PHYS 331 – Numerical Techniques for the Sciences I

Homework 2: Python Introduction Part 1

Posted Monday, August 21st, 2023
Due Friday, September 8th, 2023

Problem 1 – Plotting in Python using Matplotlib (8 points)

(a) *(2 points)* Using the code template provided in the notebook `problem1.ipynb`, modify the function `main_a` in the first cell as indicated to plot $\tanh(x)$, the hyperbolic tangent, between $-5 \leq x \leq 5$. The NumPy and Matplotlib functions `np.arange`, `plt.show`, `plt.xlim`, `plt.xlabel`, `plt.ylabel`, and `plt.plot` may be helpful to you. Be sure to include an appropriate set of axis labels. The plot should display in a new cell below your code.

(b) *(2 points)* Write a function `plotfunc(a)` in the provided notebook that accepts a parameter `a` and plots the function $\tanh(ax)$ in the domain $-5 \leq x \leq 5$.

(c) *(4 points)* Implement code in the `main_bc` function that uses the function `plotfunc` you wrote in part (b) to plot $\tanh(ax)$ for $a = 0.5$, 1.3, and 2.2 all within the same plot. Label the x and y axes and add a legend. Note that to make a single plot that displays all three curves, you can call `plt.plot` in the function `plotfunc`, and call `plt.show` at the end of the `main_bc` function.

Problem 2 – Functions and Control Flow in Python (7 points)
Recall that the Taylor's series expansion of $\sin(x)$ is given by

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots. \tag{1}$$

Write a Python function `taylor_sin(x0, n)` which returns the value of the n-th order term in the Taylor expansion of $\sin(x)$ at the point $x = x_0$. For instance, `taylor_sin(1.7, 3)` should return the numerical floating-point value of $-\frac{(1.7)^3}{3!}$. Note that by our definition, since $\sin(x)$ is an odd function, all terms where n is even are zero. Your solution should work for any non-negative integer value of n. Implement your solution in the notebook `problem2.ipynb`.

Problem 3 – Recursive Functions (10 points)
The n-th Fibonacci number is generated by the sequence beginning with zero where the n-th value is the sum of the previous two elements at $n-1$ and $n-2$. Therefore, the first few elements are given by

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots . \tag{2}$$

Note that we define the sequence to begin with $n = 0$. One of the most natural ways to compute the n-th Fibonacci number is in terms of a *recursive function*, or a function that calls itself. In order to prevent such a function from recursively calling itself an infinite number of times, it is important to identify a *base case*, or a condition that will cause the function to immediately return for a particular input (rather than calling itself again). Implement all of the functions discussed below in the notebook `problem4.ipynb`.

(a) *(2 points)* What is an appropriate base case to use when writing an algorithm to compute the n-th Fibonacci number, where the only input to the algorithm is n?

(b) *(4 points)* Implement the function `fib(n)` using a `for` or `while` loop which calculates and returns the n-th Fibonacci number.

(c) *(4 points)* Implement the function `fib_r(n)` using recursion (and no loops) which calculates and returns the n-th Fibonacci number.

Problem 4 – Small systems of linear equations (10 points)

(a) *(5 points)* Implement a function `LinearSolve2(a, b, c, d, y1, y2)` in the notebook `problem5.ipynb` that solves a 2x2 system of equations

$$\begin{aligned} ax_1 + bx_2 &= y_1, & (3)\\ cx_1 + dx_2 &= y_2, & (4) \end{aligned}$$

for x_1, x_2. Your function should return an error message when the relevant determinant is very close to zero (say $< 10^{-6}$ in magnitude). Test your function on a case that works without problems and a case that fails.

(b) *(5 points)* Extend the previous case to 3x3 systems with a function `LinearSolve3`.

Hint: You may find it useful to define an auxiliary function that calculates determinants of 3x3 matrices. Your `LinearSolve3` function would then use that auxiliary function by taking advantage of Cramer's rule, i.e. using determinants. Write *your own* auxiliary function; you are not allowed to use someone else's linear algebra routines.

Problem 5 – Lists, NumPy arrays, and Masking (6 points)

(a) *(5 points)* Write a function `maskn(lst, i)` in the notebook `problem3.ipynb` which accepts a list of integers `lst` and a single integer `i`, and returns a list called `mask`, of the same length as `lst`, where the elements of `mask` are 0 for each number (in the original list) not divisible by `i`, and 1 for each number that is. The function should work for a list of any length.

(b) *(3 points)* Now do the same thing in the function `maskn_array(ary, i)`, but using NumPy arrays instead of Python lists. Your new function should return a NumPy array of *Booleans* that is False for each number in `ary` not divisible by `i`, and True for each number that is. See if you can write the code for the function in one line!

(c) *(2 points)* Which one do you expect to be faster? Try using the `%%timeit` command from class on both functions (pick a suitable test case, such as a list/NumPy array of integers from 0 to 10,000). Does the timing match your expectations?