Lab assignment #9: Partial Differential Equations, Part. 2

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

Due Friday, November 20th 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).

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

Room NN Ahmed Rayyan (a.rayyan@mail.utoronto.ca),
URL: `https://gather.town/mIAKeWKElOnF4uL3/PHY407-AR`
PWD: `phy407ar`

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 PDEs using basic methods.
- 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 are 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)
```

Physics background

Time-dependent Schrödinger equation Solving the time-*independent* Schrödinger equation ($\mathbf{H}\psi = E\psi$, with $\mathbf{H}$ the Hamiltonian of the system) gives the eigenstates of quantum system. However, a quantum system can evolve over time, in which case we turn to the time-*dependent* Schrödinger equation

$$i\hbar\frac{\partial\psi}{\partial t} = \mathbf{H}\psi. \tag{1}$$

In Q1, we will focus on the one-dimensional equation for a particle of mass m in a potential $V(x)$, in which case eqn. (1) reduces to

$$i\hbar\frac{\partial\psi}{\partial t} = -\frac{\hbar}{2m}\frac{\partial^2\psi}{\partial x^2} + V\psi, \tag{2}$$

We can solve the equation above if we know the potential $V(x)$ and the initial wave function $\psi(x, t = 0)$. The probability to find the particle at a time t and location x is $\psi^*\psi$, where the asterisk denotes complex conjugation, which has to satisfy

$$\int_{-\infty}^{\infty}\psi^*\psi\,\mathrm{d}x = 1. \tag{3}$$

Note that if the condition above is satisfied initially, a good numerical integration should maintain normalization. Once we know ψ, the "position" and energy of the particle at a given instant t are respectively given by

$$\langle X\rangle(t) = \int_{-\infty}^{\infty}\psi^*(x,t)x\psi(x,t)\mathrm{d}x \quad\text{and}\quad E(t) = \int_{-\infty}^{\infty}\psi^*(x,t)\mathbf{H}\psi(x,t)\mathrm{d}x. \tag{4}$$

As for V, we will be considering three different potentials in this lab. Within a potential of width L, i.e., between $-L/2$ and $+L/2$, the three cases will be

$$\text{Square well:} \quad V(x) = 0, \tag{5a}$$

$$\text{Harmonic oscillator:} \quad V(x) = \frac{1}{2}m\omega^2 x^2, \tag{5b}$$

$$\text{Double well:} \quad V(x) = V_0\left(\frac{x^2}{x_1^2} - 1\right)^2. \tag{5c}$$

Outside of $(-L/2, L/2)$, the potential V will be infinite.

In this lab, you will be able to check results related to the Ehrenfest theorem. I will however focus on one consequence of this theorem, which has a certain regime of validity, and that you should be able to confirm in your results[1], namely,

> "[f]or general systems, if the wave function is highly concentrated around a point x_0, then $V'(\langle x\rangle)$ [*where V' is the gradient of the potential*] and $\langle V'(x)\rangle$ will be almost the same, since both will be approximately equal to $V'(x_0)$. In that case, the expected position and expected momentum will approximately follow the classical trajectories, at least for as long as the wave function remains localized in position."

Electric and magnetic field in 2D resonant cavity: We can write the electric and magnetic fields within the cavity we will investigate in Q2 as

$$\vec{E} = \begin{pmatrix} 0 \\ 0 \\ E_z(x,y,t) \end{pmatrix} \quad \text{and} \quad \vec{B} = \begin{pmatrix} B_x(x,y,t) \\ B_y(x,y,t) \\ 0 \end{pmatrix}, \quad \text{respectively.} \tag{6}$$

It follows from Maxwell's equations that

$$\frac{\partial H_x}{\partial t} + c\frac{\partial E_z}{\partial y} = 0 \tag{7a}$$

$$\frac{\partial H_y}{\partial t} + c\frac{\partial E_z}{\partial x} = 0 \tag{7b}$$

$$\frac{\partial E_z}{\partial t} + c\frac{\partial H_y}{\partial x} + c\frac{\partial H_x}{\partial y} = J_z \tag{7c}$$

where c is the speed of light, $H_x = cB_x$, $H_y = -cB_y$, $J_z = -\mu_0 c^2 j_z$ is the electric current density, and μ_0 is the vacuum permeability. The above system of equations takes the form of three coupled advection equations with a source term. The boundary conditions are that the tangential electric field and the normal magnetic field must be zero at the conducting walls, explicitly stated as

1. $E_z = 0$ at all the walls (which are located at $x = 0, L_x$ and $y = 0, L_y$),
2. $H_x = \partial_x H_y = 0$ at $x = 0, L_x$, and
3. $H_y = \partial_y H_x$ at $y = 0, L_y$.

[1] lifted from https://en.wikipedia.org/wiki/Ehrenfest_theorem

Finally, the normalized current pattern associated with the (m, n) mode takes the form

$$J_z(x, y, t) = J_0 \sin\left(\frac{m\pi x}{L_x}\right) \sin\left(\frac{n\pi y}{L_y}\right) \sin(\omega t). \tag{8}$$

We will solve the Maxwell equations (7) spectrally. To do so, contrary to the Schrödinger equation problem, our first step will be to first define a Crank-Nicolson temporal procedure, and then we will design the handling of the spatial derivatives with Fourier modes. That is, we will decompose each quantity we are solving for as series of sine or cosine functions, each of which satisfying the boundary conditions for said quantity.

But first, let us discretize time as $t_n = n\tau$, with τ the duration of a time step and $n \in \{1, \dots N-1\}$, and space as $(x_p, y_q) = (pa_x, qa_y)$, with a_x, a_y the sizes of a grid cell in the x- and y-directions, respectively, and $(p, q) \in \{1, \dots P-1\}^2$. The Crank-Nicolson temporal differentiation of eqns. (7) yields

$$\frac{(H_x)^{n+1}_{p,q} - (H_x)^n_{p,q}}{\tau} + \frac{c}{2}\left(\frac{\partial E_z}{\partial y}\right)^{n+1}_{p,q} + \frac{c}{2}\left(\frac{\partial E_z}{\partial y}\right)^{n}_{p,q} = 0, \tag{9a}$$

$$\frac{(H_y)^{n+1}_{p,q} - (H_y)^n_{p,q}}{\tau} + \frac{c}{2}\left(\frac{\partial E_z}{\partial x}\right)^{n+1}_{p,q} + \frac{c}{2}\left(\frac{\partial E_z}{\partial x}\right)^{n}_{p,q} = 0, \tag{9b}$$

$$\frac{(E_z)^{n+1}_{p,q} - (E_z)^n_{p,q}}{\tau} + \frac{c}{2}\left(\frac{\partial H_y}{\partial x}\right)^{n+1}_{p,q} + \frac{c}{2}\left(\frac{\partial H_y}{\partial x}\right)^{n}_{p,q} + \frac{c}{2}\left(\frac{\partial H_x}{\partial y}\right)^{n+1}_{p,q} + \frac{c}{2}\left(\frac{\partial H_x}{\partial y}\right)^{n}_{p,q} = 0, \tag{9c}$$

where $(H_x)^n_{p,q} = H_x(x_p, y_q, t_n)$, etc.

In space, we adopt a Fourier approach, and to satisfy the boundary conditions, we compute the discrete sine transforms (DST) and discrete cosine transforms (DCT) or the quantities, as relevant to satisfy the boundary conditions, namely,

$$(E_z)^n_{p,q} = \sum_{q'=0}^{P}\sum_{p'=0}^{P} \hat{E}^n_{p',q'} \sin\left(\frac{pp'\pi}{P}\right)\sin\left(\frac{qq'\pi}{P}\right), \tag{10a}$$

$$(H_x)^n_{p,q} = \sum_{q'=0}^{P}\sum_{p'=0}^{P} \hat{X}^n_{p',q'} \sin\left(\frac{pp'\pi}{P}\right)\cos\left(\frac{qq'\pi}{P}\right), \tag{10b}$$

$$(H_y)^n_{p,q} = \sum_{q'=0}^{P}\sum_{p'=0}^{P} \hat{Y}^n_{p',q'} \cos\left(\frac{pp'\pi}{P}\right)\sin\left(\frac{qq'\pi}{P}\right), \quad \text{and} \tag{10c}$$

$$(J_z)^n_{p,q} = \sum_{q'=0}^{P}\sum_{p'=0}^{P} \hat{J}^n_{p',q'} \sin\left(\frac{pp'\pi}{P}\right)\sin\left(\frac{qq'\pi}{P}\right). \tag{10d}$$

Plugging these equations in the Crank-Nicolson scheme (eqns. 9) and projecting on the basis functions sin and cos as relevant, we ultimately end up with the relations

$$\hat{E}^{n+1}_{p,q} = \frac{(1 - p^2D_x^2 - q^2D_y^2)\hat{E}^n_{p,q} + 2qD_y\hat{X}^n_{p,q} + 2pD_x\hat{Y}^n_{p,q} + \tau\hat{J}^n_{p,q}}{1 + p^2D_x^2 + q^2D_y^2} \tag{11a}$$

$$\hat{X}^{n+1}_{p,q} = \hat{X}^n_{p,q} - qD_y(\hat{E}^{n+1}_{p,q} + \hat{E}^n_{p,q}) \tag{11b}$$

$$\hat{Y}^{n+1}_{p,q} = \hat{Y}^n_{p,q} - pD_x(\hat{E}^{n+1}_{p,q} + \hat{E}^n_{p,q}) \tag{11c}$$

where $D_x = \pi c\tau/(2L_x)$, $D_y = \pi c\tau/(2L_y)$, and $L_x, L_y = Pa_x, Pa_y$, respectively.

Computational background

Animations in Matplotlib return WARNING: creating animations is as illuminating as it is time-consuming. I (N.G.) believe that your experience will be much more enriching if you use them, but we are not requiring you to hand in any.

Last week, I proposed a very simple way to animate Matplotlib pictures. Now that we have it down, let's try a somewhat more elaborate method, based on the `matplotlib.animation.FuncAnimation` function, with the addition of saving the result in `.mp4` format.

See this method in the script `animation_demo.py`, itself based off of one of the basic examples you will find at `https://matplotlib.org/3.3.1/api/animation_api.html`. In that script, I animate the same curve as last lab's example, i.e., $\sin(x-t)$. It works on my computer (a Mac; I am fairly confident it works on Linux, but not sure about Windows). If it does not work on yours, please let me know, and I will try to amend this section and the corresponding script.

Spatial discretisation of the time-dependent Schrödinger equation As we saw in class, to solve for PDEs, we first discretise in space, followed by a discretisation in time. So, first, we write that $x = x_p = pa - L/2$, with $p \in \{1, \dots, P-1\}$, P the number of cells in the x direction, and a the step size (in other words, $L = aP$). Note that because the potentials have infinite walls, $\psi = 0$ at both ends of the domain, which is why we start the count at $p = 1$ and end it at $p = P - 1$. We can then approximate eqn. (2) as

$$i\hbar\frac{\partial \Psi}{\partial t} = \mathbf{H}_D \Psi, \tag{12}$$

where $\Psi(t)$ is an array containing all discretized values of ψ, i.e.,

$$\Psi(t) = \begin{pmatrix} \psi_1(t) \\ \vdots \\ \psi_p(t) \\ \vdots \\ \psi_{P-1}(t) \end{pmatrix} \quad \text{with} \quad \psi_p(t) = \psi(pa - L/2, t), \tag{13}$$

and where $\mathbf{H}_D$ is the discretized Hamiltonian,

$$\mathbf{H}_D = \begin{pmatrix} B_1 & A & 0 & \cdots & & & & \\ A & B_2 & A & 0 & \cdots & & & \\ 0 & \ddots & \ddots & \ddots & & & & \\ \vdots & 0 & A & B_{p-1} & A & 0 & \cdots & \\ & \cdots & 0 & A & B_p & A & 0 & \cdots \\ & & & & & \ddots & \ddots & \end{pmatrix}, \tag{14}$$

with $A = -\hbar^2/(2ma^2)$ and $B_p = V(pa - L/2) - 2A$.

Crank-Nicolson time stepper for the time-dependent Schrödinger equation Picking up where we left off, we can discretize Ψ in time, with $\Psi^n = \Psi(n\tau)$, where $n \in \{1 \dots N-1\}$, N the number of time steps, and τ is the time step (in other words, the final time of the integration is τN). Recall that the Crank-Nicolson scheme averages results from an explicit and an implicit Euler time step, that is, it works with

$$i\hbar\Psi^{n+1} = i\hbar\Psi^n + \tau\mathbf{H}_D\Psi^n \quad\Rightarrow\quad \Psi^{n+1} = \left(\mathbf{I}_{P-1} - i\frac{\tau}{\hbar}\mathbf{H}_D\right)\Psi^n, \quad \text{(explicit)} \quad \text{and} \tag{15a}$$

$$i\hbar\Psi^{n+1} - \tau\mathbf{H}_D\Psi^{n+1} = i\hbar\Psi^n \quad\Rightarrow\quad \left(\mathbf{I}_{P-1} + i\frac{\tau}{\hbar}\mathbf{H}_D\right)\Psi^{n+1} = \Psi^n \quad \text{(implicit)}, \tag{15b}$$

where $\mathbf{I}_n$ denotes the rank-n identity matrix, to form

$$\left(\mathbf{I}_{P-1} + i\frac{\tau}{2\hbar}\mathbf{H}_D\right)\Psi^{n+1} = \left(\mathbf{I}_{P-1} - i\frac{\tau}{2\hbar}\mathbf{H}_D\right)\Psi^n. \tag{16}$$

Therefore, at each time step, we need to solve a linear system of equations

$$\mathbf{L}\Psi^{n+1} = v, \tag{17}$$

with $\mathbf{L} = \left(\mathbf{I}_{P-1} + i\frac{\tau}{2\hbar}\mathbf{H}_D\right)$, $v = \mathbf{R}\Psi^n$, and $\mathbf{R} = \left(\mathbf{I}_{P-1} - i\frac{\tau}{2\hbar}\mathbf{H}_D\right)$

Some common matrix manipulations in Python The matrices in eqns. (16) have properties that we can use to our advantage: they are banded. There are two ways to go about it: to use the NumPy matrix manipulation functions, or the more efficient, but more complicated, SciPy sparse matrix manipulation. I will only discuss the former, since the latter is still very much work in progress, and harder to use. Below, I omit the prefix `numpy.` in front of all functions (or, I assume that I imported them from NumPy).

- The function `eye` primary purpose is to build an identity matrix of a certain size, though it also gives the option to choose which sub-diagonal to fill.
- The function `diag` is a more elaborate version of `eye`, in the sense that it allows you to fill the diagonal of your choice with the array of your choice, provided you know what you are doing with the dimensions (very easy to mess up!).
- The `dot` and `matmul` functions are similar and can sometimes used interchangeably. A rule of thumb is that the former should be used to take the dot product of two vectors (you can also take a look at `vdot` to take the dot product of a vector and its complex conjugate, but again, easy to mess it up), while the former should be used to multiply a matrix and a vector, or two matrices.
- In my solution, I used `solve(A, v)` to solve the $Ax = v$ systems that arise in the Crank-Nicolson method. But because the matrices in eqns. (16) are tridiagonal, Newman discusses a method to solve corresponding linear systems in § 6.1.6, with the corresponding routine `banded.py` downloadable on the online material of the textbook. You may use this less straightforward, but more efficient, method.
- (not a matrix manipulation function per se, but useful nonetheless) To take the complex conjugate of an array `A`, element-wise, use `conj(A)`.

Below is a code snippet that would compute the matrix product $x = \mathbf{M}v$, with

$$\mathbf{M} = \begin{pmatrix} 2 & +i & 0 \\ -i & 3 & +i \\ 0 & -i & 4 \end{pmatrix} \quad \text{and} \quad v = \begin{pmatrix} 5 \\ 5 \\ 5 \end{pmatrix}. \tag{18}$$

It is not the most efficient to go about in terms of number and length of lines, but it illustrates most functions I described.

```
from numpy import eye, diag, matmul, conj, arange, ones
vec_diag = arange(2, 5)
D = diag(vec_diag, k=0)  # k=0 means diagonal
Sup = 1j*eye(3, k=1)  # 3x3 matrix, with ones on 1st super-diagonal
Sub = conj(1j*eye(3, k=-1))  # 3x3 matrix, with ones on 1st sub-diagonal
M = D + Sub + Sup; print('\n', M)
v = 5*ones(3)
x = matmul(M, v); print('\n', x)
```

Computing diagnostics We are now reaching a level of complexity where it is impossible to know what the "true" solution should be, and we will have to resort to indirect measures of accuracy. In particular, on the first reflexes of a numerical physicist is to check conservation laws. In our case, we can check that normalization and total energy are conserved (you can also throw in momentum if you wish).

Computing diagnostics of your solutions means computing quantities of the form

$$D = \int_{-\infty}^{\infty} \psi^* \mathbf{O} \psi \, \mathrm{d}x, \tag{19}$$

with D some diagnostic quantity and $\mathbf{O}$ some operator (though because our potential wells will have infinitely tall walls, the integration bounds will be finite).

You can use the old methods for computing integrals that we have seen earlier in the term. When it comes to other operators however, some caution needs to be taken. Indeed, diagnosing quantities from a simulation is different than computing the integral of an analytical function, because the former has an internal consistency while the latter has a "true" value that exists regardless of how we compute it.

For example, to compute the wave function, you need to use the discretized Hamiltonian $\mathbf{H}_D$, which is essentially a differential operator. And to compute the energy, you will need to use the same discretized Hamiltonian, warts and all, for consistency's sake.

2D discrete Fourier transforms In Q2, we'll be taking the discrete cosine transform (DCT) and discrete sine transform (DST) for two dimensions, as seen in eqns. (10). The code `dcst_for_q3.py` provides functions for computing the 1D transforms and inverse transforms. In order to accomplish the goal of going from 1D to 2D transforms, we have provided the following templates. The first function is the forward transform, the second is the backward or inverse transform. Note that the direction order is reversed for the inverse transform.

```
import dcst
import numpy as np

def dXXt2(f):
    """ Takes DXT along x, then DXT along y (X = C/S)
    IN: f, the input 2D numpy array
    OUT: b, the 2D transformed array """

    M = f.shape[0]  # Number of rows
    N = f.shape[1]  # Number of columns
    a = np.zeros((M, N))  # Intermediate array
    b = np.zeros((M, N))  # Final array

    # Take transform along x
    for j in range(N):
        # DXT f[:, j] and set as a[:, j]
    # Take transform along y
    for i in range(M):
        # DXT a[i, :] and set as b[i, :]
    return b

def idXXt2(b):
    """ Takes iDXT along y, then iDXT along x (X = C/S)
    IN: b, the input 2D numpy array
    OUT: f, the 2D inverse-transformed array """

    M = f.shape[0]  # Number of rows
    N = f.shape[1]  # Number of columns
    a = np.zeros((M, N))  # Intermediate array
    f = np.zeros((M, N))  # Final array

    # Take inverse transform along y
    for i in range(M):
        # iDXT b[i,:] and set as a[i,:]
    # Take inverse transform along x
    for j in range(N):
        # iDXT a[:,j] and set as f[:,j]
    return f
```

Saving output for reuse Now that we are starting to write longer programs to handle more complex problems, it becomes important to save your data and output from intermediate steps to reuse at another time. For example, the time-integrations in this lab can take several tens of minutes. But what if you computed the field correctly, and then realize you made a mistake in plotting a figure? You may want to create an if statement such as

```
I_want_to_compute everything = False
if I_want_to_compute_everything:
    # ...
    # The time-integration procedure
    # ....
    # concludes with command to save all the results you will need for plotting
    # in a separate file
else:
    # a command to load the files you need in the separate file created above
```

A simple way to do this is through `numpy.save` and `numpy.savez` commands. For example, if you want to save arrays `x`, `y`, and `phi` to a file, you can do the following.

```
import numpy as np
...
# x, y, and phi have already been defined

np.savez('output_file', x=x, y=y, phi=phi)
```

The output statement `np.savez` takes a filename `output_file`, creates a file with the name `output_file.npz`, and saves data to that file. The other keyword/value pair arguments x=x, etc., represent variables in the file that are custom-defined by you, the user. In this example, variables x, y, and phi will be saved with their corresponding names in the output file output_file.npz, in a binary format.

Once your program wrote the output file, you can open it in another program, using np.load, and read values in, as in the following example:

```
npzfile = np.load('output_file.npz')
x = npzfile['x']
y = npzfile['y']
phi = npzfile['phi']
```

In this example, np.load returns a Python dictionary[2] with the name npzfile whose keys ('x', 'y', 'z') correspond to the variable names and whose values are the numpy arrays read from the .npz file.

Questions

In both questions, these simulations take a while, minutes to tens of minutes! When testing, reduce the number of grid cells or time steps, and only use the full integration durations and/or number of grid cells to produce the report figures.

1. **[60% of the lab] Solving the time-dependent Schrödinger equation with the Crank-Nicolson scheme:** This question, and the accompanying Physical and Computational backgrounds, are somewhat based on Exercise 9.8 (p. 439) of Newman's textbook, but not only. You may use it as support if you don't understand how to proceed.

   For this question, we will use the following values:

   - typical values for an electron, i.e., $L = 10^{-8}$ m, $m = 9.109 \times 10^{-31}$ kg or the value of m given by scipy.constants.
   - Initial condition
   $$\psi(x, t=0) = \psi_0 \exp\left(-\frac{(x-x_0)^2}{4\sigma^2} + i\kappa x\right), \qquad (20)$$
   with $\sigma = L/25$ and $\kappa = 500/L$. Note that the value of sigma ensures that $|\psi(x, t=0)|/\psi_0$ is less than machine precision away from x_0, and less than the smallest number representable by Python at $x = \pm L/2$. The value of x_0 will depend on the potential, and you will need to compute the value of ψ_0, either analytically or numerically, in order for the initial wave function to be normalized.
   - Interval $[-L/2, L/2]$ split into $P = 1024$ segments.
   - Time split into segments of duration $\tau = 10^{-18}$ s.

   (a) In a first time, code the Crank-Nicolson for the wave equation in the time-dependent Schrödinger equation in the case of an infinite square well (eqn. 5a). To do so,

   - define $x_0 = L/5$,
   - integrate for $N = 3000$ time steps, i.e., for a duration $T = N\tau$.

[2] https://realpython.com/python-dicts/

As a reminder, steps in programming this scheme should include the definition of the potential $V(x)$, the definition of the discretized Hamiltonian $\mathbf{H}_D$, the definition of one or more function(s) to compute the requested diagnostics (energy, normalization and expected position), and the loop to advance the solution in time.

In order to ensure that your code is not doing anything funny, check that energy is constant, and that the wave function remains normalized throughout.

HAND IN PLOTS OF ENERGY AND NORMALIZATION OF THE WAVE FUNCTION WITH TIME, AND COMMENTS ON IMPLEMENTATION AND PHYSICAL RESULTS.

(b) Describe the behaviour of the wave function. Alternatively, if you think that the wave function is too messy, you can comment on the probability density of the position, $\psi^*\psi$. How about the "trajectory" $\langle X\rangle(t)$: how does it compare with the out-of-context quote about the Ehrenfest theorem I used in the Physics background?

HAND IN PLOTS OF THE REAL PART OF ψ, OR ALTERNATIVELY, OF $\psi^*\psi$, AT $t = 0$, $T/4$, $T/2$, AND T. ALTERNATIVELY, YOU MAY SUBMIT VIDEO FILES OF YOUR ANIMATIONS. IN ANY CASE, CLEARLY EXPLAIN WHAT YOU ARE SUBMITTING. FINALLY, HAND IN YOUR COMMENTS.

(c) Now do the same as part (b), but for the harmonic potential (eqn. 5b), with $\omega = 3\times10^{15}$ s^{-1}, $x_0 = L/5$ again, but integrate over $N = 4000$ steps.

HAND IN PLOTS OF THE REAL PART OF ψ, OR ALTERNATIVELY, OF $\psi^*\psi$, AT $t = 0$, $T/4$, $T/2$, AND T. ALTERNATIVELY, YOU MAY SUBMIT VIDEO FILES OF YOUR ANIMATIONS. IN ANY CASE, CLEARLY EXPLAIN WHAT YOU ARE SUBMITTING. FINALLY, HAND IN YOUR COMMENTS.

(d) Now do the same as parts (b) and (c), but for the double-well (eqn. 5c), with $V_0 = 6\times10^{-17}$ J, $x_0 = L/3$, $x_1 = L/4$ and $N = 6000$ steps.

SAME INSTRUCTIONS AS FOR THE PREVIOUS TWO QUESTIONS. ADDITIONALLY, HAND IN YOUR CODE FOR AT LEAST ONE CASE.

2. **[40% of the lab] Resonant EM cavity**

Figure 1 shows a 2D resonant cavity consisting of a hollow, rectangular, perfectly conducting channel of dimensions $L_x \times L_y$. Suppose that the walls of the channel are aligned along the x- and y-axes. We shall excite this cavity in an artificial manner by imposing a z-directed alternating current pattern of angular frequency ω, which has the same spatial structure as the mode in which we are interested. We will calculate the electric and magnetic field patterns excited within the cavity by such a current pattern.

Note: Since we solve this problem in steps, it is possible to submit a single Python script for the entire question. Make sure to comment in your script which code pertains to which part.

(a) Explain why the decomposition in eqns. (10) satisfy the boundary conditions of E_z, H_x and H_y.

HAND IN YOUR NOTES.

(b) For computing the fields, we suggest the following routine:

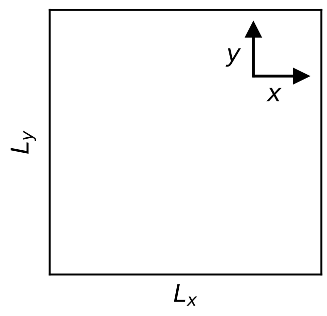

Figure 1: A $L_x \times L_y$ 2D resonant cavity with conducting walls.

- Implement the discretizations for t, x and y described in the Physics Background. Then, use eqn. (8) to generate the current pattern $J_z(x, y, t)$.
- Take the Fourier transforms of H_x, H_y, J_z, and E_z (recall eqns. 10).
- Evolve the Fourier coefficients using eqns. (11), which follow the Crank-Nicolson method.
- Reconstruct H_x, H_y, and E_z via a 2D inverse Fourier transform.
- Loop over time.

Starting from the sample code given in the computational background and the `dcst.py` script on the textbook's online material, write your own functions to compute the 2D Fourier transforms (and their inverses) for eqns. (10). Pay attention to what directions the transforms are taken, particularly for H_x and H_y. A way to check your functions is to test that $f = \mathscr{F}_{2D}^{-1}(\mathscr{F}_{2D}(f))$ is true for some 2D array f.

HAND IN YOUR CODE.

(c) Use your 2D Fourier transform functions and your pseudocode to write a Python script that calculates the electromagnetic field. Set the final time $T = N\tau = 20$, $\tau = 0.01$, $L_x = L_y = J_0 = m = n = c = 1$, $P = 32$.

With the driving frequency $\omega = 3.75$, evolve the fields over time, then plot the traces $H_x(x = 0.5, y = 0.0)$, $H_y(x = 0.0, y = 0.5)$, and $E_z(x = 0.5, y = 0.5)$ as a function of time. Explain the pattern of the traces (i.e., what is happening in the cavity?).

SUBMIT CODE, PLOTS, AND EXPLANATORY NOTES.

(d) Repeat the first question, but now vary the driving frequency uniformly between $\omega = 0$ to $\omega = 9$. Plot the maximum amplitude of $E_z(x = 0.5, y = 0.5, t)$ as a function of ω. At approximately what frequency does the amplitude peak?

The *normal frequencies* of this system are given by

$$\omega_0^{m,n} = \pi c\sqrt{(nL_x)^{-2} + (mL_y)^{-2}}. \tag{21}$$

How does the frequency at which the amplitude peaks that you computed above compare with this formula?

SUBMIT CODE, PLOTS, AND EXPLANATORY NOTES.

(e) Take $\omega = \omega_0^{1,1}$, and plot $H_x(x = 0.5, y = 0.0)$, $H_y(x = 0.0, y = 0.5)$, and $E_z(x = 0.5, y = 0.5)$ as a function of time. Compare this to the plots you made in part (a) and provide an explanation for their differences. What can you say about the energy in the cavity?

SUBMIT PLOTS AND EXPLANATORY NOTES.