Introduction to Computational Quantum Mechanics: Application-based Learning with Python

Braden M. Weight
Department of Physics and Astronomy
University of Rochester

September 13, 2023

Contents

Preface

All codes, lecture notes, and homeworks can be accessed at
https://github.com/bradenmweight/Intro_Computational_Quantum_Mechanics

Chapter 1

Python and Environments

1.1 Miniconda

Anaconda is a distribution of python is extremely convenient to install and utilize. Anaconda comes in two forms: Anaconda and Miniconda. Anaconda comes pre-installed with many things, while Miniconda is the bare minimum of pre-installed packages. I suggest everyone start with Miniconda and simply install on top of the what you need as we go along. The Anaconda website can be found here.

To install the Miniconda distribution, follow these steps:

- Log into the cluster.
- Migrate to the location you want to install Miniconda, usually in home ("∼/")
- Get the install file from the web:

```
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
```

- Verify the completed download:

```
ls -lh Miniconda3-latest-Linux-x86_64.sh
```

- Convert the file to an executable file and run it:

```
chmod +x Miniconda3-latest-Linux-x86_64.sh
./Miniconda3-latest-Linux-x86_64.sh
```

- Reload the terminal:

```
source ~/.bashrc
```

- Check to see if conda works:

```
conda --help
```

1.2 Conda Environments

By default, anaconda provides the ”base” environment, which is the default python that comes with the version of Miniconda that you installed. For now, we can use this default environment for everything that we will need for the remainder of this course. However, when one becomes involved with multiple hard-core projects involving different versions of python and, worse, interfering package versions, then one should consider to build additional environments and switch between them for long-term projects. For this projected individual, we will outline how to manage environments.

The following is a list of commands to check the new conda environment:

- List the environments currently available:

  ```
  conda env list
  ```

- Check the current environment variable:

  ```
  echo $CONDA_PREFIX
  ```

- List the available/installed packages:

  ```
  conda list
  ```

- Make sure the correct conda/user’s python is being called (not the system python...):

  ```
  which python
  ```

The following is a list of commands to manage the current conda environment:

- Install the Numpy package (useful for number operations and linear algebra):

  ```
  conda install numpy
  ```

- Install the Scipy (similar to Numpy but with some other features that we may use) and Matplotlib (plotting) packages :

  ```
  conda install scipy matplotlib
  ```

- Example for removing the XYZ and ABS packages:

  ```
  conda remove XYZ ABC
  ```

- Create a new anaconda environment called ”myENV” with a specific choice of python version (optional):

  ```
  conda create --name myENV python=3.9
  ```

- To switch between two conda environments, one needs to deactivate the current (doesn’t matter what it is called) and activate the next:

  ```
  conda deactivate
  conda activate myENV
  ```

1.3 The First Python Code!

In this section, the reader will be exposed to some common items in all python codes, namely the import block, the " __name__ == '__main__' " block (although technically optional, but highly recommended), and general, user-defined functions. Additionally, we will compose mathematical functions with Numpy and plot them using Matplotlib. Keep in mind that we will learn in later chapters about the types of variables in python and how to use them properly and efficiently. Our goal here is to introduce the reader to their first full code to make simple plots while learning about code styling and general python code layout.

```
# NAME: my_first_python_code.py

#####################
# THIS IS A SINGLE-LINE COMMENT
"""
THIS IS A BLOCK COMMENT.
IT CAN GO FOR MANY LINES.
"""
#####################



# Python modules
import random

# External modules
import numpy as np
import matplotlib
matplotlib.use('Agg') # This is required for the NDSU cluster...not sure why
from matplotlib import pyplot as plt

# User-made modules
# We don't yet have any of these.

""" Description of the above imports:
  `as np' renames the Numpy module to 'np'
  A function inside the module Matplotlib is called pyplot.
  We want it, and we rename it as plt
"""

def my_print_function(something_to_print):
    """
    This is a function comment.
    They are useful whenever your define a new function.
    You should always tell the purpose of the function
      as well as what is its input and output.

    For example:
    The purpose of this function is to read a string and print the string.
    INPUT: something_to_print [str]
    OUTPUT: None
    """
```

```
    print( "My function is printing something:" )
    print( something_to_print )

    # This is a return statement.
    # For this function, it is optional. Default is ``None''/
    return None

def main():
    # This is the main function of the code.
    # This enables one to get main from both the command line
    #   as well as from another python module.

    something_to_print = "Hello, world." # String variable

    print("Print directly:", something_to_print) # Print directly to the screen

    # Write a function to print whatever you provide to it
    my_print_function( something_to_print )

# This checks whether someone is calling this module at the command line directly,
#   or whether it is being called from another python file as a user-defined module
if ( __name__ == "__main__" ): # This will evaluate to True if run from the command line
    main()
```

- Create a new file called "my_first_python_code.py"

  ```
  touch my_first_python_code.py
  ```

- Modify the code with your favorite text editor (*i.e.*, VIM, Notepad++, Visual Studio Code, Nano, EMACS, etc.)

  ```
  vim my_first_python_code.py
  ```

- Once your code is completed, you can run the code with the following command:

  ```
  python3 my_first_python_code.py
  ```

```
# NAME: my_first_plot.py

# Python modules
import random

# External modules
import numpy as np
import matplotlib
matplotlib.use('Agg') # This is required for the NDSU cluster...not sure why
from matplotlib import pyplot as plt
```

```
# User-made modules
# We don't yet have any of these.

def get_globals():
    global X_GRID

    # Create a numpy array starting from 1 to 1000 in increments of 2
    X_GRID = np.arange( 1,1001,2 )


def get_Y_quadradtic():
    """
    The purpose of this function is to create the Y-values for a user-defined function
    INPUT:  None
    OUTPUT: Y [1D nd.array]
    """

    # Let's do a quadratic function
    Y_X = X_GRID ** 2 / 10**3 # y(x) = x^2 / 10^4

    # This is a return statement.
    return Y_X

def plot_quadratic_1D_LINE(Y_X):
    """
    The purpose of this function is to generate a 1D line plot.
      Generates image file "Y_X_LINE.jpg"

    INPUT:  Y_X [1D nd.array]
    OUTPUT: None
    """

    plt.plot( X_GRID, Y_X, "-", c='black', linewidth=3, label="My 1D curve:\n$Y(X) = \\frac{X^2}{10^3} $" )

    ##### OPTIONAL PLOTTING COMMANDS #####
    plt.legend()
    plt.xlim( X_GRID[0], X_GRID[-1] ) # Define x-axis as limits of your chosen X_GRID
    plt.ylim( 0 ) # Define one limit of the y-axis
    plt.xlabel("This is my X-Axis",fontsize=15)
    plt.ylabel("This is my Y-Axis",fontsize=15)
    plt.title("This is my title",fontsize=15)
    ######################################

    plt.savefig("Y_X_LINE.jpg",dpi=600)
    plt.clf() # This clears the internal plot in case we want to make another.

def get_Y_random():
    """
    The purpose of this function is to create the Y-values for a user-defined function
    INPUT:  None
    OUTPUT: Y [1D nd.array]
    """

    # How many Y-values do we need to get ?
    NGRID = len(X_GRID)

    #### Let's do a random function
```

```
    # These are uniform random numbers on the interval [0,1)
    RAND = random.random() # This is a single random number
    RAND_LIST = [ 2*random.random() - 1 for j in range(NGRID) ] # This is a list of random numbers
    # "for j in range(NGRID)" above just gives us the same number of randoms as the X_GRID variable
    # "2*random.random() - 1" converts from [0,1) to [-1,1)

    # Let's also do a set of random gaussian numbers with mean of 0 and width of 1.
    RAND_GAUSS_LIST = [ random.gauss(0,1) for j in range(NGRID) ]

    return RAND_LIST, RAND_GAUSS_LIST


def plot_random_1D_LINE_SCATTER_two_FUNCTIONS(Y1, Y2):
    """
    The purpose of this function is to generate a 1D line plot.
      Generates image file "Y_X_LINE.jpg"

    INPUT:  Y1 [1D nd.array]
    INPUT:  Y2 [1D nd.array]
    OUTPUT: None
    """

    plt.scatter( X_GRID, Y1, c="blue", label="Uniform Random" )
    plt.plot( X_GRID, Y2, "o", c="red", label="Guassian Random", alpha=0.25 )

    # Plot horzontal lines indicating certain quantities
    plt.plot( X_GRID, 1.0 *np.ones(len(X_GRID)), "--", c="black", lw=2, label="$\pm 1$" )
    plt.plot( X_GRID, -1.0*np.ones(len(X_GRID)), "--", c="black", lw=2 )



    ##### OPTIONAL PLOTTING COMMANDS #####
    plt.legend()
    plt.xlim( 0, 1000 ) # Define x-axis as limits of your chosen X_GRID
    plt.ylim( -5,5 ) # Define limits of the y-axis
    plt.xlabel("Random Coordinate",fontsize=15)
    plt.ylabel("Random Value",fontsize=15)
    plt.title("Uniform vs. Gaussian Random Numbers",fontsize=15)
    ######################################

    plt.savefig("Y_X_RANDOM_SCATTER.jpg",dpi=600)
    plt.clf() # This clears the internal plot in case we want to make another.

def main():

    get_globals()

    # Let's plot a quadratic function
    Y_X = get_Y_quadradtic()
    plot_quadratic_1D_LINE(Y_X)

    # Let's plot a random function
    RAND_LIST, RAND_GAUSS_LIST = get_Y_random()
    plot_random_1D_LINE_SCATTER_two_FUNCTIONS(RAND_LIST, RAND_GAUSS_LIST)

if ( __name__ == "__main__" ):
    main()
```

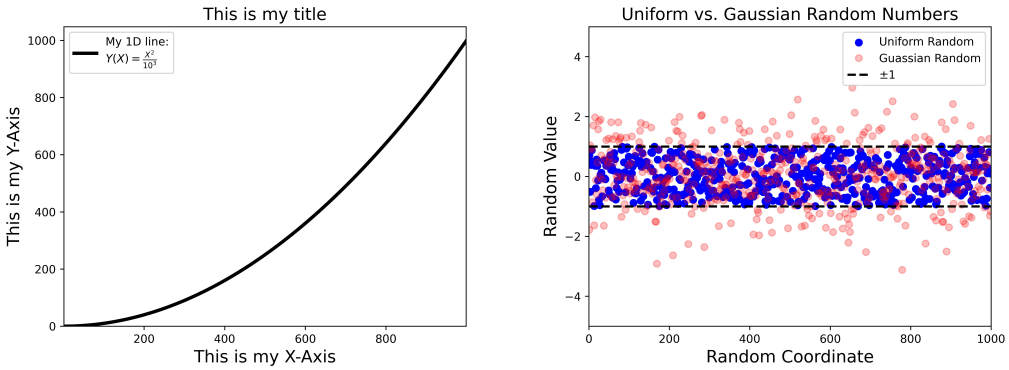

Figure 1.1: Generated from the code "my_first_plot.py". (Left) A 1D line plot of a quadratic function.

- Create a new file called "my_first_plot.py"

  ```
  touch my_first_plot.py
  ```

- Modify the code with your favorite text editor (*i.e.*, VIM, Notepad, VSCode, Nano, etc.)

  ```
  vim my_first_plot.py
  ```

- Once your code is completed, you can run the code with the following command:

  ```
  python3 my_first_plot.py
  ```

In these two example python codes, we have seen how to plot a simple 1D curve as well as how to generate and plot random data. These tools will be useful later when we discuss random motion of parties (*e.g.*, Brownian motion) using the Langevin equation. Although one can simply "copy and paste" these two examples, it is more useful to type each code out, line-by-line to make sure all commands are completely understood. The goal here is only to introduce one to the general structure of a python code and to see an example of how to plot 1D functions.

Chapter 2

Types of Variables in Python

In this chapter, we will learn about some of the types of variables that exist in Python. Some of these are common to all computer programming languages, such as the integer (int), floating point (float), string (str), boolean (bool), complex (complex) etc., while some are intrinsic to Python such as the dictionary (dict), list (lst), tuple (tuple). Here, "()" indicates the conversion (or casting) function of one variable to another, if possible.

We will also introduce the Numpy module to work with arrays/vectors, matrices, and higher-dimensional tensors and their mathematical operations. Numpy will become your most-used tool in python, as it naturally captures the mathematics of linear algebra (quantum mechanics) and has simple tools for reading and writing data to files.

2.1 Built-in Data Types

Here, we will briefly introduce some of the built-in data types in python that will be heavily used in day-to-day python coding. These variables do not need to be imported in the import block through any module.

2.1.1 Integers

Everyone probably understands what an integer is from a mathematical sense from primary school. That is to say, an integer $\mathcal{Z}$ is the set of whole numbers $\{0, 1, 2, ...\}$ plus their additive inverses $\{-1, -2, -3, ...\}$. They point here is that their decimal values are filled with zeros to the right of the deciman such that, *e.g.*, $1 = 1.0000000\cdots$. In this sense, there is no new information housed within the decimal value and so one can discard this information. From a computer science perspective, this will save memory and allow on to perform simplified mathematical operations on this type of variable in contrast to those that require additional information beyond the decimal point. For example, adding two integers will be computationally faster than adding two floating point numbers (see next section) whole decimal values require additional bitwise operations behind the scenes.

2.1.2 Floating Points

Floating point numbers are real numbers (in the mathematical sense) $\mathcal{R}$ such that they are composed of all integers $\mathcal{Z}$ plus all rational and irrational numbers in between. More importantly, unlike the integers, the values to the right of the decimal need not be zero and hence are important information to be kept. For example, a floating point number could be $\pi \approx 3.14159$ (irrational) or 2.5 (rational) or 2.00000.

2.1.3 Complex Numbers

Complex numbers are number that are composed of two real numbers, a and b, such that $z = a+ib = a+jb$. The complex numbers z live in a two-dimensional space in the sense that they can only be described by two independent numbers a and b or rather as a vector (a, b) in the basis of $\{(1, 0), (0, 1)\}$. Python understands these values in the engineering notation where the imaginary unit $i \rightarrow j$ by convention where $j = \sqrt{-1}$ as usual. The complex variables, like real variables in two-dimensional space, has a magnitude z and a phase θ_z defined as,

$$|z| = \sqrt{a^2 + b^2}, \tag{2.1}$$

$$\theta_z = \arctan(\frac{b}{a}). \tag{2.2}$$

These variables can be declared in python, for example, as $z = a + b * 1j$.

2.1.4 Strings

Strings are composed of characters. One character is any symbol (including a space) that can be written in ASCII or similar. In python, strings are noted by the enclosure by quotations. In principle, any quotations are valid. For example, 'The string ASCII text' = "The string ASCII text" = "'The string ASCII text'" = """The string ASCII text""", etc. For nested strings (*i.e.*, printing strings of strings), one can make use of this, like in " I want to show you my string, which is 'This is my string'."

2.1.5 Dictionaries

Dictionaries are variables that store other variables in the same way a real dictionary stories information. Dictionaries are composed of keys and values such that when you construct a dictionary, you provide it with a set of keys and set of values. Then you can reference the dictionary passing in a key, and the result will the be corresponding value. For example, constructing the dictionary as $my\_dict = \{key : value\}$ can be referenced as $my\_dict[key]$ returning $value$. The keys and corresponding values can be any variables type; there is no restriction.

2.1.6 Lists

Similarly to dictionaries, lists are a collection of variables; however, these collections are not composed of keys, only the values. For example, defining a list as

```
my_list = [ 1 , 2.5 , "a string", 1 + 2j ]
```

and can be referenced as my_list[2] returning "a string". There is again no restriction on the types of variables that can be housed inside the list. Additionally, a list can house other lists as

```
my_list = [ 1 , 2.5 , "a string", 1 + 2j, [1,2,3,4,5] ]
```

where my_list[4] would return the list [1,2,3,4,5]. One useful tool from lists is you can dynamically update them with the append function. Defining an empty list as

```
my_list = []
```

then you can append an element as

```
my_list.append( "a string" )
```

Now the list will look like $my\_list = [\text{"}a string\text{"}]$.

2.1.7 Booleans

Boolean variables are True or False variables. By convention in python. The first letter of each is always capitalized. These variables allow on to perform comparisons of other variables in such things as if-statements, among many others. For example, True == True will evaluate to True, as 5 == 5 will evaluate to True. 1 == 2 will evaluate to False, but 1 + 1 == 2 will evaluate to True.

```python
# NAME: built_in_variable_types.py

import math
import numpy as np

def learn_integers():
    """
    In this function, we will learn about integers.
    """

    print("\n######## BEGIN INTEGERS ########\n")

    a = 10 # Store the integer value of "10" in variable "a"
           # Python will automatically know that "10" is an integer
    # To check this, use the built-in type() function
    print(f"What type is variable a ?  --> {type(a)}" )

    # What operations can we do on integers ?
    # Addition
    print(f"a = {a}  -->  a + 1 = {a + 1}    --> {type(a + 1)}" )
    # Subtraction
    print(f"a = {a}  -->  a - 1 = {a - 1}     --> {type(a - 1)}" )
    # Multiplication
    print(f"a = {a}  -->  a*5   = {a * 5}    --> {type(a * 5)}" )
```

```
    # Division
    print(f"a = {a}  -->  a/5   = {a / 5}   --> {type(a / 5)}" )
    print("  Warning! Division produced a floating point variable.")
    print("  To achieve an integer again, we need to do integer division.")
    print("  Note that if the floating point result is 1.75, int(1.75) = 1 (i.e., rounds down)")
    # Integer Division
    print(f"a = {a}  -->  a//5      = {a // 5}         --> {type(a // 5)}" )
    print(f"a = {a}  -->  int(a/5)  = {int(a / 5)}     --> {type(int(a / 5))}" )
    print()

    # Integer Power
    print(f"a = {a}  -->  a**2   = {a ** 2}    --> {type(a ** 2)}" )
    print()

    # Other operations produce floating points
    # Fractional Power (e.g., a square-root)
    print("The following types of operations necessarily produce non-integer results:")
    print(f"a = {a}  -->  a**(0.5)       = {a ** (0.5)}    --> {type(a ** (0.5))}" )
    print(f"a = {a}  -->  a**(1/2)       = {a ** (1/2)}    --> {type(a ** (1/2))}" )
    print(f"a = {a}  -->  math.sqrt(a)   = {math.sqrt(a)}    --> {type(math.sqrt(a))}" )
    print(f"a = {a}  -->  math.log[a]    = {math.log(a)}     --> {type(math.log(a))}" )

    print("\n####### END INTEGERS #######\n")

def learn_floating_points():
    """
    In this function, we will learn about floating point variables.
    """
    print("\n####### BEGIN FLOATS #######\n")

    a = 2.8 # Store the floating point value of "2.8" in variable "a"
            # Python will automatically know that "2.8" is an float
    # To check this, use the built-in type() function
    print(f"What type is variable a ?  --> {type(a)}" )

    # What operations can we do on integers ?
    # Addition
    print(f"a = {a}  -->  a + 1 = {a + 1}     --> {type(a + 1)}" )
    # Subtraction
    print(f"a = {a}  -->  a - 1 = {round(a - 1,3)}     --> {type(a - 1)}" )
    # Multiplication
    print(f"a = {a}  -->  a*5   = {a * 5}    --> {type(a * 5)}" )
    # Division
    print(f"a = {a}  -->  a/5   = {round(a / 5,3)}    --> {type(a / 5)}" )

    print("\nWhat if we need an integer from a floating point ?")
    # Convert Result to Integer
    print(f"a = {a}  -->  int(a)         = {int(a)}     --> {type(int(a))}" )
    print(f"a = {a}  -->  math.floor(a)  = {math.floor(a)}     --> {type(math.floor(a))}" )
    print(f"a = {a}  -->  math.ceil(a)   = {math.ceil(a)}     --> {type(math.ceil(a))}" )

    print("\nWhat if we only want the first few decimal places ?")
    a = math.pi
    print(f"a = {a}  -->  round(a,0) = {round(a,0)} ")
    print(f"a = {a}  -->  round(a,1) = {round(a,1)} ")
    print(f"a = {a}  -->  round(a,2) = {round(a,2)} ")
    print(f"a = {a}  -->  round(a,3) = {round(a,3)} ")
```

```
    # Other operations work as expected
    print("\nThe other operations work as expected.")
    print(f"a = {a}  -->  a**(2)         = {a ** (2)}     --> {type(a ** (2))}" )
    print(f"a = {a}  -->  a**(0.5)       = {a ** (0.5)}    --> {type(a ** (0.5))}" )
    print(f"a = {a}  -->  a**(1/2)       = {a ** (1/2)}    --> {type(a ** (1/2))}" )
    print(f"a = {a}  -->  math.sqrt(a)   = {math.sqrt(a)}    --> {type(math.sqrt(a))}" )
    print(f"a = {a}  -->  math.log[a]    = {math.log(a)}    --> {type(math.log(a))}" )

    print("\n####### END FLOATS #######\n")

def learn_complex_floats():
    """
    In this function, we will learn about complex variables.
    """
    print("\n####### BEGIN COMPLEX #######\n")

    # Two ways to make a complex
    a = complex(2,3)
    a = 2 + 3j # Store the integer value of "2 + 3j" in variable "a"
          # Python will automatically know that "2 + 3j" is complex
    # To check this, use the built-in type() function
    print(f"What type is variable a ?  --> {type(a)}" )

    # What operations can we do on integers ?
    # Addition
    print(f"a = {a}  -->  a + 1  = {a + 1}        --> {type(a + 1)}" )
    print(f"a = {a}  -->  a + 1j = {a + 1j}        --> {type(a + 1j)}" )
    # Subtraction
    print(f"a = {a}  -->  a - 1  = {a - 1}        --> {type(a - 1)}" )
    print(f"a = {a}  -->  a - 1j = {a - 1j}        --> {type(a - 1j)}" )
    # Multiplication
    print(f"a = {a}  -->  a*5    = {a * 5}      --> {type(a * 5)}" )
    print(f"a = {a}  -->  a*5j   = {a * 5j}     --> {type(a * 5j)}" )
    # Division
    print(f"a = {a}  -->  a/5    = {a / 5}    --> {type(a / 5)}" )
    print(f"a = {a}  -->  a/5j   = {a / 5j}    --> {type(a / 5j)}" )

    print("\nWhat if we only want the first few decimal places ?")
    print("Here we can invoke Numpy to do this easily for us.")
    a = math.pi + math.pi/2 * 1.0j
    print(f"a = {a}  -->  np.around(a,0) = {np.around(a,0)} ")
    print(f"a = {a}  -->  np.around(a,1) = {np.around(a,1)} ")
    print(f"a = {a}  -->  np.around(a,2) = {np.around(a,2)} ")
    print(f"a = {a}  -->  np.around(a,3) = {np.around(a,3)} ")

    # Other operations work as expected
    print("\nThe other operations work as expected.")
    print(f"a = {a}  -->  a**(2)         = {a ** (2)}   ")
    print(f"a = {a}  -->  a**(0.5)       = {a ** (0.5)} ")
    print(f"a = {a}  -->  a**(1/2)       = {a ** (1/2)} ")
    try:
        print(f"a = {a}  -->  math.sqrt(a)   = {math.sqrt(a)} ")
    except TypeError:
        print("math.sqrt(a) didn't work because a is complex.")
    try:
        print(f"a = {a}  -->  math.log[a]    = {math.log(a)} ")
```

```
    except TypeError:
        print("math.log[a] didn't work because a is complex.")


    # Complex numbers have other operations
    # Complex magnitude = |a| = sqrt[ RE**2 + IM**2 ]
    # Complex phase     = |a| = arctan [ IM / RE ]
    rad_to_deg = 180/math.pi

    my_magnitude     = math.sqrt(a.real**2 + a.imag**2)
    numpy_magnitude  = np.abs(a)

    my_angle         = math.atan(a.imag / a.real)
    numpy_angle      = np.angle(a, deg=True) # Result in degrees rather than radians
    print("\nComplex Magnitude (degrees):", round( my_magnitude,4), round(numpy_magnitude,4) )
    print("Complex Angle     (degrees):", round( my_angle * rad_to_deg,4), round(numpy_angle,4) )

    print("\n####### END COMPLEX #######\n")

def learn_strings():
    """
    The purpose of this function is to learn strings.
    """

    print("\n####### BEGIN STRING #######\n")

    # A string is a list of characters
    my_string = "Hello, user"
    print(f"My String: '{my_string}' has {len(my_string)} characters. Yes, spaces count here.")
    print(f"The eighth character is my string is '{my_string[7]}' ")
    print(f"\nWhat if we try mathematical operations on strings ? Does it make sense ?")
    print(f" my_string + my_string = '{my_string + my_string}'  --> It became a longer string.")
    print(f" my_string * 2         = '{my_string * 2}'  --> It became a longer string.")

    try:
        print(f"my_string + 2 = '{my_string + 2}'")
    except TypeError:
        print("\tmy_string + 2 does not work")
    try:
        print(f"my_string / 2 = '{my_string / 2}'")
    except TypeError:
        print("\tmy_string / 2 does not work")


    # We can check if things are in the string
    print(f"\n Is 'user' in my string ?    --> {'user' in my_string}")
    print(f" Is 'teacher' in my string ? --> {'teacher' in my_string}")

    # Check if two strings are equal
    print(''' Is "string_5" equal to "string_5" ? -->''', "string_5" == "string_5" )
    print(''' Is "string_5" equal to "not the same string" ? -->''', "string_5" == "not the same string" )

    # We can split strings based on characters
    print(f"\n my_string.split() splits based on spaces by default --> {my_string.split()} ")
    print(f" my_string.split(' ') splits based on spaces --> {my_string.split(' ')} ")
    print(f" my_string.split(',') splits based on commas --> {my_string.split(',')} ")
    print(f" my_string.split('ll') splits based on 'll' --> {my_string.split('ll')} ")
```

```
    print(" The anove variable types are called lists. We will see this later.")

    print("\n######## END STRING ########\n")

def learn_booleans():
    """
    The purpose of this function is to learn boolean variables.
    """

    print("\n######## BEGIN BOOL ########\n")

    print("Boolean variables are True or False.")

    my_true_bool  = True  # First letter MUST be capitalized. true will give an error.
    my_false_bool = False # First letter MUST be capitalized. false will give an error.

    print(f"Is True equal to False ? --> {my_true_bool == my_false_bool}")
    print(f"Is True equal to True  ? --> {my_true_bool == my_true_bool}")

    print("\nWhat about mathematical operations, again ?")
    print(f" True * 0 = {True * 0}")
    print(f" True * 1 = {True * 1}")
    print(f" True * 5 = {True * 5}")
    print(" I use this feature quite often, even though it is strange.")
    print(" It allows quickly choosing between variables given conditions.")
    print(" We will use this later in the course.")
    print(f" 5 * (1 == 2) + 10 * (2 == 2) = {5 * (1 == 2) + 10 * (2 == 2)}")

    print("\n######## END BOOL ########\n")

def learn_lists():
    """
    The purpose of this function is to learn about lists.
    """

    print("\n######## BEGIN LIST ########\n")

    my_list = [ "Hello", 5, 2.5, True, [1,2,3,4] ]
    print(" Lists are collections of variables.")
    print(f''' "[ 'Hello', 5, 2.5, True, [1,2,3,4] ]" = {my_list}''')
    print(" They can store the same type or different types of variables within them.")
    print(" Get elements from the list:")
    print(f" my_list    = {my_list} --> {type(my_list)} ")
    print(f" my_list[0] = {my_list[0]} --> {type(my_list[0])} ")
    print(f" my_list[1] = {my_list[1]}     --> {type(my_list[1])} ")
    print(f" my_list[2] = {my_list[2]}   --> {type(my_list[2])} ")
    print(f" my_list[3] = {my_list[3]}  --> {type(my_list[3])} ")
    print(f" my_list[4] = {my_list[4]}  --> {type(my_list[4])} ")
    print(" The last element is actually another list filled with integers.")
    print(" To access these 'deeper' elements, we need to access them via a second index:")
    print(f" my_list[4][0] = {my_list[4][0]}  --> {type(my_list[4][0])} ")
    print(f" my_list[4][1] = {my_list[4][1]}  --> {type(my_list[4][1])} ")
    print(f" my_list[4][2] = {my_list[4][2]}  --> {type(my_list[4][2])} ")
    print(f" my_list[4][3] = {my_list[4][3]}  --> {type(my_list[4][3])} ")
    print("\n Lists are very powerful for storing data, but their mathemetical operations are limited:")
    print(f" my_list + my_list = {my_list + my_list} --> Longer List")
```

```
    print(f" my_list * 2 = {my_list * 2} --> Longer List")
    print(" They act like strings actually.")

    print("\n Another nice feature of lists is that you can dynamically change them:")
    my_list = []
    print(f"my_list = {my_list}")
    my_list.append("1")
    print(f"my_list = {my_list}")
    my_list.append("2")
    print(f"my_list = {my_list}")
    my_list.append("3")
    print(f"my_list = {my_list}")
    print("This is useful for when you don't know the number of elements you will need.")


    print("\n####### END LIST #######\n")

def learn_dictionaries():
    """
    The purpose of this function is to learn about dictionaries.
    """

    print("\n####### BEGIN DICTIONARY #######\n")

    print(" Dictionaries store keys and values and are able to be referenced by the keys.")
    my_dict = { "key":"value", 10:"ten", "carbon":12.007, True:"I am true." }
    print( my_dict )
    print( f" my_dict['key'] = {my_dict['key']}" )
    print( f" my_dict[10] = {my_dict[10]}" )
    print( f" my_dict['carbon'] = {my_dict['carbon']}" )
    print( f" my_dict[True] = {my_dict[True]}" )
    print(" One can add keys and values dynamically:")
    my_dict["NEW_KEY"] = "NEW_VALUE"
    print( f" my_dict['NEW_KEY'] = {my_dict['NEW_KEY']}" )


    print("\n####### END DICTIONARY #######\n")

def main():
    learn_integers()
    #learn_floating_points()
    #learn_complex_floats()
    #learn_strings()
    #learn_booleans()
    #learn_lists()
    #learn_dictionaries()


if ( __name__ == "__main__" ):
    main()
```

2.2 Numpy Arrays

A very nice introduction to Numpy can be found, for free, at W3Schools – Numpy Tutorial. In this website, they walk you through multiple examples of all aspects of Numpy with an interface to run and manipulate small codes yourself. This really enables the learner to change the examples slightly and see how it changes the result – a great way to learn !

```
# numpy_arrays.py

import numpy as np
import random
from scipy.linalg import expm as SCIPY_MAT_EXP

def learn_make_numpy_array():
    """
    In this function, we will learn about creating arrays.
    """

    print("\n######## BEGIN MAKE AND COMPARE ARRAYS  ########\n")

    # This is a one-dimensional list -- a built-in data type
    my_1D_list        = [1,2,3,4]
    print(f"my_1D_list        = {my_1D_list}")

    # This converts the list to a numpy array
    #   and is usually the easiest way to create
    #   a simple numpy array
    my_1D_numpy_array = np.array(my_1D_list)
    print(f"my_1D_numpy_array = {my_1D_numpy_array}")
    print("\nObservations:")
    print("\tNote the difference in the way they are printed.")
    print("\tNumpy arrays usually do not have commas separating the values.")

    print("\nMaking two-dimensional arrays can be the same as for lists.")
    my_2D_list  = [[1,2],[3,4]]
    my_2D_array = np.array(my_2D_list)
    print(f"my_2D_list = {my_2D_list}")
    print(f"my_2D_array = \n{my_2D_array}")
    print("\nObservations:")
    print("\tNote the difference in the way they are printed.")
    print("\tNumpy arrays attempt to print themselves like matrices when possible \
            \n\t  -- because that's what they are.")

    print("\nMaking higher-dimensional arrays can be the same as for lists \
            but is more complicated to write down.")
    my_3D_list   = [ [ [1,2] ,[3,4]  ],\
                     [ [5,6] ,[7,8]  ],\
                     [ [9,10],[11,12]] ]

    my_3D_array  = np.array(my_3D_list)
    print(f"my_3D_list = {my_3D_list}")
    print(f"my_3D_array = \n{my_3D_array}")
    print("\n\tHow can we examine these better ?")
```

```
    print("\tLet's use of for-loop to print them.")
    print("\tLet's iterate over the first index:")
    shape = my_3D_array.shape # Will return (3,2,2)
    print(f"The shape of my array is: {shape}")
    for x in range(shape[0]):
        print(x, "\n", my_3D_array[x])

    print("\n\tLet's iterate over the first and second index:")
    for x in range(shape[0]):
        for y in range(shape[1]):
            print(x,y,my_3D_array[x,y])

    print("\n\tLet's iterate over the first, second, and third index:")
    for x in range(shape[0]):
        for y in range(shape[1]):
            for z in range(shape[2]):
                print(x,y,z,my_3D_array[x,y,z])

    print("\nNow, can we do the opposite ? Compose the numpy array with a for-loop ?")
    print("We first need an empty array like we needed an empty list: A = np.zeros(shape)")
    A = np.zeros( shape ) # = np.zeros( ( 3,2,2 ) )
    for x in range(shape[0]):
        for y in range(shape[1]):
            for z in range(shape[2]):
                counter = x*shape[2]*shape[1] + y*shape[1] + z # 0,1,2,3,4,...,shape[0]*shape[1]*shape[2]
                print( counter + 1 )
                A[x,y,z] = counter + 1

    print("Are the two arrays the same now ?")
    same_check = ( A == my_3D_array ).all()
    # Checks to make sure all the elements are the same.
    # For integers, it is okay.
    print( f"A == my_3D_array --> {A == my_3D_array}" )
    print( f"\n(A == my_3D_array).all() --> {(A == my_3D_array).all()}" )
    print("\n What about floats ?")
    print("Let's make a random array of VERY small numbers:")
    random_array = np.array([ [ [random.random()*10**-10    \
                                for z in range(shape[2])]  \
                                for y in range(shape[1]) ] \
                                for x in range(shape[0]) ])
    print( random_array )
    print( f"\n(A + rand == my_3D_array).all() --> {(A + random_array == my_3D_array).all()}" )
    print(f"\nCompare floats with built-in numpy function: np.allclose()")
    print( f"np.allclose(A + rand, my_3D_array) --> {np.allclose(A + random_array, my_3D_array)}" )

    print("\n####### END MAKE AND COMPARE ARRAYS #######\n")

def learn_array_math():
    """
    Here, we will learn about simple mathematical operations on numpy arrays.
    """

    print("\n####### BEGIN ARRAY MATH #######\n")

    A = np.array([1,2,3,4,5])
    print(f"A      = {A}")
```

```
print("\nArrays with numbers:")
print(f"A + 1  = {A + 1} = A + np.array([1,1,1,1,1]) = {A + np.array([1,1,1,1,1])}")
print(f"A - 1  = {A - 1}")
print(f"A * 2  = {A * 2}")
print(f"A / 2  = {A / 2}")
print(f"A // 2 = {A // 2}")

print("\nArrays with arrays:")
A = np.array([1,2,3,4,5])
B = np.array([5,5,5])
print(f"A      = {A}")
print(f"B      = {B}")
print(f"A + B --> DOES NOT WORK ! Shapes need to be the same for all operations.\n")

A = np.array([1,2,3,4,5])
B = np.array([5,5,5,5,5])
print(f"A      = {A}")
print(f"B      = {B}")
print(f"A + B  = {A + B}")
print(f"A - B  = {A - B}")
print(f"A * B  = {A * B}")
print(f"A / B  = {A / B}")

print("\nOften in QM, one encounters the needs to get differences between\n\
sets of eigenvalues (i.e., energies) of wavefunctions as matrices.")
ENERGY = np.array([1,2,3,4,5])
E_DIFF = np.subtract.outer(ENERGY,ENERGY)
print("E_DIFF = \n",E_DIFF)
print("\nSometimes, one needs to have 1/E_DIFF (e.g., non-adiabatic coupling = d_jk ~ 1/E_DIFF for j != k):\n")
print("1/E_DIFF = \n",1/E_DIFF)
print("\nWe found 'inf' terms since we divided by 0.")
print("We should have set the diagonal elements to 1.0 before inverting. Then set back to 0.0.")
E_DIFF[ np.diag_indices(len(ENERGY)) ] = 1.0 # Set diagonal elements to one before inverting.
E_DIFF_INV = 1/E_DIFF
E_DIFF_INV[ np.diag_indices(len(ENERGY)) ] = 0.0 # Return indices to zero.
print("1/E_DIFF = \n",E_DIFF_INV)
print("\nWe will make use of this stuff once we get to quantum mechanics.")

print("\nArray with complicated math:")
A = np.array([1,2,3,4,5])
print(f"A      = {A}")
print(f"e^A    = {np.exp(A)}")
print("\t-->This raised each individual element into the power of e.")
print(f"LOG[A]    = {np.log(A)}")
print("\t-->This took the natural log of each individual element.")
print("\nWhat about matrices ? Does EXP[M] make sense the way it works with vectors ?")
A = np.array([[1,2],[2,5]])
print(f"A         = \n{A}")
print(f"np.exp()  = \n{np.exp(A)}")
print("\t--> This does not make sense ! How do we do exponentials of matrices ? \
          \n\t\t--> Open mathematical question !")
print("\t--> Recall that exponentials of matrices are very common in QM: \
          \n\t\tU|\psi(0)> ~ e^(HAMILTONIAN*t)|\psi(t)>")
print("e^A = 1 + 1/2 A^2 + 1/6 A^3 + 1/24 A^4 + ... (Taylor expansion)")
print("We can either (1) (i) diagonalize the matrix, \
          \n\t\t(ii) exponentiate the diagonals (math is okay if matrix is digonal), \
          \n\t\tthen (iii) rotate back, OR,")
```

```
    print("\t(2) we can use the Pade approximation.")
    print("\n(1) Rotation to Diagonal Space")
    Av, U = np.linalg.eigh(A)
    print(f" U @ np.diag(Av) @ U.T = \n{U @ np.diag(Av) @ U.T}")
    print(f" U @ np.diag(np.exp(Av)) @ U.T = \n{U @ np.diag(np.exp(Av)) @ U.T}")

    print("\n(2) Pade Approximation")
    print(f"SCIPY_MAT_EXP(A) = \n{SCIPY_MAT_EXP(A)}" )

    print("\tYou may ask: if we know two ways to do it, why is it an open question ?")
    print("\tThe problem is that in QM, the matrices are HUGE. Here, we did 2x2.")
    print("\tFor realistic problems, one may find matrices of NxN with N ~ 10^50")
    print("\tThe Pade approximation is much faster than (i) diagonalization, (ii) exponentiation, (iii) rotation")

    #print("\n######## END ARRAY MATH #######\n")

def main():
    learn_make_numpy_array()
    #learn_array_math()

if ( __name__ == "__main__" ):
    main()
```

Chapter 3

Numerical Calculus

3.1 Differentiation

3.1.1 Analytic Differentiation

Exact differentiation.

$$\frac{d}{dx}f(x) = f'(x) = \lim_{t \to 0} \frac{f(x+t) - f(x)}{t} \quad \text{(Exact)} \tag{3.1}$$

A worked example proving Eq. 3.1.

$$\begin{aligned} f(x) &= Ax^2 \\ f'(x) &= \lim_{t \to 0} \frac{A(x+t)^2 - Ax^2}{t} && (3.2) \\ &= \lim_{t \to 0} \frac{A(x^2 + 2xt + t^2) - Ax^2}{t} && (3.3) \\ &= \lim_{t \to 0} \frac{A(2xt + t^2)}{t} && (3.4) \\ &= \lim_{t \to 0} A(2x + t) && (3.5) \\ &= 2Ax \quad \text{(This is the correct result !)} && (3.6) \end{aligned}$$

Example analytic differentials of common functions.

$$f(x) = Ax^b \rightarrow f'(x) = Abx^{b-1} = \frac{b}{x}f(x) \tag{3.7}$$

$$f(x) = Ae^{bx} \rightarrow f'(x) = b(Ae^{bx}) = bf(x) \tag{3.8}$$

$$f(x) = Ae^{-ibx} \rightarrow f'(x) = -ib(Ae^{-ibx}) = -ibf(x) = \frac{b}{i}f(x) \quad \left(\text{Note}: i = -\frac{1}{i}\right) \tag{3.9}$$

$$f(x) = A\sin(bx) = A\frac{1}{2i}\left[e^{ibx} - e^{-ibx}\right] \rightarrow f'(x) = A\frac{ib}{2i}\left[e^{ibx} + e^{-ibx}\right] = Ab\cos(bx) \tag{3.10}$$

$$f(x) = A\cos(bx) = \frac{1}{2}\left[e^{ibx} + e^{-ibx}\right] \rightarrow f'(x) = \frac{ib}{2}\left[e^{ibx} - e^{-ibx}\right] = -Ab\sin(bx) \tag{3.11}$$

$$f(x) = A\log(bx) \rightarrow f'(x) = \frac{Ab}{bx} = \frac{A}{x} \tag{3.12}$$

3.1.2 Numerical Differentiation

First-order Finite Difference Expressions

Taylor series expansion around $f(x + \Delta x)$.

$$f(x + \Delta x) = f(x) + \Delta x f'(x) + \frac{1}{2}\Delta x^2 f''(x) + \frac{1}{3!}\Delta x^3 f'''(x) + \mathcal{O}(\Delta x^4) \tag{3.13}$$

Solve for $f'(x)$ and drop all terms proportional to Δx or smaller.

$$f'(x) = \frac{f(x + \Delta x) - f(x)}{\Delta x} - \frac{1}{2}\Delta x f''(x) - \frac{1}{3!}\Delta x^2 f'''(x) + \mathcal{O}(\Delta x^4) \tag{3.14}$$

In other words, first-order finite difference expression for the first derivative of a function can be expressed as the pre-limit exact expression,

$$f'(x) = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \quad \text{(Exact)} \tag{3.15}$$

$$\approx \frac{f(x + \Delta x) - f(x)}{\Delta x} + \mathcal{O}(\Delta x) \quad \text{(Forward Difference)} \tag{3.16}$$

There is no reason why we chose to expand around $f(x + \Delta x)$. We can equivalently expand around $f(x - \Delta x)$ as,

$$f'(x) \approx \frac{f(x) - f(x - \Delta x)}{\Delta x} + \mathcal{O}(\Delta x) \quad \text{(Backward Difference)} \tag{3.17}$$

Each of these two expressions achieve an accuracy of $\mathcal{O}(\Delta x)$. Now, if we average the forward and backward expressions, we can achieve an order of magnitude higher accuracy,

$$f'(x) \approx \frac{1}{2}\left[\frac{f(x + \Delta x) - f(x)}{\Delta x} + \frac{f(x) - f(x - \Delta x)}{\Delta x}\right] \tag{3.18}$$

$$\approx \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x} + \mathcal{O}(\Delta x^2) \quad \text{(Central Difference)} \tag{3.19}$$

We now have an expression with an accuracy of $\mathcal{O}(\Delta x^2)$. This is the best we can do in first order (*i.e.*, dropping terms proportional to Δx or smaller), but we can express higher-order approximations to the first derivative by including more terms in our expansion to achieve greater accuracy.

Higher-order Finite Difference Expressions

Second-order Central Difference:

$$f'(x) \approx \frac{-2f(x-\Delta x) - 3f(x) + 6f(x+\Delta x) - 2f(x+\Delta x)}{6\Delta x} + \mathcal{O}(\Delta x^3) \tag{3.20}$$

Fourth-order Central Difference:

$$f'(x) \approx \frac{f(x-2\Delta x) - 8f(x-\Delta x) + 8f(x+\Delta x) - f(x+2\Delta x)}{12\Delta x} + \mathcal{O}(\Delta x^4) \tag{3.21}$$

Higher-order Derivatives

Manipulating Eq. 3.13, one can arrive at expressions for the second derivative as well, to varying approximation.

$$f''(x) \approx \frac{f(x) - 2f(x+\Delta x) + f(x+2\Delta x)}{\Delta x^2} + \mathcal{O}(\Delta x) \quad \text{(Forward Difference)} \tag{3.22}$$

$$f''(x) \approx \frac{f(x) - 2f(x-\Delta x) + f(x-2\Delta x)}{\Delta x^2} + \mathcal{O}(\Delta x) \quad \text{(Backward Difference)} \tag{3.23}$$

$$f''(x) \approx \frac{f(x-\Delta x) - 2f(x) + f(x+\Delta x)}{\Delta x^2} + \mathcal{O}(\Delta x^2) \quad \text{(Central Difference)} \tag{3.24}$$

$$\tag{3.25}$$

and also higher-order second derivatives using the central difference as,

$$f''(x) \approx \frac{-f(x-2\Delta x) + 16f(x-\Delta x) - 30f(x) + 16f(x+\Delta x) - f(x+2\Delta x)}{12\Delta x^2} + \mathcal{O}(\Delta x^4) \quad \text{(Second-order Central Difference)} \tag{3.26}$$

3.2 Root-finding Algorithms

Often, we need to find the roots (or zeros) of functions. For example, when performing a geometry optimization using some quantum chemical software, the goal is to minimize the energy with respect to the change in nuclear positions. A minimum is nothing more than the zero of its first derivative function.

3.2.1 Bisection Method

The simplest of all root-finding schemes is the bisection method. This method does not use the derivative of the function at all and simply provides a scheme based on the intermediate value theorem.

Intermediate value Theorem

If $f(a)*f(b) < 0$ (*i.e.*, $f(a)$ and $f(b)$ have opposite signs), then there exists a root ($f(x_0) = 0$) between a and b in the function $f(x)$.

Algorithm

1. Define convergence criteria, $\epsilon = 10^{-N}$ (*e.g.*, $N = 15$)
2. Choose interval $\mathcal{I} \in [a_0, b_0]$ such that $f(a_0)f(b_0) < 0$ (*i.e.* they have opposite signs).

   ######## BEGIN FOR-LOOP (n) ########
3. Calculate the midpoint of the interval: $c_n = \frac{a_n+b_n}{2}$
4. Check convergence:
   (a) If $f(c_n)$ is a root within defined convergence ($|f(c_n)| < \epsilon$), then the root is $x_0 = c_n$ and exit the loop.

   Or, a more robust criterion is,
   (b) If the length of the interval is the same order of magnitude as the convergence criterion $|b_n - a_n| \leq \epsilon$, then the root is $x_0 = \frac{a_n+b_n}{2}$ and exit the loop.
5. If there is no root within convergence, bisect the interval:
   (a) If $f(a_n)f(c_n) > 0$ (*i.e.* they have the same sign), the new interval is $\mathcal{I} \in [a_{n+1}, b_{n+1}] = [c_n, b_n]$.
   (b) If $f(a_n)f(c_n) > 0$ (*i.e.* they have the opposite signs), the new interval is $\mathcal{I} \in [a_{n+1}, b_{n+1}] = [a_n, c_n]$.
6. Return to step 2.

   ######## END FOR-LOOP (n) ########

Alternatively, one can simply run this loop for a fixed number of iterations instead of checking for the convergence explicitly.

What are some of the limitations to this method ? (I) One needs to know *a priori* an interval where the minimum will be found. Can you think of any others ?

3.2.2 Newton-Raphson

The Newton-Raphson method is a more robust method in the sense that it incorporates the derivative of the function into the scheme and has a faster rate of convergence compared to the bisection method. However, this method is not guaranteed to converge to a root if the starting point is not in the neighborhood of a root.

$$y = f(x_n) + f'(x_n) * (x - x_n) \qquad \text{(Tangent Line at x = } x_n\text{)} \tag{3.27}$$

The root of this line can be easily defined exactly the point when $y = 0$. Solving for this value, the expression for the next approximation for the root is,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{3.28}$$

Choosing a convergence criterion based on the interval size,

$$\frac{|x_{n+1} - x_n|}{|x_n|} \leq \epsilon, \tag{3.29}$$

now defines the algorithm. One simply replaces the x_{n+1} and convergence criterion in the bisection procedure. This scheme is great if one knows the analytic expression for the derivative of the function $f(x)$. However, often one does not know this and must be approximated via finite difference expressions, for example, Eq. 3.18.

Secant Method (1$^{\text{st}}$-order Newton-Raphson)

The secant method is simply the Newton-Raphson approach with a first-order numerical approximation to the first derivative. This can be written as,

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}, \quad f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}} \tag{3.30}$$

Algorithm

1. Define convergence criteria, $\epsilon = 10^{-N}$ (*e.g.*, $N = 15$)
2. Choose starting position, x_0.

   ######## BEGIN FOR-LOOP (n) ########
3. Calculate the next position: $x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$
4. Check convergence:
   (a) If $\frac{|x_{n+1} - x_n|}{|x_n|} \leq \epsilon$ is satisfied, exit loop with root $r_0 = \frac{x_{n+1} - x_n}{2}$.
   (b) Else, return to step 3 with $x_n = x_{n+1}$

   ######## END FOR-LOOP (n) ########

3.2.3 Using External Modules: SciPy and Numpy

Python contains many modules that can help with differentiation and optimization of functions. SCIPY.OPTIMIZE offers many powerful algorithms for optimizing multivariate functions. Further, for general differentiation of functions, NUMPY.GRADIENT offers a simple way to approximate derivatives for multivariate.

3.3 Integration

3.3.1 Analytic Integration

The definite integral I of a function $f(x)$ over the range $[a, b]$ is defined as,

$$I = \int_a^b dx f(x) = F(b) - F(a), \tag{3.31}$$

where $F(x)$ is the indefinite integral of the function $f(x)$. A simple example would be,

$$I = \int_a^b dx A x^n = A\frac{1}{n}x^{n+1}\Big|_a^b = A\frac{1}{n}\left[b^{n+1} - a^{n+1}\right] \tag{3.32}$$

3.3.2 Numerical Integration

To obtain a useful approximation to this integral, we can Taylor expand it,

$$\begin{aligned}
I &= \int_x^{x+\Delta x} d\xi\ f(\xi) && (3.33)\\
&= F(x+\Delta x) - F(\xi) && (3.34)\\
&= [F(\xi) + \Delta x F'(x) + \frac{1}{2}\Delta x^2 F''(x) + \frac{1}{6}\Delta x^3 F'''(x) + \mathcal{O}(\Delta x^4)] - F(\xi) && (3.35)\\
&= \Delta x F'(x) + \frac{1}{2}\Delta x^2 F''(x) + \frac{1}{6}\Delta x^3 F'''(x) + \mathcal{O}(\Delta x^4) && (3.36)\\
&= \Delta x f(x) + \frac{1}{2}\Delta x^2 f'(x) + \frac{1}{6}\Delta x^3 f''(x) + \mathcal{O}(\Delta x^4). && (3.37)
\end{aligned}$$

From here, we can factor out a common $\frac{1}{2}\Delta x$ to get,

$$I = \frac{1}{2}\Delta x[2f(x) + \Delta x f'(x) + \frac{1}{3}\Delta x^2 f''(x) + \mathcal{O}(\Delta x^3)]. \tag{3.38}$$

This result (in the brackets) looks a lot like a Taylor series expansion of $[f(x)+f(x+\Delta x)] = 2f(x) + \Delta x f'(x) + \frac{1}{2}\Delta x^2 f''(x) + \mathcal{O}(\Delta x^3)$. We can multiply this expression (on both sides) by $\Delta x/2$ to obtain,

$$\int_x^{x+\Delta x} d\xi\ f(\xi) \approx \frac{1}{2}\Delta x[f(x) + f(x+\Delta x)]. \quad \text{(Trapazoidal Rule)} \tag{3.39}$$

We can approximate this further by considering only one of the two points to obtain the most basic integration technique (Riemann) as,

$$\int_x^{x+\Delta x} d\xi\ f(\xi) \approx f(x)\Delta x. \quad \text{(Riemann Sum)} \tag{3.40}$$

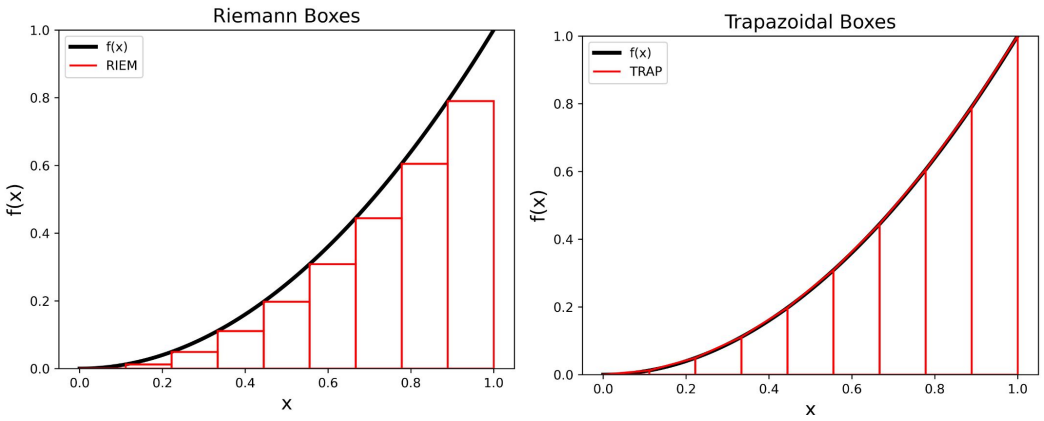

Figure 3.1: (left) Riemann sum boxes and (right) trapezoidal boxes for the integration of $f(x) = x^2$ from 0 to 1.

Chapter 4

Fourier Analysis

Fourier analysis is a ubiquitous tool necessary to understand waves and spectral quantities. Simply, it is method to "decompose" or identify the components of the "inverse space" that make up the original function. By "inverse space", I can mean a few things. The most obvious thing is the conversion between real space and reciprocal space (*i.e.*, k-space), which arises is solid state physics where reciprocal lattice vectors define the crystal structure. Another example, maybe even more useful for a broad range of application (especially in electrical engineering), is that of the time-frequency conversion, where the frequencies of a time-series (*e.g.*, time-dependent voltage in a circuit, time-dependent population in molecular dynamics simulations, or simply the frequencies that comprise a waveform). In this chapter, we will use these ideas and examples to understand how to implement such a tool in the computer to decompose various waves or, in general, arbitrary functions in Python.

4.1 Fourier Transform

The most important definitions for this chapter are the continuous Fourier transforms between one variable (*i.e.*, x) and its reciprocal (*i.e.*, k). The forward Fourier transform is defined as,

$$\phi(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} dx\ \psi(x)\ e^{-ikx} \quad \text{(Forward)} \tag{4.1}$$

which maps the real-space function $\psi(x)$ onto the reciprocal space function $\phi(k)$. The inverse of this transformation is obtained as,

$$\psi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} dk\ \phi(k)\ e^{ikx} \quad \text{(Backward)}. \tag{4.2}$$

One can directly understand this transformation in the context of Chapter 3, since the Fourier transform is simply a one-dimensional integral of a function. Here, the integral function is a multiplication of two functions, $\psi(x)e^{ikx}$, where $\psi(x)$ is the function of interest and e^{ikx} is the function that describes a single plane wave of frequency (or rather

wavevector) k as a function of position x. In a way, and as we will discuss later when we are doing quantum mechanics, this function is nothing more than a *projection* or *overlap* of the function $\psi(x)$ onto the function e^{ikx} which answers the following question: "How much of e^{ikx} is in $\psi(x)$, for a particular k ?". Some keywords that help to describe this are *overlap*, *projection*, *inner product*, *basis set transformation*, all of which we will discuss later.

Some useful examples of this transformation are as follows:

$$\psi(x) = \delta(x) \rightarrow \phi(k) = \frac{1}{\sqrt{2\pi}} \quad \text{(All possible k)} \tag{4.3}$$

$$\psi(x) = e^{ik_0 x} \rightarrow \phi(k) = \frac{1}{\sqrt{2\pi}}\delta(k - k_0) \quad \text{(A single k)} \tag{4.4}$$

$$\psi(x) = \cos(k_0 x) = \frac{1}{2}\left[e^{ik_0 x} + e^{-ik_0 x}\right] \tag{4.5}$$
$$\rightarrow \phi(k) = \frac{1}{\sqrt{2\pi}}\left[\delta(k - k_0) + \delta(k + k_0)\right] \quad \text{(Two k's)}$$

$$\psi(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \rightarrow \phi(k) = \frac{1}{\sqrt{2\pi}} e^{-2\sigma^2 k^2} \tag{4.6}$$
(Gaussian transforms to Gaussian)

One very important note on the $1/\sqrt{2\pi}$ normalization factor on the Fourier transform (Eq. 4.1). This is completely arbitrary. In fact, many other places you will find that the normalization is asymmetric in the sense that for the forward transform (Eq. 4.1) has a normalization factor of 1 and the reverse (Eq. 4.2) has a normalization factor of $1/(2\pi)$. Sometimes, that is reversed and the forward transform (Eq. 4.1) has a normalization factor of $1/(2\pi)$ and the reverse (Eq. 4.2) has a normalization factor of 1. Throughout our work in this Python course, we will always use a symmetric definition of the Fourier transform such that the normalization is $1/\sqrt{2\pi}$ for both the forward and backward transformation.

4.2 Approximate/Discrete Fourier Transformation (DFT)

To implement such as transformation in the computer, one needs to perform a similar approximation as was done in Chapter 3 when discretizing the integral. To a first approximation, one can simply perform a Riemann summation as,

$$\phi(k_m) \approx \frac{1}{\sqrt{2\pi}} \sum_{j=0}^{N} \Delta x \; \psi(x_j) \; e^{-ik_m x_j} \quad \text{(Forward)}, \tag{4.7}$$

and the backward as,

$$\psi(x_j) \approx \frac{1}{\sqrt{2\pi}} \sum_{m=0}^{N} \Delta k \; \phi(k_m) \; e^{ik_m x_j} \quad \text{(Backward)}. \tag{4.8}$$

There are some restrictions on the values of k_n. In fact, given the number of points N and the value of the real-space spacing Δx, it completely determines the reciprocal space grid as follows:

$$\Delta x \Delta k = \frac{2\pi}{N}. \tag{4.9}$$

In this way, we can also write down the maximum possible value of the reciprocal grid as,

$$k_{\max} = \frac{\pi}{\Delta x} \quad \text{(Angular Frequency/Wavevector)} \tag{4.10}$$

$$\rightarrow \frac{k_{\max}}{2\pi} = \frac{1}{2\Delta x} \quad \text{(Linear Frequency/Wavevector)} \tag{4.11}$$

It is more convenient to rewrite these expressions such that the exact values of x_n and k_m are not needed. In this case, we would call it the dimensionless Fourier transform. We would write it as,

$$\phi(k_m) \approx \frac{\Delta x}{\sqrt{2\pi}} \sum_{j=0}^{N} \psi(x_j)\, e^{-i2\pi \frac{mj}{N}} \qquad \text{(Forward)}, \tag{4.12}$$

$$\psi(x_j) \approx \frac{\Delta k}{\sqrt{2\pi}} \sum_{m=0}^{N} \phi(k_m)\, e^{i2\pi \frac{mj}{N}} \quad \text{(Backward)}, \tag{4.13}$$

which now does not depend on the actual grid at all, only the number of grid points (which is very interesting to me!). This implies that the Fourier transform does not care about the values of your grid, only the number of them! (This is weird, right ?)

Further, it turns out that we can rewrite this operation as a simple matrix multiplication. In the second homework regarding Numpy arrays, we constructed the discrete Fourier transform (DFT) matrix $\mathbf{W}$, which is written as,

$$\mathbf{W} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \gamma & \gamma^2 & \gamma^3 & \cdots & \gamma^{N-1} \\ 1 & \gamma^2 & \gamma^4 & \gamma^6 & \cdots & \gamma^{2(N-1)} \\ 1 & \gamma^3 & \gamma^6 & \gamma^9 & \cdots & \gamma^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \gamma^{N-1} & \gamma^{2(N-1)} & \gamma^{3(N-1)} & \cdots & \gamma^{(N-1)(N-1)} \end{bmatrix}, \tag{4.14}$$

where $\gamma = e^{-i2\pi/N}$. This matrix is the same as that of the plane-wave (exponential) in Eq. 4.12 for all possible indices m and j such that the mj^{th} element of $\mathbf{W}$ can be written as $e^{-i2\pi mj/N}$. In python, this can be easily achieved by the following couple lines of code,

```
### Define the Fourier matrix, W
m = np.arange(N).reshape( (-1,1) )
j = np.arange(N).reshape( (1,-1) )

### DFT Matrix
W = np.exp( -1j*2*np.pi * m * j / N )
```

With this in hand, we can simply perform a matrix multiplication of the original function $\psi(x_j)$ evaluated on the discrete grid with the DFT matrix $\mathbf{W}$ as,

$$\phi(\vec{k}) = \mathbf{W} \times \psi(\vec{x}) \tag{4.15}$$

which makes sense because matrix multiplication involves a summation of the rows for each column multiplication. The m^{th} element of $\phi(\vec{k})$ can be written as,

$$\phi(k_m) = \sum_j W_{mj}\ \psi(x_j) \tag{4.16}$$

These two equivalent statements can be written in python as follows:

```
### Matrix Multiplication (Eq. 4.15)
f_k = W @ f_x * dx

### Direct summation (Eq. 4.16)
f_x = f_x.reshape( (-1,1) )
f_k = np.sum( W[:,:] * f_x[:,:], axis=0 ) * dx
```

where "dx" = Δx is included for the integration width (or infinitesimal). One could also simply include this in the definition of the DFT matrix $\mathbf{W}$ for simplicity.

4.2.1 Centered DFT

While the DFT we constructed is useful, it will produce spurious results in the sense that only the absolute value of the complex function will be correct. In many cases, one will find that the real and imaginary parts of the inverse function (*i.e.*, $\phi(k)$) will be highly oscillatory when we expect that there is only real output without oscillations. To fix this, we will make a shift to the exponential operator by half the grid size to define the so-called "centered DFT". In this case, the matrix elements of the DFT matrix $\mathbf{W}$ can be written as,

$$W_{mj} = e^{-i2\pi \frac{(m-a)(n-a)}{N}}, \tag{4.17}$$

where $a = N/2$ if N is even and $(N-1)/2$ is N is odd. For $N >> 1$, $N \approx N-1$ and can be ignored with simple integer division as $a = N//2$. In this case, the phase of the resulting transform will be correct. A good example of such a correction is the DFT of the standard Gaussian function (see Fig. 4.1). With the one-sided DFT (original, un-shifted formulation), there exists highly oscillatory motion in the real component with non-zero imaginary parts. However, this is completely eliminated by performing the centered DFT. The simplest first example would be the Fourier transform of a sinusoidal function, which should have one frequency (with both positive and negative components!). See Fig. 4.2.

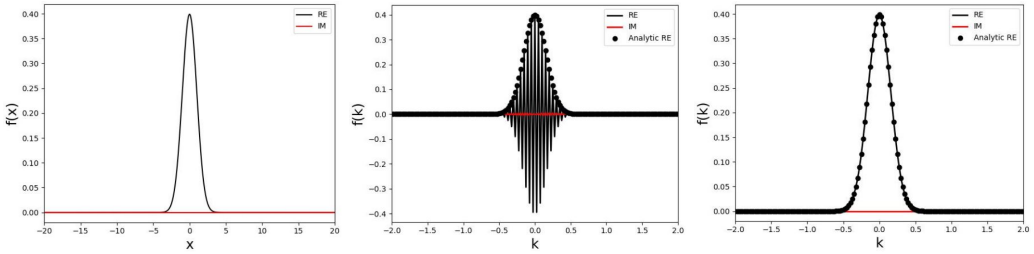

Figure 4.1: Generated from the code "1_Fourier_Transform_on_a_Discrete_Grid.py". (Left) $f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$. (Middle, Right) Numerical discrete Fourier transform of f(x) using the (Middle) one-sided, Eq. 4.17, and (Right) centered DFT expressions, Eq. 4.12. The analytic Fourier transform is shown as black circles.

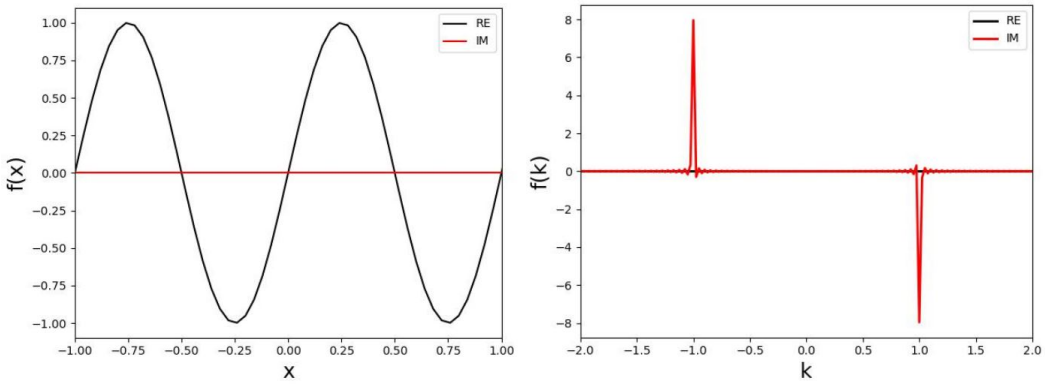

Figure 4.2: Generated from the code "1_Fourier_Transform_on_a_Discrete_Grid.py". (Left) $f(x) = sin(2\pi x)$. (Right) Numerical discrete Fourier transform of f(x).

4.3 Useful Fourier Tricks

4.3.1 Gaussian Transforms to a Gaussian

Gaussian functions have a unique property when undergoing Fourier transforms. In fact, they transform to themselves (see Fig. 4.3). This can be stated (without proof) as,

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \rightarrow f(k) = \frac{1}{\sqrt{2\pi}} e^{-2\sigma^2 k^2 \pi^2}. \tag{4.18}$$

4.3.2 Fourier Shift Theorem

There exists a very nice "trick" when working with Fourier transforms called the Fourier shift theorem. It is a simple result stemming from complex exponential functions. In essence, if one performs a shift on the real-space function such that $f(x) \rightarrow f(x - x_0)$, then the resulting

Fourier transform contains an additional *phase factor* that mixes the real and imaginary parts while retaining the overall amplitude of the function (see Fig. 4.4).

$$f(x) \to f(x - x_0) \implies f(k) \to f(k)e^{-ikx_0} \tag{4.19}$$

The amplitude is retained since,

$$|f(k)| \to |f(k)e^{ikx_0}| = |f(k)|\,|e^{ikx_0}| = |f(k)|. \tag{4.20}$$

4.3.3 "Reverse" Fourier Shift Theorem

In the opposite sense, if one adds a complex phase factor to the real-space function, then one obtains a momentum shift in the reciprocal space. This can be understood as,

$$f(x) \to f(x)\, e^{-ik_0x} \implies f(k) \to f(k - k_0). \tag{4.21}$$

This amplitude is also conserved using the same reasoning since $|e^{ik_0x}| = 1$ for all x.

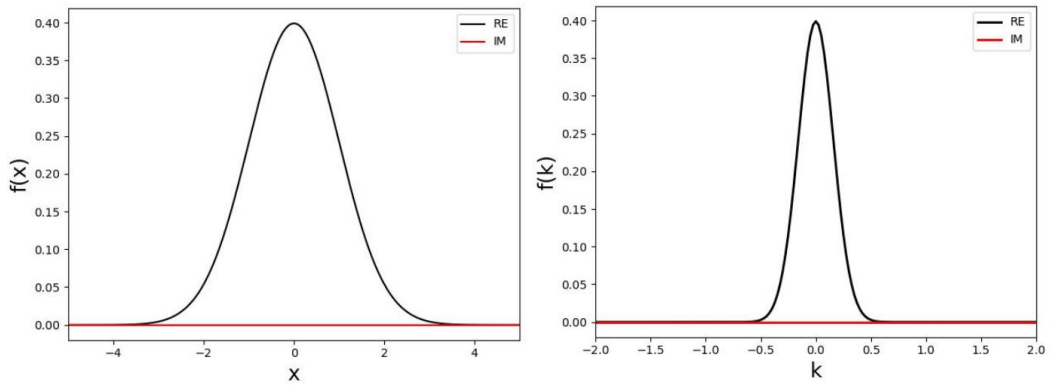

Figure 4.3: Generated from the code "1_Fourier_Transform_on_a_Discrete_Grid.py". (Left) $f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$. (Right) Numerical discrete Fourier transform of f(x).

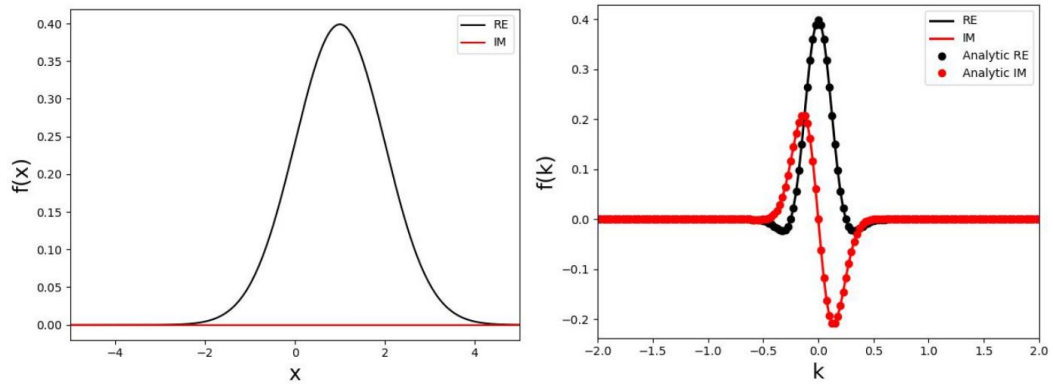

Figure 4.4: Generated from the code "1_Fourier_Transform_on_a_Discrete_Grid.py". (Left) $f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{(x-x_0)^2}{2}}$, where $x_0 = 1$. (Right, solid lines) Numerical discrete Fourier transform of f(x). (Right, circles) Analytic Result: $f(k) = \mathcal{F}[f(x-x_0)] = \mathcal{F}[f(x)]e^{-i2\pi k x_0}$, where $x_0 = 1$.

4.4 Differentiation Using the Fourier Transform

One special property of the Fourier transforms that will become important later is relationship between the Fourier transform $\mathcal{F}[f(x)] = \widetilde{f}(k)$ and the derivative $\frac{\partial}{\partial x}$. The act of Fourier transformation on a differential operator is one that maps the differential to an algebraic operator, *i.e.*, $\frac{\partial}{\partial x}\, f(x) = \mathcal{F}^{-1}[ik\, f(k)]$, often simplifying the calculation substantially, especially for highly oscillatory functions.

Recall that the inverse Fourier transform of $\widetilde{f}(k)$ is,

$$f(x) = \frac{1}{\sqrt{2\pi}} \int dk\ \widetilde{f}(k)\, e^{ikx} \quad \text{(Same as Eq. 4.2)}. \tag{4.22}$$

Differentiating both sides and simplifying gives,

$$\frac{\partial}{\partial x} f(x) = \frac{1}{\sqrt{2\pi}} \int dk\ \widetilde{f}(k) \left[\frac{\partial}{\partial x}\, e^{ikx}\right] \tag{4.23}$$

$$= \frac{1}{\sqrt{2\pi}} \int dk\ \widetilde{f}(k) \left[ik\, e^{ikx}\right] \tag{4.24}$$

$$= \frac{1}{\sqrt{2\pi}} \int dk \left[ik\widetilde{f}(k)\right] e^{ikx}, \tag{4.25}$$

where the factor ik represents the differential operator in momentum space which is now a linear algebraic operator. One can easily generalize this to arbitrary order of derivative n as,

$$\frac{\partial^n}{\partial x^n} f(x) = \frac{1}{\sqrt{2\pi}} \int dk \left[(ik)^n \widetilde{f}(k)\right] e^{ikx}. \tag{4.26}$$

4.4.1 Example: Oscillatory Real-space Function

To understand why this is useful, let's consider a perfectly oscillatory function, *i.e.*, $f(x) = \cos(k_0 x)$, which is generally one of the more difficult types of functions for numerical differentiation to achieve accurate results. The analytic Fourier transform can be written as,

$$f(x) = \cos(k_0 x) \rightarrow \widetilde{f}(k) = \frac{1}{\sqrt{2\pi}} \Big[\delta(k + k_0) + \delta(k - k_0)\Big]. \tag{4.27}$$

We can then apply the second derivative operator to the function and write the analytic result from introductory calculus, followed by the derivative via Fourier transform as,

$$\frac{\partial^2}{\partial x^2} f(x) = -k_0^2 \cos(k_0 x) \quad \text{(Direct Differentiation)} \tag{4.28}$$

$$= \frac{1}{\sqrt{2\pi}} \int dk \left[(ik)^2 \widetilde{f}(k)\right] e^{ikx} \quad \text{(Via Fourier Transform)} \tag{4.29}$$

$$= \frac{1}{\sqrt{2\pi}} \int dk \left[(ik)^2 \Big(\delta(k + k_0) + \delta(k - k_0)\Big)\right] e^{ikx}$$

$$= \frac{i^2}{\sqrt{2\pi}} \int dk \left[k^2 \delta(k + k_0) + k^2 \delta(k - k_0)\Big)\right] e^{ikx}$$

$$= -\left[(-k_0)^2 e^{i(-k_0)x} + (k_0)^2 e^{ik_0 x}\right] \text{ Note : } \mathcal{F}[\delta(k - k_0)] \sim \sqrt{2\pi}$$

$$= -k_0^2 \left[e^{-ik_0 x} + e^{ik_0 x}\right]$$

$$= -k_0^2 \cos(k_0 x)$$

Chapter 5

Differential Equations and Time Evolution

5.1 Linear First-order ODEs

Consider the following class of first-order differential equations,

$$\frac{dy}{dt} \equiv y'(t) \equiv \dot{y}(t) = f(y(t), t), \quad y(0) = y_0, \tag{5.1}$$

where the first three equalities are often used interchangeably to denote a first-order time derivative. Here, $y(0)$ is the initial value (*i.e.*, at time zero) of the goal function $y(t)$. $f(y(t), t)$ is any function of the goal function $y(t)$ and the independent variable t. Sometimes, $f(y(t), t)$ is called the "driving" function because it controls how $y(t)$ changes in time since it is equal to the derivative of $y(t)$ (*i.e.* the rate of change of $y(t)$) with respect to time, $y'(t)$. For example, if $f(y(t), t) = 0$, then there is no time-dependence of the function $y(t)$.

5.1.1 Euler Method

Forward Euler (Explicit)

If we consider the first-order forward approximation for differentiation (see Sec. 3.1) for $\frac{dy(t)}{dt}$ in Eq. 5.20, we can rewrite the left-hand side as,

$$\frac{y(t + \Delta t) - y(t)}{\Delta t} \approx f(y(t), t), \tag{5.2}$$

upon rearranging leads to,

$$y(t + \Delta t) \approx y(t) + \Delta t * f(y(t), t). \text{ (Forward Euler)} \tag{5.3}$$

This provides the forward propagation $t \to t + \Delta t$ of the primary function $y(t) \to y(t + \Delta t)$, given knowledge of the previous time and is usually referred to as the "Forward Euler"

approach and is accurate to $\mathcal{O}(\Delta t)$. The numerical version of such a propagation would look like,

$$y_{n+1} \approx y_n + \Delta t * f(y_n, t_n), \tag{5.4}$$

where t_n labels the numerical time index and y_n labels the value of the function after n steps. This method is said to be "explicit" since all information needed for the propagation is contained in already-known quantities, such as the value of the function at the previous time step.

5.1.2 Backward Euler (Implicit)

To construct an implicit version of the Euler method, one starts with the backward approximation (*i.e.*, $\frac{y(t)-y(t-\Delta t)}{\Delta t} \approx f(y(t), t)$) for the derivative in Eq. 5.20 and results in,

$$y_{n+1} = y_n + \Delta t * f(y_{n+1}, t_{n+1}), \tag{5.5}$$

where the difference from Eq. 5.4 is shown in red. In this case, the time-propagation needs knowledge of the time-propagated function y_{n+1} in order to find the time-propagated function y_{n+1}. This evidences the name of this type of propagation, "implicit". To solve for implicit propagation schemes, one needs to couple them with a root-finding approach, such as the Newton-Raphson method (see Sec. 3.2).

Example: Backward Euler

Consider the differential equation,

$$\frac{dy(t)}{dt} = e^{-y(t)}, \qquad y(0) = 0. \tag{5.6}$$

The backwards Euler equations lead to,

$$y_{n+1} \approx y_n + \Delta t * e^{-y_{n+1}}, \qquad y_0 = 0. \tag{5.7}$$

We will solve the implicit equation using the Newton-Raphson root-finding method. Recall that root-finding methods search for zeros to equations, so we need to move everything to one side in order to start, which, for our case, will look like,

$$0 \approx y_n - y_{n+1} + \Delta t * e^{-y_{n+1}} \equiv g(y_{n+1}). \tag{5.8}$$

Here, we define the "zeros" function as $g(y_{n+1})$. We also need to know the first derivative of the "zeros" function $g(y_{n+1})$ with respect to y_{n+1} to use the Newton-Raphson approach (*e.g.*, $y_{n+1} = y_n + \frac{g(y_n)}{g'(y_n)}$), which can be written as,

$$\frac{dg(y_{n+1})}{dy_{n+1}} = -1 - \Delta t * e^{-y_{n+1}}. \tag{5.9}$$

Altogether, the Newton-Raphson iterative scheme can be written simply as,

$$y_{n+1}^{j+1} = y_{n+1}^{j} + \frac{y_n - y_{n+1}^{j} + \Delta t * e^{-y_{n+1}^{j}}}{-1 - \Delta t * e^{-y_{n+1}^{j}}}, \tag{5.10}$$

where the label j indicates the Newton-Raphson iteration. In the case of differential equations, the root-finding algorithm is stopped after a set number of iterations to save time. This is in contrast to what was done in Sec. 3.2, since there the accuracy of the problem was only dictated by the accuracy of the root-finding algorithm itself. Here, the accuracy is determined by two iterative schemes – (I) the time-propagation and (II) root-finding – hence more expensive to converge than only a single iterative scheme. We will see later that the Euler propagation is usually never used since it converges very slowly with numerical timestep.

While both the forward and backward Euler methods result in the same accuracy, a major increase in stability is achieved with the backward Euler scheme. In most cases, for simple differential equations, the explicit methods work well enough. However, for highly oscillatory functions or ones which change rapidly, the implicit methods will converge the results much more efficiently (but in principle to the same accuracy) than the analogous explicit method. In other words, sometimes the explicit schemes will diverge to infinity unless a sufficiently small time step is used, and this is somewhat corrected by the implicit scheme – at the cost of the root-finding iterations.

5.1.3 Leap-frog Method

We now move to the next level of accuracy. Here, the key is to start from the central difference formula, $y'(t) \approx \frac{y(t+\Delta t)-y(t-\Delta t)}{2\Delta t}$, for the derivative to obtain,

$$y_{n+1} = y_{n-1} + 2\Delta t * f(y_n, t_n), \tag{5.11}$$

where the key change is that we look to both the previous step n as well as the one before it $n-1$ to calculate the next step $n+1$. This appraoch is accurate to second order in the timestep (*i.e.*, $\mathcal{O}(\Delta t^2)$). This propagation scheme is explicit because the following step y_{n+1} does not rely on itself but rather on previous information: y_{n-1} and y_n. The name of the method arises because of the "leaping over" the step n from $n-1$ to get to $n+1$.

The key difference between the codes of forward Euler and leap frog is that one needs the y_0 and y_1 values to initiate the leap-frog algorithm, since both are required to calculate y_2, which is the first iteration result. To achieve this, one simply performs a single-step of the forward Euler method to get y_1.

5.1.4 Runge-Kutta Method

In contrast to the previous few methods (*i.e.*, forward/backward Euler and leap-frog), which are all "single-step" propagation methods in the sense that they only require a single new calculation per time-step in order to propagate to the next step. Note that the leap-frog method seems like it needs two, y_{n-1} and y_n. However, the y_{n-1} value is re-used from the previous iteration and so only a single additional value was needed. In the next class of

methods, two or more calculations will be required in order to propagate to the next step, which we will call intermediate variables $\mathbf{k}$.

Two-step Runge-Kutta (RK2)

The leap-frog method was derived using the central difference formula using the points $n = (-1, 0, 1)$. Let's shift this slightly and consider the leap-frog method using the points $n = (0, \frac{1}{2}, 1)$, which can be considered as the midpoint of the interval. This can be written as,

$$\frac{y(t+\Delta t) - y(t)}{\Delta t} = f\left(y(t + \frac{1}{2}\Delta t), t + \frac{1}{2}\Delta t\right) \tag{5.12}$$

$$\implies y(t+\Delta t) = y(t) + \Delta t * f\left(y(t + \frac{1}{2}\Delta t), t + \frac{1}{2}\Delta t\right). \tag{5.13}$$

However, now we need to know the value of $y(t + \frac{1}{2}\Delta t)$, which we can get from a half-step Euler propagation as,

$$y(t + \frac{1}{2}\Delta t) \approx y(t) + \frac{1}{2}\Delta t * f(y(t), t). \tag{5.14}$$

Putting it all together gives the two-step Runge-Kutta method:

$$y(t+\Delta t) \approx y(t) + \Delta t * f\left(y(t) + \frac{1}{2}\Delta t * f(y(t), t), t + \frac{1}{2}\Delta t\right). \tag{5.15}$$

We can rewrite this propagation using intermediate variables k as,

$$y_{n+1} \approx y_n + k_2 \Delta t \tag{5.16}$$

$$k_1 = f(y_n, t_n)$$

$$k_2 = f\left(y(t) + \frac{1}{2}\Delta t k_1, t + \frac{1}{2}\Delta t\right) \tag{5.17}$$

Fourth-order Explicit Runge-Kutta (RK4)

One of the most common forms of Runge-Kutta methods is the four-step version. Immediately writing in the form of intermediate variables leads to,

$$y_{n+1} \approx y_n + \frac{1}{6}\Delta t(k_1 + 2 * k_2 + 2 * k_3 + k_4) \tag{5.18}$$

$$k_1 = f(y_n, t_n)$$

$$k_2 = f\left(y_n + \frac{\Delta t}{2} k_1, t_n + \frac{1}{2}\Delta t\right)$$

$$k_3 = f\left(y_n + \frac{\Delta t}{2} k_2, t_n + \frac{1}{2}\Delta t\right)$$

$$k_4 = f\left(y_n + \ \Delta t k_3, t_n + \ \Delta t\right). \tag{5.19}$$

This will be the focus of our attention due to its relative simplicity and ease of implementation. However, note that one can choose any arbitrary order of Runge-Kutta methods (RKN) to achieve arbitrary accuracy, whose error goes like $\mathcal{O}(\Delta t^{N_{\text{steps}}+1})$. Further, there exists an analogous implicit Runge-Kutta method at each order, which again requires a root-finding algorithm to be employed but again has increased stability during the time-propagation.

5.2 Linear Second-order ODEs

Consider the following class of second-order differential equations,

$$\frac{d^2y}{dt^2} = f(y(t), y'(t), t), \quad y(0) = y_0, \quad y'(0) = v_0, \tag{5.20}$$

Here, $y(0)$ is the initial value (*i.e.*, at time zero) of the goal function $y(t)$, and $y'(t)$ is the initial value of the goal function's first derivative (*e.g.*, initial velocity). $f(y(t), y'(t)t)$ is any function of the goal function $y(t)$, its first derivative $y'(t)$, and the independent variable t. Sometimes, $f(y(t), y'(t)t)$ is called the "driving" function because it controls how $y(t)$ changes in time since it is equal to the second derivative of $y(t)$ (*i.e.*, the acceleration of $y(t)$) with respect to time, $y''(t)$. For example, if $f(y(t), y'(t)t) = 0$, then there is no time-dependence of the function $y'(t) = y'(0)$ and linear time-dependence of the function $y(t) \sim y(0) + y'(0) * t$.

5.2.1 Newton's/Hamilton's Equations for Classical Dynamics

As an example of the most prominent way second-order ODEs arise and are solved, we turn to classical dynamics. Here, the first thing that should come to mind is Newton's equations, or equivalently, Hamilton's equations. To recall, Newton's equation's relates the acceleration of a particle directly to its force as,

$$m\frac{d^2x(t)}{dt^2} = F(x(t), x'(t), t), \tag{5.21}$$

where F is the force, which can, in principle depend on the position $x(t)$, the velocity $v(t) \equiv x'(t) \equiv \frac{dx(t)}{dt}$, and time itself t. However, often the force only involves the position itself.

Hamilton's equations are often the most useful to use when implementing into a computer. The idea to go from Newton's equation to Hamilton's equations is as follows: Solving one second-order equation (such as Newton's equation) is harder than solving two, coupled, first-order equations. By that, I mean we can split Newton's equation into two equations of first-order but with variables shared by both. Hamilton's equation can be written as,

$$\frac{dx(t)}{dt} = \frac{dH(x,p)}{dp(t)} \tag{5.22}$$

$$\frac{dp(t)}{dt} = -\frac{dH(x,p)}{dx(t)}, \tag{5.23}$$

$$\tag{5.24}$$

where $H(x,p)$ is the Hamiltonian and is dependent on both the position x and momentum $p \equiv x'(t) \equiv \frac{dx}{dt}$. I will not give a derivation of Hamilton's equations, but it can be easily done starting from the Lagrangian $L(x,p)$ for the system instead of the Hamiltonian $H(x,p)$.

5.2.2 Euler's Method

The first approximation one can make to solving these equation is to resort back to the Euler method for time-propagation. It is no more complicated to work with two, coupled, first-order differential equations than it was to work with one. The procedure can be written as,

$$x(t+\Delta t) = x(t) + \Delta t * v(t) \tag{5.25}$$

$$v(t+\Delta t) = v(t) + \Delta t * \frac{F(t)}{m} \tag{5.26}$$

Here, m is the mass of the particle. This can be easily derived using the forward finite-difference expression (Eq. 3.15) for the left-hand side of Eq. 5.22 and solving for $x(t+\Delta t)$ and similarly for $v(t+\Delta t)$.

5.2.3 Velocity-Verlet

One of the most powerful improvements ont eh Euler method for coupled first-order equations is the so-called velocity-Verlet (VV) approach. This approach achieves $\mathcal{O}(\Delta t^2)$ rather than the $\mathcal{O}(\Delta t)$ of the Euler method. The idea is to partially update the position with the old velocity at time t by one half step $0.5\Delta t$, then update the velocity according to the force at time t using a full step Δt. Finally, update the position using the new velocity at time $t+\Delta$ by another half step in position $0.5\Delta t$. It will make more sense written out as,

$$x(t+\frac{\Delta t}{2}) = x(t) + \frac{\Delta t}{2} * v(t) \tag{5.27}$$

$$v(t+\Delta t) = v(t) + \Delta t * \frac{F(t)}{m} \tag{5.28}$$

$$x(t+\Delta t) = x(t+\frac{\Delta t}{2}) + \frac{\Delta t}{2} * v(t+\Delta t)). \tag{5.29}$$

This simple modification is ubiquitously used in the scientific community for its ease of application.