Lab assignment #4: Solving linear and non-linear equations

Instructor: Nicolas Grisouard (nicolas.grisouard@utoronto.ca)

Due Friday, October 9th 2020, 5 pm

First, try to sign up for room 1. If it is full, sign up for room 2 (and ask your partner to join you there if they are in room 1). If room 2 is full, repeat for room 3. If not, then NG will simply go to whichever room has the most students to help the TA.

Room 1 Pascal Hogan-Lamarre (Marker, pascal.hogan.lamarre@mail.utoronto.ca),
URL: `https://gather.town/k9a7e9j0MjrTEoqw/PHY407-PHL`
PWD: `phy407-2020-phl`

Room 2 Mikhail Schee (mikhail.schee@mail.utoronto.ca),
URL: `https://gather.town/ZdDgdR4j2gGZIqvc/MS-PHY407`
PWD: `phy407-2020-MS`

Room 3 Nicolas Grisouard (nicolas.grisouard@utoronto.ca),
URL: `https://gather.town/SluBqOpSNaWQycc4/phy407-NG`
PWD: `phy407-NG`

General Advice

- Read this document and do its suggested readings to help with the pre-labs and labs.
- Ask questions if you don't understand something in this background material: maybe we can explain things better, or maybe there are typos.
- Carefully check what you are supposed to hand in for grading in the section "Lab Instructions".
- 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 from 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 them in a separate file called e.g. `MyFunctions.py`, and call and use them in your answer files with:

```
import MyFunctions as fl2  # make sure file is in same folder
ZehValyou = 4.
ZehDubble = fl2.MyFunc(ZehValyou)
```

Computational background

Solving Linear Systems (for Q1): In your linear algebra course, you learned how to solve linear systems of the form $\boldsymbol{Ax} = \boldsymbol{v}$ (where $\boldsymbol{A}$ is a matrix and $\boldsymbol{x}$ and $\boldsymbol{v}$ are column vectors) using Gaussian elimination (Newman Section 6.1.1-2). Newman's program `gausselim.py` (p.219) does this.

- I (should have) attached a module `SolveLinear.py` with a supplied function `GaussElim` which can be called as follows

```
# make sure that SolveLinear.py is in the same directory as your program.
from SolveLinear import GaussElim
...
x = GaussElim(A,v)
```

  `GaussElim()` is like Newman's `gausselim.py` but accepts `complex` (as well as `float`) arrays and does not change the input arrays `A` and `v` so you can use them for subsequent work.

- To avoid pitfalls involving divide by 0's, a technique called "partial pivoting" must be implemented. See Newman § 6.1.3 for more background on how to do this.

- The "LU" decomposition of a matrix $\boldsymbol{A}$ is just another way to represent Gaussian elimination. It is used so often that there is a `numpy` function `solve` that implements it:

```
from numpy.linalg import solve
...
x = solve(A, v)
```

  `solve` also works with float or complex arguments.

- Before solving for the eigenvalues and eigenvectors of a matrix, make sure to choose the most efficient `numpy.linalg` method for your particular matrix.

Random arrays Q1b asks you to create random arrays to test the different approaches to solving these systems. You did it in lab #1, here is a slightly different function (any approach works as long as you know precisely how to use the function of your choice). Use the following to guide you:

```
from numpy.random import rand
N = 20  # set N
v = rand(N)
A = rand(N, N)
#e.g. for numpy.linalg.solve
x = solve(A, v)
```

The entries in the arrays `A` and `v` will be filled with random values, uniformly distributed between 0 and 1.

Scipy constants See lab #1 for SciPy's implementation of basic physical constants.

Physics background

A complex circuit Consider the circuit problem illustrated in fig. 1.

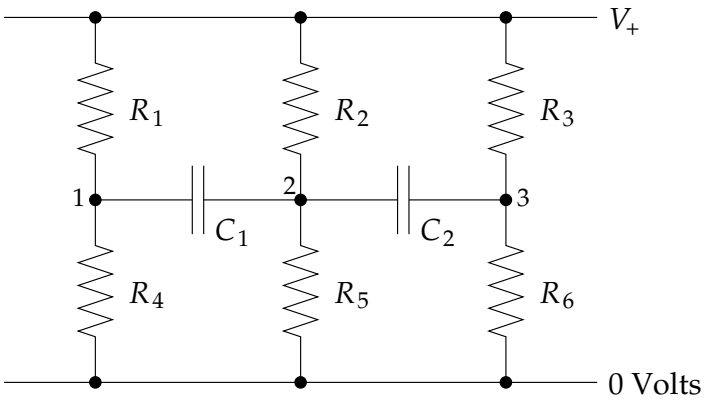

Figure 1: Electric circuit for Q1 (from exercise 6.5 of Newman).

The voltage V_+ is time-varying and sinusoidal of the form $V_+ = x_+ e^{i\omega t}$ with x_+ a constant. The resistors in the circuit can be treated using Ohm's law. For the capacitors the charge Q and voltage V across them are related by the capacitor law $Q = CV$, where C is the capacitance. Differentiating both sides of this expression gives the current I flowing in on one side of the capacitor and out on the other:

$$I = \frac{dQ}{dt} = C\frac{dV}{dt}. \tag{1}$$

Now assume the voltages at the points labeled 1, 2, and 3 are of the form $V_1 = x_1 e^{i\omega t}$, $V_2 = x_2 e^{i\omega t}$, and $V_3 = x_3 e^{i\omega t}$. If you add up the currents using Kirchoff's law that at a junction the sum of the currents in equals the sum of the currents out, you can find that the constants x_1, x_2, and x_3 satisfy the equations

$$\begin{aligned}
\left(\frac{1}{R_1} + \frac{1}{R_4} + i\omega C_1\right) x_1 - i\omega C_1 x_2 &= \frac{x_+}{R_1}, \\
-i\omega C_1 x_1 + \left(\frac{1}{R_2} + \frac{1}{R_5} + i\omega C_1 + i\omega C_2\right) x_2 - i\omega C_2 x_3 &= \frac{x_+}{R_2}, \\
-i\omega C_2 x_2 + \left(\frac{1}{R_3} + \frac{1}{R_6} + i\omega C_2\right) x_3 &= \frac{x_+}{R_3}.
\end{aligned}$$

This is a linear system of equations for three complex numbers, x_1, x_2, and x_3.

To extend the problem further, we will change this *RC* circuit with resistance and capacitance to an *RLC* circuit with resistance, capacitance, and *magnetic inductance*. Suppose we swap resistor R_6 above with an inductor L. The impact on the equations above turns out to be to replace R_6 with a *complex impedance* $\mathrm{i}\omega L$ (this is because the voltage drop across an inductor is given by $L\mathrm{d}I/\mathrm{d}t$, where I is the current). We expect that this will make this part of the circuit less resistive and therefore less dissipative. *RLC* circuits are often set up as *resonant circuits*.

Solutions to Q6.9(a, b): the asymmetric quantum well

(a) Substitute $\psi(x) = \sum_n \psi_n \sin(n\pi x/L)$ into $\hat{H}\psi = E\psi$, multiply by $\sin(m\pi x/L)$ and then integrate over x to obtain

$$\sum_{n=1}^{\infty} \psi_n \int_0^1 \sin\frac{m\pi x}{L} \hat{H} \sin\frac{n\pi x}{L} dx = E \sum_{n=1}^{\infty} \psi_n \int_0^1 \sin\frac{m\pi x}{L} \sin\frac{n\pi x}{L} dx \tag{2}$$

$$= \frac{1}{2} LE \sum_{n=1}^{\infty} \psi_n \delta_{mn} = \frac{1}{2} LE\psi_m \tag{3}$$

With the definition of H_{mn} we see that

$$\frac{1}{2} L \sum_n H_{mn}\psi_n = \frac{1}{2} LE\psi_m \Rightarrow \sum_n H_{mn}\psi_n = E\psi_m, \tag{4}$$

or, equivalently,

$$\mathbf{H}\psi = E\psi, \tag{5}$$

where $\mathbf{H}$ is the matrix with elements H_{mn}.

(b) Splitting the integral into two terms and evaluating using the orthogonality rule given in the textbook, one finds

$$H_{mn} = \begin{cases} 0 & \text{if } m \neq n \text{ and both even/odd,} \\ -\dfrac{8amn}{\pi^2(m^2-n^2)^2} & \text{if } m \neq n \text{ one even, one odd, and} \\ \dfrac{1}{2}a + \dfrac{\pi^2\hbar^2 m^2}{2ML^2} & \text{if } m = n. \end{cases} \tag{6}$$

The matrix is real and it is symmetric because if we interchange m and n we get the same expression for the off-diagonal elements ($m \neq n$).

Questions

1. **[35%] Solving linear systems**

   (a) (See Exercise 6.2 of Newman) Modify the module `SolveLinear.py` to make a function that incorporates partial pivoting. You can call this `PartialPivot(A, v)` if you'd like. Check that it gives answer (6.16) to Equation (6.2).

NOTHING TO SUBMIT.

Hint: You are going to have to flip 2 rows at the same time. The easiest way to do this is with the `copy` *command. You will have to import this command from numpy. Here is an example of how to use the copy command to flip rows* `i` *and* `j` *of a matrix:*

```
A[i, :], A[j, :] = copy(A[j, :]), copy(A[i, :])
```

Also, don't forget you also have to flip the corresponding elements of the `v` *array.*

(b) We will now test the accuracy and timing of the Gaussian elimination, partial pivoting, and LU decomposition approaches (which are all mathematically equivalent). Create a program that does the following:

- For each of a range of values of `N` creates a random matrix `A` and a random array `v`.
- Find the solution `x` for the *same* `A` and `v` for the three different methods (Gaussian elimination, partial pivoting, LU decomposition).
- Measures the time it takes to solve for `x` using each method.
- For each case, checks the answer by comparing `v_sol = dot(A, x)` to the original input array `v`, using `numpy.dot`. The check can be carried out by calculating the mean of the absolute value of the differences between the arrays, which can be written `err = mean(abs(v-v_sol))`.
- Stores the timings and errors and plots them for each method.

Implement this code for values of `N` in the range of 5 to a few hundred. You will find that the differences between the methods are large enough that they are hard to put on the same plot, so it is a good idea to plot the timings and errors on a logarithmic scale.

How do the accuracies of the methods compare? How do the times taken by each method compare?

Note that because the arrays are random you will get somewhat different answers each time. You don't need to analyze this aspect.

SUBMIT PLOTS AND EXPLANATORY NOTES (NO NEED TO SUBMIT CODE).

(c) (See Newman Exercise 6.5b on p. 230) Write a program to solve for x_1, x_2, and x_3 from the Physics Background about the electric circuit, with

$$\begin{aligned} R_1 &= R_3 = R_5 = 1\,\text{k}\Omega, \\ R_2 &= R_4 = R_6 = 2\,\text{k}\Omega, \\ C_1 &= 1\,\mu\text{F}, \qquad C_2 = 0.5\,\mu\text{F}, \\ x_+ &= 3\,\text{V}, \qquad \omega = 1000\ \text{rad.s}^{-1}. \end{aligned}$$

Use your partial pivoting routine from Q1a-b. Notice that the matrix for this problem has complex elements. You will need to define a complex array to hold it, but your routine should be able to work with real or complex arguments. Using your solution, have your program calculate and print the amplitudes of the three voltages $|V_1|$, $|V_2|$, and $|V_3|$ and their phases (i.e. the phases of the coefficients x_1, x_2, x_3) at $t = 0$ in degrees.

Hint: the built-in `abs()` will compute the magnitude, and `numpy.angle()` will compute the phase of a complex number. Look up their manual entries.

Next, plot the (real) voltages V_1, V_2, and V_3 as a function of time for one or two periods of the oscillations.

Finally, replace the resistor R_6 with an inductor L. For this example, set $R_6 = 2\,\text{k}\Omega$, $\omega = 1000$ rad.s^{-1} and $L = R_6/\omega = 2\,\text{H}$, where H ("Henry") is the MKS unit of inductance. Again, print and calculate the amplitudes and phases of V_1, V_2, and V_3, and plot V_1, V_2 and V_3 as a function of time. Briefly describe the impact of replacing the resistor R_6 by the imaginary impedance of the same magnitude, iR_6.

HAND IN CODE, PRINTED OUTPUT, PLOTS, AND A BRIEF COMMENT ON THE COMPARISON BETWEEN THE TWO CASES.

2. **[30%] Asymmetric quantum well**. Complete Exercise 6.9 parts (c, d, e). Parts (a) and the beginning of part (b) are done for you in the "Physics background" section above. Beware units in this problem, make sure they are compatible. You will need to write the program mentioned at the end of part (b) to complete (c, d, e).

SUBMIT PRINTED OUTPUTS, PLOTS, EXPLANATORY NOTES FOR ALL PARTS WHENEVER RELEVANT, AND CODE FOR PART (E).

Hints:

- *For part (c) the text discusses the issue about the Python indices vs the algebraic expression indices. Make sure that your m and n values start from 1 but that they get stored starting from 0. Below is a possible way to do this.*

```
for m in range(1, mmax+1):
    for n in range(1, nmax+1):
        H[m-1, n-1] = Hmatrix(m, n)
```

  where `mmax` and `nmax` have been defined previously and `Hmatrix` is a function that evaluates the elements of the matrix.

- *Part (d) shouldn't take a lot of programming, just change your matrix size.*

- *Part (e): Start from your program written for part (c) and/or (d), but now calculate both the eigenvalues and eigenvectors. After you have calculated these, you can write some code to plot your wave functions. The normalization of your wave function won't work automatically (i.e. $\int_0^L |\psi(x)|^2 dx \neq 1$ in general). This is due to the fact that eigenvectors have arbitrary magnitude. You can always multiply an eigenvector by a constant, and it is still an eigenvector. So, this means that after you find your wave functions, you should calculate $A = \int_0^L |\psi(x)|^2 dx$ (perhaps using one of your integration functions from labs 2 and/or 3). Then you can divide your wave function by $\sqrt{A}$ in order for the normalization to be correct.*

- *Part (e) again: it might be confusing to understand if the eigenvectors are the lines or the columns of the array. Try both, remember what the shape of the well is, and on which side of it the particles are more likely to find themselves.*

3. **[35%] Solving non-linear equations**

(a) Complete exercise 6.10 parts (a, b).

HAND IN YOUR PLOT.

(b) Complete exercise 6.11 parts (b, c, d).

FOR PART (C), HAND IN YOUR CODE AND A BRIEF DISCUSSION OF THE COMPARISON OF HOW MANY ITERATIONS IT TOOK TO CONVERGE IN PART (B) COMPARED TO PART (C). FOR PART (D): HAND IN YOUR SHORT ANSWER.

(c) Complete exercise 6.13 parts (b, c).

You don't need to do part (a) but you will need the results given in that part.

In addition to the binary search method, also try the relaxation and Newton's methods in part (b). Count the number of iterations each method takes and then comment on their relative efficiencies.

HAND IN ANALYSIS OF THE DIFFERENT METHODS (PRINTOUT OR A PLOT OR TWO, AND WRITTEN ANSWERS) AND THE CODE.

Hints:

- *You **do not** want the obvious root at $x = 0$, you want the other root.*
- *For the binary search method, you will need to find two initial x values that bracket the root. To do this, I recommend plotting the function and estimating some good starting values from there.*
- *For the comparison between the 3 methods, you should start at the same initial x value (so choose one of the initial values from the binary search method and use it as your starting value for the relaxation and Newton's methods). You may want to try different initial values, some very far from the root to get a sense of how efficient the methods are at finding the root.*
- *For Newton's method, you will need to analytically calculate the derivative before implementing it.*
- *Part (c) should involve just adding a few lines to your code from the previous question to solve for the temperature.*