MOD510: Project 1

Deadline: 15. September 2024 (23:59)

Group members

- Simen Sæverud Gåsland
- Nataliia Aksamentova

Abstract

Computers represent numbers with finite precision, which can lead to unexpected results when performing arithmetic with floats in Python. In the first exercise examine how these floats are represented, and why some floats cannot be represented using a finite number of bits. The second exercise introduces the NumPy library, where we look at some widely used functions and recreate them using native Python lists. Doing this will give us a deeper understanding of the library. In the third exercise, we explore how limited precision leads to numerical errors and strategies to minimize them. Finally, we introduce automatic differentiation, which is an alternative to numerical differentiation that avoids error and improves computational performance.

Introduction

Numerical methods are essential tools in science and engineering. The real world is complex, and nature cannot always be described using simple equations. With numerical methods allows us to solve problems that cannot be solved analytically. Solving problems numerically also comes at a cost. Computers operate with finite precision, which can lead to numerical errors.

Throughout this project we will cover fundamental topics such as floating-point representation, functions in the NumPy library, finite difference approximation and automatic differentiation.

Exercise 1: Finite-precision arithmetic

A float in Python is represented using 64 bits. In this exercise we look at how these 64 bits are distributed, and what is the largest and smallest possible float. Furthermore, we look at how we sometimes can run into unexpected results when doing arithmetic with these floats in Python.

Part 1

[1]:
```
import sys
sys.float_info
```

[1]:
```
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308,
min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15,
mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

A float in python is 64 bit double presistion floating point number. Using the sys.float_info we can among other things see that maximum and minimum value that a float can represent in Python. Below is an explanation of each output. [1][2]

max: This is the largest representable finite floating-point number. Any number larger than this will represented as inf (infinity).

max_exp: This is the maximum exponent for a floating-point number.

max_10_exp: This is the maximum base 10 exponent. It's related to **max_exp** but in terms of powers of 10 instead of 2.

min: This is the smallest positive floating-point number.

min_exp: This is the minimum exponent for a floating-point number.

min_10_exp: This is the minimum base 10 exponent. It's related to **max_exp** but in terms of powers of 10 instead of 2.

dig: This is the number of digits that can be faithfully represented in a float.

mant_dig: This is the number of bits in the mantissa. The mantissa represents the precision of the number.

epsilon: This is the machine presision ϵ_M. Meaning that this the smallest number we can add to one and still get a value larger than one.

radix: This indicates the base of the exponent representation.

rounds: This is the rounding mode. With a value of 1 this means "round to the nearest".

Part 2

A floating point number can be represented as:

$$\pm q 2^{E-e} \tag{1}$$

In formula (1):

q **:** is the mantissa which is written as $1.F$ where F is the fraction which is stored with 52 bits.

E **:** is the exponent which is stored with 11 bits.

e **:** is the bias a fixed value, and allows us the a postive and negativ powers of 2.

The bias e has fixed value, and for 64-bits this value is $2^{11-1} = 1023$. The exponent E is stored with 11-bits from 0 to $2^{11} - 1 = 2047$. Before we continue we have to note that $E = 2047$ and $E = 0$ are reserved to represent infinity and zero respectivly. With all this we can now calculate the largest (2) and smallest exponent (3).

$$E_{max} - e = 2046 - 1023 = 1023 \tag{2}$$

$$E_{min} - e = 1 - 1023 = -1022 \tag{3}$$

Now we only have to find the largest and smallest possible mantissa q. Since $q = 1.F$, where F is stored with 52 bits the largest (4) and the smallest (5) have to be:

$$q_{max} = 1.11 + 50 \text{ more ones} \tag{4}$$

$$q_{min} = 1.00 + 50 \text{ more zeros} \tag{5}$$

Putting all this into (1) we get the largest (6) and smallest (7) floating point numbers

$$x_{max} = q_{max} \cdot 2^{E_{max}-e} = 1.11 + 50 \text{ more ones} \cdot 2^{1023} = 1.7976931348623157 \cdot 10^{308} \tag{6}$$

$$x_{min} = q_{min} \cdot 2^{E_{min}-e} = 1.00 + 50 \text{ more zeros} \cdot 2^{-1022} = 2.2250738585072014 \cdot 10^{-308} \tag{7}$$

Note that we can get a smaller number than in (7), but then we would have to allow unnormalized values (mantissa q would then be 0.00...01). Which means we also lose accuracy.

Lastly, we find the machine presision ϵ_M, and this is the smallest possible value of the mantissa which is $0.00...01 = 2^{-52}$. Therefore:

$$\epsilon_M = 2^{-52} = 2.220446049250313 \cdot 10^{-16} \tag{8}$$

Part 3

The reason is that the computer uses the binary system, and in the binary system there is no way of representing 0.2 and 0.3 with a finite number of bits. As an example 0.2 in the binary system is: $0.2_{10} = 0.0011001100...2(= 23 + 24 + 27 + 28 + 211 +)$. Therefore in the computer 0.2 is be represented as 1.9999.... So when we add 0.1 we get really close to 0.3, but not equal to 0.3. [1]

[2]:
```
print(0.2 + 0.1 == 0.3) #Prints False
```

```
False
```

Part 4

Using the method in part 3 is not a good way to test whether two floating-point numbers are equal. Since $0.2 + 0.1 == 0.3$ gives **False** a better way to check is by using **math.isclose()** function. This function has an adjustble tolerance which by default is rel_tol $= 1e - 09$. [3]

[3]:
```
import math
math.isclose(0.1 + 0.2, 0.3) #Prints True
```

[3]:
```
True
```

Exercise 2: Get up to speed with NumPy

NumPy is an incredibly useful library that we have used a lot already in the computational engineering program. In this exercise we look at some useful functions in the NumPy library and try to recreate them using native Python lists. At the end we look at how we can extract a subset of value from an array.

Part 1

To create a list that has the same elements as the numpy array, we used a for-loop. Our first thought was to set a start value in the **range()** function as 0, stop value as 1 and step value as 0.1, but since the **range()** function only allows integers, we had to finesse our way around this. You can see how we solved this in the code below. Also worth noticing is that inside the **append()** function we have written $(10 - 1)$ in the denominator. According to the **np.linspace()** documentation the endpoint is by default included, so for the elements in our list to match the array we had to write $(10 - 1)$ in the denominator. [4]

[4]:
```
import numpy as np
```

[5]:
```
x = []
for i in range(0,10):
    x.append(i/(10-1))
```

Now we can compare the two and see that they both produce the same output. The only difference is that x is a list and **np.linspace()** gives us an array, but the values are the same. Also worth noticing is that the list has more decimals. By using the **round()** function we could give them the same amount, but for now we do not care about that.

[6]:
```
print(x)
print(np.linspace(0, 1, 10))
```

```
[0.0, 0.1111111111111111, 0.2222222222222222, 0.3333333333333333,
0.4444444444444444, 0.5555555555555556, 0.6666666666666666, 0.7777777777777778,
0.8888888888888888, 1.0]
[0.         0.11111111 0.22222222 0.33333333 0.44444444 0.55555556
 0.66666667 0.77777778 0.88888889 1.        ]
```

When we put the list x into the **np.exp()** function it seems to convert it to an array and gives the same output as when x was an array.

[7]:
```
#Works fine
print('exp(x) =', np.exp(x))
```

```
exp(x) = [1.         1.11751907 1.24884887 1.39561243 1.5596235  1.742909
 1.94773404 2.17662993 2.43242545 2.71828183]
```

When we try to compute $-x$ on the other hand we get an error. When we in class preformed multiplications on lists like for example $2 \cdot [1, 2, 3]$ we get $[1, 2, 3, 1, 2, 3]$. It repeats the list 2 times. Writing $-x$ is the same as saying $(-1) \cdot x$. We cannot repeat a list negative one times. This is why we think we get an error.

[8]:
```
#Gives an error
try:
    print('exp(-x) =',np.exp(-x))
except Exception as e:
  print(f'Line has failed with error: {e}')
```

```
Line has failed with error: bad operand type for unary -: 'list'
```

Generally, if we had to evaluate a function on all elements of a native Python list like the NumPy does with its arrays, we would do this in a for-loop. In the code below we calculate **np.exp(x)** using only vanilla Python.

[9]:
```
import math

new_list = []

for i in range(len(x)):
    calc_val = math.exp(x[i])
    new_list.append(calc_val)
```

When we compare the vanilla method to the NumPy method we get the same answer, the only difference being that one is a list and the other one an array.

[10]:
```
print(new_list)
print(np.exp(x))
```

```
[1.0, 1.1175190687418637, 1.2488488690016821, 1.3956124250860895,
1.5596234976067807, 1.7429089986334578, 1.9477340410546757, 2.1766299317162483,
2.4324254542872077, 2.718281828459045]
[1.         1.11751907 1.24884887 1.39561243 1.5596235  1.742909
 1.94773404 2.17662993 2.43242545 2.71828183]
```

Here is an easier and much cleaner way to calculate **np.exp**$(-x)$ using the **np.exp()** instead of the **math.exp()**. The result will also be an array. And when we compare the two, we can see that we get exactly the same result.

[11]:
```
print(np.exp([-number for number in x]))
print(np.exp(-np.linspace(0, 1, 10)))
```

```
[1.         0.89483932 0.8007374  0.71653131 0.64118039 0.57375342
 0.51341712 0.45942582 0.41111229 0.36787944]
[1.         0.89483932 0.8007374  0.71653131 0.64118039 0.57375342
 0.51341712 0.45942582 0.41111229 0.36787944]
```

Part 2

np.zeros() returns an array containing zeros. The shape of the array is determined by the first input. In the example the input is just 20. The function returns a 1 x 20 array. By default, the data type of the array is a float. [5]

[12]:
```
np.zeros(20)

#Example of other uses
#np.zeros((2,3), dtype = 'int') #returns a 2 x 3 array with int datatype
```

[12]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0.])
```

Following code is our way of recreating np.zeros() with vanilla python. For simplicity the following code returns a list of zeros with a given length and datatype is float.

```
[13]: length = 20
      zeros_list = []

      for i in range(length):
          zeros_list.append(0.)

      print(zeros_list)
```

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0]
```

np.ones() returns an array containing ones. Like the zeros function the shape of the array is determined by the first input. In the example the input is just 20. The function will then return a 1 x 20 array. By default, the data type of the array is a float. [6]

```
[14]: np.ones(20)

      #Example of other uses
      #np.zeros((3,4), dtype = 'int') #returns a 3 x 4 array with int datatype
```

```
[14]: array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
             1., 1., 1.])
```

To recreate **np.ones()** with vanilla python we only need to change the input in the **append()** function. Also, here for simplicity the following code returns a list of ones with a given length and datatype is float.

```
[15]: length = 20
      ones_list = []

      for i in range(length):
          ones_list.append(1.)

      print(ones_list)
```

```
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0]
```

The **np.linspace()** function returns an array with evenly spaced numbers on a given interval. In the example below the starting value is 0, the end value is 10 and the number of samples is 11. [4]

```
[16]: np.linspace(0, 10, 11)
```

```
[16]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

To recreate this with vanilla Python we are using a for-loop. First, we define a start value and stop value, the value num is the number of samples we want. Next, we create an empty list and define the first value to add to the list, which must be the start value. Furthermore, we calculate the increment. The $(num - 1)$ in the denominator is because the endpoint is included (as previously explained in exercise 2 part 1). Lastly, a for-loop adds the numbers to a list.

[17]:
```
start = 0
stop = 10
num = 11

linspace_list = []
num_to_add = start
increment = (start + stop) / (num - 1)

for i in range(start,num):
    linspace_list.append(num_to_add)
    num_to_add += increment

print(linspace_list)
```

```
[0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
```

In the following code the endpoint is not included (endpoint = False).

[18]:
```
np.linspace(0, 10, 11, endpoint=False)
```

[18]:
```
array([0.        , 0.90909091, 1.81818182, 2.72727273, 3.63636364,
       4.54545455, 5.45454545, 6.36363636, 7.27272727, 8.18181818,
       9.09090909])
```

To recreate this with vanilla Python we just have to make a small adjustment to our previous code. That adjust is that we remove the -1 from $(num - 1)$ as follows.

[19]:
```
start = 0
stop = 10
num = 11

linspace_list = []
num_to_add = start
increment = (start + stop) / num #removed the -1

for i in range(start,num):
    linspace_list.append(num_to_add)
    num_to_add += increment

print(linspace_list)
```

```
[0, 0.9090909090909091, 1.8181818181818181, 2.727272727272727,
3.6363636363636362, 4.545454545454545, 5.454545454545454, 6.363636363636363,
7.2727272727272725, 8.181818181818182, 9.09090909090909]
```

The **np.arrange()** function is remarkably similar to the **np.linspace()** function. Both return an evenly spaced array given a start, stop and step value. The **np.arrange()** on the other hand does not seem to have any endpoint handling. When only writing **np.arrage(5)** the function will return an open interval $[0, 5)$ which gives an array like this [01234]. By adding +1 to the array each element in the array increase by 1 returning [12345]. By multiplying the array with 2 every element in the array gets multiplied and this gives us [245810]. [7]

```
[20]: vector = np.arange(5) + 1
      2*vector
```

```
[20]: array([ 2,  4,  6,  8, 10])
```

We recreated this in vanilla Python using a for-loop. The loop will run 5 times and since the initial value is $i = 0$ the end point will not be included. The math operations adding by one and multiplying by two we have performed in the **append()** function so that it preforms these operations on every element just like intended.

```
[21]: arange_list = []
      stop = 5

      for i in range(0,stop):
          arange_list.append(2*i + 2) #Adding the + 1 and muliply ever element with 2.␣
       ↪Could have written 2*(i + 1)

      print(arange_list)
```

```
[2, 4, 6, 8, 10]
```

Part 3

```
[22]: array_of_numbers = np.array([4, 8, 15, 16, 23, 42,0,5])
      nnz = np.count_nonzero(array_of_numbers)
      print(f'There are {nnz} non-zero numbers in the array.')
      is_even = (array_of_numbers % 2 == 0)
      is_greater_than_17 = (array_of_numbers > 17)
      is_even_and_greater_than_17 = is_even & is_greater_than_17
```

```
There are 7 non-zero numbers in the array.
```

The code above work fine, but this following line of code does not run.

```
[23]: try:
          is_even_and_greater_than_17 = is_even and is_greater_than_17
      except Exception as e:
        print(f'Line has failed with error: {e}')
```

```
Line has failed with error: The truth value of an array with more than one
element is ambiguous. Use a.any() or a.all()
```

Both **is_even** and **is_greater_than_17** are arrays. The **and** operator expets a boolean value on both sides, here there is an array on both sides [8]. Therefore we have to use **&** instead which is the bitwise operator. Below is a working code that gives the expeted value, which is the number 42 that is the only number in the array that is both even and greater than 17. It is specified that we have to **np.logical_and()**, but according to the NumPy documentation both **np.logical_and()** and the bitwise operator **&** work the same. [9]

```
[24]: #is_even_and_greater_than_17 = is_even & is_greater_than_17
      is_even_and_greater_than_17 = np.logical_and(is_even, is_greater_than_17)
      print(is_even_and_greater_than_17)

      #Prints the numbers that are even and > 17
      print(array_of_numbers[is_even_and_greater_than_17])
```

```
[False False False False False  True False False]
[42]
```

The following code does not work.

```
[25]: try:
          print(array_of_numbers % 2 == 0 & array_of_numbers > 17)
      except Exception as e:
        print(f'Line has failed with error: {e}')
```

```
Line has failed with error: The truth value of an array with more than one
element is ambiguous. Use a.any() or a.all()
```

It may seem that Python does not read the line inside of print as intended. By adding parentheses, we make sure Python reads it as intended. The code below seems to work, and it gives us the same result.

```
[26]: print((array_of_numbers % 2 == 0) & (array_of_numbers > 17))
```

```
[False False False False False  True False False]
```

Part 4

The **numpy.where()** function takes in a condition (in our case: **array_of_numbers** > 17) and returns an array with elements location that follows the set condition. For the **array_of_numbers** the numbers that are greater than 17 are 23 and 42 which are element number 4 and 5 (the first element is 0). The function returns a tuple so the [0] is added to get the first element in the tuple, since the second element is just blank. [10]

```
[27]: np.where(array_of_numbers > 17)[0]
```

```
[27]: array([4, 5], dtype=int64)
```

You can also spesify what you want reurned when the condition is **True** or **False**. In the following line of code you get a 0 if the element is not greater than 17 and 1 if the element is greater then 17.

```
[28]: np.where(array_of_numbers > 17, 1, 0)
```

```
[28]: array([0, 0, 0, 0, 1, 1, 0, 0])
```

Exercise 3, Part I: Finite Differences (FD) with Functions

In this exercise we create different functions. We start off creating a function that plots the Gaussian wave packet over an interval. Furthermore, we create functions to return the analytical solution to the wave packet function, before we create functions to calculate the forward and central difference. Lastly, we compare the numerical errors the forward and central difference produce against the analytical solution and visualize this in a plot.

Part 1

We start by defining the wave packet function like shown in the problem set.

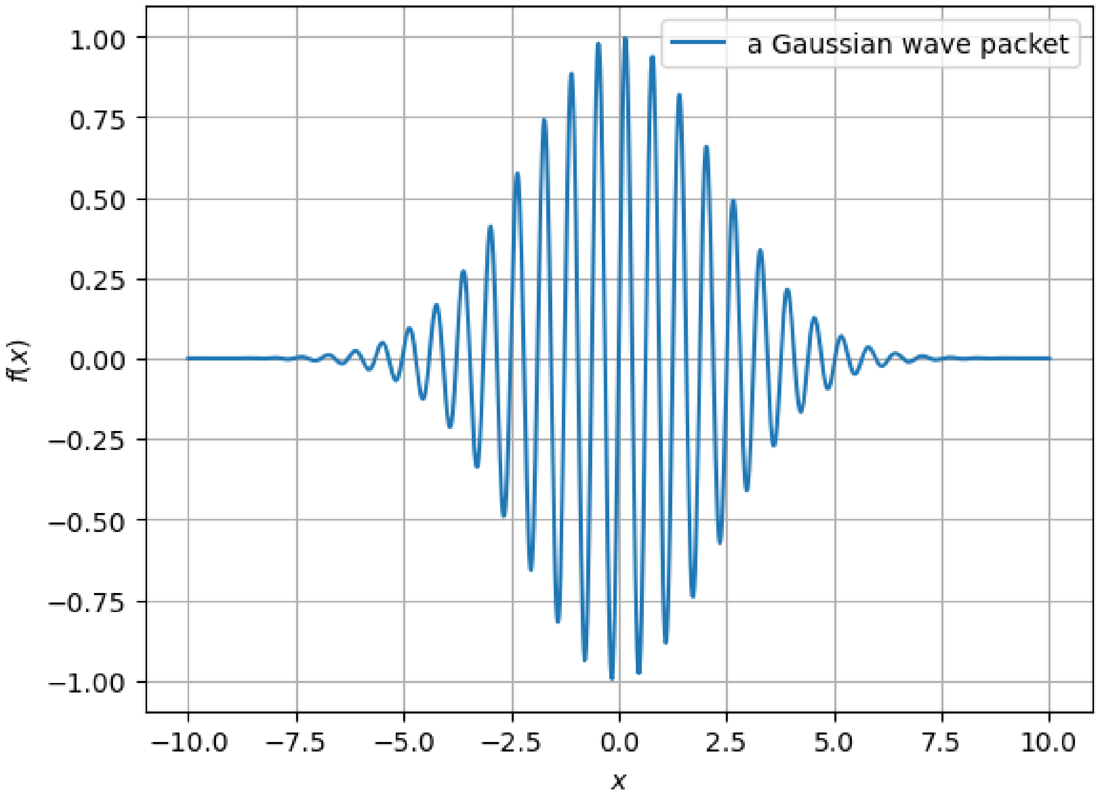
[29]:
```
def f(x, a=0.1, b=10):
    return np.sin(b*x)*np.exp(-a*x*x)
```

Below is a function that plots the wave packet function over an arbitrary interval. The function takes in a function, an interval start and end value and number of datapoint. The plotter function also has a ****kwargs** input. This is so that when the user uses the plotter function, they can also change the values for **a** and **b** in the wave packet function. We used ****kwargs** instead of ***args** because we thought passing keyword arguments are more readable than positional arguments. [11]

[30]:
```
import matplotlib.pyplot as plt

def plotter(func, start, stop, num, **kwargs):
    '''
    This functions generates a plot of a function over an
    arbitrary closed interval.

    Input:
    ------
        func: A function.
        start: x values starting point .
        stop: x values stop point.
        num: number of data points
    '''
    x = np.linspace(start, stop, num)
    y = f(x, **kwargs)

    fig, ax = plt.subplots()
    ax.plot(x, y, '-', label = 'a Gaussian wave packet')
    ax.set_xlabel(r'$x$')
    ax.set_ylabel(r'$f(x)$')
    ax.legend()
    ax.grid()
    plt.show()
```

Below is an example of the plotter function plotting the Gaussian wave packet function (with $a = 0.1$ and $b = 10$) over the interval $[-10, 10]$ with 1000 data points.

[31]:
```
plotter(func = f, a = 0.1, b = 10, start = -10, stop = 10, num = 1000)
```



Part 2

Below is a function that returns the analytical solution of the wave packet function.

[32]:
```
def fd(x, a=0.1, b=10):
    '''
    Returns the derivative of the wave packet function
    '''
    return b*np.cos(b*x)*np.exp(-a*x*x) - 2*a*x*np.sin(b*x) * np.exp(-a*x*x)
```

Part 3

The function **forward_difference()** takes in a function, an x-value and a step size h. The function also takes in ****kwargs** to allow a and b to be changed in the wave packet function. The function returns the result from formula (9) which is the forward difference.

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{9}$$

[33]:
```python
def forward_difference(func, x, h, **kwargs):
    '''
    Returns the forward difference
    '''
    return (func(x + h, **kwargs) - func(x, **kwargs)) / h
```

An example of use at $x = 1$ and with step size $h = 10^{-2}$:

[34]:
```python
forward_difference(func = f, x = 1, h = 1e-2)
```

[34]:
```
-7.220096595246589
```

To see how good the approximation of the forward difference is at $x = 1$ with step size $h = 10^{-2}$ we can calculate the numerical error by subtracting the forward difference from the analytical answer. We take the absolute value of this subtraction since we do not care about whether the error is positive or negative. The numercal error here is:

$$\epsilon \approx 0.2737$$

[35]:
```python
error = abs(fd(x = 1) - forward_difference(func = f, x = 1, h = 1e-2))
print(error)
```

```
0.27368643245679003
```

Part 4

Likewise, we here **central_difference()** takes in a function, an x-value and a step size h. This function also takes in ****kwargs**. The function returns the result from formula (3) which is the central difference.

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{3}$$

[36]:
```python
def central_difference(func, x, h, **kwargs):
    '''
    Returns the central difference
    '''
    return (func(x + h, **kwargs) - func(x - h, **kwargs)) / (2*h)
```

By using the same x- and h-value as we did in the forward difference example, when calculating the numerical error we can compare the two results. Here using central difference, we get less numerical error than we did with the forward difference. The numerical error here is:

$$\epsilon \approx 0.012$$

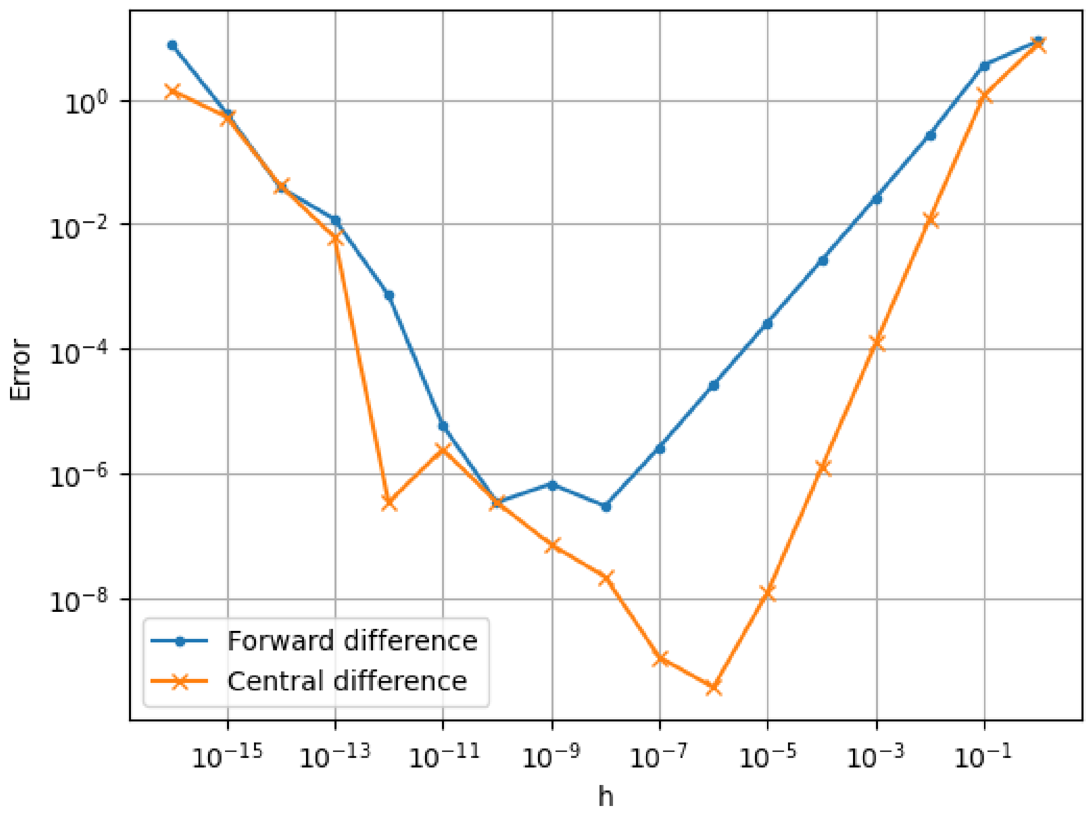
[37]:
```python
error = abs(fd(x = 1) - central_difference(func = f, x = 1, h = 1e-2))
print(error)
```

```
0.012215278172568844
```

Part 5

As we saw in part 3 and 4 the numerical errors between forward and central difference are different. In this part we look at how the numerical error at $x = 1$ changes over different step sizes h. In the plot below we plot the numerical error for both methods over the interval $\left[10^{-16}, 10^{0}\right]$. Thinking mathematically, one would expect the error to decrease as the step size h also decreases, but from the plot we can clearly see that the numerical error at first decreases but eventually starts to increase despite the step size h decreasing. The reason for this is that we have two competing errors: **truncation error** and **roundoff error**. If we look at the graph for the central difference, we can see that the numerical error is the smallest at $h = 10^{-6}$, after this point the roundoff error starts to dominate and the error increases. To conclude, if you want to minimize the numerical error the trick is not to choose a super small step size h but rather to find the "golden middle way" between truncation error and roundoff error, which for the central difference here is at $h = 10^{-6}$ and $h = 10^{-8}$ or $h = 10^{-10}$ for the forward difference. [1]

[38]:
```python
x = 1
h = np.logspace(-16,0, 17)

forward_diff = forward_difference(f, x, h)
central_diff = central_difference(f, x, h)
analytical = fd(x)

forward_error = abs(forward_diff - analytical)
central_error = abs(central_diff - analytical)

fig, ax = plt.subplots()
ax.plot(h, forward_error, '.-', label ='Forward difference')
ax.plot(h, central_error, 'x-', label ='Central difference')
ax.set_xlabel('h')
ax.set_ylabel('Error')
ax.set_xscale('log')
ax.set_yscale('log')
ax.legend()
ax.grid()
plt.show()
```



Exercise 3, Part II: FD with Classes

In this part of the exercise, we implement what we did in part I, but this time we put everything in one class.

Below is the code for the entire exercise in one kernel to avoid having several copies of the same thing. Which part of the code is written to which part of the exercise is marked with comments. After the code kernel we will elaborate on each part.

[39]:
```python
class WavePacket:
    '''
    A class representation of a wave packet-function.
    '''

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __call__(self, x):
        return np.sin(self.b*x)*np.exp(-self.a*x*x)
    def plot(self, x_min=-10, x_max=10, dx=0.01):
        '''
        A simple plotting routine for plotting f(x) in some range.
        '''
        x = np.arange(x_min, x_max, dx)
        y = self(x)
        fig = plt.figure()
        plt.plot(x, y)
        plt.grid()

    #--------------------------- Part 1 ---------------------------
    def forward_difference(self, x, h):
        '''
        Returns the forward difference
        '''
        return (self(x + h) - self(x)) / h

    def central_difference(self, x, h):
        '''
        Returns the central difference
        '''
        return (self(x + h) - self(x - h)) / (2*h)
    #---------------------------------------------------------------

    #--------------------------- Part 2 ---------------------------
    def error_plotter(self, x):
        '''
        Plots the numerical error (in range [1e-16, 1]) produced by forward and
        central difference at an arbitrary x-value.
        '''

        h = np.logspace(-16,0, 17)

        forward_diff = self.forward_difference(x, h)
        central_diff = self.central_difference(x, h)
        analytical = self.b*np.cos(self.b*x)*np.exp(-self.a*x*x) - 2*self.a*x*np.
↪sin(self.b*x) * np.exp(-self.a*x*x)

        forward_error = abs(forward_diff - analytical)
        central_error = abs(central_diff - analytical)

        fig, ax = plt.subplots()
        ax.plot(h, forward_error, '.-', label ='Forward difference')
        ax.plot(h, central_error, 'x-', label ='Central difference')
        ax.set_xlabel('h')
        ax.set_ylabel('Error')
        ax.set_xscale('log')
        ax.set_yscale('log')
        ax.legend()
        ax.grid()
        plt.show()
        #-------------------------------------------------------------
```

Part 1

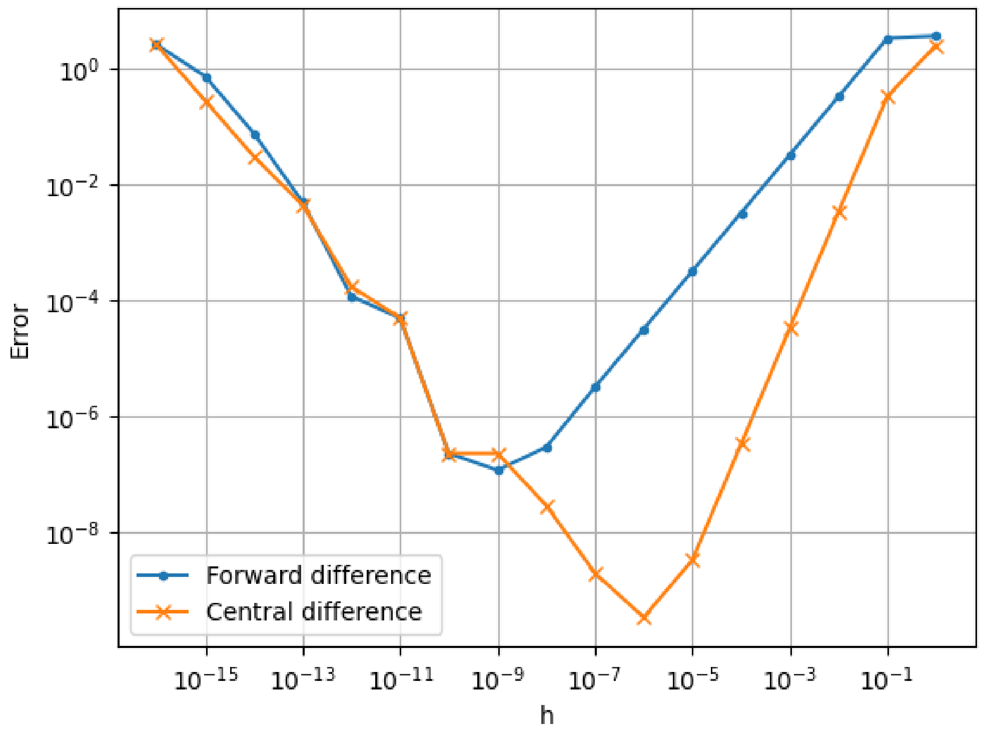
In this part we have added two functions (instace methods) that returns the forward and central difference. The implementation of these functions are very similary to what we did in part I, but these functions do not take the a function and ****kwargs** as inputs. Here instead argument for a and b are taken in when an instance of the **wavepacket** class is created. Furthermore, we have replaced the **func** input by creating the **__call__** method. This method is triggerd when we use the class (self) as a function. [12]

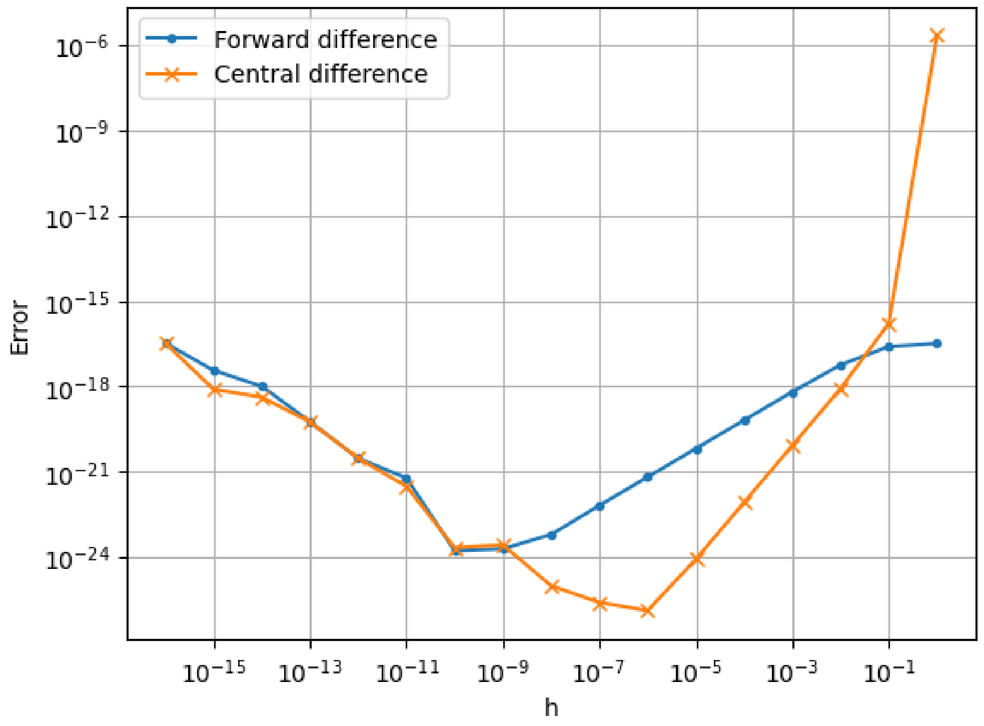
Part 2

The error plot we have here created as a function within the class (a method). It produces the same plot as in Part I, and has only been modified so that it can run as a method.

Now that we have created this class it is super easy to define wave packet functions with different a and b parameters. Now we can analyze the numerical error with a few lines of code like below.

[40]:
```
WP1 = WavePacket(a = 0.1, b = 10)
WP2 = WavePacket(a = 10, b = 0.1)

x = 2
WP1.error_plotter(x)
WP2.error_plotter(x)
```


Error
h
Forward difference
Central difference
10^{0}
10^{-2}
10^{-4}
10^{-6}
10^{-8}
10^{-15}
10^{-13}
10^{-11}
10^{-9}
10^{-7}
10^{-5}
10^{-3}
10^{-1}


Error
h
Forward difference
Central difference
10^{-6}
10^{-9}
10^{-12}
10^{-15}
10^{-18}
10^{-21}
10^{-24}
10^{-15}
10^{-13}
10^{-11}
10^{-9}
10^{-7}
10^{-5}
10^{-3}
10^{-1}

Exercise 4: Automatic for the people?

Numerical differentiation can be slow and as we have seen influenced by round off errors. In this exercise we explore the core ideas of automatic differentiation. We humans can calculate derivatives analytically using a set of rules. The core idea of automatic differentiation is that computers can do the same if we explain the rules. This is done by using a fixed set of operations: addition, subtraction, multiplication, and division. We also use some other elementary functions like sin, log and exp.

Like in exercise 3, the code for the entire exercise in one kernel to avoid having several copies of the same thing. Which part of the code is written to which part of the exercise is marked with comments. After the code kernel we will elaborate on each part.

[41]:
```python
class duple:
    '''
    Class for automatic differentiation
    top: function value
    bottom: derivative of function
    '''

    def __init__(self,top, bottom=0):
        self.top = top
        self.bottom = bottom

    def __add__(self, d): # u+v, u'+v'
        return duple(self.top + d.top, self.bottom + d.bottom )

    #--------------------------- Part 1 ---------------------------
    def __repr__(self):
        return "["+str(self.top)+","+str(self.bottom)+"]"

    def __str__(self):
        return "["+str(self.top)+","+str(self.bottom)+"]"
    #--------------------------------------------------------------

    #--------------------------- Part 2 ---------------------------
    def __pos__(self):
        return duple(+ self.top, + self.bottom)

    def __neg__(self):
        return duple(- self.top, - self.bottom)

    def __sub__(self, d):
        return duple(self.top - d.top, self.bottom - d.bottom)
    #--------------------------------------------------------------

   #--------------------------- Part 3 ---------------------------
    def __mul__(self,d):
        return duple(self.top * d.top, (self.bottom * d.top) + (self.top * d.
↪bottom))

    def __truediv__(self,d):
        return duple(self.top / d.top, ((self.bottom * d.top) - (self.top * d.
↪bottom)) / (d.top * d.top))
    #------------------------------------------------------------

#------------------------------ Part 4 ---------------------------
class dfunction:
    def __init__(self, f, df):
        self.f = f
        self.df = df
    def __call__(self,d: duple): #chain rule sending in and returning duple
        return duple(self.f(d.top), (self.df(d.top) * d.bottom))
#-----------------------------------------------------------------
```

Part 1

When creating a **duple** object e.g. $x = \text{duple}(1, 2)$ and trying to print it, we get a bunch of nonsense. When printing x, it would be ideal to have a result that looked like this: $[1, 2]$. To achieve this, we have been asked to implement **__repr__** and **__str__**. Both these will give us a string representation of the object, but **__str__** is used to create a string that easily readable for the user, while **__repr__** is meant to give more details that can be used in debugging. [13]

[42]:
```
#Testing the code
x = duple(1,2)
print(x) #Should return [1,2]
```

```
[1,2]
```

Part 2

Despite having implemented the **__add__** method we get an error when writing $+x$, and when writing $-x$. To implement these unary operators, we use **__pos__** and **__neg__**. The unary plus operator does not really do anything other than indicating a positive value. Note that in our code we have written +self.top and +self.bottom which is redundant, but we think it adds to the readability of the code. The unary minus operator **__neg__** on the other hand negates the values [14]. Lastly, we were asked to implement the $-$ operator using the **__sub__** method. To implement this, we subtracted the top values and the bottom values.

[43]:
```
#Testing the code
x = duple(1,2)
y = duple(3,4)
print(x - y) #Should return [-2,-2] since 1-3=-2 and 2-4=-2
```

```
[-2,-2]
```

Part 3

Here we were asked to implement the $*$ operator and the $\div$ operator using **__mul__** and **__longdiv__** respectively. For the multiplication operator we multiplied the top of the duple like normal and for the bottom part we used the product rule. Similarly, for the division operator we divided the top part like normal and used the quotient rule for the bottom part.

[44]:
```
#Testing the code

#Here compare the auto diff and analytical solution.
#These two should give the same answer.

x = 1.2
One = duple(1.,0.) #Derivative of a constant is zero
X = duple(x,1) #Derivative of x with respect to x is 1

print("auto diff = ",X*X*X)
print("analytical = ",x*x*x, 3*x*x)

print("auto diff = ", One/(One+X))
print("analytical = ", 1/(1+x),-1/(1+x)**2)

Exp = duple(np.exp(x),np.exp(x)) #Derivative of exp(x) is exp(x)
Sin = duple(np.sin(x),np.cos(x)) #Derivative of sin(x) is cos(x)

print("auto diff = ", Exp*Sin)
print("analytical = ", np.exp(x)*np.sin(x), np.exp(x)*np.sin(x)+np.exp(x)*np.
 ↪cos(x))
```

```
auto diff =  [1.728,4.32]
analytical =  1.728 4.319999999999999
auto diff =  [0.45454545454545453,-0.20661157024793386]
analytical =  0.45454545454545453 -0.20661157024793386
auto diff =  [3.0944787419716917,4.297548854694511]
analytical =  3.0944787419716917 4.297548854694511
```

Part 4

If we want to use automatic diffrentiation on composite function we need to add the class **dfunction**. When creating an instance of this class you have to send in a function f and its derivative fd. Using the **__call__** method the class **dfunction** can be called like a funtion, and will then return a duple where the top value is $f(g(x))$, where $f(x) = \text{self.f}$ and $g(x) = \text{d.top}$, and the bottom value is the result of the chain rule.

[45]:
```
#Testing the code
x = 1.2
One = duple(1.,0.) #Derivative of a constant is zero
X = duple(x,1) #Derivative of x with respect to x is 1
```

```
def f(x):
    return np.log(x)

def df(x):
    return 1/x

Log = dfunction(f,df)
Exp = dfunction(np.exp,np.exp)

print("auto diff ", Log(One+Exp(X)))
print("analytical ", np.log(1+np.exp(x)),np.exp(x)/(1+np.exp(x)))
```

```
auto diff  [1.4632824673380311,0.7685247834990176]
analytical  1.4632824673380311 0.7685247834990176
```

[46]:
```
#Testing the code on the wavepacket

a=0.1
b=10
x=1

A=duple(a) #second argument is default 0 i.e. a constant
B=duple(b) #second argument is default 0 i.e. a constant
X=duple(x,1)

Sin=dfunction(np.sin,np.cos)

print('auto diff', Sin(B*X)*Exp(-A*X*X))
print('analytical ', np.sin(b*x)*np.exp(-a*x*x),b*np.cos(b*x)*np.
 ↪exp(-a*x*x)-2*a*x*np.sin(b*x)*np.exp(-a*x*x))
```

```
auto diff [-0.49225065733419177,-7.493783027703379]
analytical  -0.49225065733419177 -7.493783027703379
```

Part 5

Automatic differentiation is equivalent to analytical differentiation of elementary functions along with propagation using the chain rule. It is most used in areas such as Machine learning and in any field that requires optimization, for example, engineering design, computational fluid dynamics, robotics. This is also widely used in areas like scientific computing and data assimilation and computer graphics and vision. To sum up, it is widely used in fields that require high precision. [15]

Compared with numerical differentiation, automatic differentiation is more accurate and precise, while numerical differentiation has the advantage of simplicity.

Strengths: * The first advantage is accuracy and precision

- The second advantage: it allows us to work with many inputs.
- Versatility: Automatic differentiation can be applied to complex computer programs.

- Scalability: Can compute gradients for models with millions or even billions of parameters.

Weaknesses:

- Complexity: Developing and work from scratch is a complex and challenging task.
- Non-Differentiable Functions: not suitable for functions that are not differentiable or functions where the internal operations are unknown.
- Memory Usage: requires storing the intermediate values from the forward pass to compute the backward pass which can be a significant memory overhead for very deep or complex computational graphs.

Conclusion

Through the exercises we have seen how numerical computation depends on both mathematical methods and the underlying representation of numbers in the computer. Floating-point arithmetic can introduce small but important errors, which must be considered when designing algorithms. The NumPy library provides efficient tools for handling large scale numerical data, while finite difference methods give a straightforward way to approximate derivatives. At the same time, automatic differentiation is more accurate and less computationally expensive.

Overall, this project has given us a better understanding of both the strengths and limitations of numerical methods. The knowledge we have gathered from this project will undoubtedly become handy further into our studies in computational engineering.

Self-reflections

Simen :

The workload for this project was larger than anticipated, but luckily, I started working on the project as soon as it was published. This allowed me to work on the project in smaller session every day. This also allowed for extra time at the end to make a better report. I am happy with the work that has been done and how I worked with the project. Throughout the project I have been persistent to not only get a code that works but also try to really understand why it works. For the next project I plan to work in a similar way.

Nataliia :

I have got some knowledge about the difference between numerical and automatic differentiation. It was also interesting to know how different computers solve some equations, for example, that $0.1 + 0.2$ is not equal 0.3 because of the binary system. I had a problems with coding because it was extremely difficult for me to get to know Python just 3 weeks ago and now implement such complex code, but I was lucky with my group partner. But I think that the most important knowledge that I got from this project is information about the different errors in programming, how to write new classes and functions, and have learned some interesting Python libraries.

To sum up, from my own experience, it was difficult but at the same time interesting and useful.

Bibliography

[1] A. Hiorth, *Computational Engineering and Modeling.* [Online]. Available: https://github.com/ahiorth/CompEngineering. [Accessed: 14-Sep-2025].

[2] Python Software Foundation, "sys" *Python 3.13.7 Documentation.* [Online]. Available: https://docs.python.org/3/library/sys.html. [Accessed: 14-Sep-2025].

[3] Python Software Foundation, "math.isclose" *Python 3.13.7 Documentation.* [Online]. Available: https://docs.python.org/3/library/math.html#math.isclose. [Accessed: 14-Sep-2025].

[4] NumPy Developers, "numpy.linspace" *NumPy v2.4.dev0 Manual.* [Online]. Available: https://numpy.org/devdocs/reference/generated/numpy.linspace.html. [Accessed: 14-Sep-2025].

[5] NumPy Developers, "numpy.zeros" *NumPy v2.3 Manual.* [Online]. Available: https://numpy.org/doc/2.3/reference/generated/numpy.zeros.html. [Accessed: 14-Sep-2025].

[6] NumPy Developers, "numpy.ones" *NumPy v2.2 Manual.* [Online]. Available: https://numpy.org/doc/2.2/reference/generated/numpy.ones.html. [Accessed: 14-Sep-2025].

[7] NumPy Developers, "numpy.arange" *NumPy v2.1 Manual.* [Online]. Available: https://numpy.org/doc/2.1/reference/generated/numpy.arange.html. [Accessed: 14-Sep-2025].

[8] R. Singh, "Difference between 'and' and '&' in Python" *TutorialsPoint.* [Online]. Available: https://www.tutorialspoint.com/difference-between-and-and-amp-in-python. [Accessed: 14-Sep-2025].

[9] NumPy Developers, "numpy.logical_and" *NumPy v2.0 Manual.* [Online]. Available: https://numpy.org/doc/2.0/reference/generated/numpy.logical_and.html. [Accessed: 14-Sep-2025].

[10] NumPy Developers, "numpy.where" *NumPy v2.2 Manual.* [Online]. Available: https://numpy.org/doc/2.2/reference/generated/numpy.where.html. [Accessed: 14-Sep-2025].

[11] Geeks for Geeks, "*args and **kwargs in Python," *GeeksforGeeks.* [Online]. Available: https://www.geeksforgeeks.org/python/args-kwargs-python/. [Accessed: 14-Sep-2025].

[12] Geeks for Geeks, "__call__ in Python" *GeeksforGeeks*, Jul. 12, 2025. [Online]. Available: https://www.geeksforgeeks.org/python/__call__-in-python/. [Accessed: 14-Sep-2025].

[13] Geeks for Geeks, "str() vs repr() in Python" *GeeksforGeeks*, 23 Jul. 2025. [Online]. Available: https://www.geeksforgeeks.org/python/str-vs-repr-in-python/. [Accessed: 14-Sep-2025].

[14] Geeks for Geeks, "Dunder or magic methods in Python" *GeeksforGeeks*, 7 Aug. 2024. [Online]. Available: https://www.geeksforgeeks.org/python/dunder-magic-methods-python/. [Accessed: 14-Sep-2025].

[15] A. Gezerlis, *Numerical Methods in Physics with Python.* Cambridge, UK: Cambridge University Press, 2020.