PHY407 Computational Lab 4
Solving Linear, Nonlinear, and Eigenvalue Systems

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. Paul wrote 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. In Q1b you will be asked to implement this technique. See Newman section 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.

- **Some tricks: random arrays, complex numbers, and plotting on a logarithmic scale (for Q1b-c).**

  1. Q1b asks you to create random arrays to test the different approaches to solving these systems. Use the following to guide you:

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

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

2. In Q1b it is suggested you plot quantities using a logarithmic scale for the y axis. Here is some example code:

```
from pylab import yscale, log
plot(N_Array, error_GaussElim ...)
yscale('log')
```

3. To solve Q1c, you will need to load up an array with complex values. You can do this using `complex(x,y)` or using `x+1j*y`.

```
x,y = [2.0,3.0]
#one form
z1 = complex(x,y)
#another form which gives the same result
z2 = x+1j*y
```

When you calculate complex voltages in Q1c, to plot the voltage corresponding to, for example, $V_1 e^{i\omega t}$ you can use something like the following (in your problem `V1` is a complex number you solve for):

```
from numpy import linspace, exp, pi
from pylab import plot
t = linspace(0,2*pi/omega,101) #cycle through one period
e_i_omega_t = exp(1j*omega*t) #length is 101, e^{i omega t}
V1 = 3+2j
V1_complex = V1*e_i_omega_t
#get real part for plotting
V1_signal = V1_complex.real #do you understand this step?
plot(t, V1_signal) #plot, etc.
```

- **Finding roots and extrema of functions (Q2).** In this lab you will try three methods for solving nonlinear systems: relaxation methods, Newton's method, and the bisection method. The text discusses the various pros and cons of each of the methods. Newton's method is probably the most commonly used but it's good to know about alternatives.

  Finding maxima and minima of functions is really just another "finding roots" problem. It's just that you are finding roots of the derivative of your function. However, it's not always practical to use Newton's method for this. The golden ratio method is intuitive although a bit complicated to code.

- **Solving Eigenvalue/Eigenvector Systems (Q3).** Since solving eigenvalue/eigenvector problems is so common in Physics there are a set of linear algebra routines specifically written for most programming languages (like the ones in `numpy.linalg`). For this lab you will use `eig` or `eigh` to solve these systems.

Physics Background

- **Circuit problems (Q1c):** Exercise 6.5 shows a circuit consisting of resistors and capacitors that is being driven by a periodic voltage $V_+ = x_+ e^{i\omega t}$. You won't need to solve Exercise 6.5a, but, just so you know, here's how to do it: the current through each resistor is $I = V/R$ and through each capacitor $I = CdV/dt$. To get the equations you add up the currents using Kirchoff's law: at a junction the sum of the currents in equals the sum of the currents out.

- **The Ising model and statistical physics (Q2).** The Ising model provides an elementary model of ferromagnetism arising from the collective behaviour of interacting particles with spin. Under the simplifying approximation of mean field theory, a simple relationship arises between sample magnetization m and temperature T (see Newman 6.89):

$$m = \tanh\left(\frac{m}{T}\right).$$

  This is an implicit relation for m which is difficult to solve analytically but can be solved numerically. The solution (see Newman Fig. 6.2) shows that below a critical temperature of $T = T_c = 1$ magnetization increases from zero and approaches 1 as $T \to 0$. Below the critical temperature, T satisfies *critical behaviour* of the form $m \sim (T_c - T)^\beta$, where β is known as a critical exponent. This means that $\log m = \beta \log(T_c - T) + const.$ - a relationship we will ask you to investigate. Such behaviour is common in second order phase transitions of the kind taking place in this model.

- **Time independent Schrodinger equation solutions (Q2 and Q3)** In Q2b, you will be asked to solve for the energy levels (eigenvalues) of the square finite quantum well. This requires solving a nonlinear equation using a binary search method.

  In Q3, you will be asked to find the eigenstates of an asymmetric quantum well. We now outline the solutions to Q6.9(a,b):

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

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

  With the definition of H_{mn} we see that

$$\frac{1}{2} L \sum_n^{\infty} H_{mn}\psi_n = \frac{1}{2} LE\psi_m \Rightarrow \sum_n^{\infty} H_{mn}\psi_n = E\psi_m$$

  or equivalently:

$$\mathbf{H}\psi = E\psi$$

  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:

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

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

Lab Instructions

For grading purposes, only hand in solutions to the following parts. Ensure that your codes are well commented and readable by an outsider:

- Q1a: **Don't hand in.**
- Q1b: Code, plot, written answers.
- Q1c: Plots and written answers.
- Q2a: Code, plot, written answers.
- Q2b: Code, numerical answers (six roots).
- Q2c: Code and plots for Part a, numerical answer for Part b.
- Q3: For Exercise 6.9c: Hand in your code and a printout of the first 10 eigenvalues. For Exercise 6.9d: Hand in your short answer. For Exercise 6.9e: Hand in your code and your plot. Make sure to label the different curves on the graph.

Lab Questions

1. **Solving linear systems (40% of lab mark):**

   (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). *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 array, 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 does the time 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.

(c) Do Exercise 6.5b on pp.230-231. Do not hand in the solution to 6.5a. Then extend the exercise by plotting the (real) voltages V_1, V_2, and V_3 as a function of time for one or two periods of the oscillations.

2. **Finding roots and extremizing functions.**

(a) **Newton and relaxation methods for finding roots:** Example 6.3 on pp.257-258, using the code `ferromag.py`, solves for the magnetization of the Ising model according to mean-field theory. It solves a transcendantal equation using the relaxation method to find the equation's roots, which gives the magnetization m as a function of temperature T using relaxation. Do the following:

- Extend the code to also solve for m using Newton's method.
- Plot $m(T)$ to verify that you are using both methods correctly.
- Then plot the number of iterations required for each method, for each value of temperature, starting from the initial magnetization of `m1=1.0` as in the sample code and aiming for a target accuracy of 10^{-6} in each case. *Comment: For the relaxation method, the error is given by* (6.83) *but for Newton's method the error is given by* (6.101)*. Make sure you understand what each equation means. Set* `accuracy = 1e-6` *for both cases.*
- As we approach the critical temperature $T_c = 1$ from below, the magnetization can be shown to vary as a powerlaw: $m \sim (T_c - T)^\beta$.Verify, by plotting your solution as a function of $T_c - T$ for $T < T_c$ that β is about 1/2.

(b) **Binary search method for finding roots.** Do Newman 6.14b on p.268. Newman 6.14a is very helpful in solving this problem; we have plotted the functions in the script `Lab04-Q2a.py` to show you where to start looking for the roots.

(c) **Golden ratio search method for finding a maximum.** Complete exercise 6.18 parts (a, b). Feel free to use your previous functions for gaussian quadrature from Lab 2 for the integral. *Hint: The example program buckingham.py on page 284 provides a template for the golden ratio algorithm which you can use. HOWEVER, it is looking for a minimum of a function, so make sure to adapt it properly to your scenario, i.e. min of $f(x)$ = max $(-f(x))$. Your maximum temperature should be close to 7000K. The answer to part (c) is that this is clearly not practical since the melting temperature of tungsten, glass and many other things is much lower than this.*

3. Complete Exercise 6.9 parts (c,d,e). Parts (a) and (b) are done for you in the "Physics Background" section above. Beware units in this problem, make sure they are compatible.

   Hints:

   - For part (c) the text discusses the issue about the python indices vs the algebraic expression indices. What you will want to do is make sure that your m and n values start from 1 but that they get stored starting from 0. Here is 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 mmax and nmax values.

   - Part (e): Start from your program in question 2, 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. $\int_0^L |\psi(x)|^2 dx$ will not automatically = 1.). This is due to the fact that eigenvectors have arbitrary magnitude. You can always multiply an eigenvector by a constant and its 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 lab 2). Then you can divide your wave function by $\sqrt{A}$ in order for the normalization to be correct.