

Table of Contents

- [A Simple Example Using OpenCV \(Python\)](#)
- [Key Considerations](#)
- [Summary](#)
- [1. Compute the Intrinsic Parameters](#)
- [2. Incorporate the Extrinsic \(Pose\) Information](#)
- [3. Stereo Rectification](#)
- [4. Compute Disparity and then Depth](#)
- [Bringing It All Together](#)
- [Final Remarks](#)

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 \times 15 μm
- **Image resolution:** 1024 \times 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 \times 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.