ICC-P Project: Falling Ring

Introduction

A metal ring is dropped into a pool of liquid, with both the ring and the liquid initially at non-uniform temperatures. This programming assignment aims to model and simulate the heat transfer between the ring and the liquid and to visualize the simulation results.

Assume the ring is oriented horizontally and falls vertically at a constant speed through the liquid. The ring can be modeled as a circle, represented by a finite number of discrete, equidistant points along its perimeter. Similarly, time will also be treated as discrete. Thus, during the simulation, as the ring falls, the vertical positions of its points change from depth d_i to d_{i+1}, where $|d_{i+1} - d_i|$ is the distance the ring travels between two discrete time steps, t_i and t_{i+1}.

Having a finite number of points on the ring allows us to model the physical properties of both the ring and the liquid it passes through. As the ring falls and time progresses (i.e., as depth and time increase), we track the points on the ring, which together form the surface of an imaginary cylinder. The space-time region represented by this cylindrical surface can be "unraveled" into a plane, as illustrated in Figure 1.a. This plane can then be transformed into a two-dimensional image for visualization.

The plane formed by unrolling the cylinder can be represented as a two-dimensional (2D) array of data, indexed by depth and time. Consider Figure 1.b. The x-axis (p) represents the points around the ring at a specific depth, while the y-axis (d) represents depth. The depth at the liquid surface is zero, as the ring is falling vertically.

Modeling the ring: The 2D array will be used to keep the temperature of every point of the ring while the ring is falling. A row in this 2D array will model the temperature of the entire ring at the corresponding depth and moment in time. A column in this 2D array will model the temperature of a particular point of the ring at various depths during the ring's fall. This 2D array can be stored as a list of lists in Python.

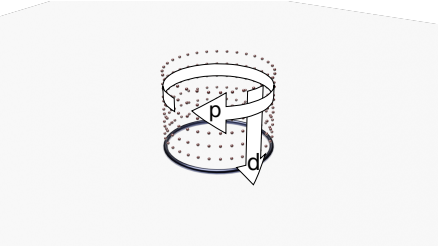
Modeling the liquid: Similarly, a 2D array models the temperature of the cylindrical liquid surface that comes in contact with the falling ring for each time step.

This assignment consists of three main parts:

- **Part 1 [50 pts]:** Implement the functionality of converting 2D arrays into color images for visualization purposes.
- **Part 2 [20 pts]:** Simulate the heat transfer between the ring and the surrounding liquid.
- **Part 3 [30 pts]:** Simulate the heat transfer between discrete equidistant points of the ring.

Part 1 is divided into several smaller sub-problems. While we recommend starting with Part 1 to visualize the simulation results, the three parts are independent, and you may work on them in any order. Helper functions are provided where needed, such as for visualizing images generated by your code in Part 1.

(a) While falling, the ring traverses an imaginary cylinder, which can be unraveled onto a two-dimensional surface, which will be shown as an image. The ring is shown as a full circle, even though we model it as a set of equidistant points. The liquid points in contact with the falling ring are shown as dots.



(b) The correspondence between the two-dimensional surface (i.e., image) coordinates and the falling ring.

Figure 1: An illustration of how the points of the ring and liquid are represented as an image.

Testing and Submitting Your Code for Grading

The file `falling_ring.py` contains placeholders for the functions you need to implement. **This is the file you will submit to Moodle for automated grading.**

Do not modify the function headers, i.e., their names or arguments (names, order, or type). If you modify any of these, the score will be zero for the affected function. Moreover, do not add any additional code to this file. On the flip side, you do not need to worry about dealing with arguments of the wrong type. For instance, if we ask you for a function that takes a real value, the grading system will never give it anything else than a float.

To test your code locally (without automated grader on Moodle), modify the `quick_tests` function at the top of `simulator.py`. This function (with your own tests) will be called when you run `simulator.py` with the test flag as follows:

```
>> python simulator.py --test
```

Throughout this document, we provide examples of inputs and outputs for each graded function, which you can copy to `quick_test` and use to test your code. The `simulator.py` script can also run more complex tests that generate images. We provide the required commands and expected output in later sections.

Finally, we also provide automated tests similar to those used by our automated grader in Moodle. You can run the tests as follows:

```
>> python tests.py
```

The results of each test will be printed. **These tests are the best way to check your code before submitting it for grading and will give you more detailed feedback than from an automated grader on Moodle.** Feel free to add your tests to debug your code, as this file

will not be submitted to Moodle.

Note that all values returned by functions are rounded to two digits after the decimal point before printing to remove the rounding errors. In practice, for the entirety of this project, what is returned by your functions will be considered "correct" if every single value is within a margin of error of ± 0.005 centered on what we expect (i.e., $|a-b| < 0.005$ where a is your value and b is the reference value). **You should not round the return values. The grader will take care of this comparison.**

You are advised to use the virtual machine you use in the labs to code this project. We recommend using `Python3.10`, the version of `Python` provided in the virtual machine. Moreover, the external libraries needed for this project are *matplotlib* and *numpy* and are already installed in the virtual machine.

Submission Instructions

You will submit the assignment to Moodle (https://moodle.epfl.ch/) in groups. The grading will be done automatically on Moodle. Each function will be graded separately. For all the functions, you are awarded points based on the total number of tests passed. Each test carries equal weight. Moodle outputs your total score and generates a `report.txt` file, containing a list of passed and failed test cases. **Note that Moodle displays the total score of your last submission and not the highest score across all submissions.**

In addition, before submitting your code for grading on Moodle, ensure that the local tests we provided pass successfully. Moreover, we recommend the following Python Style Guide: https://peps.python.org/pep-0008/#introduction

For more information (e.g., instructions on how to access, the deadline to submit the code, the maximum allowed number of trials, etc.), please visit the Moodle page of the course.

Part 1: Visualization [50 pts]

As described in the introduction, there are 2D arrays for storing the temperature of the ring and liquid during the simulation. The 2D arrays need some processing to be visualized. To do so, we will implement a way to map a color to a temperature. The simplest way to do this is to first assign a discrete set of colors to a discrete set of temperatures. Accordingly, the colors can be mixed to create an appropriate intermediate color for intermediate temperatures.

Finding the Right Interval [15 pts]

In this part, you will write the following function:

```
def get_interval(
    query, # float, the temperature to find an interval for
    values # a sorted list of floats, the temperatures for which the colors are known
)  # returns an int, the index of the lower bound of the interval "query" belongs to
```

This function should find and return `i`, $0 \leq$ `i` $<$ `len(values)`, such that:
`values[i]` $\leq$ `query` $<$ `values[i+1]`.

Consider these special cases:

- If `query` is lower than `values[0]`, the function should return zero.
- If `query` is higher or equal than `values[-1]`, the function should return `len(values)-1`.

Here are a few test cases for this function:

```
get_interval(0.5, [-1, 0, 1, 2, 3])
# -> 1

get_interval(5, [-100, 0, 3, 9])
# -> 2

get_interval(2.5, [0, 1, 2])
# -> 2

get_interval(0.5, [2, 6, 50])
# -> 0

get_interval(3, [1, 2, 3, 4])
# -> 2
```

Interpolating Colors [15 pts]

In this part, you will write the following function:

```
def interpolate_color(
    query, # float, the temperature to get the color for
    values, # a sorted list of floats, the temperatures for which we know the
        corresponding colors
    colors # list of lists, each of these lists has three floats, representing the red,
        green, and blue components of the color for the temperatures in "values"
) # returns a list of 3 floats, the red/green/blue components for the query
    temperature's color.
```

This function takes `query` as a given temperature, `values` as a sorted list of temperatures, and `colors` declaring the corresponding color of each temperature in `values`. Accordingly, it computes and returns the color of `query`. The two lists, `values` and `colors`, have at least two and the same number of elements.

`colors` is a list where each color is represented by a list of three floating-point numbers ranging from 0 to 1. A color displayed on a screen comprises three light components: red, green, and blue. Accordingly, each of the three floats corresponds to the intensity of one of these colors: 0 represents no light, while 1 represents maximum light. For example:

- The color [1, 0, 0] contains only a red component at the highest intensity, with no green or blue components.

- The color [0, 1, 0] contains only a green component.
- The color [0, 0, 1] contains only a blue component.
- `values=[-1, 2]` and `colors=[[1, 0, 0], [0, 0, 1]]` means that temperatures -1 and 2 degrees should be displayed as red and blue, respectively.

The function first finds the corresponding temperature interval in `values` for `query` by calling the `get_interval` function you implemented before. Using the obtained index `i`, the `interpolate_color` function should read the two corresponding colors from the list `colors` and mix them linearly. In the formulas below, `P` represents the relative distance of `query` from the start of the corresponding interval. This distance is used to compute the `query`'s color by calculating the weighted average of the colors associated with the temperatures defining `query`'s interval. Note that each element in `colors` is a three-element list, and multiplying the list by a float means that float scales all list elements. Please find these formulas below, applicable to `0 <= i < (len(values)-1)`:

$$\texttt{P} = \frac{\texttt{query - values[i]}}{\texttt{values[i+1] - values[i]}}$$

$$\texttt{P\_color} = \texttt{(1-P)}\cdot\texttt{colors[i] + P}\cdot\texttt{colors[i+1]}$$

Your function should implement the corner cases below:

- If `query` $<$ `values[0]`, its color is `colors[0]`.
- If `query` $\geq$ `values[-1]`, its color is `colors[-1]`.

Here are a few test cases for this function:

```
def round_list(L):
    # rounds each value in a list to two digits after the decimal point
    return [round(x,2) for x in L]

round_list(interpolate_color(0.6, [-1,0,1],[[0,0,0],[0,1,0],[0,1,1]]))
# -> [0.0, 1.0, 0.6]

round_list(interpolate_color(5, [0,3,9], [[1,0,0],[0,1,0],[0,0,1]]))
# -> [0.0, 0.67, 0.33]

round_list(interpolate_color(2.5, [0,1,2], [[1,0,0],[0,1,1],[0,0,1]]))
# -> [0, 0, 1]

round_list(interpolate_color(0.5, [2,6,50], [[1,0,0],[0,1,1],[0,0,1]]))
# -> [1, 0, 0]
```

Applying a Colormap to an Image [20 pts]

Now that we have the `interpolate_color` function to map a color to a temperature, all that is left for the visualization is to apply this function to every point of the 2D array we want to display.

As we described in the introduction, a 2D array can model (1) the temperatures of the falling ring (during the time interval it takes for the ring to fall) or (2) the temperatures of the liquid (for each

time step). In any of the two cases, we have a 2D array of temperatures, which we can visualize as a 2D color image.

These 2D arrays, in Python terminology, are lists of lists; if the name of the list is `L`, then `L[d][p]` is the temperature of the ring at point `p` and depth `d` (see Fig. 1b).

In this part, you will write the following function:

```
def map_to_color(
    in_data, # a list of lists of floats
    out_data, # a list of lists of lists of floats
    values, # a list of floats
    colors, # a list of lists of floats
)  # does not return anything
```

The meaning of the function parameters is as follows:

- `in_data` is a list of lists, where each list corresponds to one specific depth and contains as many floating point numbers as there are equidistant points on the ring. The list at index zero corresponds to the starting position of the ring. Each value in this 2D list is a temperature, and you must map these values to their corresponding colors. This list should be accessed as `in_data[d][p]`, where `d` and `p` are the depth and the index of a point on the 2D surface (see Figure 1.b).
- `out_data` is a three-dimensional list. The first dimension corresponds to the depth, the second to the ring points, and the third to the point's color. Therefore, it should be accessed as `out_data[d][p][color_channel]`, where `color_channel` is 0 for red, 1 for green, and 2 for blue. No other value is accepted for `color_channel`.
- `values` and `colors` are simply the same `values` and `colors` described in previous sections.

You can assume that `out_data` and `in_data` have the same length, all the lists in `in_data` and `out_data` have the same length, and the lists within the lists of `out_data` are of length three (the number of color channels). Moreover, it is safe to pass `colors` and `values` directly to `interpolate_color`.

Here are a few test cases for this function:

```
def round_color_image(img):
    # rounds each value in a list of list of lists, to two digits after the decimal point
    return [[[round(x,2) for x in pixel] for pixel in row] for row in img]

in_data = [[-0.6, -0.4], [-0.3, 3.1]]
out_data = [ [[0]*3 for j in range(2)] for i in range(2) ]

map_to_color(in_data, out_data, [-1,0,1], [[1,0,0],[0,1,0],[0,0,1]])
print(round_color_image(out_data))
# [ [ [0.6, 0.4, 0.0], [0.4, 0.6, 0.0] ],
#   [ [0.3, 0.7, 0.0], [0.0, 0.0, 1.0] ] ]

in_data = [[-0.6, -0.4, -0.3, -3.7, 3.4],
      [-0.3, 3.1, -3.9, -0.2, 0.7],
      [4.5,  4.8, 2.3, 4.6, 2.4],
```

```
      [-1.8, -3.1, 3.1, -0.2, 2.5],
      [-1.7, -2.0, 0.6, -3.1, 4.6]]
out_data = [ [[0]*3 for j in range(5)] for i in range(5) ]

map_to_color(in_data,out_data,[-1,0,1,2,3],
    [[0.0,0.0,0.0],[0.0,0.0,1.0],[1.0,0.0,1.0],[1.0,0.0,0.0],[1.0,1.0,1.0]])
print(round_color_image(out_data))
# [ [ [0.0, 0.0, 0.4], [0.0, 0.0, 0.6], [0.0, 0.0, 0.7], [0.0, 0.0, 0.0], [1.0, 1.0,
    1.0] ],
#   [ [0.0, 0.0, 0.7], [1.0, 1.0, 1.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.8], [0.7, 0.0,
    1.0] ],
#   [ [1.0, 1.0, 1.0], [1.0, 1.0, 1.0], [1.0, 0.3, 0.3], [1.0, 1.0, 1.0], [1.0, 0.4,
    0.4] ],
#   [ [0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [1.0, 1.0, 1.0], [0.0, 0.0, 0.8], [1.0, 0.5,
    0.5] ],
#   [ [0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.6, 0.0, 1.0], [0.0, 0.0, 0.0], [1.0, 1.0,
    1.0] ] ]
```

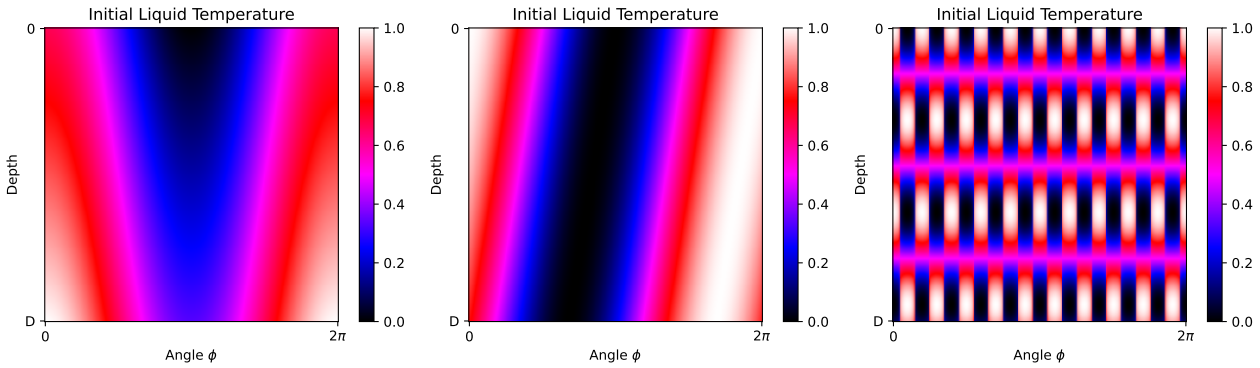
One way to test your function is by visualizing the initial temperature of the liquid using the following command. Three inputs are available, which can be tested by replacing `N` with 1, 2, or 3:

```
>> python simulator.py --visualize N
```

You should obtain the images below if you implement your functions correctly. Note that the y-axis shows depth, which ranges from zero (on top, where the ring starts falling) to `D` (the maximum depth). The x-axis shows the ring points. For convenience, instead of numbering the points starting from zero to the total number of points, we convert the index of a ring point p to the equivalent radial angle using the following formula:

$$\phi = 2\pi \frac{p}{\text{number\_of\_points\_in\_ring}}$$

On the right side of the figure, we display the mapping between the temperatures and colors.



(a) N = 1 (b) N = 2 (c) N = 3

Figure 2: The output of `map_to_color` for the available inputs.

Part 2: Thermal Exchange [20 pts]

In this section, we model the heat transfer within the simulated system. We compute the heat transfer between two points by computing the *next* temperature (one time step later) of a point (T_{next}) based on its *current* temperature (T_{current}) and the *current* temperature of the point it is exchanging heat with. Three constants reflect the physical properties involved in this specific heat transfer. These constants *implicitly* contain information on (1) the distance between the points and (2) the time step duration.

First, we should compute the amount of energy transferred between the points `a` and `b` during the time step, which depends on the **transfer coefficient** k_t:

$$E_{a\to b} = k_t(T_{a,\text{current}} - T_{b,\text{current}})$$

Using the computed transferred energy $E_{a\to b}$ between the points, we can use the **thermal capacity coefficients** of the two points, c_a and c_b, to compute the effect of energy transfer on the temperatures of the two points. Please note how the direction of the energy transfer affects the second equation:

$$T_{a,\text{next}} = T_{a,\text{current}} - \frac{E_{a\to b}}{c_a}$$

$$T_{b,\text{next}} = T_{b,\text{current}} + \frac{E_{a\to b}}{c_b}$$

If a given point interacts with more than one other point, all the energy transfers for that point need to be computed from the current temperature. Note that the temperatures of points in contact will tend to get closer to each other unless the value of k_t, which implicitly depends on the time step, is too large. Moreover, *conservation of energy* is held in these equations: the system's total amount of thermal energy, equivalent to the sum of the values of $c_i T_i$ for all points i, is constant.

As a first step in modeling the heat transfer, let us model the heat exchange between the ring and the surrounding liquid. For every point of the ring, we shall only consider a single interaction with the point of the liquid having the same (d, p) coordinates.

In this part, you will write the following function:

```
def thermal_exchange(
    liquid_temp, # list of lists of float
    ring_temp, # list of lists of float
    current_depth, # int, nonnegative
    transfer_coefficient=1.0, # float, positive
    capacity_coefficient_liquid=1.0, # float, positive
    capacity_coefficient_ring=1.0 # float, positive
    ) # doesn't return anything
```

This function modifies the temperature of the points of the ring and the points of the liquid in contact with the ring to simulate the exchange of thermal energy between them. Note that:

- `liquid_temp` and `ring_temp` are two 2D lists, one being the *current* temperature of the liquid, and the other the *history* of all temperatures the ring had during its fall.

- `current_depth` is the current depth of the ring, corresponding to the *row index* in the 2D list `ring_temp` as well as the moment in time when the thermal exchange is taking place.

You can assume that `liquid_temp` and `ring_temp` have the same size. Moreover, `current_depth` will be given such that `0 <= current_depth < len(liquid_temp)`.

Here are a few test cases for this function:

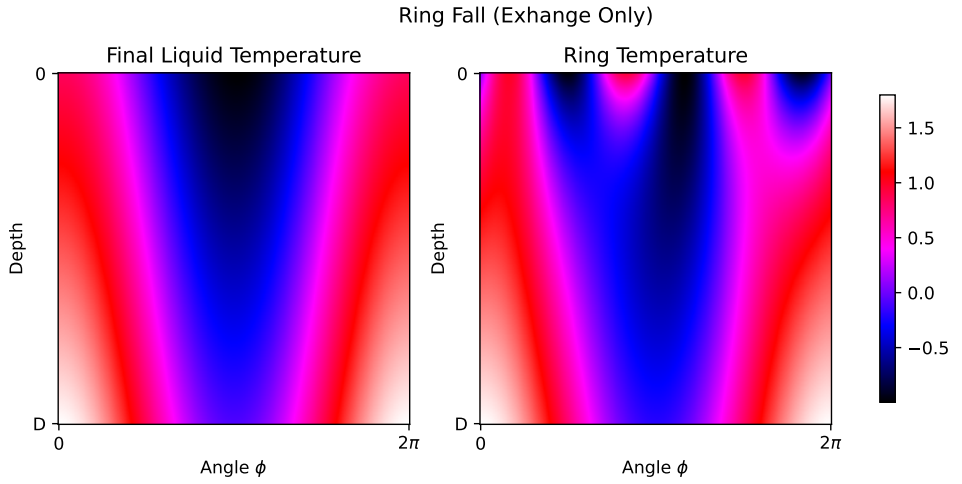
```
def round_image(img):
    # rounds each value in a list of lists, to two digits after the decimal point
    return [[round(x,2) for x in row] for row in img]

print("example 1")
liquid_temp = [[0.0,0.1,0.2,-0.3,-0.4,0.5]]
ring_temp = [ [0]*6 ]
thermal_exchange(liquid_temp, ring_temp, 0, 0.1, 1.0, 1.0)
print(round_image(liquid_temp)) # [[0.0, 0.09, 0.18, -0.27, -0.36, 0.45]]
print(round_image(ring_temp)) # [[0.0, 0.01, 0.02, -0.03, -0.04, 0.05]]

print("example 2")
liquid_temp = [[0,0.1,0.2,0.3,0.4,0.5]]
ring_temp = [ [0]*6 ]
thermal_exchange(liquid_temp, ring_temp, 0, 0.1, 1.0, 0.25)
print(round_image(liquid_temp)) # [[0.0, 0.09, 0.18, 0.27, 0.36, 0.45]]
print(round_image(ring_temp)) # [[0.0, 0.04, 0.08, 0.12, 0.16, 0.2]]

print("example 3")
liquid_temp = [[0,0.1,0.2,0.3,0.4,0.5], [0,0.1,0.2,0.3,0.4,0.5]]
ring_temp = [[0]*6, [0]*6]
thermal_exchange(liquid_temp, ring_temp, 1, 0.1, 1.0, 0.25)
print(round_image(liquid_temp)) # [[0.0, 0.1, 0.2, 0.3, 0.4, 0.5],
             #  [0.0, 0.09, 0.18, 0.27, 0.36, 0.45]]
print(round_image(ring_temp)) # [[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
           #  [0.0, 0.04, 0.08, 0.12, 0.16, 0.2]]
```

Note that `ring_temp` has the same length as `liquid_temp`. Therefore, the values in `ring_temp` that are beyond `current_depth` are just dummy values to reserve space for `ring_temp` as the simulation goes forward. Notice how `thermal_exchange` has been used in the `simulator.py` file. In the `run_thermal_exchange`, the simulator starts with an initial temperature for the ring. Then, it iterates over all values for `current_depth`, starting from 0 to the maximum depth. Initially, the current temperature of the ring is equal to its temperature in the previous depth in each iteration. Then `thermal_exchange` is called to update the current temperature of the ring in the current depth. Understanding how `thermal_exchange` is used in the simulation can help you correctly implement this function.

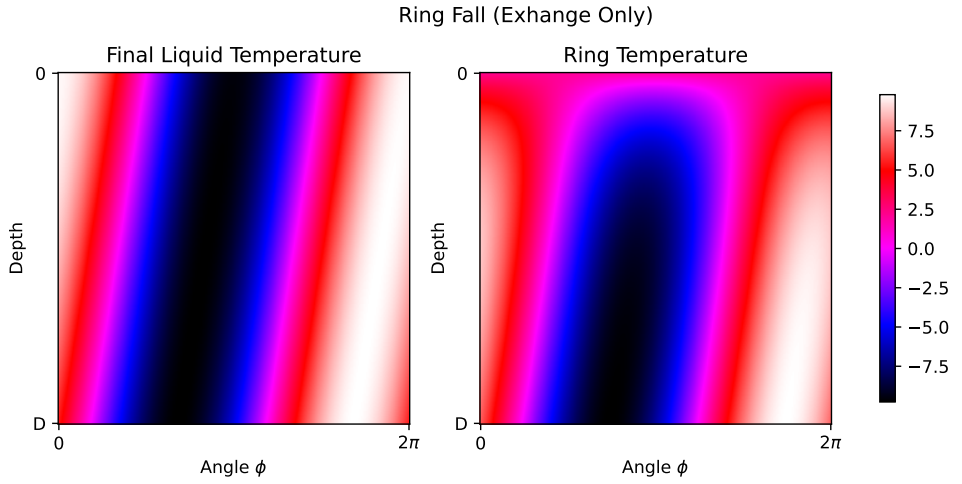
You can use the simulation script to test the function on larger inputs with the following command (replace N with 1, 2, or 3):

```
>> python simulator.py --thermal-exchange N
```

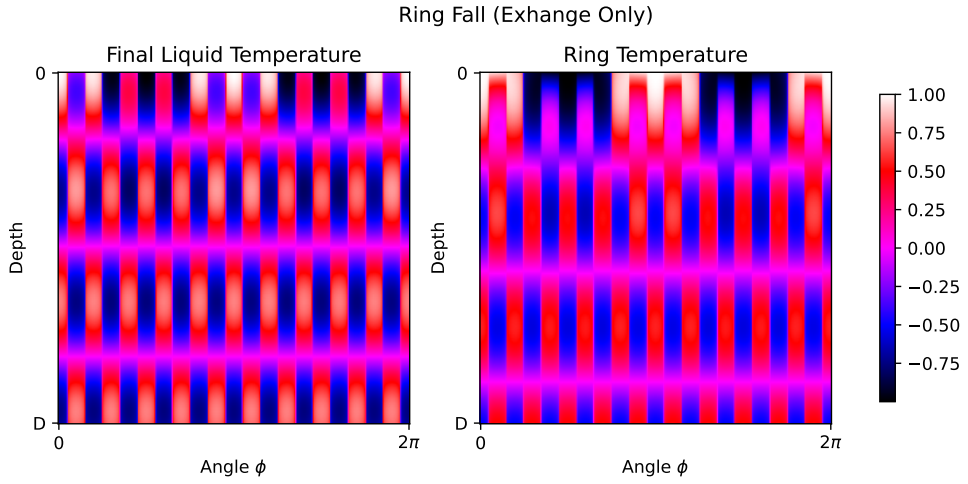
If you implement the function correctly, you should obtain the images below.



(a) N = 1



(b) N = 2



(c) N = 3

Figure 3: The output of the thermal exchange for various available inputs.

Part 3: Thermal Conduction [30 pts]

So far, we have considered the heat transfer between the liquid and the ring. However, due to thermal conduction, temperature tends to get homogeneous within the ring itself. Let us now model the temperature transfer between the ring points, applying the same physical mechanism explained before.

In this part, you will write the following function:

```
thermal_conduction(
    ring_temp, # list of lists of floats, all with the same length
    current_depth, # nonnegative int
    transfer_coefficient=1.0, # positive float
    capacity_coefficient_ring=1.0: # positive float
    )  # does not return anything
```

This function simulates the thermal exchange inside the ring and at the given depth. Note that each ring point (`ring_temp[current_depth][p]`) interacts with both of its neighboring points (`ring_temp[current_depth][p-1]` and `ring_temp[current_depth][p+1]`). Note that these two interactions need to read the same current temperature of the given point. Moreover, the points at `ring_temp[current_depth][0]` and `ring_temp[current_depth][-1]` are neighboring (recall how the ring point coordinates and temperatures are transformed onto a 2D image).

You can assume that all the lists in `ring_temp` will be of the same size, and `0 <= current_depth < len(ring_temp)`.

Here are a few test cases for this function:

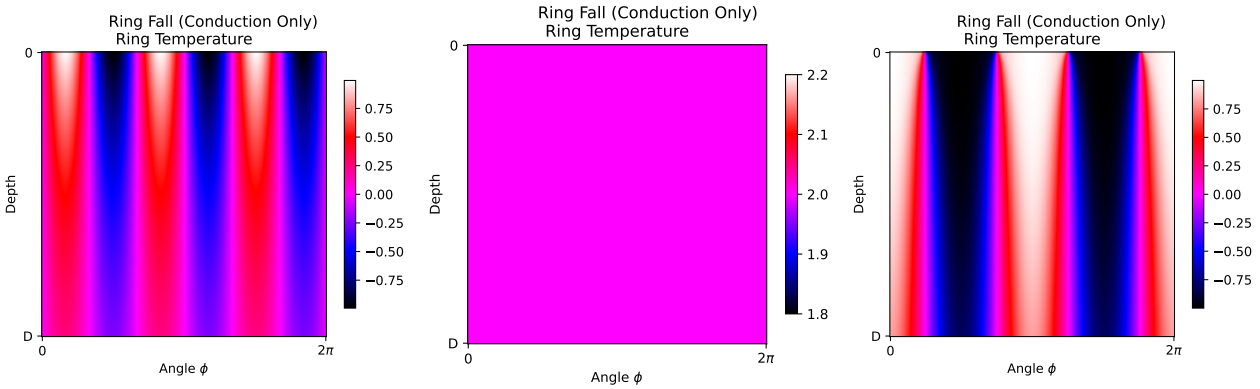
```python
def round_image(img):
    # rounds each value in a list of lists, to two digits after the decimal point
    return [[round(x,2) for x in row] for row in img]

print("example 1")
ring_temp = [[0,0.1,0.2,0.4,0.2,0.1]]
print("sum before:", round( sum(ring_temp[0]), 2) ) # -> "sum before: 1.0"
thermal_conduction(ring_temp,0, 0.1,1.0)
print(round_image(ring_temp)) # [[0.02, 0.1, 0.21, 0.36, 0.21, 0.1]],
print("sum after:", round( sum(ring_temp[0]), 2) ) # -> "sum after: 1.0"
# note how the amount of energy has not changed

print("example 2")
ring_temp = [[0,0.0,0.0,0.0,0.0,0.0]]
thermal_conduction(ring_temp,0, 0.1,1.0)
print(round_image(ring_temp)) # [[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]

print("example 3")
ring_temp = [[0,0,0,0,0,1]]
thermal_conduction(ring_temp,0, 0.1,1.0)
print(round_image(ring_temp)) # [[0.1, 0.0, 0.0, 0.0, 0.1, 0.8]]
# please also note how the heat traveled from one end of the list to the other,
# because the ends correspond to two points of the ring which are neighboring

print("example 4")
```

```
ring_temp = [[0,0.1,0.2,0.4,0.2,0.1]]
thermal_conduction(ring_temp,0, 0.1,1.0)
print(round_image(ring_temp)) # [[0.02, 0.1, 0.21, 0.36, 0.21, 0.1]]
```

You can test your function on larger inputs with the simulation script as shown below (replace N with 1, 2, or 3):

```
>> python simulator.py --thermal-conduction N
```

If you implement the function correctly, you should get the figures below:



(a) N = 1 (b) N = 2 (c) N = 3

Figure 4: The output of `thermal_conduction` for various available inputs.

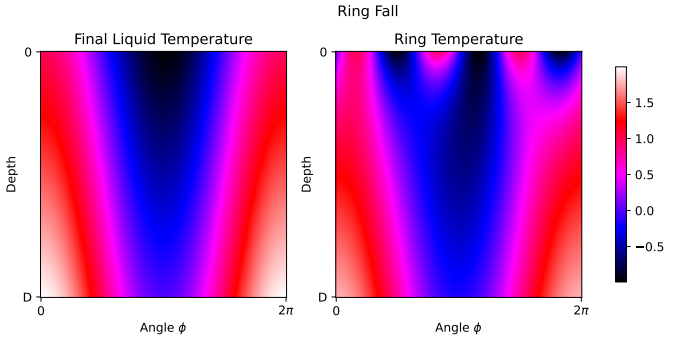
Simulating the Falling Ring

You can test your entire implementation by running the following simulator script (replace N with 1, 2, or 3):

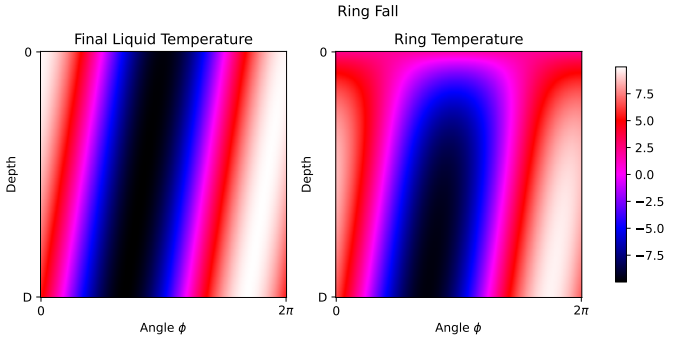
```
>> python simulator.py --run-simulation N
```

The output will show the simulation results, including the final temperature of the liquid and the history of the ring's temperature as it sank.

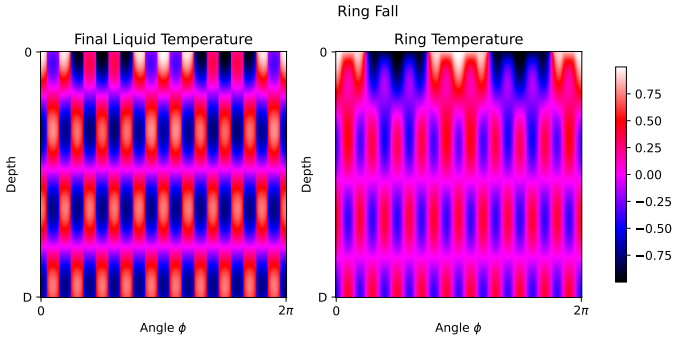
You should get the following figures if you implement all functions correctly.



(a) N = 1



(b) N = 2



(c) N = 3

Figure 5: The output of the entire simulation for various available inputs.