Spring 2023 Numerical Analysis II Course Project

1 Background

In this semester, we have learned a set of numerical methods for solving initial value problems (IVPs) defined by ordinary differential equations (ODEs). The standard form of an IVP with an ODE is given by

$$\begin{cases} \boldsymbol{y}'(t) = \boldsymbol{f}(t, \boldsymbol{y}(t)), \quad t \in [a, b], \\ \boldsymbol{y}(a) = \boldsymbol{\alpha}, \end{cases} \tag{1}$$

where $[a, b]$ is the given time interval, $\boldsymbol{f} : [a, b] \times \mathbb{R}^n \to \mathbb{R}^n$ is the defining function, and $\boldsymbol{\alpha}$ is the initial value. By solving the IVP (1) we meant to find the function $\boldsymbol{y}(t)$ such that it satisfies both the ODE and the initial value.

The ODE in (1) is called the first-order since only the first-order derivative of $\boldsymbol{y}$ is involved. We spent most time working on the case when the dimension of $\boldsymbol{y}$ is $n = 1$, i.e., $\boldsymbol{y}(t) = y(t) \in \mathbb{R}$, but realized that it is trivial to apply all methods we learned to the general case where $n \geq 1$. Moreover, we showed that a high-order ODE can be easily converted to a first-order ODE as in (1). Therefore, the key is to solve the IVP defined by a first-order ODE like (1).

The first method we learned is called the Euler's method. The standard version of the method with constant step size $h > 0$ by partitioning $[a, b]$ into N equal-sized intervals has the following scheme: first compute the step size $h = (b - a)/N$ and set the time mesh points $t_i = a + ih$ for $i = 0, 1, \ldots, N$, and then compute the following scheme

$$\begin{cases} \boldsymbol{w}_0 = \boldsymbol{\alpha}, \\ \boldsymbol{w}_{i+1} = \boldsymbol{w}_i + h\boldsymbol{f}(t_i, \boldsymbol{w}_i), \quad i = 0, 1, \ldots, N - 1. \end{cases} \tag{2}$$

An implementation of the Euler's method (2) using Python programming language is shown in Figure 1. In Figure 1, the inputs required by the Euler's method are the defining function `f`, the left and right end points `a` and `b` of the interval, the initial value `ya`, and the number of segments `N`. The outputs include `t`, which is a Python numpy array containing the values of the $N + 1$ mesh points $t_0, t_1, \ldots, t_N$, and `y`, which is a numpy array of size $n \times (N + 1)$ ($n = 2$ and $N = 10$ in this example) such that its ith column (Python index starts from 0) is the approximation to $\boldsymbol{y}(t_i)$ for $i = 0, 1, \ldots, N$.

We can use the function `euler` defined in Figure 1 to solve an IVP to demonstrate its performance. Consider the IVP (from lecture notes page 78) with defining function

$$\boldsymbol{f}(t, \boldsymbol{y}(t)) = \begin{bmatrix} f_1(t, y_1, y_2) \\ f_2(t, y_1, y_2) \end{bmatrix} = \begin{bmatrix} -4y_1 + 3y_2 - 6 \\ -2.4y_1 + 1.6y_2 + 3.6 \end{bmatrix} \tag{3}$$

and initial value $\boldsymbol{y}(0) = \boldsymbol{0}$. Then we can code this function $\boldsymbol{f}$ and use the function `euler` in Figure 1 to solve the IVP with this $\boldsymbol{f}$ as the defining function. The implementation is given in (we set the time interval to

```python
# import useful packages
import numpy as np

# Euler method
def euler(f, a, b, ya, N):

    n = len(ya)             # dimension of y
    h = (b - a) / N         # step size

    t = a + np.arange(0, N+1) * h    # mesh points t
    y = np.zeros((n, N+1))           # assign space for y (y is an n by N+1 array)

    y[:, 0] = ya    # set initial value (ya is assigned to the first column of y)

    # main iterations
    for i in range(0, N):

        ti = t[i]         # current mesh point ti
        yi = y[:,i]         # current estimate of y(ti)

        # compute y at next time point
        y[:, i+1] = yi + h * f(ti, yi)

    return t, y
```

Figure 1: Python implementation of the Euler's method.

$[0, 1]$ and $N = 10$ here) in Figure 2. In Figure 2, line 8–18 defines the function $\boldsymbol{f}$, and the line 20 computes `t` and `y`.

In the textbook and lecture notes, the values of $\boldsymbol{y}$ at some time points are shown along with the true values, denoted by $\boldsymbol{y}^*$, to demonstrate how good the approximation is (note that this requires the knowledge of $\boldsymbol{y}^*$ which may not be available in practice). This is not an effective way to show results when there are a large number of time points, because nobody would like to read a large table of long numbers. A more concise and informative comparison is done by showing some summarized quantities, such as average and standard deviation of errors at different times. An even better and more common way to show comparisons is using figures. For example, we can plot the result $\boldsymbol{y}(t) = [y_1(t); y_2(t)]$ of the Euler's method and the true solution $\boldsymbol{y}^*(t) = [y_1^*(t); y_2^*(t)]$ as in the left panel of Figure 3. Here $[\cdot;\cdot]$ is the MATLAB syntax which stacks the arguments vertically. The true solution is

$$y_1(t) = -3.375e^{-2t} + 1.875e^{-0.4t} + 1.5, \tag{4a}$$

$$y_2(t) = -2.25e^{-2t} + 2.25e^{-0.4t}. \tag{4b}$$

In the left panel of Figure 3, we use solid blue and green curves to denote $y_1(t)$ and $y_2(t)$ respectively, and dotted blue and green curves for $y_1^*(t)$ and $y_2^*(t)$ respectively.

We can see from Figure 3 that the approximation $\boldsymbol{y}(t)$ obtained by the Euler's method is fairly close to the true solution $\boldsymbol{y}^*(t)$. If we further decrease the step size h, or equivalently increase N, then the approximation is expected to be more accurate. Moreover, the Euler's method has a local truncation error at the order of $O(h)$. The other methods we learned after the Euler's method, such as the midpoint method,

```python
# Test problem
a = 0
b = 1
ya = np.array([0,0])

N = 10 # number of segments of [a, b]

def f(t,y):

    y1 = y[0]
    y2 = y[1]

    dy1 = -4 * y1 + 3 * y2 + 6
    dy2 = -2.4 * y1 + 1.6 * y2 + 3.6

    dy = np.array([dy1, dy2])

    return dy

t, y = euler(f, a, b, ya, N)
```

Figure 2: A Python implementation to use the Euler's method to solve the IVP with the defining function of the ODE given in (3).

the 4th-order Runge-Kutta method, and the predictor-corrector method, are much more accurate than the Euler's method. In such cases, the curves of $\boldsymbol{y}(t)$ is very close to $\boldsymbol{y}^*(t)$ and they would almost overlap. To see their difference, it is recommended to show the absolute error of $\boldsymbol{y}$ from $\boldsymbol{y}^*$, defined by $|\boldsymbol{y}(t) - \boldsymbol{y}^*(t)|$, or the componentwise absolute errors as in the right panel of Figure 3. This is particularly useful when we need to show and compare the accuracy of different methods in a single plot—the lower the curve is, the smaller the error is, and thus the more accurate the method is.

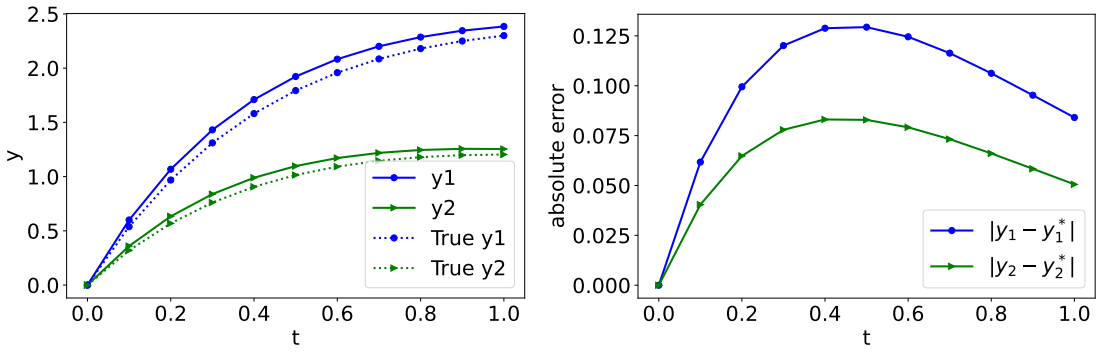

Figure 3: Approximation $\boldsymbol{y}(t) = [y_1(t); y_2(t)]$ obtained by the Euler's method for solving the IVP with defining function given in (3) and initial value $\boldsymbol{y}(0) = \boldsymbol{0}$. The true solution $\boldsymbol{y}^*(t)$ is given by (4). Left plot: $\boldsymbol{y}(t)$ and $\boldsymbol{y}^*(t)$ versus t. Right plot: The absolute errors $|y_1(t) - y_1^*(t)|$ and $|y_2(t) - y_2^*(t)|$ versus t.

It is often overlooked but in fact very important to show the plots clearly in a scientific paper or report. In the attached demo code, you can see an example setting to generate the plots in Figure 3. Some tips and tricks are also shown in the demo code. In addition, when the difference between the curves is very large,

for example, one curve is at 10^{-2} and the other one is at 10^{-7}, it is recommended to use `plt.semilogy` to make the plot which has the y-axis in the logarithmic scale.

2 Project Problems

These project includes three tasks to be completed. The first two tasks are about actual computer programming of several methods we learned during class. You can use either Python or MATLAB to implement these methods. The third task is to complete a written report that includes descriptions of the methods in Tasks 1 and 2, results of numerical experiments, and discussions. Details about these tasks are given below.

2.1 Task 1

The first task of this project is to implement the following methods we learned during class:

1. The Euler's method (this is already done in Section 1)
2. The modified Euler's method.
3. The 2nd-order Runge-Kutta method (also known as the midpoint method).
4. The 4th-order Runge-Kutta method (also known as RK4).
5. The Adams-Bashforth 4th-order explicit method.
6. The Adams 4th-order predictor-corrector method.

You need to implement these methods using Python or MATLAB similar to the Euler's method done in Figure 1. Specifically, you need to implement the following functions:

```
euler(f, a, b, ya, N)
mod_euler(f, a, b, ya, N)
rk2(f, a, b, ya, N)
rk4(f, a, b, ya, N)
adams_explicit4(f, a, b, ya, N)
adams_pc4(f, a, b, ya, N)
```

For each of these functions, the output should be `t` and `y` as in Figure 1. You can write the main body of these functions as you wish, but the function names must be as above, and the format and order of the input `(f, a, b, ya, N)` and the output `(t,y)` must be the same as done in Figure 1. This is because your implemented functions will be directly called to solve some test problems during evaluation, and hence it is important that they are consistent.

If you use Python to implement these functions, then you can write a single Python script named by "`123456789.py`" (replace `123456789` with your own 9-digit Panther ID) to include all these functions. Your script will be called in my evaluation script as

```
import 123456789 as pj

t_mod, y_mod = pj.mod_euler(f, a, b, ya, N)
t_rk2, y_rk2 = pj.rk2(f, a, b, ya, N)
t_rk4, y_rk4 = pj.rk4(f, a, b, ya, N)
t_ae4, y_ae4 = pj.adams_explicit4(f, a, b, ya, N)
t_pc4, y_pc4 = pj.adams_pc4(f, a, b, ya, N)
```

where I will provide the inputs `(f, a, b, ya, N)`. These outputs will be plotted along with the true solution to check whether your functions are implemented correctly.

If you use MATLAB, then you need to write one script for each of these functions (e.g., `rk4`, `adams_pc4`, etc.) and include all of them into a single folder named as "`123456789`" (rename the folder as your 9-digit Panther ID) and zip it as a single file called "`123456789.zip`". The functions will be called in an evaluation script similarly to check correctness.

The exact schemes of the methods mentioned above can be found either in the textbook or the lecture notes. Note that by the Adams 4th-order predictor-corrector method we meant the one given on page 65 of the lecture notes. For both Adams-Bashforth explicit method and Adams predictor-corrector method, use RK4 method to compute the first four steps.

If you would like to test the correctness of your implementation, which is highly recommended, you can use the test problem in Figure 2 and implement a few others shown in the lecture notes or in the exercises of the textbook.

2.2 Task 2

The second task of this project is to implement the Runge-Kutta-Fehlberg (RKF) method (Algorithm 5.3 on page 297 of the textbook) and the Predictor-Corrector method with variable step sizes (Adams-VS) (Algorithm 5.5 on page 318 of the textbook). These methods does not require a fixed number of segments N or step size h. Instead, they can tune the step size automatically and thus requires a prescribed error tolerance `TOL` and the minimum and maximum step sizes `hmin` and `hmax` respectively. Therefore they are also called RK4 and Predictor-Correct methods variable step sizes, respectively. In this task, you should implement these two methods as

```
rkf(f, a, b, ya, TOL, hmin, hmax)
adams_vs(f, a, b, ya, TOL, hmin, hmax)
```

Each of these two methods should return `t, y, h`, where `t` is a numpy array containing the actual time points $t_0, t_1, \ldots$ in order, `y` is a numpy array containing the approximations $\boldsymbol{y}(t_0), \boldsymbol{y}(t_1), \ldots$ as columns, and `h` records the step sizes. See the two algorithms on the textbook for more details. Both of these functions should be included in the same Python script file (or as separate scripts in the same folder as others if you use MATLAB) in Task 1. They will be evaluated in a similar way as in Task 1.

2.3 Task 3

The third and last task of this project is to write a report on your evaluation of the methods implemented in Tasks 1 and 2. The goal of your report is to explain these methods and demonstrate their performance to educated readers who know calculus but have never taken Numerical Analysis. You can decide the structure and content of your written report. The following outline just serves as a reference for you to write your report:

1. Introduction: Provide a brief introduction to IVPs defined by ODEs. Explain that you are trying to implement several methods we learned and test their performance.

2. Methods: Detailed discussion of the numerical methods in Tasks 1 and 2, including their schemes, error orders, number of function evaluations, etc.

3. Numerical Experiments: Conduct some numerical experiments on test problems of your choice, and illustrate the results similar as in Section 1. Describe your observations and provide a discussion on the computational cost and accuracy of these methods.

It is strongly recommended that you write your report using LaTeX, which can produce PDF file with highly complex mathematical notations, figures, tables and so on. If you have never used LaTeXbefore, check out this 30-min tutorial at here.

3 Project Submission and Evaluation

Both of the report and computer implementation done in Tasks 1–3 should be uploaded to iCollege by the due time. For Tasks 1 and 2, you should upload a single Python script (or a single zip file containing all the `.m` functions if you use MATLAB) to the "Project Code" folder under "Assignment" on iCollege. For Task 3, you should upload a single PDF of your report, named as "`123456789.pdf`" (rename it using your own 9-digit Panther ID) to the "Project Report" folder on iCollege.

Your project will be evaluated based on the quality of your report in Task 3 and the correctness of your implementations in Tasks 1 and 2. You will earn up to 10, 5, and 10 points for your work done in Tasks 1, 2, and 3 respectively. The total is 25 points for this project.