PHY407-Lab04

Due Oct 7 2022

Computational Background

Solving Linear Systems 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.

- The attached module `SolveLinear.py` includes 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.

Flipping rows in a matrix This is a common task when solving linear systems; the easiest way to do this is with the `copy` command from numpy. For example, to flip rows `i` and `j` of a matrix `A`:

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

Random arrays There are various ways to create random arrays. You can use any method as long as it works properly, here's a good one:

```
from numpy.random import rand
N = 20  # set N
v = rand(N)
A = rand(N, N)
```

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

Physics Background

Asymmetric quantum well Continuing last week's fun with quantum mechanics: see Exercise 6.9 in the textbook. The solutions to part (a), and some of part (b), are as follows.

- 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

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

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

  With the definition of H_{mn} we see that

$$\frac{1}{2} L \sum_n H_{mn}\psi_n = \frac{1}{2} LE\psi_m \Rightarrow \sum_n H_{mn}\psi_n = E\psi_m, \tag{3}$$

  or, equivalently,

$$\mathbf{H}\psi = E\psi, \tag{4}$$

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

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

$$H_{mn} = \begin{cases} 0 & \text{if } m \neq n \text{ and both even/odd,} \\ -\dfrac{8amn}{\pi^2(m^2-n^2)^2} & \text{if } m \neq n \text{ one even, one odd, and} \\ \dfrac{1}{2}a + \dfrac{\pi^2\hbar^2 m^2}{2ML^2} & \text{if } m = n. \end{cases} \tag{5}$$

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. **[20%] 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). *Hint: You are going to have to flip 2 rows of the matrix, and also flip the corresponding elements of the vector.*

   Submit just the printout for the check. The PartialPivot code itself will be incorporated in your submission for the next part.

   (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`.
   - Finds 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 `vsol = 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-vsol))`.
   - 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: because the arrays are random you will get somewhat different answers each time you run the program. You don't need to analyze this aspect.

   Submit plots, written answers, pseudocode, and code.

2. **[40%] Asymmetric quantum well**. Exercise 6.9 in the textbook. Beware units throughout this problem, make sure they are compatible!

   (a) The guinea piggies have done this part for you in the "Physics background" section above.

   Nothing to submit.

(b) The guinea piggies have done the first half of the work for you, in the "Physics background" section. Do the rest, by writing the program (to evaluate the expression for H_{mn}) specified in the second half of this part.

Nothing to submit, since the code will be incorporated into the subsequent parts.

(c) When you're doing this part, beware of the issue mentioned in the textbook 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.

Submit your code and printed outputs.

(d) This part shouldn't take a lot of programming, just change your matrix size.

Submit your printed outputs and brief written answer.

(e) For this part, start from your program written for the previous parts, but now calculate both the eigenvalues and eigenvectors. After you have calculated these, you can write some code to plot your wave functions.

Hints:

- *The normalization of your wave function won't work automatically (i.e. in general, $\int_0^L |\psi(x)|^2 dx \neq 1$). 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 a previous lab). Then you can divide your wave function by $\sqrt{A}$ in order for the normalization to be correct.*
- *It might be confusing to understand if the eigenvectors are the lines or the columns of the array. You can try both, then figure out which is correct by considering the shape of the well (and on which side of it the particles are more likely to find themselves).*

Submit your graph, pseudocode, and code.

3. [**40%**] **Solving non-linear systems** Refer to the textbook for the following exercises.

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

Submit your pseudocode, code, and plot.

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

Submit printout for part (b), printouts and pseudocode and code for part (c), a brief discussion of the comparison of how many iterations it

took to converge in part (b) compared to part (c), and written answer for part (d).

(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.

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. There are different ways to do this; one good possibility is plotting the function and estimating some good initial 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 also want to try different initial values for the relaxation and Newton's methods, some very far from the root, to get a sense of how sensitive to the initial value the performance of these methods are.*
- *For Newton's method, you will need to analytically calculate the derivative before implementing it.*

Submit written answers, pseudocode, and code.