COMP 208

Fall 2023

Assignment 4

Posted: Monday, November 20^{th}

Due: Monday, December 4^{th}, 11:59 p.m.

Question 1:	100 points
	100 points total

Please read the entire PDF before starting. It is very important that you follow the directions as closely as possible. The directions, while perhaps tedious, are designed to make it as easy as possible for the TAs to mark the assignments by letting them run your assignment, in some cases through automated tests. While these tests will never be used to determine your entire grade, they speed up the process significantly, allowing the TAs to provide better feedback and not waste time on administrative details.

To get full marks, you must follow all directions below:

- Make sure that all file names and function names are **spelled exactly** as described in this document. Otherwise, a 50% penalty per question will be applied.
- Make sure that your code **runs without errors**. Code with errors will receive a very low mark.
- Write your name and student ID in a comment at the top of all `.py` files you hand in.
- Name your variables appropriately. The purpose of each variable should be obvious from the name.
- Avoid writing repetitive code, but rather call helper functions! You are welcome to add additional functions if you think this can increase the readability of your code.
- Lines of code should NOT require the TA to scroll horizontally to read the whole thing. If the line is getting way too long, then it's best to split it up into two different statements.
- Vertical spacing is also important when writing code. Separate each block of code (also within a function) with an empty line.
- **Up to 30% can be removed for bad indentation of your code, omission of docstrings, and/or poor coding style as discussed in our lectures.**

Hints & tips

- **Start early.** Programming projects always take more time than you estimate!
- Do not wait until the last minute to submit your code. **Submit early and often**—a good rule of thumb is to submit every time you finish writing and testing a function.
- Write your code **incrementally**. Don't try to write everything at once. That never works well. Start off with something small and make sure that it works, then add to it gradually, making sure that it works every step of the way.
- Seek help when you get stuck! Check our discussion board first to see if your question has already been asked and answered. Ask your question on the discussion board if it hasn't been asked already. Talk to a TA during office hours if you are having difficulties with programming. Go to the instructor's office hours if you need extra help with understanding a part of the course material.

  - Note that small amounts of code can be posted on the discussion board as *private* posts, which only the instructors and T.A.'s can see. But if more than a few lines of code are in question, then better to seek help in person. Often the problem requires a different *approach* such as proper use of the debugger, and the discussion board is a poor medium for this kind of help. (You could also post a single line of code and error message (error traceback) as a public post.)

- If you come to see us in office hours, please do not ask "Here is my program. What's wrong with it?" We expect you to at least make an effort to start to debug your own code, a skill which you are meant to learn as part of this course. Reading through someone else's code is a difficult process—we just don't have the time to read through and understand even a fraction of everyone's code in detail.
  - However, if you show us the work that you've done to narrow down the problem to a specific section of the code, why you think it doesn't work, and what you've tried in order to fix it, it will be much easier to provide you with the specific help you require and we will be happy to do so.

Policy on Academic Integrity

See the Course Outline Sec. 5 for the policies, as well as the discussion in lecture 5. Here also is a detailed summary of what is allowed and what is not allowed. When in doubt, ask.

Late policy

Late assignments will be accepted up to two days late and will be penalized by 10 points out of 100 per day. You will submit through codePost as usual. Note that if you submit one minute late, it is equivalent to submitting 23 hours and 59 minutes late, etc. So, make sure you are nowhere near that threshold when you submit.

Revisions

Search for the keyword *updated* to find any places where the PDF has been updated.

Part 2

The questions in this part of the assignment will be graded.

The main learning objectives for this assignment are:

- Create dictionaries and store and access key-value pairs.
- Apply what you have learned about object oriented programming: defining classes and instance methods, creating instance attributes, creating objects and calling methods on them.
- Create NumPy arrays and perform basic operations on them.
- Understand when and how to use broadcast operations and vectorization when working with NumPy arrays

Note that this assignment is designed for you to be practicing what you have learned in our lectures up to and including Lecture 20, plus a small part in Lecture 21 on NumPy. For this reason, you are NOT allowed to use anything seen after that lecture or not seen in class at all. You will be heavily penalized if you do so.

For full marks, in addition to the points listed on page 1, make sure to add the appropriate documentation string (docstring) to all the functions you write. The docstring must contain the following:

- The type contract of the function.
- A description of what the function is expected to do.
- At least three (3) examples of calls to the function. You are allowed to use at most one example per function from this PDF.

Examples

For each question, we provide several **examples** of how your code should behave. All examples are given as if you were to call the functions from the shell.

When you upload your code to codePost, some of these examples will be run automatically to test that your code outputs the same as given in the example. However, **it is your responsibility to make sure your code/functions work for any inputs, not just the ones shown in the examples.** When the time comes to grade your assignment, we will run additional, private tests that will use inputs not seen in the examples. You should make sure that your functions work for all the different possible scenarios. Also, when testing your code, know that mindlessly plugging in various different inputs is not enough—it's not the quantity of tests that matters, it's having tests that cover all of the possible scenarios, and that requires thinking about possible scenarios.

Furthermore, please note that your code files **must not contain any function calls in the main body of the program** (i.e., outside of any functions). Code that contains function calls in the main body **will automatically fail the tests on codePost and thus be heavily penalized**. It is OK to place function calls in the main body of your code for testing purposes, but if you do so, make certain that you remove them before submitting. You can also test your functions by calling them from the shell. Please review what you have learned in our lecture on Modules if you'd like to add code to your modules which executes only when you run your files.

Safe Assumptions

For all questions on this assignment, you can safely assume that the **type** of the inputs (both to the functions and those provided to the program by the user) will always be correct. For example, if a function takes as input a string, you can assume that a string will always be provided to it during testing. The same goes for user input. At times you will be required to do some input validation, but this requirement will always be clearly stated. Otherwise, your functions should work with any possible input that respect the function's

description. For example, if the description says that the function takes as input a positive integer, then it should work with all integers greater than 0. If it mentions an integer, then it should work for *any* integer. Make sure to test your functions for edge cases!

Code Repetition

One of the main principles of software development is DRY: Don't Repeat Yourself. One of the main ways we can avoid repeating ourselves in code is by writing functions, then calling the functions when necessary, instead of repeating (copy-and-pasting) the code contained within them. Please pay careful attention in your code for this assignment to not repeat yourself, and instead call previously-defined functions whenever appropriate. As always, you can also add your own helper functions if need be, with the intention of reducing code repetition as much as possible.

Question 1: Terrains (100 points)

Background

Many problems in science and engineering involve data defined on a regular grid of points. Perhaps the most common example is an image defined on a 2D grid, or a video defined on a 3D grid where the third dimension is time). Other intensive physical quantities (pressure, temperature, density) can be defined over 2D and 3D grids as well. To analyze such data sets, one typically computes basic properties such as how the data varies from point to point (the rate of change in intensity, density, etc.), where the maxima and minima lie, and what the values are that lie between data points.

In this assignment, we will implement such basic operations for a particular type of data called a **terrain**. We will work with functions of two variables $f(x, y)$, which could include images or 3D densities or other intensive quantities. However, for some of the computations we perform, it is more intuitive to think of the functions as representing the heights (or elevations) of a continuous surface above the ground. So we will use the word *elevation* for the values of our functions $f(x, y)$. Such elevation functions are often called terrains.

For any such function $f(x, y)$, we will use a NumPy `ndarray` to represent a discrete sampled version of the function, meaning we represent the continuous function $f()$ at discrete points. Such a discrete function will be defined on a 2D grid or matrix with some given number of rows and columns. Each (row, column) grid point corresponds to a (y, x) value in the matrix. Note we write (y, x) there rather than (x, y) since we are referring to the (row, column) grid points. This is follows the tradition of linear algebra where the matrix notation M_{ij} means row i and column j. Heads up: you may find that some of your coding errors are due to the wrong order of these coordinates in your function parameters. Being mindful of this issue should help you reduce the frequency of such errors.

We next turn to an overview of the mathematical ideas that underlie some of the operations we will perform on these terrains.

Partial derivatives and their approximation

In Calculus 1, you learned about taking derivatives of a function $f(x)$ of one variable, and you wrote the derivative as $\frac{df(x)}{dx}$. We can also define derivatives when we have a function of two or more variables. In particular, we can take the derivative with respect to any one of the variables while holding the other variable(s) constant. Such derivatives are called *partial derivatives* and you will learn about them in Calculus 3 or other courses. A slightly different notation is used for a partial derivative. For example, the partial derivative with respect to the x variable is written $\frac{\partial f(x,y)}{\partial x}$ with symbol ∂ instead of d, and similarly for the derivative in the y direction. For this assignment, we will work with functions of two variables $f(x, y)$ and their partial derivatives.

For discrete functions, we cannot define partial derivatives since a derivative is formally defined only over a continuous domain. Instead, we will *approximate* the partial derivatives of a discrete function using the following **finite difference formulas**.

$$\frac{\partial f(x,y)}{\partial x} \approx \frac{f(x+1,y) - f(x-1,y)}{2}$$

$$\frac{\partial f(x,y)}{\partial y} \approx \frac{f(x,y+1) - f(x,y-1)}{2}$$

We will assume that the sampled grid values of x and y are integers.

As you also learned in Calculus, for a smooth function $f(x, y)$, the **gradient** is a 2D vector defined as

$$\nabla f(x,y) \equiv (\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y})$$

As you will learn in Calculus 3, the direction of the gradient vector is the direction in the (x, y) coordinate space where the function $f(x, y)$ is increasing at the fastest rate when you move in that direction. The

magnitude of the gradient vector is this rate of increase. For our sampled functions, we will work instead with the approximation of the gradient vector, which is defined just by plugging in the above finite differences into the gradient formula above. So when we talk about the "gradient" of a function on our grid, we're talking about this 2D vector of finite differences.

ASIDE: In many scientific computation problems, one uses the gradient vector to compute properties of a terrain surface (or a 2D intensity function), for example, finding local maxima or minima by searching along the surface. One way to do so is to use *gradient ascent* or *gradient descent*, respectively, which involves taking small successive steps in the gradient direction or opposite to the gradient direction, respectively. Think of this search procedure as walking in the locally defined *up hill* or *down hill* directions. Implementing such searches along a surface is well beyond the scope of this assignment, but hopefully you can imagine how this is a useful thing to do. In some of the most common and exciting applications these days (machine learning methods that use artificial neural networks), one of the core operations is gradient descent in very high dimensional spaces – thousands of dimensions, not just two! This is all very exciting and it is something that some of you may eventually be involved in eventually if you are analyzing large data sets.

Interpolation

Another common operation performed with sampled functions is to estimate a value of a function at a 2D position that does not lie on a grid point. This is called *interpolation* and there are various ways to do it. We will use *bilinear interpolation* which is defined mathematically in two steps.

Suppose first that a function $f(x, y)$ is defined on a unit square with corners having x and y in $\{0, 1\}$. Suppose we are given the value of the function only at these corners of this unit square. We write these given values as f_{00}, f_{10}, f_{01}, f_{11} where the first coordinate is row and the second coordinate is column. We interpolate the value $f_{\alpha,\beta}$ at an arbitrary point in the square as follows. (Note we use the notation $f_{\alpha,\beta}$ to emphasize that α is a row and β is a column.) We first define how to interpolate along the horizontal line segment from $(0, 0)$ to $(0, 1)$:

$$f_{0,\beta} = (1 - \beta)f_{00} + \beta f_{01}$$

and along the horizontal line segment from $(1, 0)$ to $(1, 1)$:

$$f_{1,\beta} = (1 - \beta)f_{10} + \beta f_{11}$$

Then, for any $\beta \in [0, 1]$, we interpolate $f_{\alpha,\beta}$ along the vertical line segments between $(0, \beta)$ and $(1, \beta)$:

$$\begin{aligned} f_{\alpha,\beta} &= (1 - \alpha)\ f_{0,\beta} + \alpha\ f_{1,\beta} \\ &= (1 - \alpha)((1 - \beta)\ f_{00} + \beta\ f_{01}) + \alpha((1 - \beta)\ f_{10} + \beta\ f_{11}) \\ &= (1 - \alpha)(1 - \beta)\ f_{00} + (1 - \alpha)\beta\ f_{01} + \alpha(1 - \beta)\ f_{10} + \alpha\beta f_{11} \end{aligned}$$

The interpolation function is said to be *bilinear* because for a fixed β it gives a linear interpolation in α and for a fixed α it gives a linear interpolation in β. You can implement bilinear interpolation by using the formula on the last line above.

To apply this interpolation method to any point (x, y) that is contained within the column and range row of the grid, you will need to consider a unit square that contains this point and whose four corners are on the grid so that the function values at these grid points are defined.

This completes the mathematical background you need for the assignment. Let's now turn to the programming task.

`Terrain` class

Define a class `Terrain`. Objects of the class should have the following attributes, which you will assign in the methods that follow. You should refer to these attribute descriptions when reading the Methods section later.

Attributes

- `n_rows` : number of rows. This corresponds to the y variable, increasing downward.
- `n_cols` : number of columns. This corresponds to the x variable, increasing rightward.
- `elevation` : an `ndarray` of type `float` with shape `(n_rows, n_cols)`. This array is the grid on which the sampled function $f()$ will be defined.

  Some of the methods described below will require comparing the `elevation` value at some position in the array with the `elevation` value at a neighboring position, or taking the difference between two such `elevation` values. To define such operations at the boundary of the array – when the neighboring position may be "off the grid" – we use the following wraparound policy: treat the `elevation` array as if it were periodic. For any column, the `elevation` value at row -1 is taken to be the `elevation` value at row `n_rows-1`. Similarly, for any row, the `elevation` at column -1 is taken to be the `elevation` at column `n_cols-1`. Note this assumption conveniently corresponds to Python indexing syntax, which can use negative numbers.

  [Updated: Nov. 21] To compute the `elevation`, it will be helpful to define the following two attributes. These attributes will also be used by the `compute_extrema` method described later.
- `col_mesh` : an `ndarray` with shape `(n_rows, n_cols)` which specifies the x coordinates of the mesh points. For simplicity, the x and y values taken by the mesh points will correspond to the column and row indices, respectively. You may use the NumPy `meshgrid` function to compute `col_mesh` and the below `row_mesh`. See here for an explanation of the `meshgrid` function. We will also discuss this in class in lecture 21.
- `gradient` : an `ndarray` of type `float` with shape `(2, n_rows, n_cols)` representing the discrete derivatives (finite differences) defined above. The two slices (index 0 and 1) defined by the first coordinate (first axis) correspond to the derivatives in the x and y directions, respectively.
- `extrema` : a dictionary representing the local minima and maxima of the `elevation`; the keys should be grid positions represented as (x, y) tuples and the values should be either `'min'` or `'max'`, indicating whether the grid position indicated by a key is a local minima or maxima of `elevation` according to the `min_elevation` and `max_elevation` methods.
- `row_mesh` : an `ndarray` with shape `(n_rows, n_cols)` which specifies the y coordinates of the mesh points. Same discussion as above for y and rows.

Methods

Implement each of the following methods.

It will be tempting to use `for` or `while` loops for some methods. However, **for this assignment you are not allowed to use loops, except for one method that we identify later.** The reason is that loops are very slow when working with large arrays, especially higher dimensional arrays. (Your examples will not involve large arrays since your main concern is that your methods are correct, but in a real application you typically *would* be using large arrays and you might not be able to afford using loops!) Instead of using loops, you will use the *broadcasting* and *vectorization* techniques that were discussed in the NumPy lecture. Note that it can be difficult at first to use these techniques. You'll have to consider which NumPy function(s) and operations to use in each case. But we think that once you get the hang of it, you will like it!

Also, note that for each of the methods defined below, the first parameter is `self`. However, when we refer to the 'inputs' of each of the methods, we will not include that parameter. The reason is that we will be referring to the arguments of the methods (the objects we pass to the methods), rather than to the parameters of the methods (the variables defined in the method headers).

- `__init__`: The constructor takes as input a `function` object, and optional arguments `n_rows` and `n_cols)` that determine the shape of the `elevation` array, and hence that determine the x and y integer values for sampling the function. The default values of the optional arguments both should be 5. The constructor sets the values for the attributes `col_mesh`, `row_mesh`, and `elevation`. [**Updated Nov. 21:** To obtain the `elevation` array, the row and column mesh arrays can be passed as arguments to the function that was given as an input.]

  For grading purposes, the `gradient` and the `extrema` attributes are not set in the constructor. Values for these attributes will be set by other methods below.

```
>>> t = Terrain(ramp_x, 2, 3)
>>> t.elevation
array([[0, 1, 2],
       [0, 1, 2]])
>>> t.row_mesh
array([[0, 0, 0],
       [1, 1, 1]])
>>> t.col_mesh
array([[0, 1, 2],
       [0, 1, 2]])
```

- `add`: takes as input a Terrain and an optional string argument `operator` whose default value is `'+'`. Modifies the `elevation` field by adding the `elevation` of the input Terrain to the current `elevation`, or subtracting if the value of `operator` is `'-'`. If the input Terrain has a different shape than the self Terrain, then raise an `AssertionError` with an appropriate message. If the operator is not one of `'+'` or `'-'`, then raise an `AssertionError` with an appropriate message.

  One way you can use this `add` method in your testing is to construct more complicated terrains than those defined in the basic function definitions given in **`helpers.py`**. For example, the `gaussian` function defined in `helpers.py` constructs a "Gaussian blob" which is a smooth bump of some given width and centered at some specific position in the grid. You could construct an `elevation` that is a sum and/or differences of such blobs, or add such blobs to the quadratic or 2D sine functions that are provided in `helpers.py`. Simple examples of how to build an `elevation` function using `add` are given in that file.

```
>>> t = Terrain(ramp_x, 2, 5)
>>> t1.elevation
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])
>>> t = Terrain(ramp_y, 2, 5)
>>> t2.elevation
array([[0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1]])
>>> t1.add(t2)
>>> t1.elevation
array([[0, 1, 2, 3, 4],
       [1, 2, 3, 4, 5]])
>>> t2.add(t1, '-')
>>> t2.elevation
array([[ 0, -1, -2, -3, -4],
```

```
       [ 0, -1, -2, -3, -4]])
```

- `threshold_elevation`: takes a float input `threshold`, and returns an `ndarray` of type `bool` with the same `shape` as `elevation`, indicating for each position if the value of elevation is strictly above that threshold.

```
>>> t = Terrain(ramp_x, 2, 5)
>>> t.elevation
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])
>>> t.threshold_elevation(2)
array([[False, False, False,  True,  True],
       [False, False, False,  True,  True]])
```

- `compute_gradient`: takes no inputs; computes the gradientbased on the `elevation` array attribute, using the finite difference formulas discussed in the Background section, and assigns it into the `gradient` attribute. Returns nothing.

  Note for those using doctest: if you put this example in a docstring, instead of placing a new line in the docstring example, you must write `<BLANKLINE>` so that doctest knows there should be a blank line instead of thinking that you are going to the next example.

```
>>> t = Terrain(ramp_x, 4, 5)
>>> t.compute_gradient()
>>> t.gradient
array([[[-1.5,  1. ,  1. ,  1. , -1.5],
        [-1.5,  1. ,  1. ,  1. , -1.5],
        [-1.5,  1. ,  1. ,  1. , -1.5],
        [-1.5,  1. ,  1. ,  1. , -1.5]],
<BLANKLINE>
       [[ 0. ,  0. ,  0. ,  0. ,  0. ],
        [ 0. ,  0. ,  0. ,  0. ,  0. ],
        [ 0. ,  0. ,  0. ,  0. ,  0. ],
        [ 0. ,  0. ,  0. ,  0. ,  0. ]]])
>>> t = Terrain(ramp_y, 4, 5)
>>> t.compute_gradient()
>>> t.gradient
array([[[ 0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.]],
<BLANKLINE>
       [[-1., -1., -1., -1., -1.],
        [ 1.,  1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.,  1.],
        [-1., -1., -1., -1., -1.]]])
```

- `threshold_magnitude_gradient`: takes a float input `threshold`, and returns an `ndarray` of type `bool` with the same `shape` as `elevation`, indicating for each position if the magnitude of the gradient vector is above the threshold.

```
>>> t = Terrain(f_sum_x_y, 5, 5)
>>> t.elevation
array([[0, 1, 2, 3, 4],
       [1, 2, 3, 4, 5],
       [2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7],
       [4, 5, 6, 7, 8]])
>>> t.compute_gradient()
>>> t.gradient
array([[[-1.5,  1. ,  1. ,  1. , -1.5],
        [-1.5,  1. ,  1. ,  1. , -1.5],
        [-1.5,  1. ,  1. ,  1. , -1.5],
        [-1.5,  1. ,  1. ,  1. , -1.5],
        [-1.5,  1. ,  1. ,  1. , -1.5]],

       [[-1.5, -1.5, -1.5, -1.5, -1.5],
        [ 1. ,  1. ,  1. ,  1. ,  1. ],
        [ 1. ,  1. ,  1. ,  1. ,  1. ],
        [ 1. ,  1. ,  1. ,  1. ,  1. ],
        [-1.5, -1.5, -1.5, -1.5, -1.5]]])
>>> t.threshold_magnitude_gradient(1.8)
array([[ True,  True,  True,  True,  True],
       [ True, False, False, False,  True],
       [ True, False, False, False,  True],
       [ True, False, False, False,  True],
       [ True,  True,  True,  True,  True]])
```

- `max_elevation`: takes no inputs; returns an `ndarray` of type `bool` with the same shape as `elevation` which indicates whether the elevation value at a position is strictly greater than the elevation values of its four neighbors (i.e., the positions to the immediate left, right, above, and below).

  For points on the boundary, assume the array is periodic as mentioned above. Also, as mentioned earlier, your solution may not use loops here and instead you must use vectorization and/or broadcasting. Note that for this function, you may achieve vectorization using the NumPy function `logical_and` which broadcasts the `and` operator across the two arrays on an element-by-element basis.

```
>>> t = Terrain(f_sum_x_y, 3, 4)
>>> t.max_elevation()
array([[False, False, False, False],
       [False, False, False, False],
       [False, False, False,  True]])
```

  As an aside, note that this is an oversimplified definition of a local maximum. For example, consider the following 3×3 matrix [[-1, 0, 2], [0, 1, 0], [2, 0, -1]] which might be part of a larger matrix. The center point would be considered a local max according to the above definition, but if you also were to examine the four other neighbors in the diagonal directions then you would see it is a local max in one diagonal but a local min in the other diagonal. (Technically, the middle point in this matrix is known as a *saddle* point since it has the geometry of a horse's saddle.) A better way to define a local maximum would be to consider all 8 nearest neighbors, but for this assignment the four neighbor definition given above is good enough.

- `min_elevation`: takes no inputs; returns an `ndarray` of type `bool` with the same shape as `elevation` and that indicates whether the elevation value at a position is strictly less than the elevation values of its four neighbors (i.e,. the positions to the immediate left, right, above, and below). Again, assume the array is periodic.

  Hint: avoid code repetition by making use of the `max_elevation` method here.

```
>>> t = Terrain(f_sum_x_y, 3, 4)
>>> t.min_elevation()
array([[ True, False, False, False],
       [False, False, False, False],
       [False, False, False, False]])
```

- `compute_extrema`: takes no inputs and returns `None`; assigns the `extrema` attribute, namely a dictionary representing the local minima and maxima of the `elevation` which are computed by the above two methods; the keys should be (x, y) tuples and the values should be either `'min'` or `'max'`, indicating whether the position indicated by a key is a local minima or maxima according to the `min_elevation` and `max_elevation` methods.

  This is the one method where we will allow you to use a loop. Note that you can use boolean indexing (lecture 21) rather than a loop to identify the extrema, and we strongly encourage you to do so. However, you would still need to use a `for` loop to construct your dictionary.

  Hint: to use boolean indexing, you will need to use the `row_mesh` and `col_mesh` attributes.

```
>>> t = Terrain(egg_carton, 32, 32)
>>> t.compute_extrema()
>>> t.extrema == {(4, 4): 'max', (20, 4): 'max', (4, 20): 'max', (20, 20): 'max',
                  (12, 12): 'min', (28, 12): 'min', (12, 28): 'min', (28, 28): 'min'}
True

>>> g = gaussian(15, 20, 4)
>>> t = Terrain(g, 64, 64)
>>> t.compute_extrema()
>>> t.extrema == {(15, 20): 'max', (63, 63): 'min'}
True
```

The next (and last) two methods will make use of a `bilinear_interpolate` function that is defined in the next section below. This function is not defined within the class since it does not depend on a specific object. You will need to implement that function before you can implement the next two methods.

- `interpolate_elevation`: takes as input two floats x and y and returns the bilinearly interpolated elevation value at that position. If the (x, y) values are not in the range of the mesh grid, then it treats the elevation map as periodic and essentially infinite in extent in both x and y directions.

```
>>> t = Terrain(ramp_y, 5, 5)
>>> t.interpolate_elevation(3.75, 2.25)
2.25
```

- `interpolate_gradient`: takes as input two floats x and y and returns a tuple of length 2 with the interpolated values of `gradient` at that (x, y) position.

```
>>> t = Terrain(ramp_x, 6, 6)
>>> t.interpolate_gradient(2.75, 3.25)
(1.0, 0.0)
>>> t = Terrain(ramp_y, 6, 6)
>>> t.interpolate_gradient(2.75, 3.25)
(0.0, 1.0)
```

Functions

There is just one to implement:

- `bilinear_interpolate`: takes as input two floats `alpha` and `beta` which are both in the unit interval [0,1], and four floats `f_00, f_01, f_10, f_11` which are the values of some function defined at the four corners of a unit square. The 00, 01, 10, 11 indices represent the upper left, upper right, lower left, and lower right corners of the square, respectively, and the `alpha` variable is associated with y or row coordinate and the `beta` variable is associated with the x or column coordinate. The method returns the bilinearly interpolated elevation at position `(alpha, beta)` in the unit square. See the Background section for details and discussion of notation.

```
>>> t = Terrain(ramp_x)
>>> t.bilinear_interpolate(.25, 0, 1, 2, 3, 4)
1.5
>>> t.bilinear_interpolate(0, .25, 1, 2, 3, 4)
1.25
>>> t.bilinear_interpolate(1, .75, 1, 2, 3, 4)
3.75
>>> t.bilinear_interpolate(.5, .75, 1, 2, 3, 4)
2.75
```

What To Submit

You must submit all your files on codePost (https://codepost.io/). The files you should submit are listed below. Any deviation from these requirements may lead to lost marks.

`terrain.py`

`README.txt` In this file, you can tell the TA about any issues you ran into while doing this assignment.

Remember that this assignment like all others is an `individual` assignment and must represent the entirety of your own work. You are permitted to verbally discuss it with your peers, as long as no written notes are taken. If you do discuss it with anyone, please make note of those people in this `README.txt` file. If you didn't talk to anybody nor have anything you want to tell the TA, just say "nothing to report" in the file.

You may make as many submissions as you like, but we will only grade your final submission. (All prior ones are automatically deleted.)

Note: If you are having trouble, make sure the names of your files are exactly as written above.

Assignment meeting

In the week(s) following the due date for this assignment, you will be asked to meet with a TA for a 25-30 minute meeting. In this meeting, the TA will discuss with you what you should improve for future assignments.

Only your code will determine your grade. You will not be able to provide any clarifications or extra information in order to improve your grade. However, you will have the opportunity to ask for clarifications regarding your grade.

You may also be asked during the meeting to explain portions of your code. Answers to these questions will not be used to determine your grade, but inability to explain your code may suggest that you did not write it, and may lead to your code being examined more closely for possible plagiarism.

Details on how to schedule a meeting with the TA will be shared with you in the days following the due date of the assignment.

If you do not attend a meeting, you will receive a 0 for your assignment.