Introduction

When providing solutions for autonomous driving through external sensors and cameras, it is paramount to keep track of irregularities and errors in the incoming data flow. Continuous monitoring of all sensor inputs becomes necessary to ensure system reliability. Among potential irregularities, a crucial concern arises if cameras, tasked with detecting obstacles, become tilted in varying directions. Such misalignment can precipitate system failures, posing a significant hazard to pedestrians crossing paths with autonomous vehicles due to erroneous data interpretation. To address this challenge, I devised an algorithm capable of detecting shifts in camera orientation, enabling precise calculation of pitch, roll, and yaw angle

Algorithm for detecting camera tilt

To calculate the tilt of the camera I came up with the following algorithm which is separated into the following steps:

1. **Image Resizing:** Resize images to a fixed size to avoid distortions
2. **Feature Detection and Matching:** Use SIFT to detect key points and compute descriptors. Match descriptors using BFMatcher with KNN
3. **Homography Estimation:** Estimate the homography matrix using RANSAC
4. **Camera Intrinsics:** Define the camera intrinsic matrix based on focal length and image center
5. **Homography Decomposition:** Decompose the homography matrix to exact possible rotation matrices
6. **Euler Angles Calculation:** Convert rotation matrices to Euler angles

The implementation of the here-explained algorithm can be seen in the following GitHub repository: https://github.com/jj0054/CameraAngleCalculation

1. Image Resizing

After reading the two images, it is important to bring both images to a common size. This is necessary since this might lead to distortion while calculating the angles in the next steps. OpenCV provides the functionality of changing the format of an image by passing the desired size and the image to the resize() function.

```python
# Resize the Images to a predefined size
def resizeImages(image1, image2, target_size):
    resized_img1 = cv2.resize(image1, target_size, interpolation=cv2.INTER_AREA)
    resized_img2 = cv2.resize(image2, target_size, interpolation=cv2.INTER_AREA)
    return resized_img1, resized_img2
```

Resizing of the input images to prevent disortion.

2. Feature detection and matching

To figure out how much a camera has tilted, we need to first extract similarities from both images. By doing so, the different feature coordinates (key points) can be compared and the tilt determined.

```
1 usage
def featureDetection(image1, image2):
    # Apply SIFT on pictures to find key points and descriptors
    sift = cv2.SIFT_create()
    keypoints_1, descriptors1 = sift.detectAndCompute(image1, None)
    keypoints_2, descriptors2 = sift.detectAndCompute(image2, None)

    return keypoints_1, descriptors1, keypoints_2, descriptors2
```

FeatureDetection function with SIFT.

A commonly used method is the so-called SIFT (Scale-Invariant Feature Transform) algorithm. It is a computer vision technique used for feature detection and description. As a result of applying this algorithm on both images, key points and descriptors are being returned. Key points represent the coordinates of extracted feature points on each image. Descriptors are vectors that give the key points a signature so the features can be found on another image with a different perspective.

The official paper on the SIFT algorithm provides more detailed information on this topic [https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf]

The SIFT algorithm can be implemented by the following code:

```
# Match descriptors to find pairs
1 usage
def featureMatching(descriptors1, descriptors2):
    bf = cv2.BFMatcher()
    matches = bf.knnMatch(descriptors1, descriptors2, k=2)

    return matches
```

Matching descriptors from image1 with image2 and find pairs to extract the same features on each image

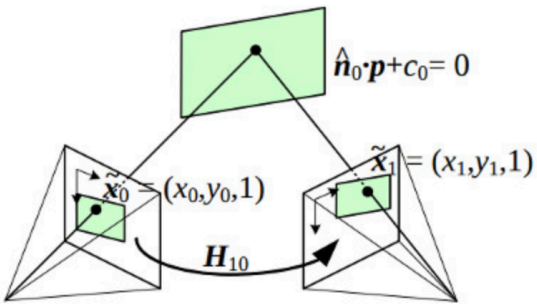
After finding the key points and descriptors of each image, it is important to find their matching features. With a function by OpenCV, it is easy to match the descriptors from both images.

The result of SIFT and the feature-matching can be visualized in the following picture:

Extracted key points matched with the corresponding key points on the second image

3. Homography Estimation

For the purpose of this program, I will be using a mathematical concept known as homography in order to calculate the tilt of the camera. As a brief summary, a homography matrix is used to describe the transformation between two pictures, in particular how a point (x/y) can be projected as (x'/y') on another picture. This matrix can be displayed as a 3x3 matrix H:



$$H = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix}$$

This type of algorithm deals with common problems associated with computer vision. Common cases are determining the relative angles and position of two cameras that take a picture of the same object. However, the tilt problem that this protocol is addressing concerns a special case—the pure camera rotation problem. In this case, two pictures are taken by the same camera but with different rotations.

The first step of applying the homography matrix is to have the coordinates of the extracted key points of each image. Additionally, it is necessary to filter out all coordinates that don't align with the data and are considered outliers. For example, some pairs of key points do not match or poorly match each other, as shown in my example. The RANSAC algorithm (Random sample consensus) can be applied while calculating H with the findHomography function provided by OpenCV. This additional step can effectively clean up the data and increase the accuracy of the result.

```python
def homographyEstimation(keypoints_1, keypoints_2, good_matches):

    # Extract coordinates from key points
    src_pts = []
    dst_pts = []
    for m in good_matches:
        src_pts.append(keypoints_1[m.queryIdx].pt)
        dst_pts.append(keypoints_2[m.trainIdx].pt)

    src_pts = np.float32(src_pts).reshape(-1, 1, 2)
    dst_pts = np.float32(dst_pts).reshape(-1, 1, 2)

    # Get homography-matrix and apply RANSAC to filter datapoints
    H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)

    return H, mask
```

Extract coordinates from key points and get homography-matrix after applying RANSAC algorithm

4. Camera Intrinsics

One significant problem that can occur when applying the homography matrix is camera specifics. This involves knowing the focal length of the camera. Ideally, those specifics are known to the program so that it can have a more accurate estimate of the distance between the object and the camera. If that information is unknown, a camera calibration process might be used but this leads to greater margins of errors and inaccuracy. This process involves taking multiple images of a known calibration pattern, such as a checkerboard, from different angles. It detects the corners of the pattern in each image and solves for the parameters that best describe the transformation between the 3D world and the 2D image plane. Through this step, a rough estimate of camera intrinsics can be obtained.

Once the camera intrinsics are known, they can be represented by a 3×3 matrix K, which includes the focal length and the principal point (center coordinates of the image). With the intrinsic matrix K determined, it becomes possible to decompose the homography matrix to extract the camera tilt angles, giving insights into the camera's orientation in space.

$$K = \begin{pmatrix} focallength & 0 & cx \\ 0 & focallength & cy \\ 0 & 0 & 1 \end{pmatrix}$$

Different cameras will vary with regard to the intrinsics, and hence, it is crucial to be aware of its different focal lengths or perform the calibration step whenever a different camera is used. This ensures accurate angle extraction and 3D reconstruction. In the provided code, an iPhone 12 camera was used, therefore the camera specifics were provided to construct K. When using a different camera the focal length has to be adjusted.

5. Homography Decomposition

Once the camera intrinsics are determined, the next step is to decompose the homography matrix to extract the rotation matrices, which contain the tilt angles of the camera. This process is crucial for understanding the camera's orientation in space. OpenCV offers a convenient function, *decomposeHomographyMat*, which facilitates this task by returning the rotation matrix, translation vectors, and normal vectors of the planes involved.

```
def homographyDecomposition(H, K):
    _, rotations, translations, normals = cv2.decomposeHomographyMat(H, K)

    return rotations, translations, normals
```

Decompose homography matrix with the camera intrinsics

However, be aware that given the complexity of decomposing the 3D plane, it is possible to have multiple solutions of rotation. This would be a potential shortcoming for this approach as one rotation needs to be manually chosen.

6. Euler Angles Calculation

The function rotationMatrixToEulerAngles(R) serves to extract Euler angles from a rotation matrix, a crucial step in understanding the camera's orientation in three-dimensional space. These angles

```python
# Extract the euler angles from the rotation matrix
2 usages
def rotationMatrixToEulerAngles(R):
    sy = math.sqrt(R[0, 0] * R[0, 0] + R[1, 0] * R[1, 0])

    if sy < 1e-6:
        singular = True
    else:
        singular = False

    if not singular:
        x = math.atan2(R[2, 1], R[2, 2])
        y = math.atan2(-R[2, 0], sy)
        z = math.atan2(R[1, 0], R[0, 0])
    else:
        x = math.atan2(-R[1, 2], R[1, 1])
        y = math.atan2(-R[2, 0], sy)
        z = 0

    return np.array([x, y, z])
```

Function to determine the angles from the rotation matrices

denote rotations around the X, Y, and Z axes. To ensure accurate angle calculation, the function incorporates a precautionary measure: it computes the sine of the rotation around the Y-axis and verifies its proximity to zero, thereby detecting potential singularities in the rotation matrix. This robust approach guarantees precise insights into the camera's orientation, even when confronted with matrices nearing singularity.

Application of this function is necessary for every rotation matrix previously determined through the decomposition of the homography matrix. Given that a single homography matrix can encapsulate multiple combinations of rotation, translation, and scaling between images, determining the angles of each rotation matrix is imperative. However, before these angles can be utilized for further applications, it's essential to validate the most feasible rotation matrix to ensure accuracy and reliability.

7. Application and Conclusion

Before ending this protocol, I would like to take the opportunity to discuss the application and potential shortcomings of my approach to the tilt problem.

Application and Advantages:

1) Wide application in Computer Vision: as mentioned earlier, the homography approach can deal with a wide range of problems in computer vision that are not limited to pure camera rotation. Other possible applications are panorama creation and object tracking.
2) Simplicity: as my code will demonstrate, calculating the camera rotation under this approach does not require too many lines of code and can free up computational capacity for other tasks.

3) Efficiency: there exist many libraries and algorithms already for homography estimation. This can effectively simplify the task of developers.

However, we should also take the shortcomings of this approach into account when applying it to real-life problems.

1) Camera specifics: although in most cases, the camera intrinsics are well-known or can be easily obtained, the estimate of rotation is less accurate if that information is unavailable.
2) Multiple estimations: The homography approach might sometimes come up with multiple solutions and unreliable results when dealing with a more complex 3D plane. Further manul validations might be necessary.
3) Picture quality: for an accurate result, homography requires quality images. Noises or distortions in the images can either render this approach unavailable or largely inaccurate results. This means that images might need to be pre-processed before being feed into the homography algorithm.

In conclusion, the homography approach and the algorithm that I developed in this protocol are valuable in dealing with the camera tilting problem. However, we also need to be aware of its limitations and develop additional algorithms to ensure its accuracy.