Physics 129L: Problem Set 3

Problem Set Submission Guideline

A) Github Submissions Starting with problem set 2, we will use GitHub for problem set submissions. To access the problem set, please **fork** and **clone** the **forked** repository to your local virtual machine. **Please complete the problem set in this forked directory.** Submit **a pull request** for merging into the main branch before the problem set due date.

B) .tar.gz File compression and submission on Github For each problem set, you are asked to submit the compressed version of the problem set to GitHub via git operation. Here is a step-by-step guideline:

1. Use the **tar** command to compress the problem set directory into a **single** ".tar.gz" file.
2. Obtain the sha256sum by running "sha256sum P2.tar.gz".
3. Echo the **full sha256sum** to a text file named "sha25sum_problem_set.txt".
4. Initialize a git repository named "Archive_P# (#: problem set number) on your local machine, and move both the "tar.gz" file and the "sha25sum_problem_set.txt" file to the repository.
5. Create an empty **public** directory under the **same name** in **your own GitHub account**.
6. **Push** this local repository to the remote repository.

Problem 1: Time complexity

Time complexity is a fundamental concept in computer science and algorithm analysis. It measures the efficiency of an algorithm, helping researchers understand how the algorithm's performance scales as the input size increases. In essence, it provides insights into how long it will take for an algorithm to complete its task as a function of the input's size. Time complexity is typically expressed using "Big O notation," $\mathcal{O}(...)$ that represents the upper bound on the number of basic operations an algorithm performs concerning the size of the input. This notation allows us to compare and contrast algorithms and make informed decisions about which one to use for a particular problem. The table below summarizes the time complexity of few known examples.

Time Complexity	Example	Algorithm
$\mathcal{O}(1)$ - Constant	Accessing an element in an array	Basic arithmetic operations
$\mathcal{O}(\log(n))$ - Logarithmic	Binary search in a sorted array	Binary search
$\mathcal{O}(n)$ - Linear	Iterating through a list or array	Linear search
$\mathcal{O}(n\log(n))$ - Linearithmic	Divide and Conquer	Merge Sort, Quick Sort
$\mathcal{O}(n^2)$ - Quadratic	Nested loops to compare all pairs of elements	Bubble Sort
$\mathcal{O}(n^k)$ - Polynomial	Algorithms with nested loops or recursion	Polynomial regression, dynamic programming
$\mathcal{O}(2^n)$ - Exponential	Brute-force algorithms	Subset generation

Time Complexity	Example	Algorithm

We will study and visualize the time complexity of various algorithms using Python. For the following algorithms, you are asked to identify their time complexity and create visualizations. You will explore the time complexity differences among various implementations.

Sort Algorithm: List v.s. dictionary

Here is a simple example algorithm that preform sort operation using **list**. The algorithm tests various input sizes and plot the corresponding time complexity.

In [ ]:
```
import random
import time
import matplotlib.pyplot as plt

def sort_using_list(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

def generate_random_list(n):
    return [random.randint(1, 10000) for _ in range(n)]

# Test different input sizes
input_sizes = [100, 500, 1000, 5000, 10000]
execution_times = []

for size in input_sizes:
    input_list = generate_random_list(size)

    start_time = time.time()
    sort_using_list(input_list)
    end_time = time.time()

    execution_time = end_time - start_time
    execution_times.append(execution_time)

# Create a plot
plt.plot(input_sizes, execution_times, marker='o')
plt.title("Sort Algorithm Using List: Time Complexity")
plt.xlabel("Input Size")
plt.ylabel("Execution Time (seconds)")
plt.grid(True)
plt.show()
```

A)

The above algorithm uses **list** to store data, but in general, we can use **dictionary** to preform the same task. Rewirte the above algorithm by using **dictionary**.

```
In [ ]: import random
import time
import matplotlib.pyplot as plt

'''---------------------Write your code below this line----------------------
def sort_using_dict(arr):
    pass

def generate_random_list(n):
    # this function needs to return "input_list"
    pass
    return input_list
'''---------------------Write your code above this line----------------------

# Test different input sizes
input_sizes = [100, 500, 1000, 5000, 10000]
execution_times = []

for size in input_sizes:
    input_list = generate_random_list(size)

    start_time = time.time()
    sort_using_dict(input_list)
    end_time = time.time()

    execution_time = end_time - start_time
    execution_times.append(execution_time)

# Create a plot
plt.plot(input_sizes, execution_times, marker='o')
plt.title("Sort Algorithm Using dictionary: Time Complexity")
plt.xlabel("Input Size")
plt.ylabel("Execution Time (seconds)")
plt.grid(True)
plt.show()
```

B)

You want to compare the difference in time complexity between the usage of a **list** and a **dictionary**. Create a plot with a (1,2) layout that includes the following:

First plot: Display two time complexity curves on the same graph and include a legend.

Second plot: Plot the difference in time complexity.

Be sure to label the axes, and is there any difference?

```
In [ ]: import matplotlib.pyplot as plt

        '''---------------------Write your code below this line---------------------



        '''---------------------Write your code above this line---------------------
```

C)

We observe that the time complexity, denoted as T, depends on the input size n, represented as $T(n)$. Extract the amplitudes T_0 and decay constants n_0 for the two time complexity curves discussed in part ** B)** using the exponential function:

$$T(n) = T_0 e^{-n/n_0}.$$

Which decay constant is larger, and what does it imply?

```
In [ ]: import scipy as sp
        '''---------------------Write your code below this line---------------------




        '''---------------------Write your code above this line---------------------
```

D)

Now, we can compare the two methods above with existing sorting functions in different packages. We will create a (1,2) plot that shows the following:

1. numpy.sort()
2. sorted()
3. .sort()
4. scipy.sort()
5. your sort method that uses **list**
6. your sort method that uses **dictionary**

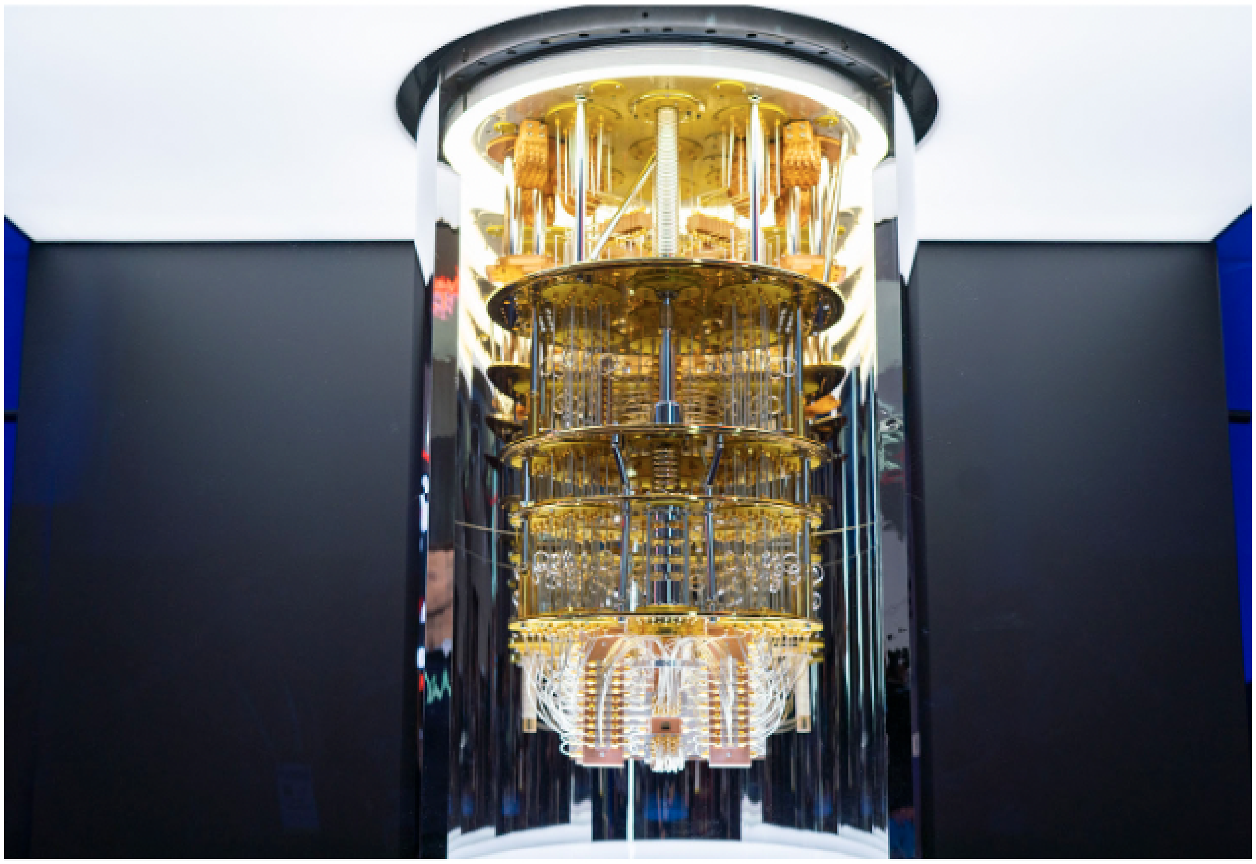
We will test these methods with various input sizes: input_sizes = [100, 500, 1000, 5000, 10000]. The plot will include a legend containing the corresponding index for each method.

For each mentioned method, we will calculate the decay constants and plot them in descending order, labeling the axes by their respective indices. What observations can you make?

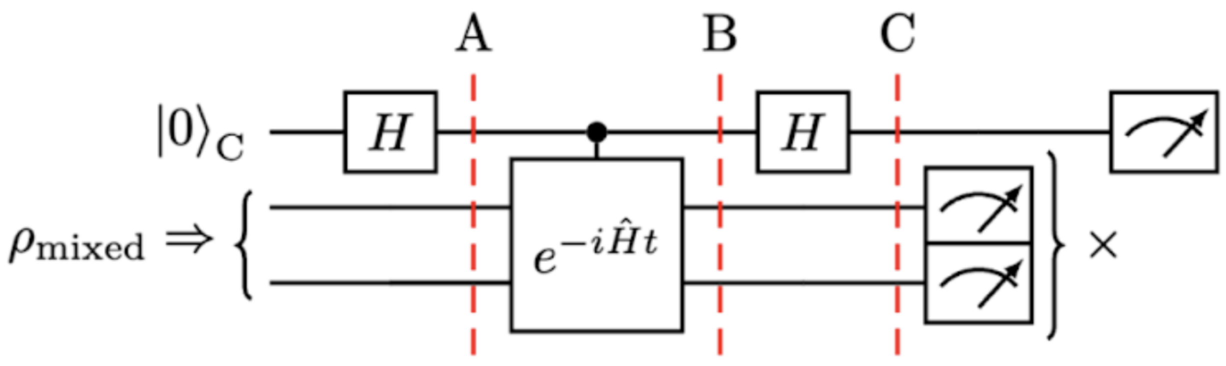
In [ ]:
```python
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import random
import time

'''---------------------Write your code below this line---------------------



'''---------------------Write your code above this line---------------------
```

Problem 2: Visualizing Quantum Time Evolution

Quantum computing is a revolutionary approach to computation that leverages the principles of quantum mechanics to perform complex calculations that would be practically impossible for classical computers.



In this problem, you will be exploring a dataset generated by the advanced quantum computer in IBM Quantum. The quantum circuit has a general form,



We found that the quantum evolution of qubits can be linked to the evolution of a general quantum state with a quantum Hamiltonian,

$$H = -\begin{bmatrix} J+2B & 0 & 0 & 0 \\ 0 & -J & 2J & 0 \\ 0 & 2J & -J & 0 \\ 0 & 0 & 0 & J-2B \end{bmatrix},$$

where J and B are unknown parameters.

A)

Import the dataset, "ibm_1.0_qubits[2, 1, 3]_Y.json" from the directory "problem2/ibm_quantum_evolution". The data should be imported as a **single 2-dimensional NumPy array named 'ibm_qubit_array.** Define variables named 'time' and 'evolution_y' as follows in the code. **Plot** the result.

In [ ]:
```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt


'''--------------------Write your code below this line--------------------


time=ibm_qubit_array[:,0]
tevolution_y=ibm_qubit_array[:,1]


'''--------------------Write your code above this line--------------------
```

B)

Numerically diagonalize the Hamiltonian and obtain **all** eigenvalues $\lambda_1, \lambda_2, \lambda_3, \lambda_4$ and eigenvectors, $\{|\lambda_1\rangle, |\lambda_2\rangle, |\lambda_3\rangle, |\lambda_4\rangle\}$. We define the equal-weight state (EW-state) as

$$|\psi\rangle = \frac{1}{4}\sum_i |\lambda_i\rangle,$$

and the expectation value of the time evolution operator has the form,

$$\langle\psi|e^{-iHt}|\psi\rangle = \frac{1}{4}\sum_i e^{-i\lambda_i t}.$$

Calculate this expectation value for each value in the time array defined above by setting

In [ ]:
```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt


'''---------------------Write your code below this line---------------------



'''---------------------Write your code above this line---------------------
```

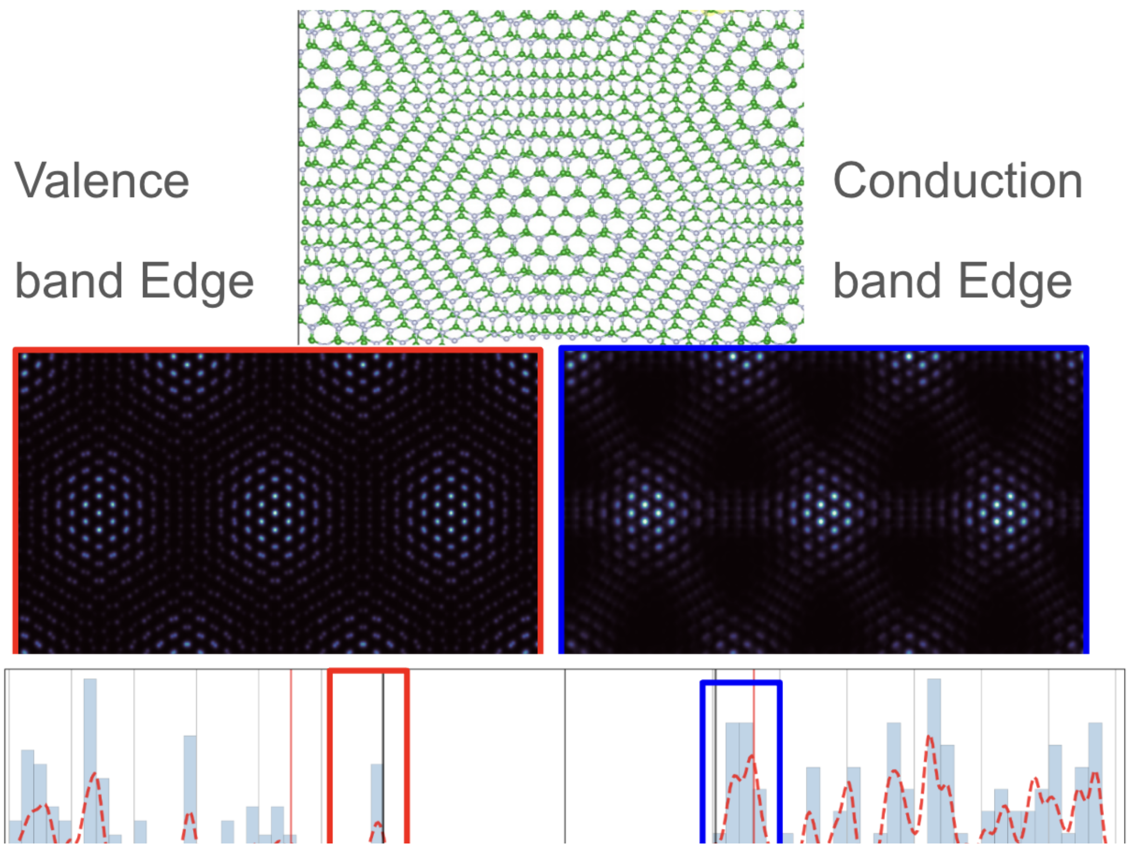
C)

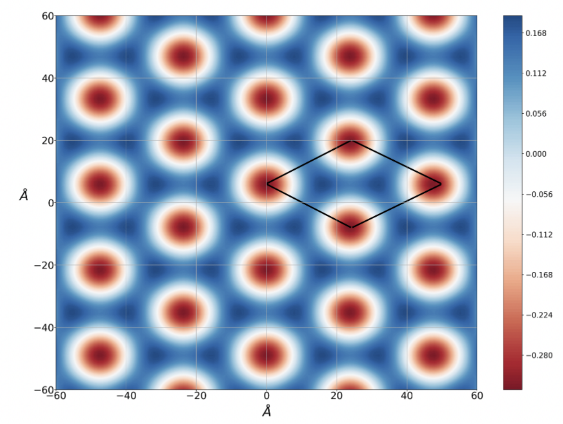
Plot a family of curves by varying $J = [1, 1.5, 2, 2.5]$ while fixing $B = 1$. How do those curves change when J becomes larger?

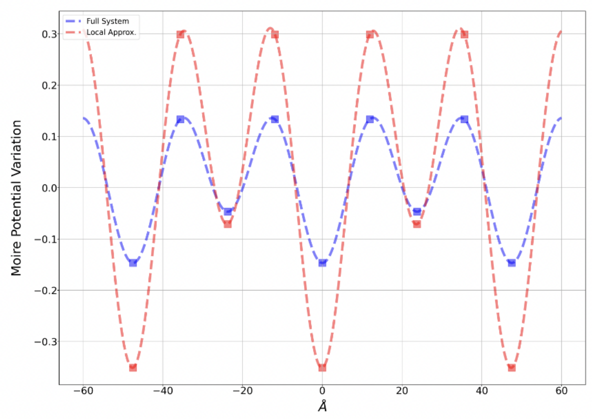
In [ ]:
```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt


'''---------------------Write your code below this line---------------------



'''---------------------Write your code above this line---------------------
```

Problem 3: Local Density of States for Electrons

The Local Density of States (LDOS), in the context of electronic structure and solid-state physics, is a fundamental concept used to describe the density of available electronic states for electrons at a specific position in a material or a solid. It is a crucial quantity in understanding the electronic properties of materials, particularly in the study of electronic band structures, energy levels, and electron behavior. In this problem, you will be looking at the local density of states for a material near the conduction and valance band edge.







Each file inside the directory 'problem3/Local_density_of_states_near_band_edge' contains the local density of states of a 2-dimensional material at a specific energy level. You want to understand the correlation between energies and the patterns of local density of states.

A)

For each text file in the directory, generate a 2-dimensional heatmap depicting the local electron density. Include a color legend to indicate the intensity. Label each heatmap with the same file index and save the images in a new directory named 'local_density_of_states_heatmap' within 'problem3/Local_density_of_states_near_band_edge'.

```
In [ ]: import numpy as np
        import scipy as sp
        import matplotlib.pyplot as plt


        '''--------------------Write your code below this line---------------------





        '''--------------------Write your code above this line---------------------
```

B)

For each text file in the directory, create a 2-dimensional surface plot where the height profile represents the local density of states. Label each height profile with the same file index and save the image in a new directory named 'local_density_of_states_height' within 'problem3/Local_density_of_states_near_band_edge'. (You'll need to refer to 3D plotting in matplotlib).

```
In [ ]: import numpy as np
        import scipy as sp
        import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import Axes3D

        '''--------------------Write your code below this line---------------------





        '''--------------------Write your code above this line---------------------
```

C)

Select a local sub-region that piques your interest, and quantitatively illustrate the changes while offering some physical speculations. For instance, the basic analysis you should present involves calculating the average local density of states within a predefined subregion for each file and plotting these changes across all indices. Feel free to explore different analyses, but ensure that their complexity is comparable to the example provided above.

```
In [ ]: import numpy as np
        import scipy as sp
        import matplotlib.pyplot as plt

        '''---------------------Write your code below this line----------------------




        '''---------------------Write your code above this line----------------------
```