EST

Submit **all** components of your solutions (written/analytical work, code/scripts, figures, plots, output, etc.) to Crowdmark in PDF form for each question (multiple pages are allowed). Note that you may resubmit as often as necessary until the due date — only the final submission will be marked.

You must *also* separately submit **a single zip file** containing any and all code/scripts you write to the Assignment 2 DropBox on LEARN, in runnable format (and if necessary, any data needed to reproduce your results).

You have a number of options of how to prepare your solutions. You can typeset your solutions in a word-processing application (MS Word, LaTeX, etc.), you can write on a tablet computer, or you can write on paper and take photos. **It is your responsibility to ensure that your submission is sufficiently legible (and not corrupted upon upload). This is particularly important if you are submitting photos of handwritten pages.** TAs have the right to take marks off for illegible answers.

Note that your solutions will be judged not only for correctness but also for the quality of your presentation and explanations.

1. **(15 marks)** Convergence of ODE Methods

   Consider the initial value problem

$$y'(t) = \frac{y(t)}{20}(10 - y(t)), \quad y(0) = 1/5.$$

   (a) Show that

$$y(t) = \frac{10}{49e^{-t/2} + 1}$$

   is an exact closed-form solution to this particular initial value problem (IVP), *including* the initial conditions. Hint: You do *not* need to analytically *solve* the ODE from scratch; just demonstrate that the proposed solution $y(t)$ satisfies the given ODE and initial conditions.

   (b) Create a single Jupyter notebook for the rest of this question, named `A2Q1.ipynb`. Define a function, `Dynamics(t,y)` that implements the dynamics function for the above problem given input values of `t` and `y`. i.e., it should return the corresponding derivative value $y'(t) = f(t, y)$ for this problem. Define a second function `ExactVal(t)` that returns the exact solution of y for a given time t, as defined in part (a).

   (c) Define a function, `y = ForwardEuler(`t_0`,`t_{final}`,N,`y_0`)`, which takes as input the initial time, t_0, the final time, t_{final}, the number of time steps, `N`, and the initial value, y_0,

and implements the Forward Euler method. It should return the computed *approximate* solution as a vector `y` of length N+1 such that

$$\mathtt{y}_i \approx y(t_i) \quad \text{for} \quad i = 0, 1, 2, ..., N$$

where $t_i = t_0 + ih, h = \frac{t_{final} - t_0}{N}$, and $y(t_i)$ denotes the exact solution at time $t = t_i$. Your `ForwardEuler` function should call the function `Dynamics` (created in part (b)) internally when needed.

(d) Next, apply your `ForwardEuler` function to solve the IVP to time $t_{final} = 20$, first with $N_1 = 5$ constant-size timesteps, and then again with $N_2 = 10$ constant-size time steps, and finally $N_3 = 20$ constant-size time steps. Plot these numerical solutions together on the same graph over the domain $0 \leq t \leq 20$, using the '+' symbol for the first solution, the 'o' symbol for the second solution, and the '*' symbol for the third solution. On the same graph, also plot the exact analytical solution using solid lines. (To create the data points for the plot of the exact solution, evaluate it at the same discrete times as the N_3 solution.)

(e) The *explicit midpoint* time stepping rule (a second order Runge Kutta scheme we saw in class) can be written as:

$$k_1 = f(t_n, y_n), \tag{1}$$
$$k_2 = f(t_n + h/2, y_n + hk_1/2), \tag{2}$$
$$y_{n+1} = y_n + hk_2. \tag{3}$$

Define another function, `y = ExplicitMidpoint(t0,tfinal,N,y0)`, which implements the method given above, similar to what you did in part (c) for Forward Euler.

(f) Next, apply both your `ForwardEuler` and `ExplicitMidpoint` functions to solve the IVP to time $t_{final} = 20$, with $N = 20$ constant-size time steps. Generate a graph plotting these two numerical solutions together over the domain $0 \leq t \leq 20$, using the '+' symbol for the FE solution and the 'o' symbol for the Midpoint solution. Also on the same graph, plot the analytical solution using solid lines. (To create the data points for the plot of the exact solution, evaluate it at the same discrete times as for the FE and Midpoint solutions.)

(g) We would now like to calculate numerical evidence illustrating the order of the Forward Euler method. Let $\text{err}(h)$ denote the absolute (global) error at t_{final} after computing with step size h (recall that this is *not the local truncation error for a single step*). For a method that is p-order accurate, we have $\text{err}(h) \approx Ch^p$ for some constant C. As a result, when we cut the step size in half we would expect to see that

$$\frac{\text{err}(h/2)}{\text{err}(h)} \approx \frac{1}{2^p}$$

i.e., the error is reduced by a factor of 2^p. Therefore, the order p of a given method can be illustrated by calculating the above ratios for decreasing h.

For the Forward Euler method, calculate a vector of successive computed values for the solution at $t_{final} = 10$ (**different than in earlier parts of this question**), by cutting the step size in half a number of times and for each h repeat the entire timestepping

process from start to finish. Specifically, use $N = 5 \times 2^i$ timesteps, for $i = 0, 1, ..., 9$. Your code should calculate the errors observed at this final time for each timestep size, and compute the ratio of errors $\frac{\text{err}(h/2)}{\text{err}(h)}$ for successive timestep choices. Finally, print this information in rows in a simple text table (you can just use the `print` function and the tab character `\t`, no need for any fancy visual formatting). The first column should be the index i, the second column the error, and the third column the error ratio relative to the row above. (The first row will not have an error ratio.) Based on your experimental evidence, what is the order of the Forward Euler method (i.e., value of p)?

(h) Repeat part (g) using your `ExplicitMidpoint` function. What is the order of the explicit midpoint method?

2. **(4 marks)** Higher Order and Systems of ODEs

Consider Newton's equations of motion for a two-body problem specified by:

$$\frac{d^2x}{dt^2}(t) = -\frac{x(t)}{(x(t)^2 + y(t)^2)^{3/2}}; \qquad x(0) = 0.4; \quad \frac{dx}{dt}(0) = 0$$

$$\frac{d^2y}{dt^2}(t) = -\frac{y(t)}{(x(t)^2 + y(t)^2)^{3/2}}; \qquad y(0) = 0; \quad \frac{dy}{dt}(0) = 2\,.$$

As t ranges from 0 to 2π, $(x(t), y(t))$ defines an ellipse. Convert this initial value problem into an equivalent first order system of ODEs which has the form:

$$\frac{d\mathbf{u}}{dt}(t) = F(t, \mathbf{u})\,.$$

3. **(8 marks)** Forward Euler and Improved Euler Methods

Solve the initial value problem by hand calculations

$$\frac{dy(x)}{dx} = x + y, \quad y(0) = 2,$$

using Forward Euler and Improved Euler methods with step size $h = 0.5$. Show your work. Compare your results with the exact solution

$$y(x) = 3\mathrm{e}^x - x - 1,$$

at $x = 0.5$ and 1. Compute the errors of the approximate solutions given by these methods. (You may use a calculator to perform the individual arithmetic steps, but be sure to show how you arrived at your result. You are not allowed to write Python code for this question.) Display your results in a table which shows the solutions given by the ODE methods and their corresponding errors at each time step.

4. **(6 marks)** Local Truncation Error

Consider solving a differential equation $y'(t) = f(t, y(t))$ using the time-stepping method given by

$$y_{n+1} = y_n + \frac{h}{4} f(t_{n-1}, y_{n-1}) + \frac{3h}{4} f(t_{n+1}, y_{n+1}). \tag{4}$$

The timestep is a constant $h = t_{n+1} - t_n = t_n - t_{n-1}$.

(a) (1 mark) State whether this is an explicit or implicit method.

(b) (1 mark) State whether this is a single-step or multi-step method.

(c) (4 marks) Determine the local truncation error for this method in the form

$$LTE = Ch^{\alpha} y^{(\beta)}(t_n) + O(h^{\alpha+1}). \tag{5}$$

That is, determine the values of α, β, and C. (The notation $y^{(\beta)}$ represents the β^{th} derivative of $y(t)$.)

5. **(6 marks)** Stability

Consider solving a differential equation $y'(t) = f(t, y(t))$ using a second-order Runge-Kutta method given by

$$y_{n+1} = y_n + \frac{h}{4}\left(f(t_n, y_n) + 3f\left(t_n + \frac{2h}{3}, y_n + \frac{2h}{3}f(t_n, y_n)\right)\right). \tag{6}$$

Analyze the stability of this method with our usual test equation

$$f(t, y(t)) = -\lambda y(t); \qquad \lambda > 0. \tag{7}$$

Explain whether (and under what conditions) the method is stable.

6. **(12 marks)** Simulating a Basic Mass-Spring System

Introduction. For this question we will use ordinary differential equations to approximate the physics of two projectiles, connected by a massless, stretchy, straight spring connecting them, being thrown in the air subject to the force of gravity. You will solve this problem (*approximately*) using SciPy's numerical tools for ODEs, instead of implementing a time-stepper yourself (such as Forward Euler, midpoint method, etc.) The assignment includes a base Jupyter notebook (`MassSpring.ipynb`) that you should start from. You will need to complete the missing components of that file (indicated by TODO code comments) to generate the correct solution for the trajectories of the two projectiles. In particular, you must implement code for the *dynamics function* for a *first order system of ODEs* describing the motion of the two projectiles. You must also determine (and specify in the code) the vector of *initial conditions*, and modify the code to halt when either of the two projectiles first impacts the ground.

Mathematical model. The fixed Cartesian coordinate system for this problem has a horizontal x-axis and a vertical y-axis. The two projectiles will move both vertically and horizontally in this coordinate system, with their positions given by the coordinates $(x_1(t), y_1(t))$ and $(x_2(t), y_2(t))$. Their corresponding horizontal and vertical velocities are given by $(u_1(t), v_1(t))$ and $(u_2(t), v_2(t))$, respectively. The forces acting to accelerate (or decelerate) the projectiles after being launched are gravity and a simple spring. The mass of the projectiles is m_1 and m_2. Recall that velocity is the first derivative of position, so $u_1(t) = x_1'(t)$ is the x-velocity of the first projectile, and $v_1(t) = y'(t)$ is its y-velocity; likewise, since acceleration is the first derivative of velocity, $u_1'(t)$ is the horizontal acceleration and $v_1'(t)$ is the vertical acceleration for the first projectile. Naturally, the same relationships apply to the second projectile.

The force of gravity introduces an acceleration of g to the vertical component of each projectile's acceleration.

The spring force imparts an equal and opposite force on each projectile that is proportional to the strength of the spring, k, and the difference of the spring's current length l_{cur} from its unstretched "rest length" l_0, in the direction $\mathbf{d}$ of the spring (i.e., the vector pointing from one projectile to the other). (We use the bold notation here to indicate vectors of length 2.) Such a spring will apply forces that pull or push the ends of the spring inward/outward in order to return it to its rest length. Mathematically, we can write the resulting forces as:

$$\mathbf{F}^{12} = -k(l_{cur} - l_0)\mathbf{d} \tag{8}$$

$$\mathbf{F}^{21} = -\mathbf{F}^{12} \tag{9}$$

where the vector $\mathbf{\Delta} = (x_1(t) - x_2(t), y_1(t) - y_2(t))$ is the vector pointing from one projectile to the other, scalar $l_{cur} = \|\mathbf{\Delta}\|$ is the current spring length, and vector $\mathbf{d} = \frac{\mathbf{\Delta}}{l_{cur}}$ is a unit (i.e., length = 1) vector pointing along the spring. The force $\mathbf{F}^{12}$ is the force on projectile 1 due to the spring pulling it towards projectile 2. The force $\mathbf{F}^{21}$ is the force on projectile 2 due to the same spring pulling it towards projectile 1. The above spring equations correspond to what is known as Hooke's Law in physics/mechanics.

Considering the situation described above, the two projectiles' motion can be described by the following system of ordinary differential equations:

$$x_1'(t) = u_1(t) \qquad y_1'(t) = v_1(t) \tag{10}$$

$$x_2'(t) = u_2(t) \qquad y_2'(t) = v_2(t) \tag{11}$$

$$u_1'(t) = \mathbf{F}_x^{12}/m_1 \quad v_1'(t) = g + \mathbf{F}_y^{12}/m_1 \tag{12}$$

$$u_2'(t) = \mathbf{F}_x^{21}/m_2 \quad v_2'(t) = g + \mathbf{F}_y^{21}/m_2 \tag{13}$$

where the x, y subscripts on the force vectors indicate which component of the vector is used. Since the state (positions, velocities) of each projectile affects the other, we must solve this as a single combined system of 8 ordinary differential equations.

Problem Parameters. The parameters you will use for the above system are: the magnitude of gravitational acceleration $g = -9.81$, the projectile masses $m_1 = 1$, $m_2 = 5$, spring rest length $l_0 = 4$, and spring strength $k = 3$. The starting time is $t_0 = 0$. The end time will be $t_{final} = 30$, unless a projectile hits the ground first (see discussion of "events" below.)

Initial Conditions. The initial position of projectile 1 is $(x_1, y_1) = (0, 25)$. The initial position of projectile 2 is $(x_2, y_2) = (5, 25)$. The launch velocity of projectile 1 is $(u_1, v_1) = (2, 20)$. The launch velocity of projectile 2 is $(u_2, v_2) = (-\frac{1}{2}, 30)$.

Instructions. The provided Jupyter notebook `MassSpring.ipynb` contains most of the code necessary to simulate this ODE system. You will need to work out the appropriate dynamics function and initial conditions from the mathematical model described above, and fill in the corresponding sections of the code indicating `TODO HERE`.

The code is (nearly) set up to halt at the instant in time when one of the projectiles hits the ground level at $y = 0$. This is done using the "event" functionality of SciPy's ODE solver. To enable this behavior, you will only need to fill in the function `ground_event`, where indicated, so that it returns the lower of the two projectile's current y values.

As you will see in the code, SciPy's `solve_ivp` function accepts as parameters the dynamics function for the ODE, the timespan of the simulation (start and end times), the initial conditions of the variables, and a few additional parameters, and returns vectors containing the discrete time instants at which the numerical solution has been calculated, and the state of

the numerical solution at those time instants. To better understand the `solve_ivp` function, you can study the example file `ode_suite_demos.ipynb` that is posted on LEARN (under Contents — Resources — Code Samples), and consult `solve_ivp`'s documentation at the following URL:

https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html

Your answer to this question should consist of the following:

- A modified version of `MassSpring.ipynb` with the dynamics function, initial conditions, and `ground_event` function filled in at the appropriate "TODO HERE" locations.
- A static plot of the trajectories produced by running `MassSpring.ipynb` after making the above modifications. (You should not need to modify the visualization/plotting parts of the code at all to produce the figure.)

OPTIONAL: The notebook contains an additional, final code block that can generate and play an animation of the motion of the two projectiles (and the spring connecting them), provided your Jupyter/Python environment is configured to support doing so. (Depending on your environment you may need further changes or debugging to make this block work.) *You are not required to run it* for the purposes of the assignment or marks; however, observing the animation can be interesting, and can provide intuition for how the projectiles are behaving over time, which can occasionally be useful for debugging.