Table of Contents

I have a stereo image pair. Can we stereo rectify it and perform some sort of triangulation or distance estimation to objects in images?

Yes, you can. The general approach involves two main stages:

1. **Stereo Rectification:**
   - **Purpose:** Rectification reprojects the images so that the epipolar lines become horizontal. This simplifies the search for corresponding points between the two images (they'll lie on the same row).
   - **How to Do It:**
     - **Calibration:** First, you typically need to calibrate your stereo camera system. Calibration gives you the intrinsic parameters (focal length, principal point, distortion coefficients) for each camera and the extrinsic parameters (rotation and translation between the cameras). If you don't have calibration data, you can estimate the epipolar geometry by computing the fundamental matrix, but full calibration is preferable for accurate depth estimates.
     - **Rectification Algorithms:** With calibration data, you can use functions such as OpenCV's `cv::stereoRectify` (or the Python equivalent) to compute rectification transforms. Once

you have these, you use `cv::initUndistortRectifyMap` and `cv::remap` to actually warp the images into a rectified coordinate system.

2. **Triangulation and Distance Estimation:**
   - **Triangulation Basics:** Once you have rectified images, you find corresponding points (using feature matching, block matching, or semi-global matching algorithms) and compute the disparity (the horizontal shift between corresponding points).
   - **Depth Calculation:** With known camera parameters (baseline and focal length), the depth Z of a point can be computed using the relation:

$$Z = \frac{f \cdot B}{d}$$

     where:

     - f is the focal length,
     - B is the baseline (distance between the two camera centers),
     - d is the disparity (difference in x-coordinates of the corresponding points in the rectified images).
   - **Triangulation Methods:** For more general 3D reconstruction, you can use triangulation. This involves using the projection matrices (obtained during calibration/rectification) to back-project the corresponding image points into 3D space. OpenCV provides the `cv::triangulatePoints` function for this purpose, which uses linear methods (often based on SVD) to compute the 3D point that best fits the two projection rays.

A Simple Example Using OpenCV (Python)

Here's a rough outline of what the process might look like:

```python
import cv2
import numpy as np

# Assume you have already obtained the following through calibration:
# cameraMatrix1, distCoeffs1 for the left camera,
# cameraMatrix2, distCoeffs2 for the right camera,
# R (rotation) and T (translation) between the cameras.

# Image sizes (width, height)
image_size = (width, height)

# Compute rectification transforms
R1, R2, P1, P2, Q, _, _ = cv2.stereoRectify(cameraMatrix1, distCoeffs1,
                                             cameraMatrix2, distCoeffs2,
                                             image_size, R, T)
```

```
# Compute the rectification maps for both cameras
map1x, map1y = cv2.initUndistortRectifyMap(cameraMatrix1, distCoeffs1, R1, P1, image_size,
cv2.CV_32FC1)
map2x, map2y = cv2.initUndistortRectifyMap(cameraMatrix2, distCoeffs2, R2, P2, image_size,
cv2.CV_32FC1)

# Load your stereo images
img_left = cv2.imread('left_image.png')
img_right = cv2.imread('right_image.png')

# Rectify the images
rect_left = cv2.remap(img_left, map1x, map1y, cv2.INTER_LINEAR)
rect_right = cv2.remap(img_right, map2x, map2y, cv2.INTER_LINEAR)

# Now, perform stereo matching to compute the disparity map.
# One common approach is using StereoSGBM:
min_disp = 0
num_disp = 16*6  # must be divisible by 16
stereo = cv2.StereoSGBM_create(minDisparity=min_disp,
                               numDisparities=num_disp,
                               blockSize=7,
                               P1=8*3*7**2,
                               P2=32*3*7**2,
                               disp12MaxDiff=1,
                               uniquenessRatio=10,
                               speckleWindowSize=100,
                               speckleRange=32)
disparity = stereo.compute(rect_left, rect_right).astype(np.float32) / 16.0

# Once you have the disparity, you can compute depth.
# For a given pixel, the depth is:
# depth = (focal_length * baseline) / disparity
# If you want a full 3D reconstruction, you can reproject the disparity image to 3D:
points_3D = cv2.reprojectImageTo3D(disparity, Q)

# Alternatively, if you have specific corresponding points (pt_left and pt_right), you can use:
points_left = np.array([[pt_left_x], [pt_left_y]], dtype=float)
points_right = np.array([[pt_right_x], [pt_right_y]], dtype=float)
# Use triangulation with projection matrices P1 and P2
point_4d_hom = cv2.triangulatePoints(P1, P2, points_left, points_right)
point_3d = point_4d_hom / point_4d_hom[3]  # convert from homogeneous to 3D
```

Key Considerations

- **Calibration Accuracy:** The quality of your distance estimation depends heavily on accurate calibration. Errors in the intrinsic/extrinsic parameters will propagate into the 3D reconstruction.
- **Matching Quality:** Good rectification makes matching easier, but the matching algorithm still needs to be robust, especially in textureless regions or areas with repetitive patterns.

- **Baseline and Disparity:** A larger baseline generally improves depth resolution for nearby objects, but too large a baseline can introduce occlusions. Also, very small disparities (for far-away objects) can be noisy.

Summary

Yes, after stereo rectifying your image pair (which aligns the images for easier correspondence matching), you can compute disparities and use triangulation (or the simple depth formula) to estimate the distance to objects in your scene. This process is widely used in stereo vision applications, from robotics to 3D reconstruction.

If you need more details or have a specific implementation question, feel free to ask!

Frame Size 1024 x 1024
Spectrum visible
quantization 8 bit grayscale
camera height 325 mm
camera cant angle (vertical) 11 degree towards ground
camera cant angle (horizontal) 0 degree
inter camera distance 240 mm
focal length 21.6 mm
detector size 15 micro x 15 micro
FOV 39 degree x 39 degree
Image size on sensor 1.5 mm x 8.5 mm

Below is an outline of how you can use your camera's parameters in a stereo setup to perform rectification and distance estimation. In a nutshell, you need to:

1. **Compute the Intrinsic Parameters**
2. **Incorporate the Extrinsic (Pose) Information**
3. **Rectify the Stereo Pair**
4. **Compute Disparity and then Depth**

I'll go through each step while showing how your given parameters come into play.

---

1. Compute the Intrinsic Parameters

Your camera parameters include:

- **Focal length:** 21.6 mm
- **Detector (pixel) size:** 15 µm × 15 µm
- **Image resolution:** 1024 × 1024 pixels

A key step is to convert the focal length into pixel units. Since 15 µm = 0.015 mm, the focal length in pixels is

$$f_{\text{px}} = \frac{21.6\,\text{mm}}{0.015\,\text{mm/px}} = 1440\,\text{px}.$$

If you assume that the principal point is at the center of the image (which is typical unless you have reason to choose otherwise), then

$$c_x = c_y = \frac{1024}{2} = 512\,\text{px}.$$

Thus, the intrinsic (camera) matrix K becomes:

$$K = \begin{bmatrix} 1440 & 0 & 512 \\ 0 & 1440 & 512 \\ 0 & 0 & 1 \end{bmatrix}.$$

Note: You mentioned an "Image size on sensor" of 1.5 mm × 8.5 mm, which is not consistent with a 1024 px array at 15 µm/pixel (which would span roughly 15.36 mm on a side). In practice you must be sure that your effective sensor size and pixel pitch are consistent. (It may be that only a sub-region of the full sensor is used.)

---

2. Incorporate the Extrinsic (Pose) Information

Your extrinsic parameters include:

- **Camera height:** 325 mm
- **Vertical cant (tilt):** 11° downward
- **Horizontal cant:** 0°
- **Inter - camera distance (baseline):** 240 mm

If you set up your coordinate system with the left camera at the origin, you would have:

- The left camera's rotation R_{left} includes a tilt of 11° about the horizontal axis (often the x-axis). For a rotation about the x-axis:

$$R_{\text{tilt}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos 11^\circ & -\sin 11^\circ \\ 0 & \sin 11^\circ & \cos 11^\circ \end{bmatrix}.$$

- The right camera is translated by 240 mm relative to the left camera. In a simple case (if both cameras are mounted with the same tilt) you can assume

$$t = \begin{bmatrix} 240 \\ 0 \\ 0 \end{bmatrix} \text{ mm.}$$

Also, the camera height (325 mm) tells you where the cameras are relative to the ground. If you wish to compute the distance to objects on the ground (or reproject points to the ground plane) you'll later use the camera's position and orientation to intersect the viewing rays with the plane at the known height.

---

3. Stereo Rectification

Before doing any matching, you must "rectify" the images so that corresponding points lie on the same horizontal line. With the intrinsic parameters K and the extrinsic parameters (rotation and translation between the two cameras), you can use a function like OpenCV's

- `cv::stereoRectify` (C++/Python)

to compute the rectification transforms. Once you have those transforms, you can "warp" the images using functions such as `cv::initUndistortRectifyMap` and `cv::remap`.

Rectification will also handle the fact that the cameras are tilted (11° downward) so that the epipolar lines in the rectified images are horizontal.

---

4. Compute Disparity and then Depth

Once the images are rectified you use a stereo matching algorithm (for example, OpenCV's `StereoSGBM` or `StereoBM`) to compute the **disparity map**. For each corresponding point, the horizontal disparity d (in pixels) is measured.

The standard depth (or triangulation) equation is:

$$Z = \frac{f \cdot B}{d},$$

where

- Z is the depth (distance along the camera's view direction),
- f is the focal length in pixel units (here, 1440 px),
- B is the baseline (240 mm),
- d is the disparity in pixels.

Thus in your case:

$$Z\,(\text{in mm}) = \frac{1440 \times 240}{d} = \frac{345600}{d}.$$

For example, if the computed disparity at a pixel is 20 px then

$$Z = \frac{345600}{20} = 17280\,\text{mm} \quad (\approx 17.3\,\text{m}).$$

Important: This equation assumes that the images are properly rectified so that disparity is measured solely along the x-axis.

If you wish to compute not only the depth Z but the full 3D coordinates of points, you can use a triangulation function (e.g. OpenCV's `cv::triangulatePoints`) that uses the projection matrices for the two cameras (which are built from your intrinsic and extrinsic parameters).

Bringing It All Together

A typical pipeline might look like this (in pseudocode/Python with OpenCV):

```python
import cv2
import numpy as np

# --- Intrinsic parameters ---
f_px = 1440  # computed from 21.6 mm / 0.015 mm/px
K = np.array([[f_px, 0, 512],
              [0, f_px, 512],
              [0, 0, 1]], dtype=np.float64)

# --- Assume zero distortion (or provide your distortion coefficients) ---
distCoeffs = np.zeros(5)

# --- Extrinsic parameters ---
# Rotation for a downward tilt of 11° about the x-axis:
theta = np.deg2rad(11)
R_tilt = np.array([[1,          0,          0],
                   [0, np.cos(theta), -np.sin(theta)],
```

```
                 [0, np.sin(theta),  np.cos(theta)]], dtype=np.float64)

# Left camera at the origin
R_left = R_tilt
t_left = np.array([[0], [0], [0]], dtype=np.float64)

# Right camera translated 240 mm along the x-axis (and same tilt)
R_right = R_tilt
t_right = np.array([[240], [0], [0]], dtype=np.float64)

# --- Image size ---
image_size = (1024, 1024)

# --- Stereo Rectification ---
R1, R2, P1, P2, Q, _, _ = cv2.stereoRectify(K, distCoeffs, K, distCoeffs,
                                            image_size, R_right @ R_left.T,
                                            t_right - t_left)

# Compute the rectification maps
map1x, map1y = cv2.initUndistortRectifyMap(K, distCoeffs, R1, P1, image_size, cv2.CV_32FC1)
map2x, map2y = cv2.initUndistortRectifyMap(K, distCoeffs, R2, P2, image_size, cv2.CV_32FC1)

# --- Load images (assumed grayscale) ---
img_left = cv2.imread('left_image.png', cv2.IMREAD_GRAYSCALE)
img_right = cv2.imread('right_image.png', cv2.IMREAD_GRAYSCALE)

# Rectify the images
rect_left = cv2.remap(img_left, map1x, map1y, cv2.INTER_LINEAR)
rect_right = cv2.remap(img_right, map2x, map2y, cv2.INTER_LINEAR)

# --- Stereo Matching ---
min_disp = 0
num_disp = 16 * 6  # must be divisible by 16
stereo = cv2.StereoSGBM_create(minDisparity=min_disp,
                               numDisparities=num_disp,
                               blockSize=7,
                               P1=8 * 3 * 7**2,
                               P2=32 * 3 * 7**2,
                               disp12MaxDiff=1,
                               uniquenessRatio=10,
                               speckleWindowSize=100,
                               speckleRange=32)
disparity = stereo.compute(rect_left, rect_right).astype(np.float32) / 16.0

# --- Depth Estimation ---
# For each pixel with disparity d:
# Z (in mm) = (1440 * 240) / d
depth_map = np.divide(345600, disparity, out=np.zeros_like(disparity), where=disparity > 0)

# Optionally, reproject to 3D using Q (which was computed during stereoRectify)
points_3D = cv2.reprojectImageTo3D(disparity, Q)
```

Final Remarks

- **Camera Tilt and Ground Plane:**
  Your cameras are mounted 325 mm above the ground and are tilted 11° downward. If you are estimating distances to objects on the ground, you can take the computed 3D points (or even work directly with the pixel coordinates and the known tilt) to compute where the viewing ray intersects the ground plane. This extra step uses the camera's extrinsic parameters (height and tilt) to "lift" the simple Z depth into a full world coordinate (e.g. computing the X, Y location on the ground).
- **Consistency of Parameters:**
  Ensure that all units are consistent. In our example, all distances are in millimeters and the focal length is converted into pixels using the pixel size. Also, verify that the sensor (or active imaging area) dimensions are consistent with your pixel count and pixel size.

By using your provided parameters in this way, you can fully calibrate your stereo rig, rectify the images, compute a disparity map, and then use triangulation (or the simple depth formula) to estimate the distance to objects in your scene.

https://github.com/bvnayak/CAHVOR_camera_model
Is this relevant?

The CAHVOR model is a more sophisticated camera model than the basic pinhole (or CAHV) model because it can account for more complex lens distortions. In your stereo setup, if your calibration data or imaging system shows significant non-linearities or distortions that aren't well captured by a simple pinhole model, then converting your photogrammetric parameters into a CAHVOR model (as implemented in that repository) can indeed be relevant.

A few points to consider:

- **Enhanced Distortion Modeling:**
  The CAHVOR model extends the CAHV model by including additional parameters to model lens distortions more accurately. This is particularly useful in applications such as planetary imaging or when using cameras with wide fields of view or significant off-axis distortion.
- **Calibration and Rectification:**
  If your calibration process shows that a simple pinhole model does not fully capture the behavior of your optical system, using a CAHVOR model can improve the accuracy of the

intrinsic parameters. This, in turn, can lead to better stereo rectification and more reliable depth estimation during triangulation.

- **Conversion Utility:**
  The GitHub repository you referenced (bvnayak/CAHVOR_camera_model) provides Python tools to convert standard photogrammetric parameters into the CAHVOR format and vice versa. If your current calibration outputs or required inputs are in a photogrammetric form, this tool can be very useful in bridging that gap and ensuring that your camera model incorporates the necessary distortion parameters.

In summary, if your stereo imaging system needs to account for complex lens distortions or if your calibration process outputs photogrammetric parameters that you'd like to refine with the CAHVOR model, then yes—this repository is relevant to your project.

There are 3 options with me:
1. Find sift features and match them. Calculate disparity along x axis and generate depth map.
2. Use stereo rectify, stereo matching and then triangulation using intrinsic and extrinsic matrix with opencv
3. Convert intrinsic matrix and extrinsic matrix into cahvor and solve some equations.
Option 1 gives poor results. I have not tried option 2 and 3.
I have already shared my camera specs earlier. Let's discuss.
Let's discuss this.

Let's break down the three options and discuss their merits and challenges:

Option 1: SIFT Feature Matching

- **Approach:**
  Detect SIFT keypoints in both images, match them, compute the horizontal disparities from the matched keypoints, and then generate a (typically sparse) depth map.
- **Pros:**
  - Can work well when you have plenty of textured regions.
  - SIFT is robust to scale and rotation changes, which is beneficial if the images aren't perfectly aligned.

- **Cons:**
  - **Sparse Results:** SIFT yields only a set of keypoints rather than a full dense correspondence, so the resulting depth map may be patchy or incomplete.
  - **Poor Accuracy:** When compared to methods that exploit the full image information (dense stereo matching), the results can be noisy, especially in textureless regions.
  - **Disparity Assumption:** Calculating disparity purely along the x-axis works best if your epipolar lines are perfectly horizontal—which may not be true without proper rectification.

Since you've already found that option 1 gives poor results, it's a sign that a denser and more structured approach may be necessary.

Option 2: Stereo Rectification, Dense Stereo Matching, and Triangulation Using Intrinsic/Extrinsic Matrices (OpenCV)

- **Approach:**
  1. **Calibrate the Stereo Setup:** Use your known camera specs (focal length, pixel size, camera height, tilt, baseline, etc.) to determine intrinsic and extrinsic parameters.
  2. **Rectification:** Apply stereo rectification (using functions like OpenCV's `cv::stereoRectify`) to warp both images so that the epipolar lines are aligned horizontally.
  3. **Dense Matching:** Use algorithms like StereoBM or StereoSGBM to compute a dense disparity map from the rectified images.
  4. **Triangulation:** Convert the disparity map to a depth map using the triangulation formula $Z = \frac{f \cdot B}{d}$ (with f in pixels and B the baseline). Alternatively, use functions like `cv::triangulatePoints` to obtain full 3D coordinates.
- **Pros:**
  - **Dense Depth Map:** Provides a per-pixel depth estimation rather than a sparse one.
  - **Mature Pipeline:** This approach is well-established, with many examples and tools available in OpenCV.
  - **Epipolar Constraint:** Rectification simplifies the correspondence search, often leading to better matching performance.
- **Cons:**
  - **Calibration Sensitivity:** The accuracy of depth estimation depends heavily on the precision of your calibration (intrinsic and extrinsic parameters).

- **Rectification Quality:** If your cameras are not perfectly aligned or if the model assumptions are violated (e.g., significant lens distortion), rectification might not be ideal.

For many stereo vision applications, option 2 is the recommended baseline. Given your camera parameters (e.g., a high focal length in pixel units, known tilt, and a fixed baseline), you should be able to set up a robust rectification and matching pipeline.

Option 3: Converting Intrinsic/Extrinsic Parameters into the CAHVOR Model

- **Approach:**
  Convert your standard photogrammetric parameters (or pinhole camera model parameters) into the CAHVOR representation. The CAHVOR model extends the basic pinhole model (or CAHV model) by adding additional parameters that account for more complex lens distortions. Once converted, you would solve the CAHVOR equations for ray intersection and 3D reconstruction.
- **Pros:**
  - **Enhanced Distortion Modeling:** If your optical system exhibits significant non-linear distortions (especially off-axis), the CAHVOR model can capture these effects more accurately than the pinhole model.
  - **Potential for Higher Accuracy:** For applications where distortion must be precisely modeled (for example, in planetary imaging or when using wide-angle lenses), this approach might yield better results.
- **Cons:**
  - **Complexity:** Converting to and working with CAHVOR is mathematically more involved and may require specialized calibration and computation routines.
  - **Implementation Overhead:** You'll likely need to rely on or develop custom code (or use a library like the one you referenced on GitHub) to handle the conversion and solve the CAHVOR equations.
  - **Benefit Depends on Distortion:** If your camera's distortions are already modest or can be corrected effectively via rectification, the additional complexity might not translate into significant accuracy gains.

Option 3 becomes most attractive if your camera lens introduces significant distortions that the standard pinhole model (used in option 2) cannot handle well. If your calibration data suggests notable deviation from the ideal pinhole behavior, then exploring CAHVOR might be worthwhile.

Discussion and Recommendation

- **If your calibration is robust and lens distortions are manageable:**
  Option 2 is likely your best starting point. Stereo rectification followed by dense stereo matching is a proven pipeline. Given your camera specs (such as a high effective focal length in pixels, known tilt angles, and a fixed baseline), you can expect decent depth reconstruction performance—provided that the calibration is accurate.
- **If you suspect significant lens distortions that hurt the rectification or matching process:**
  Then option 3 might be worth investigating. Using a CAHVOR model would allow you to incorporate those distortions explicitly into your computations. However, this comes at the cost of increased complexity. You'll need to assess whether the additional computational overhead and potential calibration challenges are justified by improved depth accuracy.
- **Practical Steps:**
  1. **Try Option 2 First:** Given that it's a widely adopted approach, test the full stereo pipeline in OpenCV. Validate the rectified images and see how the disparity maps look.
  2. **Evaluate the Results:** Check whether the computed depth maps are consistent and accurate enough for your application.
  3. **If Issues Persist:** If you observe systematic errors that hint at unmodeled lens distortion (or if calibration errors seem to be at fault), consider exploring option 3 and comparing the results.

In summary, starting with option 2 makes sense for most stereo vision systems because it leverages a mature and well-documented approach. Option 3 could be reserved for when you have evidence that the standard pinhole model isn't sufficient due to complex lens distortions.

If we have the camera matrix and projection matrices, bvnayak's tool from github can generate the cahvor values
But what next?

Once you've converted your standard photogrammetric parameters (or your camera matrix and projection matrices) into CAHVOR values using bvnayak's tool, you're essentially obtaining a more complete and accurate description of your camera's imaging geometry—including explicit distortion parameters. At that point, the next steps involve using these CAHVOR parameters in your 3D reconstruction pipeline. Here's how you might proceed:

---

1. Use CAHVOR for Accurate Image Projection and Undistortion

- **Correct for Distortion:**
  The CAHVOR model includes extra parameters (beyond the basic pinhole or CAHV model) that account for lens distortions. With the CAHVOR parameters in hand, you can transform your raw image coordinates to undistorted coordinates. This process is analogous to the undistortion step in standard stereo pipelines but is now tailored to your camera's specific distortion behavior.
- **Project Rays in 3D:**
  The CAHVOR model defines how a 3D point is projected onto the image plane. By inverting these equations, you can map any image point back to a ray in 3D space. In other words, for a given pixel coordinate, you can compute the direction vector (or "ray") in space that it represents, taking into account the distortion corrections.

---

2. Triangulation Using CAHVOR-Corrected Rays

- **Obtain Correspondences:**
  As with any stereo system, you first need to match corresponding points between your left and right images. You may use dense stereo matching techniques or feature-based matching (ensuring that you take advantage of the improved accuracy from the CAHVOR undistortion).
- **Compute 3D Intersections:**
  For each corresponding pair of image points, use the CAHVOR model to compute the corresponding 3D rays from each camera. Then, solve for the intersection (or, more precisely, the point of closest approach) between these two rays. This is the triangulation step. The improved modeling of lens distortion should yield more accurate ray directions, which in turn leads to better 3D point estimates.
- **Iterative or Closed-Form Solutions:**
  Depending on the formulation of the CAHVOR model, the intersection of the rays may require solving a set of non-linear equations. Some implementations use iterative optimization (for example, minimizing the reprojection error) to refine the 3D point estimate. Alternatively, if the model is linearized sufficiently, a closed-form solution might be available.

---

3. Integration into a Full Stereo Pipeline

- **Validate the CAHVOR Model:**
  Before committing to a full reconstruction, it's often useful to validate the CAHVOR model by projecting known 3D test points onto the image and comparing them with observed pixel coordinates. This helps ensure that the conversion from photogrammetric parameters is accurate.
- **Compare with Standard Pipelines:**
  You might want to run both the standard rectification–dense matching pipeline (option 2) and your CAHVOR-based triangulation (option 3) in parallel. This allows you to assess whether the CAHVOR model yields a noticeable improvement—especially in regions where distortion would otherwise cause errors.
- **Refine Calibration:**
  Sometimes the process of converting to a CAHVOR model can reveal calibration errors or inconsistencies. Use any discrepancies to refine your calibration data, which can further improve your stereo reconstruction accuracy.

Summary

In short, once you have your CAHVOR parameters:

1. **Apply the CAHVOR model to undistort your images** so that your pixel coordinates are more representative of the true geometry.
2. **Use the CAHVOR equations to back-project image points into 3D rays.**
3. **Triangulate these rays** (from both the left and right cameras) to determine the 3D positions of the corresponding points.
4. **Integrate these steps into your overall stereo pipeline** and compare the results to your previous methods to ensure that the added complexity is justified by improved accuracy.

This approach is particularly beneficial when lens distortions are non-negligible and the simple pinhole model doesn't suffice for accurate 3D reconstruction.

Can you show me the equations or steps we need to perform using the cahvor model

Below is a summary of the key equations and steps that one typically follows when using the CAHVOR model to project 3D points into the image—and, conversely, to back‐project image points into rays for triangulation. (For a more complete derivation, see .)

---

1. CAHVOR Model Overview

In the CAHVOR model the camera is described by six main parameter groups:

- **C:** Camera center (the 3D location of the perspective center).
- **A:** The unit "camera axis" (normal to the image plane).
- **H:** The horizontal vector; it encodes both the horizontal field - of - view (or focal length expressed in horizontal image units) and the horizontal offset of the principal point.
- **V:** The vertical vector; it similarly encodes the vertical field - of - view and the vertical principal point offset.
- **O:** The optical axis (which—if the image plane is not exactly perpendicular to the optical axis—can differ slightly from A).
- **R:** A triplet of radial distortion coefficients, often written as [r_0, r_1, r_2].

---

2. Forward Projection (3D Point to 2D Image)

(a) Compute the "Ideal" (Undistorted) Projection

Given a 3D point P and the camera center C, first form the vector from the camera to the point:

$$Y = P - C.$$

Then, if there were no lens distortion, the image coordinates (in some "intermediate" coordinate system) are given by:

$$i_0 = \frac{Y \cdot H}{Y \cdot A}, \qquad j_0 = \frac{Y \cdot V}{Y \cdot A}.$$

These equations are analogous to the collinearity (or central projection) equations:

$$i = \frac{(P - C) \cdot H}{(P - C) \cdot A}, \qquad j = \frac{(P - C) \cdot V}{(P - C) \cdot A}.$$

(See equations (2a)–(2b) in Di & Li, 2004.)

(b) Apply Radial Distortion Correction

Because the lens introduces radial distortion, the model uses an additional correction based on the optical axis O and the distortion coefficients $R = [r_0, r_1, r_2]$.

1. **Project onto the Optical Axis:**
   Compute the component of Y along the optical axis:
   $$x = Y \cdot O.$$
2. **Compute the Perpendicular Component:**
   The component perpendicular to O is:
   $$l = Y - x\,O.$$
3. **Determine the "Tilt" Angle:**
   The (small) angle α between Y and O is given by:
   $$t = \tan(\alpha) = \frac{\|l\|}{x}.$$
4. **Calculate the Correction Factor:**
   The radial distortion is modeled by a polynomial:
   $$m = r_0 + r_1\,t + r_2\,t^2.$$
5. **Compute the Distortion - Corrected Vector:**
   Apply the correction to "push" the ray onto its true (undistorted) direction:
   $$Y' = Y + m\,l.$$

(c) Final Projection

Finally, the corrected image coordinates are obtained by re-projecting Y':

$$i = \frac{Y' \cdot H}{Y' \cdot A}, \qquad j = \frac{Y' \cdot V}{Y' \cdot A}.$$

This yields the observed (distorted) image coordinates. (In some implementations the distortion correction may be applied as a warping of the already computed image; conceptually it is equivalent.)

---

3. Back–Projection and Triangulation

For stereo reconstruction you need to determine the 3D ray corresponding to an image point. The process is roughly the inverse of the above:

(a) Inverting the Projection

1. **Start with a Measured Image Point (i, j):**
   This point is (ideally) the result of the above forward model.

2. **Undo the Lens Distortion:**
   Use the inverse of the distortion function (which may require numerical solution) to recover the "ideal" image coordinates (i_0, j_0).

3. **Form the Ray in the Camera's Coordinate System:**
   The "ideal" back–projection is then given by:

$$d = A + \Delta i\, H_0 + \Delta j\, V_0\,,$$

   where

   - H_0 and V_0 are the unit directions corresponding to the horizontal and vertical axes (often obtained by normalizing the portions of H and V that are orthogonal to A), and
   - Δi and Δj are the differences between the ideal image coordinates and the principal point (which is embedded in the values of H and V).

4. **Transform the Ray into World Coordinates:**
   Using the extrinsic parameters (from calibration), rotate and translate the ray from the camera coordinate system into the world coordinate system.

(b) Triangulation

Given the back–projected rays from two (or more) cameras with known centers C_1 and C_2 and ray directions d_1 and d_2, the 3D point P is found by determining the point of closest approach between the two rays. This is typically solved by finding scalars λ_1 and λ_2 such that:

$$P_1 = C_1 + \lambda_1\, d_1\,, \qquad P_2 = C_2 + \lambda_2\, d_2\,,$$

and then finding P as the midpoint of the shortest segment connecting the rays. (There are closed–form solutions for this "linear triangulation" problem.)

---

4. Summary of the Process

1. **Calibration:**
   Determine the CAHVOR parameters (C, A, H, V, O, R) from calibration images. This step may involve converting from a photogrammetric model if necessary.

2. **Forward Projection (for simulation or rectification):**
   For any 3D point P, compute $Y = P - C$, apply the ideal projection using H, V, A, and then correct for radial distortion using O and R to obtain the final image coordinates.

3. **Back–Projection (for reconstruction):**
   For each observed image point, invert the process (including undistorting the coordinates) to recover the corresponding 3D ray in the camera coordinate system; then, using known extrinsics, compute the ray in the world frame.
4. **Triangulation:**
   For corresponding points in a stereo pair, intersect (or compute the closest approach of) the two rays to estimate the 3D position.

These are the fundamental steps and equations one uses with the CAHVOR model. In practice the details (for example, inverting the distortion function) may be implemented with iterative numerical methods, and many implementations (for instance, in OpenCV or custom photogrammetry pipelines) follow these principles to achieve more accurate 3D reconstruction.

This summary is based on the derivations and conversion techniques presented by Di and Li (2004) .