CS 548: Assignment 03

Programming Assignments (95%)

python/Assign03.py

In this assignment, you will implement the Moving Least Squares (MLS) approach in Python. Except for the last function (perform_moving_least_squares), all functions assume numpy parameters.

Include the following function:

```python
def enforce_viewpoint_consistency(centroid, U, V, W, viewpoint=(0,0,30)):
    # Get vector from point to viewpoint (camera)
    to_view = viewpoint - centroid
    # Normalize
    to_view = to_view / np.linalg.norm(to_view)
    # Get dot product of that with W
    dot_val = np.dot(to_view, W)
    # If negative, flip
    if dot_val < 0:
        W = -W

    # Return results
    return U, V, W
```

Define these functions:

- **def compute_distances(center, points)**
  - Calculate the Euclidean distances of center from each of the points and return the array of distances.
- **def compute_gaussian_weights(center, points, sigma)**
  - Call compute_distances() to get the point distances and then calculate the per-point weights using a Gaussian function: $e^{\frac{-(d^2)}{2(\sigma^2)}}$
- **def compute_weighted_PCA(points, weights)**
  - Compute weighted PCA (specifically the weighted centroid, U, V, and W vectors) and return those values.
    - **WARNING:** BEFORE returning the values, make sure that the normals (W) are consistent by calling:
      U, V, W = enforce_viewpoint_consistency(weighted_centroid, U, V, W)

    - This will align the normals such that they point towards the camera (by default located at (0,0,30)). This is not an uncommon adjustment when dealing with 3D data captures from a specific perspective.
    - In our case, however, this will cause normals on the opposite side of the model to be flipped inward. This is expected behavior.
  - You may use np.linalg.eigh() to calculate eigenvalues and eigenvalues, BUT remember that the LEAST important vector (the normal) is actually the FIRST vector in the list!
- **def project_points_to_plane(points, centroid, U, V, W)**
  - Given an array of points and the data calculated via PCA (centroid, U, V, W), project the points onto the plane formed by these axes and centroid.
  - Return the projected points.
  - I would suggest usage of:
    - Numpy matrix multiplication: C = A @ B
    - The transpose function: At = np.transpose(A)
    - A matrix that transforms a COLUMN vector point into the perspective of the three axes:
      $$\begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$
- **def reverse_plane_projection(projected, centroid, U, V, W)**
  - Reverses the process of project_points_to_plane().
  - Return the UN-projected points.
- **def make_design_matrix_A(projected)**
  - Make a design matrix as described in the MLS slides for a 2^{nd} order polynomial.
  - As a reminder, each point in projected now has the coordinates (u, v, w).
  - Remember there is one row per point.
- **def make_vector_b(projected)**
  - Return the appropriate b vector as described in the MLS slides.
  - Please note this should be a COLUMN vector (one row per point).
- **def make_weight_matrix_G(weights)**
  - Make the appropriate weight matrix G as described in the MLS slides.
  - I would strongly recommend the use of np.diag().
- **def compute_polynomial_coefficients(projected, weights)**
  - Create a design matrix A.
  - Create an output matrix b.
  - Create a weight matrix G.
  - Compute the solution to the polynomial coefficients (a).
  - You may use np.linalg.inv() to get the inverse where relevant.

- **def project_points_to_polynomial(points, centroid, U, V, W, a)**
  - Project the points into the plane formed by (centroid, U, V, W).
  - Create a designed matrix for the projected points.
  - Get the predicted w coordinates: A @ a
  - Replace the original w coordinates in the projected points with the predicted ones.
  - Reverse the plane projection to get new points.
  - Return the points.
- **def fit_to_polynomial(center, points, sigma)**
  - Compute the appropriate Gaussian weights
  - Compute weighted PCA.
  - Project the points to the plane.
  - Compute the polynomial coefficients.
  - Project the center point only to the polynomial.
  - Return the updated center point AND the W vector (the normal).
- **def perform_moving_least_squares(cloud, radius, sigma)**
  - Create a KDTree using Open3D from the cloud
  - Create arrays for the output points, normals, and colors.
  - For each point in the cloud:
    - Use a radius search to find the neighbors.
    - Fit the neighborhood to a polynomial.
    - Write the updated center and normal to your output arrays.
    - For the “color” use the distance of the original query point from the updated query point as the “red” component.
  - Create an output (legacy) PointCloud and set the points, normals, and colors.
  - Return the output point cloud.

While optional, you may find it helpful to have the following functions for visualization purposes:

```
def visualize_clouds(all_clouds, point_show_normal=False):
    adjusted_clouds = []
    x_inc = 20.0
    y_inc = 20.0

    for i in range(len(all_clouds)):
        one_set_clouds = all_clouds[i]

        for j in range(len(one_set_clouds)):
            center = (x_inc*j, y_inc*i, 0)
            adjusted_clouds.append(one_set_clouds[j].translate(center))

    o3d.visualization.draw_geometries(adjusted_clouds,
            point_show_normal=point_show_normal)

def main():
    cloud = o3d.io.read_point_cloud(
            "data/assign03/input/noise_pervasive_large_bunny.pcd")

    radius = 1.0
    sigma = radius / 3.0

    mls_cloud = perform_moving_least_squares(cloud, radius, sigma)
    output_cloud = copy.deepcopy(mls_cloud)
    output_cloud.normals = o3d.utility.Vector3dVector([])

    output_points_only = copy.deepcopy(mls_cloud)
    output_points_only.colors = o3d.utility.Vector3dVector([])
    output_points_only.normals = o3d.utility.Vector3dVector([])

    output_points_normals = copy.deepcopy(mls_cloud)
    output_points_normals.colors = o3d.utility.Vector3dVector([])

    G03.visualize_clouds([[cloud, output_points_only, output_points_normals,
output_cloud]], point_show_normal=True)

if __name__ == "__main__":
    main()
```

Testing Screenshot (5%)

I have provided several files for testing:

- data/assign03
    - input/ - contains input cloud files (some with noise)
    - ground/ - contains the ground truth files (of which there are many)
- python/
    - Test_Assign03.py – the test program for the Python code

Run the testing program through the testing section of Visual Code.

You MUST run the tests and send a screenshot of the test results! Even if your program(s) do not pass all the tests, you MUST send this screenshot!

Python Tests

You may have to do “Command Palette” → “Python: Configure Tests” → pytest → python (directory)

You should then be able to run the Python tests in your testing window in Visual Code.

ALTERNATIVELY: open a terminal and enter: **pytest python/Test_Assign03.py**

…then take a screenshot of the terminal output.

Grading

Your OVERALL assignment grade is weighted as follows:

- 5% - Testing results screenshot
- 95% - Programming assignments

I reserve the right to take points off for not meeting the specifications in this assignment description. In general, these are things that will be penalized:

- **Code that is not syntactically correct (up to 60 points off!)**
- Sloppy or poor coding style
- Bad coding design principles
- Code that crashes, does not run, or takes a VERY long time to complete
- Using code from ANY source other than the course materials
- Collaboration on code of ANY kind; this is an INDIVIDUAL PROJECT
- Sharing code with other people in this class or using code from this or any other related class
- Output that is incorrect
- Algorithms/implementations that are incorrect
- Submitting improper files
- Failing to submit ALL required files