1 Natural Neighbor Interpolation

In canvas, you will find a file pixelliste.csv. This text file contains one defined pixel value per line in the format x-position, y-position, r-value, g-value, b-value, each separated by a semicolon.

1.1 Read in Pixel List (5 points)

Complete the code in the file natural_neighbor_interpolation.py.

a) Complete the unit tests (test_read_pixel_cloud_from_csv).

b) Describe your test strategy: which cases do you test and why? What could happen when reading a file, which cases are worth testing?
Test if rows in csv add up with value in row 0. Test if every row has 5 values. Test if expected types match.

I needed the following time to complete the task: 5min

1.2 Create Neighborhood Relation (10 points)

The pixels in the list are in random order. In order to interpolate, we first have to reconstruct the neighborhood relationships, i.e. we have to determine which pixel is next to which. This relationship is not unique - there are generally several solutions. We determine the neighborhood via a Delauney triangulation. This means that we build a Voronoi diagram and define two pixels as neighboring if their Voronoy cells touch.

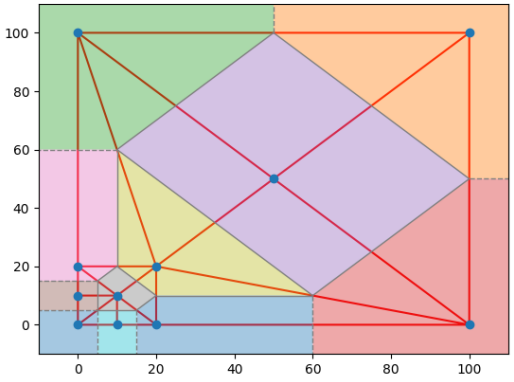
a) Familiarize yourself with the documentation of the scipy.spatial.Voronoi and scipy.spatial.Delauney package. The function create_delauney_triangulation demonstrates how to use the package. We do not bother to reprogramming the algorithm at this point.

b) Research the algorithm to create a Voronoi diagram on your own. Describe the algorithm in your own words.

c) Bonus task: Think about properties of our points: we have pixel coordinates, some of which are missing. Can you think of properties of these points that we could possibly use to create a Voronoi diagram faster than the implementation in scipy.spatial?

d) Bonus task: Argue which development environments (programming language, compiler, hardware) could be considered, and which of them would be expected to lead to fast or slow code.

The Voronoi diagram and the associated Delauney triangulation look like this:



I needed the following time to complete the task:

1.3 Barycentric coordinates (10 points)

To be able to interpolate between three points of a triangle, we need a weighting between the points. An obvious possibility would be to weight each point with the reciprocal of the distance, normalized by the sum of the weights of all three vertices.

a) Argue in your own words what the disadvantage of this method is.

b) Derive barycentric coordinates. The coordinates fulfill the following conditions: multiply each of the three vertices by its weight and add the coordinates to obtain the target point. This applies in the x-direction (condition 1) and y-direction (condition 2). In addition, the weights add up to 1 (normalizing condition). Derive a formula for Barycentric coordinates by solving for W_1, W_2 and W_3.

Note: the derivation can easily be found on the Internet. Nevertheless, solve it yourself first.

$$P_x = W_1 \cdot X_1 + W_2 \cdot X_2 + W_3 \cdot X_3 \tag{1}$$
$$P_y = W_1 \cdot Y_1 + W_2 \cdot Y_2 + W_3 \cdot Y_3 \tag{2}$$
$$1 = W_1 + W_2 + W_3 \tag{3}$$

c) The function test_compute_barycentric_coordinates already contains some test cases. Argue: are these test cases sufficient, or which test cases should be added?
Adding a test for Px, Py = x1 or x2 or x3 can be good.
Also maybe test for coordinates outside of the triangle.
Also add a test to check if $1 = W_1 + W_2 + W_3$

d) Add the test cases according to your argumentation from c).

e) Complete the source code in the function compute_barycentric_coordinates.

I needed the following time to complete the task: 1h

1.4 Interpolation of the pixel values (5 points)

Now implement an interpolation method for pixel values: use the barycentric coordinates to interpolate color values.

a) First add the test cases in test_compute_interpolated_pixel_value. Argue why you have added the test cases in exactly the same way. Not adding test cases if there are good reasons to do so is also a valid solution throughout the lecture. In this case, argue why you consider an addition to be unnecessary.

b) Now implement the function compute_interpolated_pixel_value.

I needed the following time to complete the task: 20min

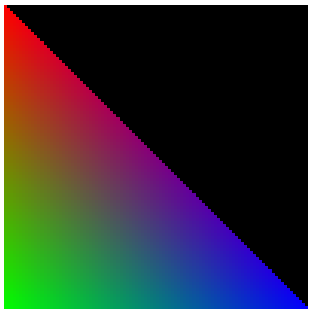
1.5 rasterization of triangles (20 points)

Based on the function for interpolating pixel values, implement the complete natural neighbor interpolation.

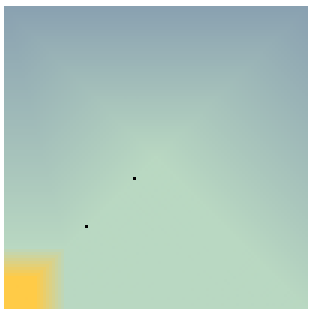
a) You rasterize a triangle by first determining the axis aligned bounding box, i.e. the smallest and largest coordinate in the x and y directions. The triangle always lies within this box. As always: first add test cases or argue why this is not necessary.
Test can help since when handling coordiantes you can quickly confuse them.

b) Now iterate all pixels within the box. Use two nested loops. Calculate the interpolated color value for each pixel and save it in the array. If necessary, first add test cases. Then implement the function.

c) Not all pixels within the box are also within the triangle. This must be dealt with. A simple test of whether a point is inside a triangle uses barycentric coordinates: if one of the coordinates is negative, the point is outside. Build in an appropriate test. For example, the interpolation function could return None if the point is outside. You could also write a separate function (pixel_inside_trangle) and not even calculate the interpolated color values if the pixel is outside.



d) You have been asked for help because a colleague's implementation has pixel errors. Maybe your implementation has the same problem? Speculate what could be causing the error and suggest a solution.
floating point errorsg



I needed the following time to complete the task: 1h