Automatic for the people

MOD510: Project 1

Deadline: 15 September 2024 (23:59)

Sep 2, 2025

Learning objectives. By completing this project, the student will:

- Get experience in structuring and writing a report.
- Write new functions and classes in Python.
- Explore numerical round-off and truncation errors.
- Learn useful Python libraries for scientific computing.

Abstract

Because computers have finite memory, numerical errors must always be taken into account when doing calculations, especially when working with floating-point numbers [1]. In the first part of this project, we investigate round-off errors and truncation errors using Python, and we discuss how different implementation strategies affect code efficiency and clarity. In particular, we show how coding with classes can simplify the implementation of numerical algorithms. Finally, you will work with automatic differentiation [3], which illustrates the power of object oriented programming. Automatic differentiation is a cornerstone in training neural networks and in many commercial simulation codes.

Remember to take a look at the Appendix for some tips, and read the guidelines for project submission at the end!

1 Exercise 1: Finite-precision arithmetic

Part 1. Run the following code snippet:

```
import sys
sys.float_info
```

- Explain the meaning of the numbers that are printed out.

Hint: Read the lecture material on the IEEE Standard for floating-point arithmetic.

Part 2.

- Show how you can calculate the printed-out values `max`, `min` and `epsilon` based on the IEEE standard.

Part 3. In Python, typing `0.1+0.2` does not (typically) produce the output `0.3`.

- Why not?

Part 4.

- Would you use the `==` operator to test whether two floating-point numbers are equal?
- Why / why not? Can you think of alternative ways to do floating-point number comparison?

2 Exercise 2: Get up to speed with NumPy

The purpose of this exercise is to learn a little bit about NumPy, which is an incredibly useful Python library. A major reason for its popularity is efficiency: doing computations with NumPy arrays (objects of the type `ndarray`) instead of using native Python lists can, by itself, speed up a program by several orders of magnitude! This is because of vectorized computation.

Vectorized functions.

Using NumPy arrays allows you to create vectorized functions; functions that operate on a whole array at once, rather than looping over the elements one-by-one inside a loop.

The way vectorization works behind the scenes is still via loops (optimized via pre-compiled C code), but as a Python programmer you do not need to worry about these details.

Part 1. The following code block gives an example of a vectorized function:

```
x = np.linspace(0, 1, 10)
np.exp(x)  # Apply f(t)=exp(t) to each element in the array x.
np.exp(-x)  # Apply the function f(t)=exp(-t) to each element of x.
```

Notice the usage of `np.exp` instead of using the exponential function provided in the built-in math library; this is an example of a universal function.

- Create a native Python list of the same size as `x` and holding the same values. Apply the same two function calls to the list. Explain what happens.
- How would you generally evaluate a function on all elements of a native Python list as opposed to a NumPy array?

Part 2. As already hinted, the NumPy library comes with a plethora of useful features and functions. The code snippets below show some examples:

```
np.zeros(20)
```

```
np.ones(20)
```

```
np.linspace(0, 10, 11)
```

```
np.linspace(0, 10, 11, endpoint=False)
```

```
vector = np.arange(5) + 1
2*vector
```

- Explain what each line of code does.
- How would you produce the same output using native Python lists?

Part 3. Frequently you will want to extract a subset of values from an array based on some kind of criterion. For example, you might want to count the number of non-zero numbers, or identify all values exceeding a certain threshold. With NumPy, suchs tasks are easily achieved using boolean masking, e.g.:

```
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
```

However, neither of the following codes lines will execute:

```
is_even_and_greater_than_17 = is_even and is_greater_than_17
print(array_of_numbers % 2 == 0 & array_of_numbers > 17)
```

- Explain why this code fails.
- Use `np.logical_and` to make the code work

Part 4. The function np.where can also be used to select elements from an array.

- Explain the output of the following two lines of code:

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

```
np.where(array_of_numbers > 17, 1, 0)
```

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

In scientific computing one often needs to calculate derivatives of functions. For many practical problems, exact formulas may not be available and numerical estimates are needed. However, to evaluate the correctness of the numerical methods, it is a good idea to test the code on simple functions where the derivative is known.

In this exercise, we consider a function that is relevant for describing wave phenomenona:

$$f(x) = \sin(bx) \cdot e^{-ax^2} \tag{1}$$

One way to implement the function in Python is:

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

We have chosen to define `a` and `b` as *default arguments*, which allows us to evaluate the function at $x = 1$ by simply typing `f(1)`; this is equivalent to the command `f(1, 0.1, 10)`. If you want to change the `b` parameter, you can do, e.g., `f(1, b=2)`. Note also that the function works both when `x` is a single number *and* when it is a Numpy array. This is because we use the Numpy versions of the sine (`np.sin`) and exponential (`np.exp`) functions.

Python functions are first-class!

An important feature of Python is that functions are first-class objects, meaning that you can assign them to variables, you can store them inside various containers and data structures, they can be passed as input arguments to other functions, and they can be return values of other functions.

We will exploit this property of Python several times during this project.

Part 1. It is always a good idea to start by visualizing the function in a plot.

- Make a Python function that plots $f(x)$ from equation (1) over an arbitrary closed interval.
- Use the function to plot $f(x)$ in the range $[-10, 10]$. Try to make your figure similar to the one shown in figure 1

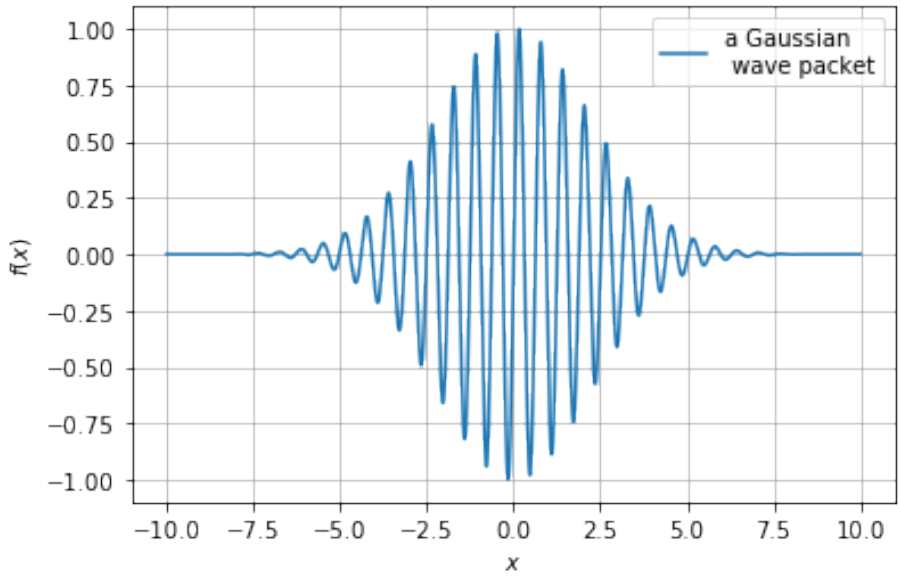

Figure 1: A plot of $f(x)$.

Part 2. The analytical derivative of $f(x)$ is

$$f'(x) = b\cos(bx) \cdot e^{-ax^2} - 2ax\sin(bx) \cdot e^{-ax^2} \tag{2}$$

- Write a Python function that calculates the derivative defined in equation (2) (*not* the numerical derivative)

Part 3. Next, you are going to write a Python function that calculates the numerical derivative of an *arbitrary* single-variable function `f` at a point x using finite differences.

- Write a Python function that calculates the derivative of an arbitrary function using the *forward difference* method (see section 3.3 in [2]).
- Apply your function to the particular case of equation (1) and $x = 1$. Use a suitable value of h, e.g. $h = 1e - 2$, and check that your estimate agrees reasonably well with the *analytical solution*.

Hint: Check out some tips in Appendix A, we would recommend to use args in Python to write a numerical derivative function. See also section 1.3.4 in [2].

Part 4.

- Write another Python function that calculates $f'(x)$ with the *central difference* method (see section 3.4 in [2]).

Part 5. Next, we wish to quantify the error in our numerical derivative approximations for the point $x = 1$.

- For each of the two derivative approximations to $f'(1)$ (forward difference and central difference), make a scatter plot that shows the *absolute error* of the approximation on the y-axis versus the step size, h, on the x-axis. Include both error curves in the same figure. The step sizes should be varied logarithmically between $h = 10^{-16}$ and $h = 10^0 = 1$.
- Comment on what you observe in the figure you made. When is the numerical error smallest, and why? Is it what you expect from a theoretical analysis using Taylor's formula?

4 Exercise 3, Part II: FD with Classes

Implementing numerical algorithms with free functions, as we did in the previous exercise, is perfectly fine, and you can complete the course by only coding in this way. However, experience has taught us that it is easy to introduce errors when using this approach. In many cases you are better off by also using classes, and object-oriented design. In this exercise, you will get some practice in coding with classes. This knowledge will come in handy in later projects, and it is a good tool to have in your programming toolkit.

Previously, we worked with a function having two input parameters, a and b. Implementing numerical algorithms using free functions was then simple. However, in a more complicated situation, there could be dozens, or even hundreds, of parameters to keep track of. Most of these parameters might have fixed values, but frequently you will want to re-run a model with slightly different parameters. If you are not using classes, it is very easy to use the wrong parameters. This is especially true when working in a Jupyter notebook, because it is possible to run code blocks in any order. If you forget to excute a cell that is responsible for updating one of your variables, your subsequent calculations might use incorrect input, and thus end up being wrong!

Key take-away: Classes provide encapsulation.

By wrapping parts of your code into classes, and particular realizations of classes (objects), you facilitate code re-use, and it can make your code easier to understand and work with, thus reducing the probability of introducing bugs which may be hard to track down.

4.1 A Crash Course on Classes

To get started, there are really only a couple of things you need to know. First, all of your classes should include a special function called `__init__`, in which you declare the variables (attributes) you wish an instance / object of the class to keep track of.

Second, when setting, updating, or fetching attributes stored within the class, you should always use the prefix `self`, followed by a dot. Furthermore, the functions you define inside the class should have `self` as the first function argument (there are exceptions, but we will not consider that in this project). All of this is best understood via an example:

```
class WavePacket:
    """
    A class representation of a wave packet-function.
    """
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def f(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.f(x)
        fig = plt.figure()
        plt.plot(x, y)
        plt.grid()
```

Besides the initialization method and a function that calculates $f(x)$ from equation (1), the class includes a simple plotting routine. A major difference from before is the following: when our function $f(x)$ is defined inside a class, we do not have to pass around a and b as arguments to the function `f`. Instead, we simply access a and b from the class itself (using the `self`-prefix).

Below is an example of how to use the class:

```
# Create two WavePacket objects, having their own parameter values
WP1 = WavePacket(0.1, 2)  # a=0.1, b=2
WP2 = WavePacket(0.1, 10)  # a = 0.1, b=10

# Evaluate the two functions at a specific point
x = 1
print(WP1.f(x))
print(WP2.f(x))

# Plot the two functions
WP1.plot()
WP2.plot()
```

Although we had to write slightly more code, we hope you appreciate how easy this makes running parallel simulations with different parameters. Actually, Python provides a way for us to simplify even further, by defining the special `__call__` method for the class:

```
class FancyWavePacket:
    """
    A slightly more fancy class representation of a wave packet-function.

    In this version, we define the dunder (double-underscore) method __call__,
    which lets us treat objects of the class as if they were real functions!
    """
    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)
```

Compared to the first example of the class, observe that we have replaced the function `f` by `__call__` (with two underscores on both sides of "call"). This way, we can write our code *as if FancyWavePacket was a function*:

```
WP1 = FancyWavePacket(0.1, 2)  # a=0.1, b=2
WP2 = FancyWavePacket(0.1, 10)  # a = 0.1, b=10

# Evaluate the two functions at a specific point
x = 1
print(WP1(x))  # If WP1 had been a function, the syntax would be the same here!
print(WP2(x))  # Again, we no longer have to type "WP2.f(x)", we can do "WP2(x)".
```

READ THIS BEFORE ANSWERING THE EXERCISES BELOW.

To avoid code duplication, you should only submit a single version of the WavePacket class. In other words, while different exercises will ask you to add a new function to the class, your final delivery should only include the full class which contains all of the functions.

Part 1.

- Add a function (instance method) to the class that returns the forward difference approximation to the derivative of the function `f` at a point `x`. Include both `x` and the step-size `h` as input arguments to the function.
- Add a second function which calculates the central difference approximation.

Part 2.

- Make a third class function that, for any input x, creates scatter plots showing the absolute error of the two finite difference approximations of $f'(x)$ versus step size. As before, choose step sizes in the range from 10^{-16} to 1 (with logarithmic spacing).

Hint: You should re-use the first two functions when making the third one.

5 Exercise 4: Automatic for the people?

5.1 Introduction to automatic differentiation

At the core of computational models, the computer evaluates *gradients*. Gradients may be evaluated using numerical differentiation. Numerical differentiation is usually slow and could be influenced by round off errors. Humans can calculate derivatives of complicated functions using a set of rules. Why cannot computers do the same, provided we explain the rules? This idea was explored by Wengert in 1964 [3].

The core idea behind automatic differentiation is the fact that when we evaluate a function, it is done by using a set of fixed operations: addition, subtraction, multiplication, division and a set of elementary functions, $\sin, \log, \exp$ etc [3].

So how can we tell the algebraic rules of differentiation to the computer? We will do this using Python and object oriented programming. We want to calculate both the function value *and* the derivative at the same time. To achieve this we implement these operations using a vector of size 2

$$\begin{pmatrix} f(x) \\ f'(x) \end{pmatrix}. \tag{3}$$

At the top we have the function and at the bottom we have the derivative of the function. First we start by constructing a class

```
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
```

5.2 Addition and subtraction

Next, we want to implement some simple operations, lets say we want to do

```
x=duple(1,2) #f(x)=1, f'(x)=2
y=duple(3,4) #g(x)=3, g'(x)=4
x+y # f(x)+g(x)=4, f'(x)+g'(x)=6
```

clearly we want to have as output `[4,6]`. If you run the code, Python will give an error message `TypeError: unsupported operand type(s) for +: 'duple' and 'duple'`. This is because `duple` is a class we have defined, and the `+` operator is not defined for this class. However, in Python you can implement these operations using *operator overloading*

```
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 )
```

it is now possible to run the following code

```
x=duple(1,2)
y=duple(3,4)
z=x+y
print(z.top, z.bottom)
```

to get the expected result. It would also be nice to simply do `print(x+y)` or `x+y` and get the result `[4,6]`, instead of accessing variables by e.g. `z.top`. Fortunately Python has functionality for this, `__repr__` and `__str__`. The `__repr__` and `__str__` is what is produced if you simply write `x+y` or `print(x+y)` in the terminal respectively.

Part 1.

- add the following functions to your class, what happens when you do `x+y`?

```
def __repr__(self):
        return "["+str(self.top)+","+str(self.bottom)+"]"

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

will allow you to get nice formatted output.

Part 2. In the code above we have implemented the `+` operator, however if you try to execute `+x` or `-x`, Python will give an error message `TypeError: bad operand type for unary -: 'duple'`.

- Extend your code to make it possible to perform those operations. (Hint: check out `__neg__` and `__pos__` in the documentation)
- Implement the `-` operator using `__sub__`, test that if you do `x-y` you get `[-2,-2]`

5.3 Multiplication and division

So far we have treated the function (top) and derivative (bottom) equally in terms of addition and subtraction. That is because addition and subtraction is a linear operation, i.e. $(f(x) + g(x))' = f'(x) + g'(x)$. For multiplication and division we have different rules

$$(f(x) \cdot g(x))' = f'(x) \cdot g(x) + f(x) \cdot g'(x), \tag{4}$$

$$\left(\frac{f(x)}{g(x)}\right)' = \frac{f'(x) \cdot g(x) - f(x) \cdot g'(x)}{g(x)^2}. \tag{5}$$

Hence we need to implement a different rule for the multiplication operator

Part 3.

1. use the `__mul__` function to implement the `*` operator, use normal multiplication for the top, and equation (4) for the bottom.
2. use the `__truediv__` function to implement the `/` operator, use normal division for the top, and equation (5) for the bottom.

Test of code. Before proceeding to more advanced functions, we can actually do quite a lot with our small code and calculate derivatives of simple functions

```
x=1.2 # just for making it easy to use other values
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))
```

5.4 Adding more functions

So far we have only implemented functions where we explicitly know the derivative, but it would be more convenient if we could implement more complicated functions, such as $\ln(1+e^x)$, without explicitly calculating the derivatives. Actually the only thing we need to do is to implement the kernel rule for the bottom part

$$f(g(x))' = f'(g(x)) \cdot g'(x). \tag{6}$$

Part 4. Finish the code below

```
class dfunction:

    def __init__(self, f, df):
        self.f = f
        self.df = df

    def __call__(self,d: duple): # kernel rule sending in and returning duple
        return duple(self.f(d.top), #fill inn kernel rule here!)
```

If your code works you should be able to run the following code

```
def f(x):
    return np.log(x)
def df(x):
    return 1/x

Log=dfunction(f,df) #note no arguments to f - we are passing the function
Exp=dfunction(np.exp,np.exp) # same here
print("auto diff ", Log(One+Exp(X)))
print("analytical ", np.log(1+np.exp(x)),np.exp(x)/(1+np.exp(x)))
```

Next, we can easily evaluate the function that we evaluated previously using finite differences

```
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)
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))
```

Part 5.

- Write a few sentences about the use of automatic differentiation in some areas
- Make some comments about the strength and weaknesses of automatic differentiation compared to numerical differentiation

6 Appendix A: Passing arguments to functions

In this project one of the tasks is to write a function that can calculate the numerical derivative of a function that needs two additional parameters (`a` and `b`) to be evaluated (see equation (1)). Let us first take a look on how the call signature would be i `f` only depends on x (i.e. if `a` and `b` were fixed), this is easy: we can simply take `f`, `x`, and `h` (or Δx) (step-size) as input arguments to our derivative function. Assuming it is called `calc_derivative`, it could work something like this:

```
df_x = calc_derivative(f, x=1.0, h=1.0e-3)
```

However, the function in equation (1) depends on two additional input parameters, `a` and `b`. We can of course add these two as extra arguments to the derivative function, but then we would lose generality, for not every function has the same two parameters. One way out of this dilemma could be to create a new function for each combination of `a` and `b` that you use, for example:

```
def one_version_of_f(x):
    """
    The function f(x) = sin(10x) * exp(-0.1x^2).
    """
    return f(x, 0.1, 10.0)

def another_version_of_f(x):
    """
    The function f(x) = sin(3.14x) * exp(-0.5x^2).
    """
    return f(x, 0.5, 3.14)
```

Since both of these example functions have their own values for `a` and `b` hard-coded inside them, we do not need to pass anything extra to `calc_derivative`, that is, we can do:

```
df1_x = calc_derivative(one_version_of_f, x=1.0, h=1.0e-3)
df2_x = calc_derivative(another_version_of_f, x=1.0, h=1.0e-3)
```

Another possibility is to use the the args mechanism, which lets you pass around a variable number of parameters to a function. An example of how this works is:

```
def calc_derivative(f, x, h, *args):
    return (f(x, *args) - f(x-h, *args))/h
```

If you use this method, you need to pass the values of `a` and `b` to the function that calculates the derivative, e.g.:

```
calc_derivative(f, x=1.0, h=1.0e-3, a=0.1, b=10.0)
calc_derivative(another_version_of_f, x=1.0, h=1.0e-3, a=0.5, b=3.14)
```

7 Guidelines for project submission

You should keep the following points in mind when working on the project:

- Write the name of all persons working in the group at the top of the notebook.
- The final project report must start with an abstract, the abstract is a self contained summary of the project and should contain quantitative statements.
- Start your notebook by providing a short introduction in which you outline the nature of the problem(s) to be investigated.
- End your notebook with a brief summary of what you feel you learned from the project (if anything). Also, if you have any general comments or suggestions for what could be improved in future assignments, this is the place to do it.
- All code that you make use of should be present in the notebook, and it should ideally execute without any errors (especially run-time errors). If you are not able to fix everything before the deadline, you should give your best understanding of what is not working, and how you might go about fixing it.

- Avoid duplicating code! If you find yourself copying and pasting a lot of code, it is a strong indication that you should define reuseable functions and/or classes.
- If you use an algorithm that is not fully described in the assignment text, you should try to explain it in your own words. This also applies if the method is described elsewhere in the course material.
- In some cases it may suffice to explain your work via comments in the code itself, but other times you might want to include a more elaborate explanation in terms of, e.g., mathematics and/or pseudocode.
- In general, it is a good habit to comment your code (though it can be overdone).
- When working with approximate solutions to equations, it is very useful to check your results against known exact (analytical) solutions, should they be available.
- It is also a good test of a model implementation to study what happens at known 'edge cases'.
- Any figures you include should be easily understandable. You should label axes appropriately, and depending on the problem, include other legends etc. Also, you should discuss your figures in the main text.
- It is always good if you can reflect a little bit around *why* you see what you see.

References

[1] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM computing surveys (CSUR)*, 23(1):5–48, 1991.

[2] Aksel Hiorth. *Computational Engineering and Modeling.* https://github.com/ahiorth/CompEngineering, 2021.

[3] Robert Edwin Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.