Problem Set 1

- *Submission format*—Submit your solution via NTULearn, with **one Python source file per problem (`.py` not `.ipynb`)**. Each file should begin with the lines

  ```
  import numpy as np
  import scipy
  import matplotlib.pyplot as plt
  ```

  You may also import other Scipy modules or standard Python modules, but do not use non-standard modules (e.g., stuff from PyPI).

- *Coding assistants*—AI coding assistants may be used if you want, though all assignments can be completed without them. If you use an AI assistant, you must carefully scrutinize, test, and fix up the code it generates. But do not perform verbatim copying of other people's work, including other students taking this course; that is treated as plagiarism.

- *Code quality*—For full marks, the submitted programs must meet these criteria:

  1. Keep all other top-level code, aside from import statements and function/class definitions, to a minimum. Most of your code should live inside functions. Your program should work if we `import` it as a module and call its functions.
  2. If the assignment asks you to write a function, follow the specification exactly. Do not modify the order or definitions of the inputs/outputs to suit yourself. You are free to define additional helper functions, classes, or data structures.
  3. Use comments to document important code blocks. In particular, if you define helper functions or class methods, document their inputs/outputs clearly.
  4. Avoid cryptic function or variable names like `abc`.
  5. Nontrivial numerical constants should be grouped appropriately (e.g., the start of a function), not "hard-coded" deep inside code blocks.
  6. Functions should run appropriate "sanity checks" (e.g., using `assert`) on inputs.
  7. Every generated plot should be properly formatted, with appropriate axis ranges, a proper figure title and axis labels, a legend if there are multiple curves per graph, etc.

0. GAUSSIAN ELIMINATION (15 MARKS)

In this problem, you will implement and test the Gaussian elimination algorithm. Given a square matrix A and vector $\vec{b}$, the algorithm solves $A\vec{x} = \vec{b}$ to find $\vec{x}$.

(a) (3 marks) Write a function to perform the row reduction phase of the algorithm:

`def row_reduce(A, b, n):`	
Inputs	
`A`	A 2D array specifying a square matrix A.
`b`	A 1D array specifying a vector $\vec{b}$.
`n`	The pivot row index (an integer).

This function has no return values, and performs row reduction by directly modifying the contents of the `A` and `b` arrays, so that all elements underneath `A[n,n]` become zero:

$$\text{For each } m > n, \begin{cases} \text{(i)} & b_m \rightarrow b_m - \left(\frac{A_{mn}}{A_{nn}}\right) b_n \\ \text{(ii)} & A_{mk} \rightarrow A_{mk} - \left(\frac{A_{mn}}{A_{nn}}\right) A_{nk} \quad \text{for each } k. \end{cases} \tag{0}$$

Hint—The function should detect the case where A appears to be non-invertible, which is associated with the denominators in Eq. (0) approaching zero. Such error cases should be handled by raising an exception via the `raise` statement.

(b) (2 marks) Write a function to perform pivoting:

`def pivot(A, b, n):`	
Inputs	
`A`	A 2D array specifying a square matrix A.
`b`	A 1D array specifying a vector $\vec{b}$.
`n`	The pivot row index (an integer).

The function has no return values, and directly modifies `A` and `b` to perform "pivoting":

- Search rows $m \geq n$, and finding the row with the largest $|A_{mn}|$
- If a row $m \neq n$ was found,
  - Swap rows m and n in A.
  - Swap elements m and n in $\vec{b}$.

(c) (2 marks) Write a function to perform Gaussian elimination:

`def gauss_eliminate(A, b):`	
Inputs	
`A`	A 2D real or complex array specifying a square matrix A. Will not be altered.
`b`	A 1D real or complex array specifying a vector $\vec{b}$. Will not be altered.
Return value	
`x`	A 1D array containing the solution to $A\vec{x} = \vec{b}$.

Unlike the functions from (a) and (b), the `gauss_eliminate` function should *not* alter the contents of the input arrays `A` and `b`.

(d) (3 marks) To verify that your code from (a)–(c) is working correctly, write a set of *unit tests* for the `row_reduce`, `pivot`, and `gauss_eliminate` functions. The unit tests should use Python's `unittest` framework, and can be submitted as a separate `.py` file. It is up to you to design these tests, and to ensure that they achieve good coverage of the tested functions' specifications and functionality.

(e) (3 marks) Write a function, `gauss_eliminate_profile()`, to profile the performance of Gaussian elimination. You should compare the performance of (i) the `gauss_elimination` solver you wrote, and (ii) Scipy's own `solve` function. Present your results as "log-log" plots of the runtime t_N versus the problem size N, along with fitted power law trend-lines

$$t \sim N^p \quad \Leftrightarrow \quad \log(t) \approx p \log(N) + q. \tag{1}$$

In your comments, discuss whether the scaling is as expected, and any other relevant details.

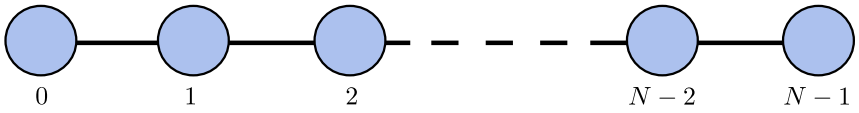
(f) (2 marks) Write a function, `matrices_profile()`, to profile the performance of the Numpy/Scipy library functions for the following matrix operations:

- Finding all eigenvalues.
- Finding all eigenvalues and eigenvectors.
- Determinant.
- Matrix exponential.

For each operation, use the same approach as in (e) to determine how the runtime scales with problem size. Label all plots clearly and discuss your findings in code comments.

1. 1D TIGHT-BINDING MODELS (15 MARKS)

In condensed matter physics, tight-binding models are common class of toy models that describe a quantum particle living on a discrete lattice. Consider a set of N discrete positions, or "sites", laid out in a 1D chain and indexed by $n = 0, 1, \ldots, N-1$:



The quantum state of the particle is described by a complex vector,

$$|\psi\rangle = \left.\begin{bmatrix} \psi_0 \\ \vdots \\ \psi_{N-1} \end{bmatrix}\right\} N \text{ complex numbers} \tag{2}$$

which is normalized to unity (a.k.a. probability conservation):

$$\langle\psi|\psi\rangle = \sum_{n=0}^{N-1} |\psi_n|^2 = 1. \tag{3}$$

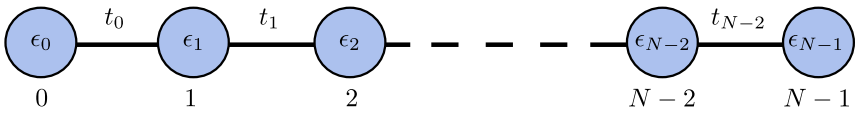
An energy eigenstate $|\psi_n\rangle$ is a solution to the time-independent Schrödinger equation

$$H|\psi_n\rangle = E_n|\psi_n\rangle, \tag{4}$$

where the Hamiltonian H is a Hermitian matrix and E_n is the "eigen-energy". For each H, there are N distinct energy eigenstates. We will assume H involves only "on-site" and "nearest neighbour" couplings, corresponding to the following nonzero matrix elements:

$$\left.\begin{aligned} H_{nn} \equiv \epsilon_n \in \mathbb{R} \quad &(\text{"on-site energy"}) \\ H_{n,n+1} = H^*_{n+1,n} = t_n \in \mathbb{C} \quad &(\text{"nearest neighbour hopping"}) \end{aligned}\right\} \quad n = 0, 1, \ldots \tag{5}$$

Hence, H is determined by the real numbers $\epsilon_0, \ldots, \epsilon_{N-1}$, and the complex numbers $t_0, \ldots, t_{N-2}$. Note that there are N sites but $N-1$ hopping terms. All other matrix elements are zero.



(a) (3 marks) Write the following function to create the Hamiltonian for a 1D tight binding model:

`def chain_hamiltonian(epsn, t):`	
Inputs	
`epsn`	On-site energies (real 1D array).
`t`	Nearest-neighbour hoppings (real or complex 1D array).
Return value	
`H`	The Hamiltonian matrix (complex 2D array).

Hint—As usual, be sure to do reasonable sanity checks on the function inputs.

(b) (6 marks) We will now study a *random* tight-binding chain with the following properties: (i) each on-site energy has the form $\epsilon_n = \mathcal{E}_0 \chi_n$, where $\mathcal{E}_0$ is an overall energy scale and χ_n is a random variable drawn independently from the standard normal distribution; (ii) $t_n = t_0$ for all n.

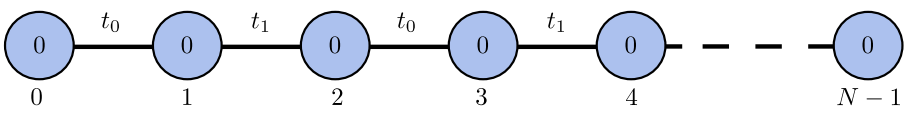
Write a function `random_chain_study()`, which takes no inputs and produces a series of plots demonstrating the properties of the random chain model:

(i) A "band diagram" showing the eigen-energies $E_0, \ldots, E_{N-1}$ (vertical axis) versus the on-site energy scale $\mathcal{E}_0$ (horizontal axis). All eigenenergies should be plotted together in one graph. Use the same set of random variables $\{\chi_n\}$ throughout, rather than re-drawing them for each $\mathcal{E}_0$. Keep N and t_0 fixed. Use your own judgment to choose N, t_0, and the range of $\mathcal{E}_0$.

(ii) Several "mode distribution" plots showing $|\psi_n|^2$ versus the spatial index n, for a few representative energy eigenstates. Show results (labelled clearly) for at least three regimes: $|\mathcal{E}_0| \ll |t_0|$, $|\mathcal{E}_0| \sim |t_0|$, and $|\mathcal{E}_0| \gg |t_0|$. For clarity, add indicators to the band diagram of part (i) showing the locations of your parameter choices in part (ii).

You should ensure that each plot is properly labelled and documented in code comments. In additional code comments, discuss the physical meaning of your results.

(c) (6 marks) We will now study a so-called Su-Schrieffer-Heeger (SSH) model, consisting of a 1D tight-binding chain with $\epsilon_n = 0$ and alternating hoppings t_1 and t_2:



Note that the last hopping in the chain depends on whether the number of sites is even or odd (i.e., t_0 if N is even, and t_1 if N is odd).

Write a function `ssh_chain_study()`, which takes no inputs and produces a series of plots demonstrating the properties of the SSH model:

(i) Band diagrams showing the variation of the eigen-energies $E_0, \ldots, E_N$ versus t_0, for t_1 fixed. Show two cases: even N and odd N.

(ii) Mode distribution plots, $|\psi_n|^2$ versus n, for some representative energy eigenstates. Show results for various choices of t_0, t_1, and even/odd N, and pay particular attention to the "zero modes" having $E_n = 0$ (but don't focus only on them).

You should ensure that each plot is properly labelled and documented in code comments. In additional code comments, discuss the physical meaning of your results.