GCV. HW1

TODO1:

```python
#  TODO: write your code to constrict a world-frame point cloud from a depth image,
#  using known intrinsic and extrinsic camera parameters.
#  Hints: use the class `RaycastingImaging` to transform image to  points in camera frame,
#  use the class `CameraPose` to transform image to points in world frame.
pose_i = CameraPose(extrinsics[i])
imaging_i = RaycastingImaging(intrinsics_dict[i]['resolution_image'], intrinsics_dict[i]['resolution_3d'])
points_i = pose_i.camera_to_world(imaging_i.image_to_points(image_i))
```

The first line inits camera, defined by its extrinsics matrix, which describes the camera's location in the world, and what direction it's pointing. The second line inits Raycasting by intrinsics dic - we need this line to generate rays that will be used for transformation. The third line leverages the two previous lines by firstly, transforming image to points, and then converting these points to world frame.

TODO2:

```python
#  TODO: your code here: use functions from CameraPose class
#  to transform `points_j` into coordinate frame of `view_i`
reprojected_j = pose_i.world_to_camera(points_j)
```

This piece of code converts points from world frame to camera coordinates. Thus, we transform points `j` to view `i`.

TODO3:

```python
# TODO: your code here: use cKDTree to find k=`nn_set_size` indexes of
#  nearest points for each of points from `reprojected_j`
uv_i = imaging_i.rays_origins[:, :2]
_, nn_indexes_in_i = cKDTree(uv_i).query(reprojected_j[:, :2], nn_set_size)
```

`imaging_i.rays_origins` define (u, v) coordinates of `points_i` in the pixel grid of `view_i`. We use all except the last coordinate. After that for each reprojected point we look for nearest points among (u, v) pairs. The amount is limited by `nn_set_size`.

TODO4:

```
# Build an [n, 3] array of XYZ coordinates for each reprojected point by taking
# UV values from pixel grid and Z value from depth image.
#  TODO: your code here: use `point_nn_indexes` found previously
#         #  and distance values from `image_i` indexed by the same `point_nn_indexes`
         point_from_j_nns = np.concatenate((uv_i[point_nn_indexes], image_i_flat[point_nn_indexes][..., None]), axis=1)
```

I flattened image in advance (for not to repeat in each loop call). Here we get 2d array with last dimension equal to 3 after concatenation.

TODO5:

```
# TODO: compute a flag indicating the possibility to interpolate
# by checking distance between `point_from_j` and its `point_from_j_nns`
# against the value of `distance_interpolation_threshold`
        distances_to_nearest = np.sqrt(((point_from_j - point_from_j_nns) ** 2).sum(axis=1))
        interp_mask[idx] = np.all(distances_to_nearest < distance_interpolation_threshold)
```

In vectorized case, I had to unsqueeze one dimension, here it worked without it. So, I extracted one point from several points (or vice versa, doesn't matter) and calculated euclidian norm of obtained vectors. After that I obtained boolean mask as result of comparison of distances with the threshold.

TODO6:

```
# TODO: your code here: use `interpolate.interp2d`
#                 #  to construct a bilinear interpolator from distances predicted
#                 #  in `view_i` (i.e. `distances_i`) into the point in `view_j`.
#                 #  Use the interpolator to compute an interpolated distance value.
                interpolator = interpolate.interp2d(point_from_j_nns[:, 0], point_from_j_nns[:, 1], distances_i_flat[point_nn_indexes])
                distances_j_interp[idx] = interpolator(point_from_j[0], point_from_j[1])
```

I use points as (x, y, z) to fit interpolator and after that use (x, y) to estimate z by fitted interpolator.