Geometric Computer Vision (HW)

Denis Rollov.

```
# 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.
```

In this #TODO we need to calculate the following variables: points_i, pose_i, imaging_i (not incorporate image_i, distances_i).

In order to transform image to points in world frame, we merely need to create object CameraPose. To transform image to points in camera frame we need to define RaycastingImaging via two variables: **resolution_image** (it is just the resolution of our image (shape[0])) and **resolution_3d** (mm/pixel) as well. In order to calculate **points_i** initially it is important to transform **images_i** to points and then convert the points from the camera to world. Subsequently, we will be able to construct a world-frame point cloud from a depth image.

```
# Reproject points from view_j to view_i, to be able to interpolate in view_i.
# We are using parallel projection so this explicitly computes
# (u, v) coordinates for reprojected points (in image plane of view_i).
# TODO: your code here: use functions from CameraPose class
#  to transform `points_j` into coordinate frame of `view_i`
```

Simple step: create a transformation from world to camera via method **world_to_camera**.

```
# For each reprojected point, find K nearest points in view_i,
# that are source points/pixels to interpolate from.
# We do this using imaging_i.rays_origins because these
# define (u, v) coordinates of points_i in the pixel grid of view_i.
# TODO: your code here: use cKDTree to find k=`nn_set_size` indexes of
#  nearest points for each of points from `reprojected_j`
```

```
```

In order to realize this part of the code it is necessary to find the solution what to input into **cKDTree**. The input is x, y coordinates of **rays_origins**. After that it is necessary to query the kd-tree for nearest neighbors (inputs in query are reprojected x and y and number of the nearest neighbors to return).

```
# 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
```

Before this step it is important to initialize flattened vectors of image and distance (in order to speed-up our code and not repeat the same calculations). So, I just took number of neighbours indexes and concat x,y (array R^{n, 2} and z (column-vector).

```
# 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`
```

In order to calculate the distance it is necessary to use Euclidean distance. The next step is applying mask into our obtained distances.

```
# 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.
```

Bilinear interpolator from predicted distance can be constructed via three variables: x-values, y-values, flatten vector of number of corresponding number of neighbours. Subsequently, interpolator just takes two values as input: xnew and ynew from reprojected matrix (transformation world to camera).