FP.4 Compute Camera-based TTC

Compute the time-to-collision in second for all matched 3D objects using only keypoint correspondences from the matched bounding boxes between current and previous frame.

$$TTC_{camera} = \frac{-dt}{(1 - \frac{h_1}{h_0})}$$

The idea is that by calculating the scale change in the image coordinate, we can calculate the Time-To-Collision with the formula given above. The scales are distance between matched keypoints in a given frame (h_1 current frame, h_0 previous frame).

For Robust calculation, points below were considered:

- To not select two points that are very close to each other, `minDist=100` is chosen (it's a hyper-parameter and can be tuned further). By not considering the close points we are reducing the potential affect of the outliers (otherwise in case of outlier the difference in the scale could be huge!).
- Another way to reduce the effect of outliers is to calculate the mean scale change and then exclude the ones that are much larger than the mean value (specifically scale changes that are larger than `2*stddev`).
- Finally, we use the `median` of the calculated distance ratios to compute the TTC (more robust than using the mean value in the presence of potential outliers)

The code is implemented in the function `computeTTCCamera`:

1. Loops over the not repeated combination of two `kptMatches` from current bounding-box.
    a. using two for loops where inner for loop starts from one element ahead of the outer loop and goes to the end (outer loop starts from 0 and goes to one element before the last one).

```
for (auto it1 = kptMatches.begin(); it1 != std::prev(kptMatches.end()); ++it1)
    {
        ...

        // start from the one element ahead of outer iterator to prevent repeated distance calculation
        for (auto it2 = std::next(it1); it2 != kptMatches.end(); ++it2)
        {
            ...
        }

        ...
    }
```

2. Calculate the scale (or distance) between two matched points in both current frame and previous frame and then calculate the difference (scale change) and added it to `distChangeVec`
3. Calculate the `mean` and `stddev` of `distChangeVec`
4. Loop over all elements in `distChangeVec` and only consider the ones that are not far away from the mean (hence, removing the outliers)
    a. calculate the distance ratio vector to be used in TTC computation

```
vector<double> distRatioVec;
for (int i = 0; i < distChangeVec.size(); ++i)
{
    if (abs(distChangeVec[i] - meanDistChange) < 2.0*stddevDistChange)
    {
        distRatioVec.push_back(distanceCurrVec[i] / distancePrevVec[i]);
```

```
    }
}
```

5. Calculate the median of `distRatioVec` to use in computation of TTC

```
// compute camera-based TTC from the median of distance ratios
sort(distRatioVec.begin(), distRatioVec.end());
int vecSize = distRatioVec.size();
double medianDistRatio;
if (vecSize % 2 == 0)
{
    // even number of points; need to calculate the mean of two elements in the middle of the sorted array
    medianDistRatio = (distRatioVec[vecSize/2 - 1] + distRatioVec[vecSize/2]) / 2;
}
else
{
    // odd number of points; just take the middle element of the sorted array
    medianDistRatio = distRatioVec[vecSize/2];
}

double dt = 1.0 / (frameRate + 1e-8);
double denom = 1 - medianDistRatio;
if (abs(denom) < 1e-6)
{
    TTC = NAN;
    cerr << "Warning: Calculated medianDistRatio is close to 1 (i.e.: no scale change): " << medianDistRatio << endl;
}
else
{
    TTC = -dt / denom;
    cout << "Camera-TTC= " << TTC << endl;
}
```