ECSE 343: Numerical Methods in Engineering

Assignment 0: Floating Point and Vectorized Python

Due: Thursday, January 20th, 2022 at 11:59pm EST on myCourses

Final weight: **0% + 5% bonus**

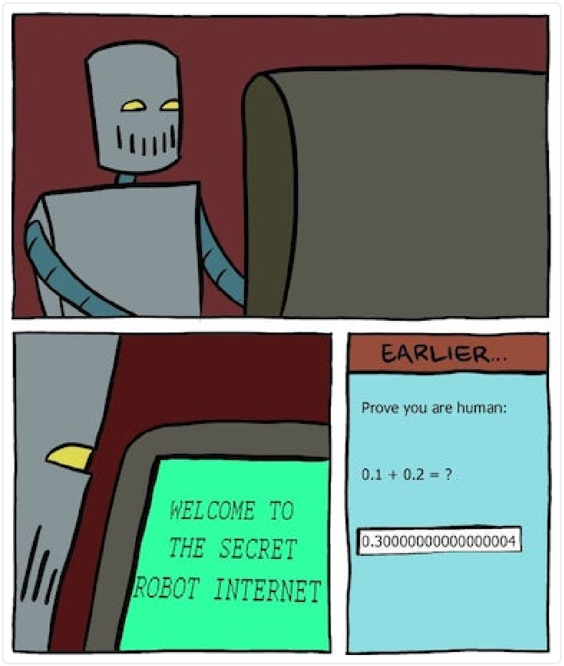


image courtesy of www.smbc-comics.com/index.php?db=comics&id=2999

Contents

1 Assignment Policies and Submission Process
1.1 Late policy
1.2 Installing `Python`, Libraries and Tools
1.3 `Python` Language and Library Usage Rules
1.4 Let's get started... but let's also not forget...
2 Floating point number systems
2.1 Fictitious Floating Point Systems
3 Vectorizing **`Python`** with **`numpy`**
3.1 Slicing and dicing `numpy.arrays`
3.2 Avoiding `for` loops
4 Computer arithmetic
4.1 Catch the NaN
4.2 Floating point errors
5 You're Done!

1 Assignment Policies and Submission Process

Download and modify the standalone `Python` script we provide on *myCourses*, renaming the file according to your student ID as

`[YourStudentID].py`

For example, if your id is **234567890**, your submission filename should be **234567890.py** and should include all the solutions you plan to submit for this assignment, according to the instructions below.

ⓘ Every time you submit a new file, your previous submission will be overwritten. We will only grade the **final submitted file**, so feel free to submit often as you progress through the assignment.

1.1 Late policy

All the assignments are to completed individually, unless stated otherwise. You are expected to respect the late policy and collaboration/plagiarism polices.

1.2 Installing `Python`, Libraries and Tools

The assignments and group project are to be implemented using `Python 3.7` and the latest version of the `numpy` and `matplotlib` libraries.

You are free to install and configure your development environment, including IDE choices, as you wish. One popular, more self-contained installation package that we recommend is the **Individual Edition** of the Anaconda software installer.

After following your platform specific installation instructions, the Anaconda Navigator provides a simple graphical interface that allows you to:

- define isolated development *environments* with the appropriate `Python` version (3.7)
- download and install the required libraries (`numpy` and `matplotlib`), including their dependencies, into the environment, and
- [optionally] to pick between a variety of development IDEs.

1.3 `Python` Language and Library Usage Rules

`Python` is a powerful language, with many built-in features. You should feel free to explore the base language features and apply them as a convenience, whenever necessary. A good example is that, if you need to sort values in a list before plotting them, you should feel free to use the built-in `sort` function rather than implementing your own sorting algorithm (although, that's perfectly fine, too!):

```
myFavouritePrimes = [11, 3, 7, 5, 2]

# In ECSE 343, learning how to sort a list is NOT a core learning objective
myFavouritePrimes.sort() # 100% OK to use this!

print(myFavouritePrimes) # Output: [2, 3, 5, 7, 11]
```

We will, however, draw exceptions to the use of (typically external) library routines that allow you to shortcut through the *core learning objective(s)* of an assignment. For example, if we ask you to develop a linear solver and apply it to a problem, and you instead rely on calling one of `numpy`'s built-in solvers, you will be penalized. When in doubt as to whether a library (or even a built-in) routine is “safe” to use in your solution, please **contact the TA**.

Python 3.7 has a built-in convenience `breakpoint()` function which will break code execution into a debugger, where you can inspect variables in the debug REPL and even execute code! This is a very powerful was to test your code *as it runs* and to tinker (e.g., inline in the REPL) with the function calling conventions and input/output behaviour of code.

Be careful, as you can change the execution state (i.e., the debug environment is not isolated from your scripts execution stack and heap), if you insert REPL code and then continue the execution of your script from the debugger.

To help, the (purposefully minimal) base code we provide you with includes a superset of all the library `import`s we could imagine you using for the assignment.

You must not use any additional `imports` for your solution, other than the ones provided by us.

Doing so will result in a score of **zero (0%)** on the assignment.

```
import matplotlib.pyplot as plt   # for plotting
import numpy as np                # all of numpy, at least for this assignment
```

As you may have noticed, this course will rely *heavily* on `numpy` — in fact, you'll likely learn just as much about the power (and peculiarities) of the `Python` programming language as you will about the `numpy` library. This library not only provides convenience routines for matrix, vector and higher-order tensor operations, but also allows you to leverage high-performance vectorized operations if you're careful about restructuring your data/code in a vectorizable form. This assignment will briefly expose you to some of these nuances; those of you with `MATLAB` experience will find this coding paradigm familiar, but also a little painful, since many foundational conventions are different (e.g., 0- versus 1-indexing, column- vs. row-major default behaviours, broadcasting conventions, etc.)

1.4 Let's get started... but let's also not forget...

With these preliminaries out of the way, we can dive into the assignment tasks. **Future assignment handouts will not include these preliminaries, although they will continue to hold true.** Should you forget, they will remain online in this handout for your reference throughout the semester.

2 Floating point number systems

Any *floating point number system* can be characterized by four parameters, (β, t, L, U), where

- β is the *base* of the number system,
- t is its *precision* (i.e., number of significant digits),
- L is the lower bound on the exponent e, and
- U is the upper bound on the exponent e.

Given an instance of any such system, we can express a real number x in its floating point representation $\mathrm{fl}(x)$ as:

$$\mathrm{fl}(x) \equiv \underbrace{\pm}_{\text{sign}} \left(\underbrace{\frac{d_0}{\beta^0} + \frac{d_1}{\beta^1} + \cdots + \frac{d_{t-1}}{\beta^{t-1}}}_{\text{mantissa}} \right) \times \underbrace{\beta^e}_{\text{exponent}}$$

where the base β is an integer greater than 1, the exponent e is an integer between L and U (inclusive, i.e., $L \leq e \leq U$) and the digits d_i are integers in the range $0 \leq d_i \leq \beta - 1$. The number $\mathrm{fl}(x)$ is usually an approximation of x, unless it happens to fall on one of the (relatively few) numbers that can be *perfectly represented* in the floating point system. For any non-zero value, we normally force $d_0 \neq 0$ by adjusting the *exponent* e so that leading zeros are dropped. As such, the smallest (in magnitue) perfectly representable *non-zero* number has a mantissa of $(1.0 \cdots 0)_\beta$.

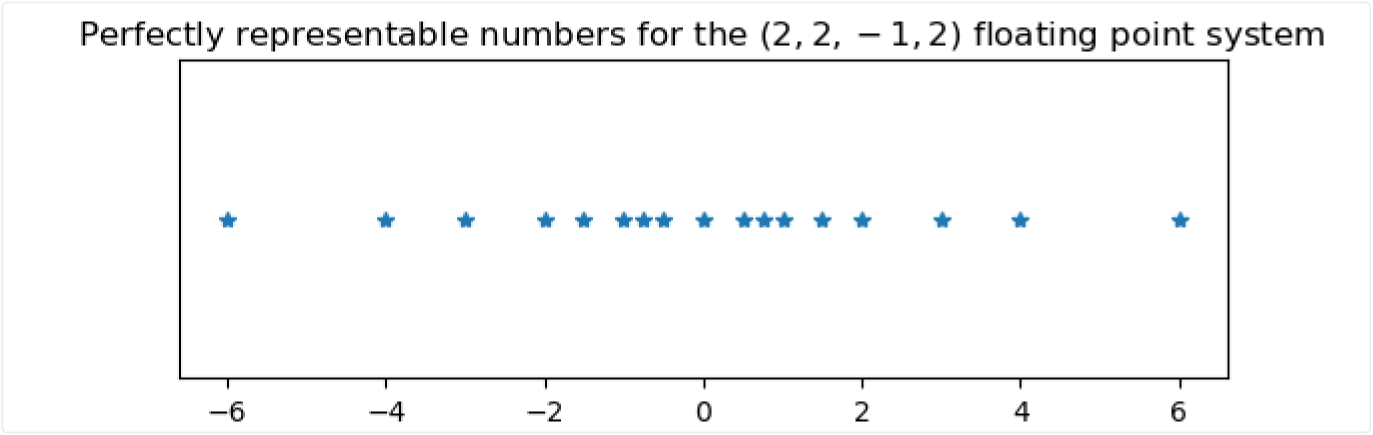
2.1 Fictitious Floating Point Systems

Let's get a better sense of how far removed representable floating point numbers can get from real numbers.

☑ **Deliverable 1 [20%]**
Complete the implementation of the function `FloatSystem` that returns a 1D `numpy.array` of all the *perfectly representable numbers* (in base 10) for a given floating point number system. You can safely ignore the NaN and $\pm\infty$ cases, and explicitly add `0` to your representable numbers output (i.e., without having to treat any special case exponent settings).

Here's a code snippet with an example visualization of the expected output of this function for a fictitious floating point system with $\beta = 2$, $t = 2$, $L = -1$ and $U = 2$. Any real numbers that don't fall *exactly* on the stars in the plot below cannot be *perfectly* represented in this fictitious floating point number system.

```python
# Plot an asterisk at each perfectly representable value along the real line
plt.title('Perfectly representable numbers for the $(2,2,-1,2)$ floating point
system')
tmp = FloatSystem(2, 2, -1, 2)
plt.plot(tmp, zeros(tmp.shape[0]), '*')
plt.yticks([])
plt.show()
```



The perfectly representable real numbers in a fictitious floating point system example.

3 Vectorizing Python with numpy

Writing numerical code requires balancing several (sometimes competing) design parameters: correctness of the code, numerical stability (i.e., in the floating point sense of the word), and the efficiency and scalability of the code are among these parameters.

Python is undoubtedly a flexible and powerful language, affording numerical software developers with many tools to tackle their modeling and simulation tasks — however, as an interpreted language, Python's performance cannot compete with lower-level optimized code generated from, e.g., a compiler. Luckily, Python allows for callable modules and libraries that need not be implemented in Python but rather in any number of higher performance compiled languages. Moreover, Python's vast ecosystem of specialized libraries often comprise high-performance compiled backends: in this sense, Python serves just as much as a high-level "glue" language as it does as a standalone one.

For numerical computation, numpy is one such library that is implemented in highly-optimized machine code. When used appropriately, numerical code implemented in a manner that leverages numpy's ability to efficiently perform data-parallel operations over **vectors** and **higher-order tabular data** can be *several orders of magnitude* more efficient than its Python-only equivalent.

One could easily teach an entire course on how to write efficient numpy code, and that is not the main goal of ECSE 343; however, learning to *think* about numerical operations in *vectorized* form whenever appropriate will open up the opportunity for cleaner, more readable and (much) more efficient code.

3.1 Slicing and dicing numpy.arrays

Multi-dimensional arrays are fundamental data structures in numerical computation, and numpy implements sophisticated indexing schemes that respect specialized broadcasting rules, in addition to treating multi-dimensional arrays as first-class objects in all the library's exposed functions.

The next deliverable will give you a brief sense of the power and flexibility of *some* of numpy's indexing notation. It is not meant, by any means, to be comprehensive; instead, the learning goal here is to open the door to your independent exploration of numpy in order to facilitate implementation tasks, e.g., in future assignments.

Deliverable 2 [20%]
This is a written answer-only deliverable: answer these questions using Python comments (i.e., not code) in your submission .py file. Feel free to experiment with indexing schemes using the Python REPL or the __main__, in support of your written answers (i.e., do not regurgitate online

documentation but, rather, run tiny code snippets to support any understanding you gain with the support of online documentation.)

Answer the questions below given two `numpy.array` variables, the first (**a**) has a `shape` of `(3,)` and the second (**b**) has a `shape` of `(4,5)`:

1. What do **`b[0:3]`** and **`b[:, 0:3]`** do?
2. Why does **`b[:,:] = b[1,:]`** work and how would you make **`b[:,:] = b[:,1]`** work? *Hint:* one of the more elegant solutions relies on using `None`.
3. Write a one-line `Python`/`numpy` expression that returns a `numpy.array` with shape of `(5,)` with elements **`[a[0], a[1], a[2], a[0], a[1]]`**, but without this explicit parameter list (i.e., your answer should **not** be **`numpy.array([a[0], a[1], a[2], a[0], a[1]])`** but should use an indexing expression on **a**.) *Hint*: one of the many valid ways to do this requires using a `Python` *list comprehension*.
4. How are the outputs of **`a[2]`**, **`a[[2]]`** and **`a[[2,np.newaxis]]`** different?
5. Why is **`a[a % a.shape[0]]`** guaranteed to work whereas **`a[a]`** may not?

Without coming anywhere close to exhaustively enumerating effecient `numpy` coding practices, to first-order approximation, the following advice is a good place to start:

- avoid `for` loops by restructuring data (if needed) into (potentially high-dimension) `numpy.array`s, and then performing operations *across* subsets of the data,
- map and reduce strategies, often applied across `numpy.array` dimensions are a common strategy, and
- leverage `numpy`'s many built-in vectorized conditional, mathematical, and logical utility functions.

3.2 Avoiding `for` loops

Deliverable 3 [20%]
Perhaps the simplest example of a vectorizable mathematical operation is the computation of the scalar dot product of two 1D vectors. Complete the implementation of the `SlowDotProduct` and `FastDotProduct` functions, both of which accept two 1D `numpy.array`s (you can assume they have equal shape) and returns the scalar dot product of the two vectors. The `SlowDotProduct` routine should not use any `numpy` utility functions and, instead, rely on `Python` language math operators and `for` loops. The `FastDotProduct` function should instead use the full functionality of `numpy`.

4 Computer arithmetic

Let's learn about errors due to floating point number representations, using simple numerical algorithms as examples.

In `Python`, unlike other languages like C/C++/Java, any expression containing an integer and floating point arithmetic is automatically cast to the highest precision (64-bit) floating point representation to favour precision over efficiency. While `Python` itself is a loosely typed language, `numpy` has explicit facilities to force underlying numerical type representations, such as `numpy.float64` and `numpy.float32` for double- and single-precission floating point values.

4.1 Catch the NaN

In the IEEE floating point standard, NaNs are "infectious". You can use the `Python` REPL to explore how NaNs behave with different arithmetic, inequality and logical operations. To do so, you can use `numpy`'s NaN as `nan = np.float64("nan")` and then tinker with expressions like, e.g., `nan < 4` or `nan == nan` or `3 + nan`, etc.

☑ **Deliverable 4 [20%]**
Tracking down a NaN in a numerical algorithm can sometimes be a real pain. You will implement a routine `CatchTheNaN` that identifies the source of a single spurious NaN in a very simple numerical algorithm. The function takes as input a single (potentially very large) 2D matrix $\mathbf{M}$ with many NaNs in it. Behind the scenes (i.e., before the function is called), it turns out that $\mathbf{M}$ was constructed as the **outer product** of two 1D column vectors $\mathbf{x}$ and $\mathbf{y}$, i.e., such that $\mathbf{M} = \mathbf{x}\,\mathbf{y}^{\mathrm{T}}$. Both $\mathbf{x}$ and $\mathbf{y}$ each have a *single* NaN (i.e., in only one of their vector elements), and `CatchTheNaN` should return an `np.array` of shape `(2,)` of the indices of these NaNs, where element `[0]` is the index of the NaN in $\mathbf{x}$ and element `[1]` is the index of the NaN in $\mathbf{y}$. We will grade you on the correctness **and** performance of your code, i.e., avoid using brute force search with `for` loops.

4.2 Floating point errors

There are many types of floating point error your numerical code will be susceptible to, some of which you can guard against... some of the time. Examples include

- *catastrophic cancellation* resulting in loss in significant digits when, e.g.,
    - summing numbers with different scales, e.g., $10^{38} + 10^{-24}$, or

    - summing numbers with differences in low-precision bits, e.g., $1.38383 \times 10^{38} - 1.38382 \times 10^{38}$,
- *round-off errors* that accumulate, e.g., $\sum_{i=0}^{10^{38}} 1.0$, and
- *overflow* and *underflow*, e.g., $10^{24} \times 10^{50}$, to name a few.

Consider the following concrete example: when evaluating the natural logarithm of the sum of N exponentials of a set of values x_i,

$$y = \log \left(\sum_{i=1}^{N} \exp(x_i) \right) ,$$

the differences in the magnitude of the exponential terms of the summands may lead to overflow during accumulation, depending on the value of N.

After careful mathematical manipulation, leveraging the properties of logarithms and exponentials, we can rewrite this equation as

$$y = m + \log \left(\sum_{i=1}^{N} \exp(x_i - m) \right) ,$$

where m is the *maximum* value among the x_i,

$$m = \max_i x_i \ .$$

This alternative mathematical formulation — when implemented in a floating point system — will be more stable to the variations in the scale of the exponential terms.

☑ **Deliverable 5 [20%]**
Implement and compare a naive version of this sum (in `LogSumExpNaive`) and a more numerically robust version (in `LogSumExpRobust`). Each function takes a single 1D `numpy.array` with a shape of `(N,)` as input and returns the scalar sum. You can use the `__main__` test suite to explore the differences in the outputs of these functions, particularly for large N and/or x_i elements in different ranges of minimum and maximum magnitude.

Include a brief (i.e., one- to two-sentence long) comment in your solution implementation of `LogSumExpRobust` that explains *why* this reformulation leads to more robust floating point calculations.

5 You're Done!

Congratulations, you've completed the zeroth assignment. Review the submission procedures and guidelines at the start of the handout before submitting the `Python` script file with your assignment solution.