The deadline for this exercise sheet is **Monday, 18.06.2018, 10:00.**

Introductory Words

Remember that you need proper documentation to pass the homework. The documentation doesn't need to be *perfect*, but everything that needs a docstring, should have a docstring.

Please make sure that your libraries are up to date. For numpy your version must be ≥ 1.13 and for matplotlib ≥ 2.2. You can check the installed version either using pip: `pip show numpy matplotlib` or directly in Python:

```
import numpy as np
import matplotlib as mpl

print("Numpy:", np.__version__)
print("Matplotlib:", mpl.__version__)
```

If you need to upgrade either, use conda or pip: `conda install numpy matplotlib` or `pip install numpy matplotlib`. This should install the latest version.

1 Warm-Up

1.1 Number Stuff with NumPy

Given the following ndarrays:

```
import numpy as np



a = np.array([1, 2, 3])
b = np.array([-1, 2, -3])
c = np.arange(1, 10).reshape(3,3)
d = np.array([[1,2],[-2,-1],[5,5]])
```

calculate the following operations. Order matters.

- The dot product of a and b **Solution:** `np.dot(a, b)`
- The dot product of c and a **Solution:** `np.dot(c, a)`
- The matrix multiplicative of c and d **Solution:** `np.matmul(c,d)`
- The matrix multiplicative of the transpose of d and c **Solution:** `np.matmul(d.T, c.T)`
- the overall mean of a and b together **Solution:** `np.mean([a,b])`

Solution:

```python
"""A simple script to test some numpy functionality"""
import numpy as np


# the arrays we are going to work with
a = np.array([1, 2, 3])
b = np.array([-1, 2, -3])
c = np.arange(1, 10).reshape(3, 3)
d = np.array([[1, 2], [-2, -1], [5, 5]])

# sine none of the results are further needed we do the calculation directy
# in the output function
# Note that f-strings evaluate the statements in curly braces as python
# statements before converting them to string. Therefore, we can do all the
# numpy calculations inside the f-strings
print(
    f"a: {a}", f"b: {b}", f"c:\n{c}", f"d:\n{d}",
    f"dot(a, b): {np.dot(a, b)}",
    f"dot(c, a): {np.dot(c, a)}",
    f"c * d:\n{np.matmul(c, d)}",
    f"d^T * c^T:\n{np.matmul(d.T, c.T)}",
    f"mean(a, b): {np.mean([a,b]): .3f}",
    sep="\n"
)
```

1.2 Plot, plot away

1.2.1 Crossed out

Plot the following functions from -4 to 8 into one figure. Use at least 100 points.
Note: You only need to create *one* ndarray for the x-coordinates.

- $y_1 = 2x - 2$
- $y_2 = -2x + 6$
- $y_3 = 0.25(x - 2)^2 + 2$
- $y_4 = -0.25(x - 2)^2 + 2$

1.2.2 Markers all around

From NumPy's random module use the normal function to create 1000 randomly drawn numbers. Draw from a normal distribution with center (loc) 0 and a standard deviation (scale) of 5. Then plot those numbers into a histogram, with 25 bins, and a bar width of 0.9.
Have a look at the numpy documentation for random.normal.

1.2.3 Who is that plot?!

In the accompanying .zip archive you can find the `plot_me` module. Import the `get_x_y` function from it. The function will return two numpy ndarrays in the form `x, y`, which are x and y coordinate pairs. Plot those pairs using a scatter plot, with a scale (`s`) of 0.5.
Note: The function can take a few seconds to compute.

Solution:

```python
"""
This script implements the functionality of the whole task 1.2

Each subtask is plotted in its own figure so everytime one is closed the
next one opens to show the result.

Only works with the module plot_me present.
"""
import numpy as np
import matplotlib.pyplot as plt

from plot_me import get_x_y # convenience

# ------------------------------------------------------------------------------
# Task 1.2.1: Crossed Out
# ------------------------------------------------------------------------------
# since the functions are rather simple we can implement them as lambdas
y1 = lambda x: 2*x - 2
y2 = lambda x: -2*x + 6
y3 = lambda x: -0.25*(x - 2)**2 + 2
y4 = lambda x: 0.25*(x - 2)**2 + 2

# 100 should be enough for a rather smooth curve
x = np.linspace(-4, 8, 100)

# and plot them all. We could have done that in one plot function, but I prefer
# the longer version.
fig = plt.figure()
plt.title("Crossed Out") # they all meet in one point :)
plt.plot(x, y1(x))
plt.plot(x, y2(x))
plt.plot(x, y3(x))
plt.plot(x, y4(x))
plt.show()

# ------------------------------------------------------------------------------
# Task 1.2.2: Markers All Around
#
```

```
------------------------------------------------------------------------
fig = plt.figure()
# draw the sample from a normal distribution, with center 0 and std 5
# we want a 1d array with 10000 entries
y = np.random.normal(0, 5, (10000))
plt.title("Markers All Around")
# and plot the thing. bins as an integer automatically divides the whole range
# into evenely spaced bins. rwidth as .9 so they are cleared distinguishable
plt.hist(y, bins=25, rwidth=.9)
plt.show()

# ------------------------------------------------------------------------
# Task 1.2.3: Who Is That Plot
# ------------------------------------------------------------------------
fig = plt.figure()
x, y = get_x_y() # get the data from the supplied function
plt.title("Who Is That Plot?!") # it's a pikachu!
# setting the scale makes it more visible as the dots are very close
plt.scatter(x, y, s=0.5)
plt.show()
```

2 The great Plotting

2.1 Wavy Waves

Using your new gained knowledge, recreate the figure in the image below 1. The graphs are plotted from -2π to 2π. You do not have to modify the ticks on the x-axis. If you want to recreate those as well, look into the rc doc, which allows you to set all kind of formatting. Especially interesting is the `usetex` boolean property of `text`. Also see this explanation.

2.2 Bonus: All alive.

Let's bring this plot to life! Animate the sine waves in such a way, that the top left sine wave moves to the left and the top right sine wave moves to the right. Wave-y!

2.3 Bonus-Bonus: Wave the wave

Now that we got that animation up and going, let's animate the bottom sine wave as well. This time not just left and right, but up and down as well. Move the whole sine wave on a sine wave. So that the whole curve shifts to the left

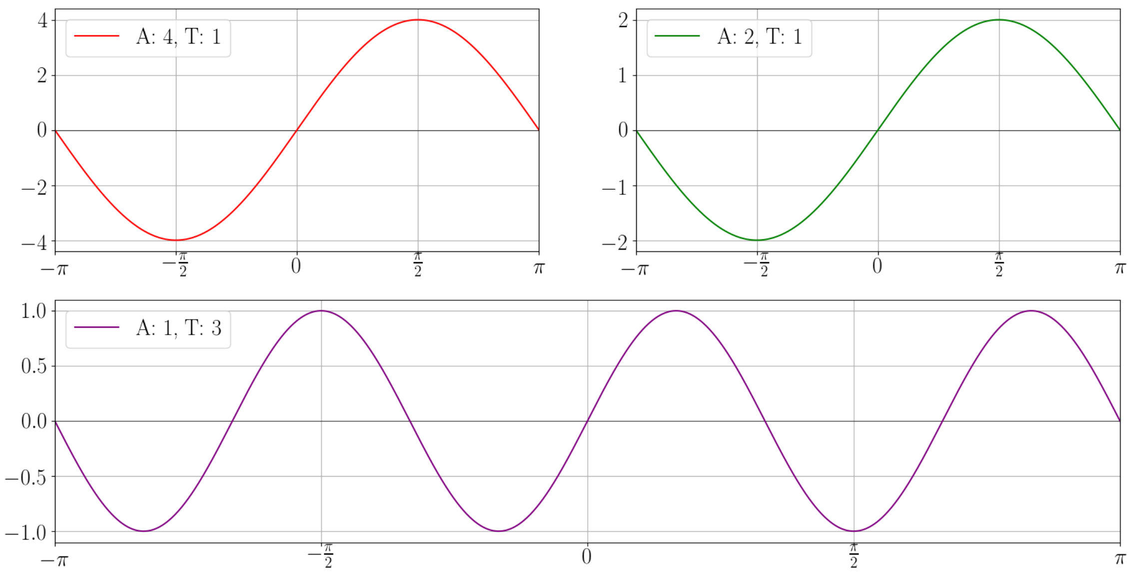

Figure 1: Three sine waves with different properties

as well as up and down as if it was moving along another sine wave.
This is a tricky one.

Solution:

```
"""
This script plots three animated sine curves.

The sine curves have different amplitudes and periods, but are animated
using the same amount of time to complete one period. Thus one sine wave moves
"faster".

There are constants at the top of the file which change the output. Since the
animation of three waves with a somewhat complex function (sine is not that
easy to calculate) is computation heavy, the animation might be lagging.

Bundled with this file comes the "animation.mp4" file, which contains the full
animation, without lag. If ffmpeg is present it can be recreated by uncommenting
the second last line of the file.
"""
```

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


# -----------------------------------------------------------------------------
# constants
# -----------------------------------------------------------------------------
# how many samples to draw and thus how many frames there will be
# higher numbers take more computation time, but is a more accurate curve
# Defaulted to 500.
SAMPLES = 500
# How long one circle of animation should play in ms. Defaulted to 4 seconds.
ANIM_LENGTH = 4000
# How big the ouput figure should be in inches. Defaults to a 12 by 8 canvas
FIG_SIZE = (12, 8)
# The font size of the labels
FONT_SIZE = 14

# -----------------------------------------------------------------------------
# config of matplotlib parameters
# -----------------------------------------------------------------------------
# we will mark the x-axis every half-pi steps
xtick = np.arange(-np.pi, np.pi+np.pi/2, np.pi/2)
# we will use mathtext to display the pi symbol and fractions
# one label for each tick in xtick. The syntax is similar to latex maths.
xlabel = [r'$-\pi$', r'$-\frac{\pi}{2}$',
          r'$0$', r'$\frac{\pi}{2}$', r'$\pi$']
# config paramters for matplotlib to change the font
font = {'family': 'sans-serif',
        'sans-serif': ['Calibri', 'Ubuntu'],
        'size': FONT_SIZE}

plt.rc('font', **font)
# -----------------------------------------------------------------------------
# functions
# -----------------------------------------------------------------------------
def create_sine_func(x, amplitudes, periods):
    """
    Creates a function to create sine waves with the given parameters and
```

```
    changeable offsets.

    Creates a function that creates a sine wave for each entry in amplitudes and
    periods. The amplitudes and periods, as well as the x-values that the sine
    waves are calculated off, are unchangeable. But the sine waves can still be
    shifted along the x- and y-axis. This allows for animation of those waves.

    Args:
        x: 1d array-like
            The list of x values to use to calculate the sine waves
        amplitudes: 1d array-like
            The list of amplitudes to use for the different sine waves
            Must be the same size as amplitudes.
        periods:
            The list of periods to use for the different sine waves.
            Must be the same size as amplitudes.

    Returns:
        sine_waves: function
            A function which takes two lists the same size as amplitudes
            and then returns a list of sine curves' y-values.
    """
    def sine_waves(x_offs, y_offs):
        """
        Creates a list of sine-wave arrays offset by the values in x_offs and
        y_offs.

        For each entry in amplitudes there must be a value in x_offs and y_offs.
        The function than calculates that many sine waves with the amplitude and
        period from the outer function's lists. Each sine wave can be offset on
        the x- and the y-axis, which the values from x_offs and y_offs are used
        for.

        Args:
            x_offs: 1d array-like
                The x-offsets for the sine waves. Must be the same length as
                amplitudes.
            y_offs: 1d array-like
                The y-offsets for the sine waves. Must be the same length as
                amplitudes.

        Returns:
            sines: 1d list
                The list containing the sine arrays. Each entry is the array of
```

```
                the y-values of a sine wave with the supplied parameters.
        """
        sines = []
        # a set of amplitude, period, x offset and y offset defines one sine curve
        # so for all sets, create one sine wave, based on the supplied x values
        for a, T, x_off, y_off in zip(amplitudes, periods, x_offs, y_offs):
            # amplitude changes the "height" of the curve into positive and
            # negative direction. period changes how long the curve needs
            # for one repetition. x-offset moves the curve along the x-axis,
            # and y-offset along the y-axis respectively.
            sine = a * np.sin(T * (x + x_off)) + y_off
            sines.append(sine)
        return sines

    return sine_waves


def animate(idx, *fargs):
    """
    Animate the sine waves by shifting them along the axes.

    Calculates the sine waves anew shifted a tiny bit more.
    fargs needs to supply the tuple: (sine_waves, lines, x_offsets, y_offsets)
    The order is important. sine_calc is the output of create_sine_func, lines
    is the list of the plots in the figure, and x- and y-offsets are the list
    of offsets to be applied to the sine waves.

    Since the top-left and bottom curve move to the left, the x-offset is
    supplied as is. For the top-right curve it is negated to move the curve to
    the right. Only the bottom curve is shifted on the y-axis, hence the
    y-offsets for the top curves are 0.

    The index of the animation is important to access how much to shift the curves.

    Args:
        idx: int
            The index of the animation playing. Used to access offsets.
        fargs: tuple
            A tuple containing the sine_calc function, a list of 2D-lines to
            replace the y-data on, the x-offsets and y-offsets to index with idx

```

```
    Returns:
        lines: list
            The list of altered 2D-Line objects to be plotted in the next frame
    """
    sine_waves = fargs[0] # extract function object
    lines = fargs[1] # extract list of artists
    x_off = fargs[2][idx] # extract offset for this animation frame
    y_off = fargs[3][idx]

    # create the lists supplied to the sine_waves function
    x_offs = [x_off, -x_off, x_off]
    y_offs = [0, 0, y_off]

    # calculate the new sine waves
    sines = sine_waves(x_offs, y_offs)

    # and replace the data
    for i in range(len(lines)):
        lines[i].set_ydata(sines[i])

    return lines

# -----------------------------------------------------------------------------
# set-up of figure and axes objects
# -----------------------------------------------------------------------------
# create a new figure object
fig = plt.figure("Inspecting Sines", figsize=FIG_SIZE)
# ... and give it a title
fig.suptitle("The Waving Sines")
# create the subplots for the three waves
# tl: top-left
ax_tl = plt.subplot(221)
# tr: top_right
ax_tr = plt.subplot(222)
# b: bottom
ax_b = plt.subplot(212)

# And iterable for the loops
axes = [ax_tl, ax_tr, ax_b]

# -----------------------------------------------------------------------------
# Setting up and initialising plots
# WIHTOUT THE BONUS TASK THIS IS ALL THAT WAS NEEDED
# -----------------------------------------------------------------------------
# create the SAMPLES x-values
x = np.linspace(-np.pi, np.pi, SAMPLES)
# the amplitudes our sine waves will have in order tl -> tr -> b
amps = [4, 2, 1]
```

```
# the periods of the sine waves
periods = [1, 1, 3]
# the colour of the graphs
colours = ['red', 'green', 'purple']

# create the sine_waves functions with the x-values and the amplitudes and
# periods we want
sine_waves = create_sine_func(x, amps, periods)
# initialise the unshifted sine waves
sines = sine_waves([0, 0, 0], [0, 0, 0])

# create a list of all the plots
lines = []
for i in range(len(amps)):
    # the label to describe the sine wave
    label = f"A: {amps[i]}, T: {periods[i]}"
    # and plot the plot
    line = axes[i].plot(x, sines[i], color=colours[i], label=label)
    lines += line

# all subplots (axes objects) share some layout options
# so we can define them in a loop. No need to write the same code thrice.
for ax in axes:
    # set the show-limits from -pi to pi
    ax.set_xlim((-np.pi, np.pi))
    # show a grid for better readability
    ax.grid(True)
    # usually matplotlib leaves a bit of space on the side of the graphs
    # this would look silly with animations though. Therefore we get rid of them
    ax.margins(xmargin=0)
    # up above we defined the xticks and their labels, here we apply them
    ax.set_xticks(xtick)
    ax.set_xticklabels(xlabel)
    # pack the legend into the top left corner of each graph
    ax.legend(loc=2)
    # just so that there is no confusion...
    ax.set_xlabel("x")
    ax.set_ylabel("y")

###### MAIN TASK IS DONE HERE

# a small addon for the bottom graph. Since we defined the layout now with the
# unshifted curve, moving it along the y-axis will put it out of bounds and clip
# the curve. Since we move the bottom curve on a sine wave, the maximum offset in
# y is 1. So we take the amplitude (the maximum height of the unshifted curve)
# and add 1.1 and make those the limit, and having some margin.
b_yrange = amps[2] + 1.1
ax_b.set_ylim((-b_yrange, b_yrange))
```

```

# ------------------------------------------------------------------------------
# initialise figures
# ------------------------------------------------------------------------------
# we need as many x_offsets as we will have frames.
# we have the same amount of frames as datapoints. This is so that there is no
# weird behaviour when a curve is shifted inbetween two data points (sometimes
# that leads to a small "bump" in the animation)
x_offs = np.linspace(0, 2*np.pi, SAMPLES)
# the bottom sine wave is supposed to move on a sine wave. So the y-offsets
# must be sine values
y_offs = np.sin(x_offs)

# build the tuple to be supplied to our animation function
fargs = (sine_waves, lines, x_offs, y_offs)

# Phew. Done. And finally: Here. We. Go.
anim = animation.FuncAnimation(
    fig,
    animate,
    frames=SAMPLES,
    interval=(ANIM_LENGTH / SAMPLES),
    fargs=fargs,
    save_count=25
)

# if you have ffmpeg in this folder or your PATH you can save the animation into
# a video file.
# anim.save('animation.mp4')

# actually we only start going here. but it was more dramatic up there.
plt.show()
```