Table of Contents

The CAHV camera model represents a camera using four 3D vectors:

- **C**: the camera center (or "eye"),
- **A**: the unit vector along the optical axis,
- **H**: a vector in the "horizontal" direction (often scaled by the focal length in the x direction),
- **V**: a vector in the "vertical" direction (often scaled by the focal length in the y direction).

A common assumption is that **A** is a unit vector and that **H** and **V** are chosen so that

$$H \cdot A = 0 \quad \text{and} \quad V \cdot A = 0.$$

(If this is not exactly true, slight modifications are needed, but in many applications it is.)

With these parameters the projection of a 3D point

$$P \in \mathbb{R}^3$$

onto the image is given by the two equations

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

Here, (u, v) are the image coordinates (often with the origin at the principal point).

The 3D 2D Projection

The formulas above show how to go from a 3D point P to its image coordinates. In words:

1. Compute the vector from the camera center to the point: $D = P - C$.
2. Find the component of D along the optical axis A; that is, compute $d = D \cdot A$.
3. The "horizontal" coordinate is then given by the ratio of the projection of D onto H to d, and similarly for the "vertical" coordinate with V.

The Inverse Mapping: 2D 3D (Back Projection)

Of course, given an image point (u, v) the best one can do is to "back project" it to a ray in space (since depth is lost in the projection). In the CAHV model the back–projection is given by the set of points

$$\{\, P : P = C + \lambda\, d(u, v),\ \lambda > 0 \,\},$$

where the direction $d(u, v)$ is chosen so that if a point P on that ray is inserted into the projection formulas, it gives the image point (u, v).

A very natural choice is to "normalize" the ray by setting the component along A equal to 1. That is, we look for a vector d of the form

$$d = A + \alpha\, H + \beta\, V$$

such that

$$d \cdot A = 1, \qquad d \cdot H = u, \qquad d \cdot V = v.$$

If we assume

$$H \cdot A = 0 \quad \text{and} \quad V \cdot A = 0,$$

and also that A is a unit vector (so that $A \cdot A = 1$), then

$$d \cdot A = A \cdot A = 1,$$

automatically. (In other words, we "force" the ray's A–component to be 1.) Then, since

$$d \cdot H = \alpha\,(H \cdot H) = u, \qquad \text{and} \qquad d \cdot V = \beta\,(V \cdot V) = v,$$

one obtains

$$\alpha = \frac{u}{H \cdot H}, \qquad \beta = \frac{v}{V \cdot V}\ .$$

Thus, one acceptable formula for the ray direction is

$$d(u, v) = A + \frac{u}{H \cdot H}\ H + \frac{v}{V \cdot V}\ V,$$

and the full back–projection is then

$$P = C + \lambda \left(A + \frac{u}{H \cdot H}\ H + \frac{v}{V \cdot V}\ V \right), \qquad \lambda > 0\,.$$

Stereo Vision

In a stereo system, you have two cameras (say, with parameters $C_1\,, A_1\,, H_1\,, V_1$ and $C_2\,, A_2\,, H_2\,, V_2\,$). Given a pair of corresponding image points $(u_1\,, v_1\,)$ and $(u_2\,, v_2\,)$, you can back–project them to obtain two rays:

$$P_1 = C_1 + \lambda_1 \left(A_1 + \frac{u_1}{H_1 \cdot H_1} H_1 + \frac{v_1}{V_1 \cdot V_1} V_1 \right),$$

$$P_2 = C_2 + \lambda_2 \left(A_2 + \frac{u_2}{H_2 \cdot H_2} H_2 + \frac{v_2}{V_2 \cdot V_2} V_2 \right),$$

and then triangulate (usually by finding the closest point between these two rays) to obtain the 3D location of the corresponding scene point.

A Note on Conventions

Sometimes the vectors H and V are chosen so that they are the "scaled" unit vectors in the horizontal and vertical directions. For example, if one writes

$$H = f_x \, \hat{h}, \quad V = f_y \, \hat{v},$$

with f_x and f_y the effective focal lengths and $\hat{h}$ and $\hat{v}$ unit vectors, then the projection formulas become

$$u = \frac{(P - C) \cdot H}{(P - C) \cdot A}, \quad v = \frac{(P - C) \cdot V}{(P - C) \cdot A},$$

and the back–projection ray is sometimes written as

$$P = C + \lambda \left(A + \frac{u}{f_x} \, \hat{h} + \frac{v}{f_y} \, \hat{v} \right), \qquad \lambda > 0 \, .$$

Both forms are equivalent up to the scaling of H and V.

Summary Answer

Thus, one acceptable answer is:

- **3D 2D (Projection):**

$$u = \frac{(P - C) \cdot H}{(P - C) \cdot A}, \quad v = \frac{(P - C) \cdot V}{(P - C) \cdot A} \, .$$

- **2D 3D (Back–Projection):**

$$P = C + \lambda \left(A + \frac{u}{H \cdot H} H + \frac{v}{V \cdot V} V \right), \quad \lambda > 0 \, .$$

In stereo vision, you use these formulas for each camera and then triangulate the two rays to recover the 3D point.

Any answer that expresses the back–projection in these terms is correct.

Below is a worked through example that shows how you might use the 2D 3D back–projection formula for each camera and then triangulate to recover the 3D point. In this example, we assume a simple stereo rig with two identical cameras that are laterally displaced (i.e. their optical axes are parallel). For simplicity, we use a CAHV formulation that mimics a pinhole camera with:

- **C**: the camera center,
- **A**: the unit optical axis,
- **H**: a horizontal vector whose length is the focal length f (so that $H \cdot H = f^2$),
- **V**: a vertical vector whose length is also f.

We also assume that

$$H \cdot A = 0, \quad V \cdot A = 0, \quad \text{and} \quad A \cdot A = 1 \, .$$

Setup of the Stereo Cameras

1. **Camera 1 (Left Camera):**
   - **Center:** $C_1 = (0, 0, 0)$
   - **Optical axis:** $A_1 = (0, 0, 1)$
     (pointing along the positive z–axis)
   - **Horizontal vector:** $H_1 = (f, 0, 0) = (1000, 0, 0)$
     (so that $H_1 \cdot H_1 = 1000^2 = 10^6$)
   - **Vertical vector:** $V_1 = (0, f, 0) = (0, 1000, 0)$
     (so that $V_1 \cdot V_1 = 10^6$)
2. **Camera 2 (Right Camera):**
   - **Center:** $C_2 = (B, 0, 0) = (100, 0, 0)$
     (i.e. the right camera is shifted by $B = 100$ units along the x–axis)
   - **Optical axis:** $A_2 = (0, 0, 1)$
     (same orientation as camera 1)
   - **Horizontal vector:** $H_2 = (f, 0, 0) = (1000, 0, 0)$
   - **Vertical vector:** $V_2 = (0, f, 0) = (0, 1000, 0)$

For both cameras, the 2D 3D back–projection (or “ray”) for an image point (u, v) is given by:

$$P = C + \lambda \left(A + \frac{u}{H \cdot H} \, H + \frac{v}{V \cdot V} \, V \right), \quad \lambda > 0 \, .$$

Because $H \cdot H = V \cdot V = 10^6$, the expression inside the parentheses becomes

$$A + \frac{u}{10^6}\, H + \frac{v}{10^6}\, V.$$

Notice that with $H = (1000, 0, 0)$ and $V = (0, 1000, 0)$, we have:

$$\frac{u}{10^6}\, H = \left(\frac{u}{1000}, 0, 0\right), \quad \frac{v}{10^6}\, V = \left(0, \frac{v}{1000}, 0\right).$$

Thus the direction for the ray becomes:

$$d(u, v) = \left(\frac{u}{1000}, \frac{v}{1000}, 1\right).$$

This is exactly what we expect from a pinhole camera, where the image coordinates (if the principal point is at the origin) are scaled by the focal length:

$$u = f\frac{X}{Z} \quad \Rightarrow \quad \frac{X}{Z} = \frac{u}{f}\,.$$

Choosing Image Coordinates for a Scene Point

Let's say a scene point $P = (X, Y, Z)$ is observed by both cameras. Suppose that in the image of **Camera 1** the point appears at:

$$(u_1\,, v_1\,) = (50,\, 0)\,.$$

And in the image of **Camera 2** it appears at:

$$(u_2\,, v_2\,) = (-50,\, 0)\,.$$

These image coordinates are chosen so that the corresponding 3D point will be in front of both cameras. (In a real system, the coordinates would be determined by a feature matching algorithm.)

Step 1. Back–Project Each 2D Point to a 3D Ray

For Camera 1:

Using the formula, the back–projected ray is:

$$P_1\,(\lambda_1\,) = C_1 + \lambda_1 \left(A_1 + \frac{u_1}{10^6}\, H_1 + \frac{v_1}{10^6}\, V_1\right).$$

Plug in the numbers:

- $C_1 = (0,0,0)$
- $A_1 = (0,0,1)$
- $u_1 = 50,\ v_1 = 0$
- $\frac{50}{10^6}\ H_1 = \frac{50}{10^6}\ (1000,0,0) = \left(\frac{50}{1000},0,0\right) = (0.05,\ 0,\ 0)$
- $\frac{0}{10^6}\ V_1 = (0,0,0)$

Thus,

$$P_1(\lambda_1) = (0,0,0) + \lambda_1\ \big((0.05,0,0) + (0,0,1)\big) = \lambda_1\ (0.05,0,1).$$

In component form:

$$x_1 = 0.05\,\lambda_1, \quad y_1 = 0, \quad z_1 = \lambda_1\ .$$

For Camera 2:

Similarly, the back–projection ray is:

$$P_2(\lambda_2) = C_2 + \lambda_2\ \left(A_2 + \frac{u_2}{10^6}\ H_2 + \frac{v_2}{10^6}\ V_2\right).$$

Plug in the numbers:

- $C_2 = (100,0,0)$
- $A_2 = (0,0,1)$
- $u_2 = -50,\ v_2 = 0$
- $\frac{-50}{10^6}\ H_2 = \frac{-50}{10^6}\ (1000,0,0) = \left(\frac{-50}{1000},0,0\right) = (-0.05,\ 0,\ 0)$
- $\frac{0}{10^6}\ V_2 = (0,0,0)$

Thus,

$$P_2(\lambda_2) = (100,0,0) + \lambda_2\ \big((-0.05,0,0) + (0,0,1)\big) = (100,0,0) + \lambda_2\ (-0.05,0,1).$$

In component form:

$$x_2 = 100 - 0.05\,\lambda_2, \quad y_2 = 0, \quad z_2 = \lambda_2\ .$$

Step 2. Triangulate to Find the 3D Point

The goal in triangulation is to find parameters λ_1 and λ_2 such that the two rays $P_1(\lambda_1)$ and $P_2(\lambda_2)$ meet (or, in practice, are as close as possible). In our idealized example the rays intersect exactly.

We set:

$$P_1(\lambda_1) = P_2(\lambda_2).$$

In components, equate the x and z coordinates (the y coordinate is trivially 0 in both):

1. **z–coordinate:**

   $$\lambda_1 = \lambda_2 \,.$$

   Let $\lambda_1 = \lambda_2 = \lambda$.

2. **x–coordinate:**

   $$0.05\,\lambda = 100 - 0.05\,\lambda\,.$$

   Solve for λ:

   $$0.05\,\lambda + 0.05\,\lambda = 100 \quad \Rightarrow \quad 0.1\,\lambda = 100\,,$$

   $$\lambda = \frac{100}{0.1} = 1000\,.$$

Now substitute $\lambda = 1000$ back into one of the ray equations:

- **From Camera 1:**

  $$P_1(1000) = 1000\,(0.05, 0, 1) = (50,\ 0,\ 1000).$$

Thus, the triangulated 3D point is:

$$P = (50,\ 0,\ 1000)\,.$$

Interpreting the Result

- The z–coordinate 1000 represents the depth of the point (i.e. 1000 units in front of the cameras).
- The x–coordinate 50 is consistent with the fact that camera 1 saw the point at a horizontal offset that, when scaled by the depth and focal length, gives $x = 50$. Likewise, camera 2's measurement (after accounting for its 100 unit offset) agrees with this result.
- In a real stereo system, noise or calibration errors mean the two rays may not intersect exactly; in that case one typically finds the point that minimizes the distance between the rays (for example, by solving a least–squares problem). Here, however, our simple example yields an exact intersection.

Summary of the Process

1. **Back–Projection:**
   For each camera, use the formula
   $$P = C + \lambda \left(A + \frac{u}{H \cdot H}\, H + \frac{v}{V \cdot V}\, V \right)$$
   to convert the image point into a 3D ray.
2. **Express the Rays in Parametric Form:**
   Write the rays from Camera 1 and Camera 2 in parametric form:
   - $P_1(\lambda_1) = \lambda_1\ (0.05,\ 0,\ 1)$
   - $P_2(\lambda_2) = (100, 0, 0) + \lambda_2\ (-0.05,\ 0,\ 1)$
3. **Solve for the Parameters (λ_1 and λ_2):**
   Set the corresponding coordinates equal (here, x and z are sufficient) and solve for λ. In our example, we found $\lambda = 1000$.
4. **Recover the 3D Point:**
   Substitute λ back into one of the ray equations to obtain the 3D coordinates $P = (50,\ 0,\ 1000)$.

This example demonstrates how the back–projection formulas are practically used to compute the rays from 2D image points, and then how stereo triangulation recovers the 3D position by finding the intersection of these rays.

Yes, essentially the CAHV model is an alternative parameterization of the standard pinhole camera model that uses intrinsic and extrinsic matrices. Both representations describe how 3D points are mapped to 2D image coordinates, but they organize the information differently. Here's a breakdown of the correspondence:

1. **Intrinsic vs. CAHV's H and V:**
   - In the standard model, the **intrinsic matrix** (often denoted as K) encodes the focal length(s), skew, and principal point of the camera.
   - In the CAHV model, the vectors **H** (horizontal) and **V** (vertical) encode the scaling and orientation of the image axes. Their magnitudes (e.g., $H \cdot H$ and $V \cdot V$) are directly related to the focal lengths along the x and y directions, and they also incorporate any skew or non-square pixel effects if necessary.
2. **Extrinsic vs. CAHV's C and A:**

- In the standard approach, the **extrinsic parameters** are given by the rotation matrix R and the translation vector t that position the camera in the world.
- In the CAHV model, **C** represents the camera center (analogous to the translation) and **A** is a unit vector representing the optical axis (which is related to the rotation R). The vector A gives the direction in space where the camera is "looking."

3. **Mapping Equivalence:**
   - With the intrinsic-extrinsic formulation, a 3D point P in world coordinates is typically projected using:
     $$p \sim K\,[\,R \mid t\,]\,P,$$
     where p is in homogeneous image coordinates.
   - In the CAHV formulation, the projection is given by:
     $$u = \frac{(P - C)\cdot H}{(P - C)\cdot A}, \quad v = \frac{(P - C)\cdot V}{(P - C)\cdot A},$$
     which encodes the same geometry.
4. **Why Use CAHV?**
   - The CAHV model can be more intuitive in some contexts—especially in remote sensing or planetary imaging—because it directly ties the projection to geometrically meaningful vectors.
   - It often simplifies certain computations (like back–projection) and can be more robust in certain calibration contexts.
5. **In Stereo Vision:**
   - When triangulating, you compute the back–projection ray for each camera from its image point. Whether you derive this ray using a standard K, R, and t decomposition or via the CAHV model, you're essentially recovering the same ray in space.
   - The CAHV representation just expresses that ray in terms of the camera center **C** and the directional combination $A + \frac{u}{H\cdot H}\,H + \frac{v}{V\cdot V}\,V$.

Summary

The CAHV model isn't a fundamentally different approach; it's just a different way to package the camera's intrinsic and extrinsic parameters into four vectors. Both methods are mathematically equivalent in that they capture the same 3D-to-2D projection, and in stereo vision both are used to compute rays that are then triangulated to recover 3D scene geometry.

Yes. The CAHV model is simply an alternative way to package a camera's intrinsic (like focal length and pixel scale) and extrinsic (position and orientation) parameters. In your case, you have specific numbers that let you “fill in” the CAHV parameters for each camera. Let's walk through how you might do that.

---

1. Converting Your Specifications

Focal Length and Detector Pitch

- **Focal length:** 21.6 mm
- **Detector (pixel) size:** 15 µm × 15 µm

To work in consistent units (typically pixels when you're dealing with image coordinates), you can convert the focal length from millimeters to pixels. Since

$$1 \text{ mm} = 1000 \ \mu\text{m},$$

the focal length in micrometers is

$$21.6 \text{ mm} \times 1000 = 21600 \ \mu\text{m}.$$

Then the focal length in pixels is

$$f_{\text{pixels}} = \frac{21600 \ \mu\text{m}}{15 \ \mu\text{m}/\ \text{pixel}} = 1440 \text{ pixels}.$$

In a CAHV model, the **H** and **V** vectors encode the focal lengths along the horizontal and vertical directions, respectively. (They also may account for any non-square pixels or skew—but here your detector is square and we assume no skew.)

---

2. Setting Up the CAHV Parameters

Recall that in the CAHV model you have four vectors:

- **C:** the camera center.
- **A:** a unit vector along the optical axis.
- **H:** a vector in the horizontal direction. Its length (or more precisely, $H \cdot H$) is related to the focal length (in pixels, if your image coordinates are in pixels).

- **V:** a vector in the vertical direction, similarly scaled.

Intrinsic Parameters (H and V)

In a “standard” (untilted) pinhole camera with the principal point at the image center and no skew, you might choose in the camera's own coordinate system:

- $H = (f_{\text{pixels}}, 0, 0) = (1440, 0, 0)$
- $V = (0, f_{\text{pixels}}, 0) = (0, 1440, 0)$
- And if you were not tilting the camera, the optical axis would be $A = (0, 0, 1)$.

Notice that

$$H \cdot H = 1440^2, \quad V \cdot V = 1440^2.$$

Extrinsic Parameters (C and A) with Tilt

Your cameras are mounted on a rover with the following additional information:

- **Baseline:** 24 cm (i.e. 240 mm separation between the two cameras).
- **Vertical cant (tilt):** Both cameras are canted downward by 11°.
- **No horizontal cant.**

What does a downward tilt of 11° mean? It means that the optical axis is rotated about the horizontal axis by 11°. A rotation about the horizontal axis (which we align with the **H** direction) changes the vertical and forward components of the optical axis.

A standard rotation about the x–axis (which we take as the horizontal direction) by an angle θ is given by:

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix}.$$

If we start with the untilted optical axis $A = (0, 0, 1)$ (i.e. pointing straight “forward”), then after a rotation by 11° downward the new optical axis becomes:

$$A_{\text{tilted}} = R_x(11^\circ)\,(0, 0, 1) = \big(0,\ -\sin(11^\circ),\ \cos(11^\circ)\big).$$

Using approximate values:

$$\sin(11^\circ) \approx 0.1908, \quad \cos(11^\circ) \approx 0.9816,$$

so

$$A_{\text{tilted}} \approx (0,\ -0.1908,\ 0.9816).$$

Since the rotation is about the x–axis, the horizontal vector H remains unchanged by the tilt:

$$H_{\text{tilted}} = (1440,\, 0,\, 0).$$

However, the vertical vector V will rotate. Starting from $V = (0, 1440, 0)$, we have

$$V_{\text{tilted}} = R_x\left(11^\circ\right)(0, 1440, 0) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(11^\circ) & -\sin(11^\circ) \\ 0 & \sin(11^\circ) & \cos(11^\circ) \end{pmatrix} (0, 1440, 0).$$

Performing the multiplication gives:

$$V_{\text{tilted}} = \left(0,\ 1440\cos(11^\circ),\ 1440\sin(11^\circ)\right)$$

or numerically:

$$V_{\text{tilted}} \approx \left(0,\ 1440 \times 0.9816,\ 1440 \times 0.1908\right) \approx (0,\ 1413.9,\ 274.8).$$

Camera Centers (C)

For a stereo rig with a 240 mm baseline you must decide on a coordinate system for the two camera centers. One common choice is to place them symmetrically about the rover's center. For example, you might choose:

- **Camera 1 (left camera):**
  $C_1 = (-120\ \text{mm},\, 0,\, 0)$
- **Camera 2 (right camera):**
  $C_2 = (120\ \text{mm},\, 0,\, 0)$

(Alternatively, you could set one at the origin and the other at $(240\ \text{mm},\, 0,\, 0)$. The important thing is that the distance between the two is 240 mm.)

---

3. Assembling the CAHV Models

For **each camera**, the CAHV model in world coordinates would then be:

- **Camera Center (C):**
  - $C_1 = (-120,\, 0,\, 0)$ for Camera 1
  - $C_2 = (120,\, 0,\, 0)$ for Camera 2
- **Optical Axis (A):** (same for both, since both are tilted the same way)

$$A \approx (0,\ -0.1908,\ 0.9816)$$

- **Horizontal Vector (H):** (remains unchanged by the tilt)

$$H = (1440,\, 0,\, 0)$$

- **Vertical Vector (V):** (after applying the tilt)

$$V \approx (0,\, 1413.9,\, 274.8)$$

These four vectors completely describe each camera in the CAHV model. When you use the projection formulas

$$u = \frac{(P - C) \cdot H}{(P - C) \cdot A}, \quad v = \frac{(P - C) \cdot V}{(P - C) \cdot A},$$

the numbers in H and V (specifically, $H \cdot H$ and $V \cdot V$) account for your focal length and pixel scaling, while C and A incorporate the camera's position and its 11° downward tilt.

---

4. Relating to the Intrinsic/Extrinsic Model

In the more traditional intrinsic/extrinsic approach you would have:

- **Intrinsic matrix K:**

$$K = \begin{pmatrix} 1440 & 0 & c_x \\ 0 & 1440 & c_y \\ 0 & 0 & 1 \end{pmatrix},$$

  where c_x , c_y are the principal point coordinates (often the image center).

- **Extrinsic parameters:**
  The rotation matrix R (which, in this case, is just the rotation by 11° about the x–axis) and the translation vector t (which would come from the camera center C).

The CAHV model "packs" these pieces into the four vectors C, A, H, V in a way that makes the geometry especially clear in some remote-sensing and robotics contexts.

---

5. Summary

- **Yes, the CAHV model is equivalent** to using intrinsic and extrinsic matrices.
- **Given your camera setup:**

- The focal length converts to 1440 pixels.
- The intrinsic information is encoded in $H = (1440, 0, 0)$ and $V \approx (0, 1413.9, 274.8)$ (after applying an 11° rotation about the horizontal axis).
- The extrinsics are given by the camera centers C (with a 240 mm baseline) and the optical axis $A \approx (0, -0.1908, 0.9816)$ (which is the result of tilting the untilted axis $(0, 0, 1)$ by 11° downward).

When you project a 3D point using

$$u = \frac{(P - C) \cdot H}{(P - C) \cdot A}, \quad v = \frac{(P - C) \cdot V}{(P - C) \cdot A},$$

or when you back–project an image point to a 3D ray, you are using exactly the same information that would appear in K, R, and t in the standard model.

This is how you'd "translate" your physical camera specifications into the CAHV parameters for stereo vision.

The standard triangulation formulas (or the simple back–projection equations in the CAHV model) assume that after calibration and/or image rectification the epipolar lines are horizontal—meaning that corresponding points in the two images differ only in their horizontal coordinate (i.e. disparity is along x). When you find that your SIFT feature matches show a vertical offset of, say, 4–6 pixels, it indicates that the cameras (or the images) are not perfectly rectified.

Here are a few points and options for what to do:

1. **Check the Camera Geometry and Calibration:**
   Even though you know the physical mounting (with both cameras tilted 11° downward and no horizontal cant), slight misalignments or calibration errors can lead to vertical disparities. Double-check your calibration parameters to ensure that the extrinsic (and intrinsic) parameters accurately capture the actual camera orientations and positions.
2. **Perform Stereo Rectification:**
   In practice, even if you start with a good calibration, it's common to rectify the images. Stereo rectification is a process that warps each image so that the epipolar lines become exactly horizontal. After rectification, corresponding points should lie on the same row (or nearly so), and the disparity will be purely horizontal. This is the most common solution:
   - Use the computed calibration (or the fundamental/essential matrix) to determine rectification homographies.
   - Warp both images accordingly.
   - Then re-run feature matching and triangulation using the new, rectified images.

3. **Modify the Triangulation Method:**
   If, for some reason, you choose not to rectify (or if the vertical disparity is very small and you can tolerate it), you can incorporate the full two-dimensional disparity (both horizontal and vertical components) into your triangulation algorithm. Instead of assuming the image coordinates are aligned so that disparity is horizontal only, you use the full projection equations for both cameras. For example, if your projections are given by:

$$u = \frac{(P - C) \cdot H}{(P - C) \cdot A}, \quad v = \frac{(P - C) \cdot V}{(P - C) \cdot A},$$

   you can set up two independent equations for each camera and solve for the 3D point P (often via least–squares if the rays don't intersect exactly). However, this approach is more complicated and is less common in practice.

4. **Physical Alignment:**
   If possible, physically adjust your camera rig to minimize vertical misalignment. Even small differences in mounting can cause vertical disparity. Once you have a rig that minimizes vertical offsets, the rectification step becomes easier or even unnecessary.

In Summary

- **The ideal scenario:** After proper calibration, you rectify your stereo images so that corresponding points have zero (or nearly zero) vertical disparity. Then the standard triangulation formulas that assume horizontal disparity only work perfectly.
- **If vertical disparity is present:** Either use image rectification to remove it or modify your triangulation to account for both horizontal and vertical offsets. Rectification is the standard solution because it simplifies the correspondence search and triangulation.

By rectifying the images, you align the epipolar lines, which not only makes the triangulation formulas valid (disparity purely horizontal) but also typically improves stereo matching reliability.

Below is an example Python script that uses OpenCV to:

1. Load two images from disk.
2. Convert them to grayscale.
3. Apply CLAHE histogram equalization.
4. Detect SIFT keypoints/descriptors and match them (using Lowe's ratio test).

5. Compute the disparities in both x and y directions.
6. Decide if the epipolar lines are not horizontal (i.e. if the average vertical disparity is above a threshold).
7. If needed, estimate the fundamental matrix and use OpenCV's uncalibrated stereo rectification to “fix” the images so that epipolar lines become horizontal.

You can adjust the file paths and thresholds as appropriate for your application.

---

python

```python
import cv2
import numpy as np

def load_and_preprocess(image_path):
    # Load image and convert to grayscale
    img = cv2.imread(image_path)
    if img is None:
        raise IOError(f"Cannot load image at {image_path}")
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Apply CLAHE for contrast enhancement
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
    equalized = clahe.apply(gray)
    return img, equalized

def compute_sift_features(image):
    # Create SIFT detector
    sift = cv2.SIFT_create()
    keypoints, descriptors = sift.detectAndCompute(image, None)
    return keypoints, descriptors

def match_features(des1, des2):
    # Use BFMatcher with default parameters (L2 norm is standard for SIFT)
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=False)
    # Find two nearest neighbors for each descriptor
    matches = bf.knnMatch(des1, des2, k=2)

    # Apply Lowe's ratio test
    good_matches = []
    for m, n in matches:
        if m.distance < 0.75 * n.distance:
            good_matches.append(m)
    return good_matches

def compute_disparities(kp1, kp2, matches):
```

```
    disparities = []
    for m in matches:
        pt1 = kp1[m.queryIdx].pt  # (x, y) in image 1
        pt2 = kp2[m.trainIdx].pt  # (x, y) in image 2
        disp_x = pt1[0] - pt2[0]
        disp_y = pt1[1] - pt2[1]
        disparities.append((disp_x, disp_y))
    disparities = np.array(disparities)
    avg_disp_x = np.mean(np.abs(disparities[:, 0]))
    avg_disp_y = np.mean(np.abs(disparities[:, 1]))
    return avg_disp_x, avg_disp_y, disparities

def rectify_images(img1, img2, kp1, kp2, matches):
    # Collect matching point coordinates
    pts1 = np.float32([kp1[m.queryIdx].pt for m in matches])
    pts2 = np.float32([kp2[m.trainIdx].pt for m in matches])

    # Estimate the fundamental matrix using RANSAC
    F, mask = cv2.findFundamentalMat(pts1, pts2, cv2.FM_RANSAC)
    if F is None:
        print("Fundamental matrix could not be estimated.")
        return None, None

    # Select inlier points only
    pts1_in = pts1[mask.ravel() == 1]
    pts2_in = pts2[mask.ravel() == 1]

    # Get image size (assumes both images are same size)
    h, w = img1.shape[:2]

    # Compute the rectification homographies for both images.
    # stereoRectifyUncalibrated returns a flag and two homography matrices.
    retval, H1, H2 = cv2.stereoRectifyUncalibrated(pts1_in, pts2_in, F, imgSize=(w, h))
    if not retval:
        print("Stereo rectification failed.")
        return None, None

    # Warp the images using the computed homographies
    img1_rect = cv2.warpPerspective(img1, H1, (w, h))
    img2_rect = cv2.warpPerspective(img2, H2, (w, h))
    return img1_rect, img2_rect

def main():
    # Set image file paths (change these to the paths of your images)
    img1_path = 'path_to_image1.jpg'
    img2_path = 'path_to_image2.jpg'

    # Load and preprocess images (load original color and equalized grayscale)
    img1, gray1 = load_and_preprocess(img1_path)
    img2, gray2 = load_and_preprocess(img2_path)

    # Compute SIFT keypoints and descriptors on the equalized images
```

```
    kp1, des1 = compute_sift_features(gray1)
    kp2, des2 = compute_sift_features(gray2)

    # Find good feature matches
    good_matches = match_features(des1, des2)
    print(f"Found {len(good_matches)} good matches.")

    # Compute disparities between matched keypoints
    avg_disp_x, avg_disp_y, disparities = compute_disparities(kp1, kp2, good_matches)
    print(f"Average horizontal disparity: {avg_disp_x:.2f} px")
    print(f"Average vertical disparity: {avg_disp_y:.2f} px")

    # Define a threshold for vertical disparity (in pixels)
    vertical_threshold = 2.0  # adjust as needed

    if avg_disp_y > vertical_threshold:
        print("Vertical disparity is high; epipolar lines are not horizontal.")
        print("Performing stereo rectification...")
        rect1, rect2 = rectify_images(img1, img2, kp1, kp2, good_matches)
        if rect1 is not None and rect2 is not None:
            # Display the rectified images
            cv2.imshow("Rectified Image 1", rect1)
            cv2.imshow("Rectified Image 2", rect2)
            print("Rectified images displayed. Press any key to exit.")
            cv2.waitKey(0)
            cv2.destroyAllWindows()
        else:
            print("Rectification was not successful.")
    else:
        print("Vertical disparity is within threshold; epipolar lines are near horizontal.")
        # Optionally, display the original images with feature matches
        img_matches = cv2.drawMatches(img1, kp1, img2, kp2, good_matches, None,
flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
        cv2.imshow("Matches", img_matches)
        print("Matches displayed. Press any key to exit.")
        cv2.waitKey(0)
        cv2.destroyAllWindows()

if __name__ == '__main__':
    main()
```

Explanation

1. **Loading and Preprocessing:**

   - The function `load_and_preprocess` reads an image, converts it to grayscale, and applies CLAHE.

2. **Feature Detection:**
    - SIFT is used to detect features and compute descriptors.
3. **Feature Matching:**
    - A brute-force matcher finds the two nearest neighbors for each feature. Lowe's ratio test filters out ambiguous matches.
4. **Disparity Calculation:**
    - For each good match, the script computes the difference in x and y coordinates (disparities) and then averages their absolute values.
5. **Rectification Decision:**
    - If the average vertical disparity is above a set threshold (here 2 pixels), the script estimates the fundamental matrix and performs uncalibrated stereo rectification using OpenCV's `stereoRectifyUncalibrated`.
6. **Rectification:**
    - The images are warped using the computed homographies so that the epipolar lines become horizontal.
7. **Display:**
    - The script displays either the rectified images or, if no rectification is needed, the feature matches.

This script provides a starting point; depending on your images and application, you may wish to fine–tune parameters (such as thresholds, CLAHE clip limits, etc.) or add error handling.