4 — PHY 494: Homework assignment (33 points total)

Due Thursday, Feb 15, 2018, 11:59pm.

Submission is now to your **private GitHub repository**. Follow the link provided to you by the instructor in order for the repository to be set up: It will have the name *ASU-CompMethodsPhysics-PHY494/assignments-2018-*YourGitHubUsername and will only be visible to you and the instructor/TA. Follow the instructions below to submit this (and all future) homework.

Read the following instructions carefully. Ask if anything is unclear.

1. `git clone` your assignment repository (change *YourGitHubUsername* to your GitHub username)

   ```
   repo="assignments-2018-YourGitHubUsername.git"
   git clone https://github.com/ASU-CompMethodsPhysics-PHY494/${repo}
   ```

2. run the script `./scripts/update.sh` (replace *YourGitHubUsername* with your GitHub username):

   ```
   cd ${repo}
   bash ./scripts/update.sh
   ```

   It should create three subdirectories[1] `assignment_04/Submission`, `assignment_04/Grade`, and `assignment_04/Work`.

3. You can try out code in the `assignment_04/Work` directory but you don't have to use it if you don't want to. Your grade with comments will appear in `assignment_04/Grade`.

4. Create your solution in `assignment_04/Submission`. Use Git to `git add` files and `git commit` changes.

   You can create a PDF, a text file or Jupyter notebook inside the `assignment_04/Submission` directory as well as Python code (if required). **Name your files `hw04.pdf` or `hw04.txt` or `hw04.ipynb`**, depending on how you format your work. Files with code (if requested) should be named exactly as required in the assignment.

5. When you are ready to submit your solution, do a final `git status` to check that you haven't forgotten anything, commit any uncommited changes, and `git push` to your GitHub repository. Check on *your* GitHub repository web page[2] that your files were properly submitted.

   You can push more updates up until the deadline. Changes after the deadline will not be taken into account for grading.

Homeworks must be legible and intelligible and on-time or may be returned ungraded with 0 points.

[1] If the script fails, file an issue in the Issue Tracker for PHY494-assignments-skeleton and just create the directories manually.

[2] `https://github.com/ASU-CompMethodsPhysics-PHY494/assignments-2018-`*`YourGitHubUsername`*

Bonus problems This assignment contains **bonus problems**. A bonus problem is optional. If you do it you get additional points that count towards this homework's total, although you can't get more than the maximum number of points. If you don't do it you can still get full points. Bonus problems and bonus points are indicated with an asterisk "*".

Included code and tests The homework comes with starter code in the `Submission` directory. Edit and submit code as directed in the problems. The directory also includes a file `test_hw4.py`. You can use these tests to check if your solutions are correct:

```
pytest -v test_hw4.py
```

(If you solved all coding problems, you should see "9 passed"; if you also solved the Bonus problem 4.3(c) you should see "7 xpassed". Otherwise you will be informed which problems failed.)

4.1 NumPy arrays (11 points)

Work through the NumPy tutorial.[3] Do the examples while you read it.

(a) How do NumPy array operations such as $+$, $-$, $*$, $/$ ... differ from linear algebra operations (i.e. scalar product, vector/matrix multiplication, ...)? **[2 points]**

(b) For the following, add your code to the file `problem1.py`. Given the three arrays

```
import numpy as np

sx = np.array([[0, 1], [1, 0]])
sy = np.array([[0, -1j],[1j, 0]])
sz = np.array([[1, 0], [0, -1]])
```

(i) What is the result of `result1b1 = sx * sy * sz`? Explain what NumPy array multiplication does to the arrays. (Note: your code should assigne the result to the variable `result1b1` in `problem1.py`.) **[2 points]**

(ii) Use `np.dot()` to multiply the three arrays (like $\sigma_x \cdot \sigma_y \cdot \sigma_z$). Add your code to `problem1.py` and assign your result to variable `result1b2` Show your result and explain what happened. **[2 points]**

(iii) Compute the "commutator" $[\sigma_x, \sigma_y] := \sigma_x \cdot \sigma_y - \sigma_y \sigma_x$ and show that it equals $2i\sigma_z$.[4] Add your code to `problem1.py`, assign the result to variable `result1b3`. **[3 points]**

[3]Some stuff such as the `ix_()` function is fairly esoteric for beginners but almost everything else is what you should be familiar with for your daily work with arrays.

[4]These are the Pauli matrices that describe the three components of the spin operator for a spin 1/2 particle, $\hat{\mathbf{S}} = \frac{\hbar}{2}\boldsymbol{\sigma}$. The fact that any two components of the spin operator do *not* commute is a fundamental aspect of quantum mechanics.

(iv) Given a "state vector"

$$\mathbf{v} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -i \end{pmatrix}$$

calculate the "expectation value" $\mathbf{v}^\dagger \cdot \sigma_y \cdot \mathbf{v}$ (i.e., the multiplication of the hermitian conjugate of the vector, $\mathbf{v}^\dagger$ with the matrix σ_y and the vector $\mathbf{v}$ itself) using NumPy. [5] Add your code to `problem1.py` and assign your result to variable `result1b4`.[6]) **[2 points]**

4.2 Coordinate manipulation with NumPy (16 points)

We can represent the cartesian coordinates $\mathbf{r}_i = (x_i, y_i, z_i)$ for four particles as a numpy array `positions`:

```
import numpy as np
positions = np.array(\
    [[0.0, 0.0, 0.0], [1.34234, 1.34234, 0.0], \
     [1.34234, 0.0,  1.34234], [0.0, 1.34234, 1.34234]])
t = np.array([1.34234, -1.34234, -1.34234])
```

and `t` will be a translation vector. *For the following use NumPy.* Add your code to file `problem2.py` and assign results to variables as indicated in the problems.

(a) What is the *shape* of the array `positions` and what is its *dimension*? **[1 points]**

(b) What is the *shape* of the array `t` and what is its *dimension*? **[1 points]**

(c) How do you access the coordinates of the second particle in `positions`? Assign the result to variable `result2c`. **[1 points]**

(d) For the second particle:

  (i) How do you access its y-coordinate? Assign the result to variable `result2d`. **[2 points]**

  (ii) What type of object is this output, what is its *shape* and its *dimension*? **[2 points]**

[5] The *hermitian conjugate* $\mathbf{v}^\dagger = (v_1^*, v_2^*)$ is `v.conjugate().T` where `v.T` is shorthand for `v.transpose()`. It turns out that you don't need the transposition when you use `np.dot()` but I include it here for conceptual clarity. (Including `transpose()` comes at a minor performance penalty — check with `%timeit` if you are curious.)

[6] Note for anyone having done PHY 315 (Quantum Mechanics II) that here you are calculating the quantum mechanical expectation value of the y-component of a spin $\frac{1}{2}$ particle in an eigenstate of the operator of the y-component of the spin ($\sigma_y \mathbf{v} = -\mathbf{v}$) and because $\mathbf{v}$ is normalized, you should get the eigenvalue as the expectation value.

(e) Write Python code to translate all particles by a vector $\mathbf{t} = (1.34234, -1.34234, -1.34234)$,

```
t = np.array([1.34234, -1.34234, -1.34234])
```

Add your code to `problem2.py` and assign the translated coordinates to variable `result2e`. [**3 points**]

(f) Make your solution of (e) a function `translate(coordinates, t)`, which translates all coordinates in the argument `coordinates` (an `np.array` of shape `(N, 3)`) by the translation vector in `t`. The function should return the translated coordinates as a numpy array.

Add the function to `problem2.py`. Show the results of the function applied to (1) the input `positions` and `t` from above and (2) for `positions2 = np.array([[1.5, -1.5, 3], [-1.5, -1.5, -3]])` and `t = np.array([-1.5, 1.5, 3])`. [**6 points**]

4.3 NumPy functions (6 points)

(a) We want to plot the function [7]

$$\mathrm{sinc}(x) := \frac{\sin(\pi x)}{\pi x} \tag{1}$$

over the range $-6 \leqslant x < 6$.

- Use the NumPy `arange()` function to generate an array `X` with values from -6 to 6 in steps of 0.2.[8].
- Apply the NumPy `sinc()` function to the `X` array[9] and assign it to a variable `Y`.
- Plot your data with matplotlib

```
import matplotlib.pyplot as plt
plt.plot(X, Y)
plt.xlabel("x")
plt.ylabel("y = sinc(x)")
plt.savefig("sinc.png")     # write plot to file
```

Submit your code as file `problem3a.py` together with the plot in file `sinc.png`. [**4 points**]

[7] This is the definition used in numpy.sinc function.

[8] You don't have to include the upper endpoint 6 in the range because this can be difficult to achieve with `arange()` and a floating point `step`; as an alternative you can look into using `numpy.linspace()`.

[9] You do *not* need any loops. Try `numpy.sinc(X)` and embrace NumPy!

(b) Use the NumPy `arange()`, the `sum()`, and the `sqrt()` functions to calculate the sum[10]

$$S = \sqrt{\sum_{n=1}^{100} \frac{6}{n^2}}. \tag{2}$$

Put your code into file `problem3b.py` and assign the result to a variable `mypi`. **[2 points]**

(c) Bonus: : Write a function to approximate the real-valued Riemann Zeta function

$$\zeta(s) = \sum_{k=1}^{\infty} \frac{1}{k^s} \tag{3}$$

with the finite sum up to $N_{\max}$ as

$$\zeta(s) \approx \zeta(s; N_{\max}) := \sum_{k=1}^{N_{\max}} \frac{1}{k^s} \tag{4}$$

Add your function `zeta(s, Nmax=1000)` to a file `problem3c.py` and plot $\zeta(s)$ in the range $1 < s \leqslant 10$ and for a range of $N_{\max}$ and include a figure of the plot `zeta.png`.[11] **[bonus +4*]**

You can plot multiple graphs within the same plot and add a legend with matplotlib:

```python
import numpy as np
import matplotlib.pyplot as plt

def zeta(s, Nmax=1000):
    """Approximation to the real Riemann zeta function"""
    # add your code ...

s = np.arange(1, 10, 0.1)
Nmax_values = np.array([10, 100, 1000, 100000])

fig = plt.figure(figsize=(6, 6))    # new figure
ax = fig.add_subplot(111)           # add "axes", i.e., graph to figure
```

[10] You can compare your result to the analytical solution

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}.$$

[11] Note that the sum in problem (b) $\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}$ is equal to $\zeta(2)$...

```
# plot each graph into the same axes and label by Nmax
for Nmax in Nmax_values:
   ax.plot(s, zeta(s, Nmax=Nmax), label="Nmax="+str(Nmax))

# finish the axes by adding labels and legend
ax.set_xlabel(r"$s$")          # fancy LaTeX typeset label
ax.set_ylabel(r"$\zeta(s)$")  # fancy LaTeX typeset label
ax.legend(loc="best")          # place legends

fig.savefig("zeta.png")
```

(Add the `label="something"` keyword to the `plot()` method, plot everything into the same graph (called an "axes" in matplotlib), and then call the `legend()` method and save the figure.)

4.4 BONUS: File processing in Python (15* bonus points)

The standard way to open a file in Python and to process it line by line is the code pattern

```
with open(filename) as inputfile:
   for line in inputfile:
       line = line.strip()   # strip trailing/leading whitespace
       if not line:
          continue           # skip empty lines
       # now do something with a line
       # E.g., split into fields on whitespace
       fields = line.split()
       # access data as fields[0], fields[1], ...
       x = float(fields[0])  # convert text to a float
       y = float(fields[1])
       # ...
print("Processed file ", filename)
```

In brief:

1. A file is opened for reading with `open(filename)`, which returns a *file object* (here assigned to the variable `inputfile`). The `with` statement is a very convenient way to make sure that the file is always being closed at the end: when the `with`-block exits (here at the `print` statement), `inputfile.close()` is called implicitly[12].

[12] If you were not to use `with`, your code would look like

```
inputfile = open(filename)
for line in inputfile:
   # ...
inputfile.close()
```

2. We *iterate* over all lines in the file (similar to what we did for lists) in a `for`-loop.
3. Remove leading and trailing white space with the `strip()` method of a string (`line` is a string). If you want to keep all white space, do not use `strip()`.
4. Skip empty lines: note that an empty string evaluates to `False` and thus can be used directly in the `if` statement. The `continue` statement then starts the next iteration in the loop.
5. Start processing the line. Often you know the structure of the file (e.g. a data file with 3 columns, separated by white space) so you typically split into fields (the string's `split()` method produces a list). Select the fields as needed.
6. As an example, fields 0 and 1 are assumed to represent floating point numbers. `fields[0]` contains a string but using `float(fields[0])` it can be converted ("cast") to a Python float. Similarly, integer numbers can be cast with `int()`.

Use the above information to write a Python program `evaluate_ships.py` that reads the file

`PHY494-resources/01_shell/data/starships.csv`

splits lines on commas[13], and prints out the names and cost (in credits, "CR") of all starships that cost more than 100 million CR.[14]

Submit your code `evaluate_ships.py` and your output in a file `starship_costs.dat`. **[bonus +15*]**

```
print("Processed file ", filename)
```

but with the disadvantage that when something goes wrong during the `for`-loop, your file will never be closed, which exhausts system resources. When open a file for *writing* (`open(filename, 'w')`) you will *corrupt the file* when you are not closing it properly. The `with` statement guarantees that the file will *always be closed, no matter what else happens.* Use the `with` statement!

[13] "csv" stands for "comma separated values" and is a common file format for tabular data.

[14] Hint: Turn all `"unknown"` entries into 0 and then cast numbers to floats.