Lab assignment #7: Ordinary Differential Equations, Pt. II

Instructor: Nicolas Grisouard (nicolas.grisouard@utoronto.ca)
TA & Marker: Mikhail Schee (mikhail.schee@mail.utoronto.ca)
Partial marking by Pascal Hogan-Lamarre (pascal.hogan.lamarre@mail.utoronto.ca)

Due Friday, October 29th 2020, 11:59 pm

It appears that there is no need for a second room. Should it be incorrect and the room overflows, N.G. will open up the room he had for the previous labs (grab the link on the old question documents, e.g., Lab #3).

Room 1 (also for office hour) Mikhail Schee and Nicolas Grisouard,
URL: `https://gather.town/app/faemi6rJ9YvAgbx5/MS-PHY407-2`
PWD: `phy407-2021-MS`

General Advice

- **Work with a partner!**
- Read this document and do its suggested readings to help with the pre-labs and labs.
- This lab's topics revolve around computing solutions to ODEs, coupled or not, using adaptive-steps Runge-Kutta and Bulirsch-Stoer algorithms for IVPs, and shooting methods for BVPs.
- Ask questions if you don't understand something in this background material: maybe we can explain things better, or maybe there are typos.
- Specific instructions regarding what to hand out are written for each question in the form:

  THIS IS WHAT IS REQUIRED IN THE QUESTION.

  Not all questions require a submission: some are only here to help you. When we do though, we are looking for "C^3" solutions, i.e., solutions that are **C**omplete, **C**lear and **C**oncise.
- An example of **C**larity: make sure to label all of your plot axes and include legends if you have more than one curve on a plot. Use fonts that are large enough. For example, when integrated into your report, the font size on your plots should visually be similar to, or larger than, the font size of the text in your report.
- Whether or not you are asked to hand in pseudocode, you **need** to strategize and pseudocode **before** you start coding. Writing code should be your last step, not your first step. Test your code as you go, **not** when it is finished. The easiest way to test code is with `print()` statements. Print out values that you set or calculate to make sure they are what you think

they are. Practice modularity. It is the concept of breaking up your code into pieces that as independent as possible form each other. That way, if anything goes wrong, you can test each piece independently. One way to practice modularity is to define external functions for repetitive tasks. An external function is a piece of code that looks like this:

```
def MyFunc(argument):
    """A header that  explains the function
    INPUT:
    argument [float] is the angle in rad
    OUTPUT:
    res [float] is twice the argument"""
    res = 2.*argument
    return res
```

Place these functions in a separate file called e.g. functions_labNN.py, and call and use them in your answer files with:

```
import functions_labNN as fl  # make sure file is in same folder
ZehValyou = 4.
ZehDubble = fl.MyFunc(ZehValyou)
```

Computational Background

The shooting method for boundary value problems (and the secant method for finding roots) The shooting method is an application of root finding for boundary value problems. A typical application is to look for correct values of parameters that lead to functions on the domain that match a given set of boundary conditions. You modify your guess until you get one that works. To do so, you typically have to solve a nonlinear equation for a root using methods such as the secant method.

In lecture 4 on finding roots of non-linear equations, we saw that the secant method was a version of Newton's method where you replace the derivative with its finite-difference approximation (compare Newman (6.96) to (6.104)). You use it instead of Newton's method when you do not have an analytic expression for the function you are trying to find the root of, or its derivative.

In Example 8.9 of the textbook (p. 395), the shooting method is used with RK4 to find the ground state energy in a square well potential for the time independent Schrödinger equation. Look over the code (called squarewell.py on Newman's website) and see if you can understand what it's doing: it is integrating a pair of ODEs. The solve(E) function calls the function f(r, x, E), which in turn calls the function V(x). The solution to these equations depends on the parameter E. In the secant method loop, E is adjusted until a root is found for the variable psi, which the physics in the problem requires to be zero. Stated another way, the program does the following:

- Initializes E with reasonable guesses.
- Finds how psi depends on E using solve(E).
- Adjusts E using the secant method.
- Repeats until E has converged to within a target accuracy.

Recursion in the Bulirsch–Stoer method: One way to simplify the Bulirsch–Stoer equation is to make use of recursion. Write a user–defined function called, say `step(r, t, H)` that takes as arguments the vector $\mathbf{r} = (\mathbf{x}, \mathbf{y})$ at time t with an interval length H and returns the vector at time $t+H$. This `step` function should use the 'modified midpoint extrapolation' (section 8.5.5) until the calculation converges to the desired accuracy or you exceed a threshold for the number of steps. If you exceeded the number of steps you should break the problem into smaller steps $H/2$ and try again with two calculations

```
r1 = step(r, t, H/2)
r2 = step(r1, t+H/2, H/2)
```

Correctly written, this function will call itself to sub–divide the domain as needed to improve the accuracy of your solution.

Physics Background

Quantum oscillators We've treated a similar system in Lab #4 (the asymmetric quantum well). This week, you are asked to adapt the shooting code used in Example 8.9 (`squarewell.py` on Newman's website) to a couple of new cases. Instead of an infinite potential barrier, you are solving the problem in a harmonic or anharmonic well. To solve it, Newman suggests in his Exercise 8.14 (p. 397) that you set the boundaries of the problem far from the origin $x = 0$ and that at those boundaries you set ψ to zero. For the harmonic potential, you can find exact solutions at

http://hyperphysics.phy-astr.gsu.edu/hbase/quantum/hosc5.html

to check your numerical solutions. These solutions show that the wave function rapidly decays away from the origin, justifying Newman's suggestion.

Belousov–Zhabotinsky reaction See textbook p. 402 for a description. We just wanted to add that videos of the reaction can be found on YouTube, such as

https://youtu.be/jRQAndvF4sM

created by our very own Prof. Stephen Morris.

Questions

1. **[30%] Space garbage**

   See Newman's Exercise 8.8 (p. 353). I am not asking you to do it, but to build upon the results. Use the results for part (a) as they are given in the textbook. As for the solution to part (b), it is provided in the script `Newman_8-8.py` that you downloaded with the present document.

   (a) Code it up again (or add to the starter code), this time using an adaptive step size approach (you can take the initial h to be $h = 0.01$, which is the one in the script). We wish to achieve a target error-per-second $\delta = 10^{-6}\,\mathrm{m\,s^{-1}}$ for the position (x, y). To do so, plot individual points of the adaptive step size algorithm at each time step, and overlay it on top of the one from `Newman_8-8.py`. However, for the non-adaptive scheme, you now need to use $h = 0.001$ ($N = 10{,}000$ time steps) in order to roughly achieve the same maximum accuracy over the course of the simulation (you can check that it is the case and optimize this number somewhat).

For example `plt.plot(x, y, 'k.')` would plot black points. When calculating ρ, use the formula

$$\rho = \frac{h\delta}{\sqrt{\epsilon_x^2 + \epsilon_y^2}}, \tag{1}$$

which uses the Euclidean error for the position in the (x, y) plane, $\sqrt{\epsilon_x^2 + \epsilon_y^2}$, as described in the text on pp. 359-360 (see the discussion around eqn. 8.54). Do you see the effect of the adaptive time step?

SUBMIT PLOT AND A SHORT DESCRIPTION OF WHAT YOU SEE.

(b) Once you have convinced yourself that the code is working, compare the clock time it takes to the clock time of the solution without adaptive time stepping. Compare the $\delta = 10^{-6}$ solution to the time taken for the non-adaptive time step with $h = 0.001$ ($N = 10,000$).

SUBMIT PRINTED OUTPUT OR A SHORT ANSWER.

(c) Provide a plot of the size of the time step as a function of time. Here's a simple way to do this: if your time values are in the list called `tpoints`, you can plot the time step array as follows

```
dtpoints = array(tpoints[1:]) - array(tpoints[:-1])
plot(tpoints[:-1], dtpoints)  # drop the last point in tpoints
```

Optionally, you can drop the first several points in your plot because it takes a little while for the adaptive routine to settle down into predictable behaviour.

Try to relate the time step size to the solution. Under what circumstances do the time steps tend to be relatively short or relatively long?

HAND IN THE CODE, YOUR PLOT(S), AND WRITTEN ANSWERS.

2. **[30%] Quantum oscillators**

See Newman's Exercise 8.14 (p. 397).

(a) Do part (a).

SUBMIT YOUR CODE, AND ENERGY LEVELS OF THE FIRST 3 STATES.

(b) Do part (b)

SUBMIT ANY NEW CODE (YOU MAY COMBINE EVERYTHING), AND ENERGY LEVELS OF THE FIRST 3 STATES.

(c) Do part (c). For the harmonic oscillator, overlay the analytical solutions to confirm that your calculations are correct (see Physics background; you will need to do a bit of research to make the website's formulas compatible with the parameters we defined). You will need to keep everything legible!

SUBMIT THE PLOTS OF THE WAVEFUNCTIONS FOR BOTH THE HARMONIC AND ANHARMONIC OSCILLATORS.

3. **[40%] Oscillating chemical reactions**

   Do Newman's Exercise 8.18 (p. 402).

   I suggest approaching the problem as follows:

   - Do Q1 in this lab first — or at least make sure you understand how adaptive step size works.
   - Read the problem.
   - Skim 8.5.4.
   - Read 8.5.5 and study Example 8.7 (download the code it implements), and read 8.5.6.
   - Read the hint for Exercise 8.17 about recursion.
   - Come back to the problem and sketch out a solution before starting to code.
   - Note that the constants a and b in the B-Z equations are *not* the constants `a` and `b` in `bulirsch.py` where they represent the initial and final times.

   Finally: when I ran my program, I got overflow errors for my function evaluations, but the program continued to run and produced decent results, so don't worry about these errors.

   SUBMIT YOUR CODE AND PLOT