Member-only story

A Hands-On Application of Homography: IPM

Correcting Perspectivity

Daryl Tan · Follow
Published in Towards Data Science · 8 min read · May 14, 2020

Figure 1. Left: Frontal view RGB. RIght: BEV from IPM

Homography based IPM

In computer vision, homography is a transformation matrix **H** when applied on a projective plane maps it to another plane (or image). In the case of Inverse Perspective Mapping (**IPM**), we want to produce a birds-eye view image of the scene from the front-facing image plane.

In the field of autonomous driving, IPM aids in several downstream tasks such as lane marking detection, path planning and intersection prediction solely from using a monocular camera as this orthographic view is scale-invariant. Emphasising the importance of this technique.

How does IPM work?

IPM first assume the world to be flat on a plane. Then it maps all pixels from a given viewpoint onto this flat plane through homography projection.

When does IPM work?

In practice, IPM works well in the immediate vicinity of the camera. For faraway features in the scene, blurring and stretching of the scene become more prominent during perspective projection as a smaller number of pixel is represented, limiting the application of IPM. This can be observed in figure 1, where severe undesirable distortion is produced farther away. To be exact, the lookahead distance is approximately 50m in the figure.

In addition, the following constraints must hold:

1. The camera is in a fixed position: Since the position of the road is sensitive to the camera, a slight perturbation in position or orientation will change how the 3D scene is projected onto an image plane.
2. The surface is planar: Any object with height or elevation will violate this condition. Non-planar surfaces will create artefacts/distortion in the BEV image.

3. Free of objects with height. All points above the ground will induce artefacts as a consequence of perspective projection between 2 planes lying arbitrarily in the 3D scene.

In this article, I will attempt to explain the idea of IPM. More importantly, this post is dedicated to how we can work out and apply the homography using only Python and Numpy.

Next, I will show that somewhat similar results can be obtained using OpenCV. We will be using a relatively common road scene from Cityscape dataset as an example. Feel free to message me regarding any questions/doubts or even mistakes you might have spotted.

The code for this article is available here. And all `inline` text refers to some variable or function in the code.

Setting up the problem

The problem that we are trying to solve is to transform a frontal view image into a birds-eye view image. IPM does this by removing the perspective effect from the front-facing camera and remap its image onto a top-view 2D domain. The image in BEV is one that attempts to preserve distance and parallel lines, correcting perspective effect.

The following bullet points summarise the procedure for homography based IPM:

- Model the road (X, Y, Z=0) as a flat 2D plane. Some approximation must be made with regard to resolution as the road are discretized onto the BEV image.

- Determine and construct the projection matrix **P** from known extrinsic and intrinsic parameters. Usually, this is obtained through calibration.
- Transform and warped the road by applying **P**. Also known as a perspective projection.
- Remap the frontal pixel to the new image plane.

I will go into more details in the following sections.

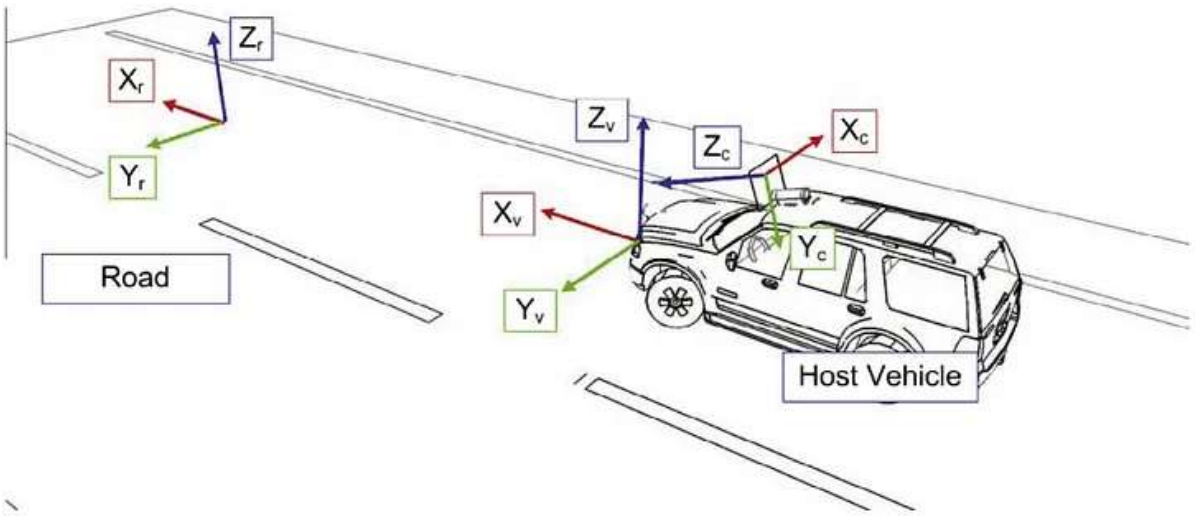


Figure 2. Source

Before we get started, it is important to know where the camera is relative to the road. For the road scene in Cityscape, the camera is mounted on top of the vehicle pitching slightly downwards. The exact position and orientation is written in the file `camera.json` . Note that the values are relative to the ego-vehicle.

Concepts

To understand IPM, some background knowledge about perspective projection and camera projective geometry is required. I will briefly

describe what you need to know about those topics in this section.

1. Perspective projection

Perspective projection precisely describes how the world around us gets mapped on a 2D plane. During this mapping, 2 parallel lines in the world (euclidean space) are transformed into a pair of line in the new plane which converges at the point of infinity. Referring to figure 3, the parallel properties of the 2 cubes placed in the world are not preserved from where we are observing right now. An example of this would be the lane lines.

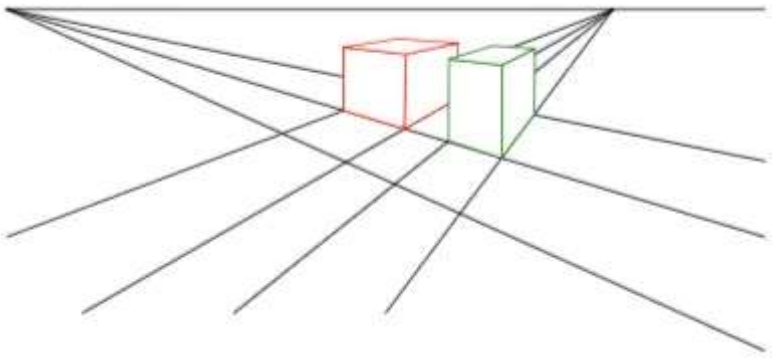


Figure 3

In the case of our example as shown in figure 4, when the road is observed from a different viewpoint, notice that the same region looks different. Parallel lies are no longer preserved in a projective transformation **P**.

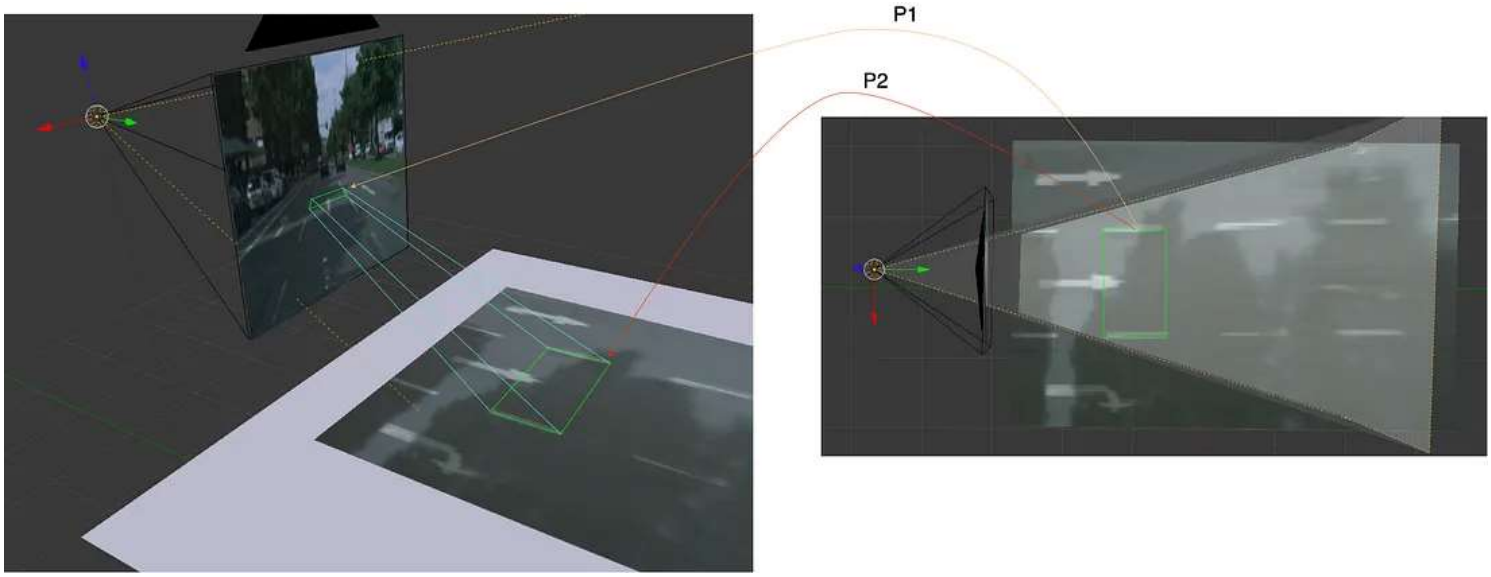
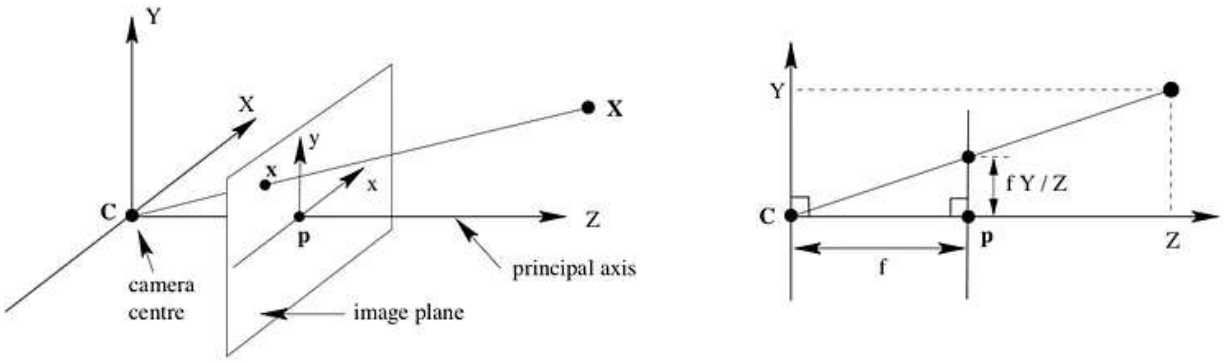


Figure 4. Left: Looking through a camera on top of a vehicle. Right: Top view of the scene.

2. Camera Projective Geometry



$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Figure 5. Pinhole camera model

The camera model describes the perspective projection from the **3D** scene to a **2D** image. The image formed depends on the intrinsic and extrinsic properties.

The **extrinsic [R|t]** describes the relative position and orientation of the world relative to the camera. It brings the scene in world coordinate into the camera coordinate system.

The **intrinsic, K** defines how the 3D scene will be warped onto the image with respect to change in the focal length and camera center. **Note:** This is a simple pinhole camera model. Other factors such as pixel skew and lens distortion are not reflected in the equation.

In our problem. the origin is set to the top left-hand corner of the image plane 2. The idea is to project the entire scene (X, Y, Z) lying on image plane 2 onto the camera image plane 1.

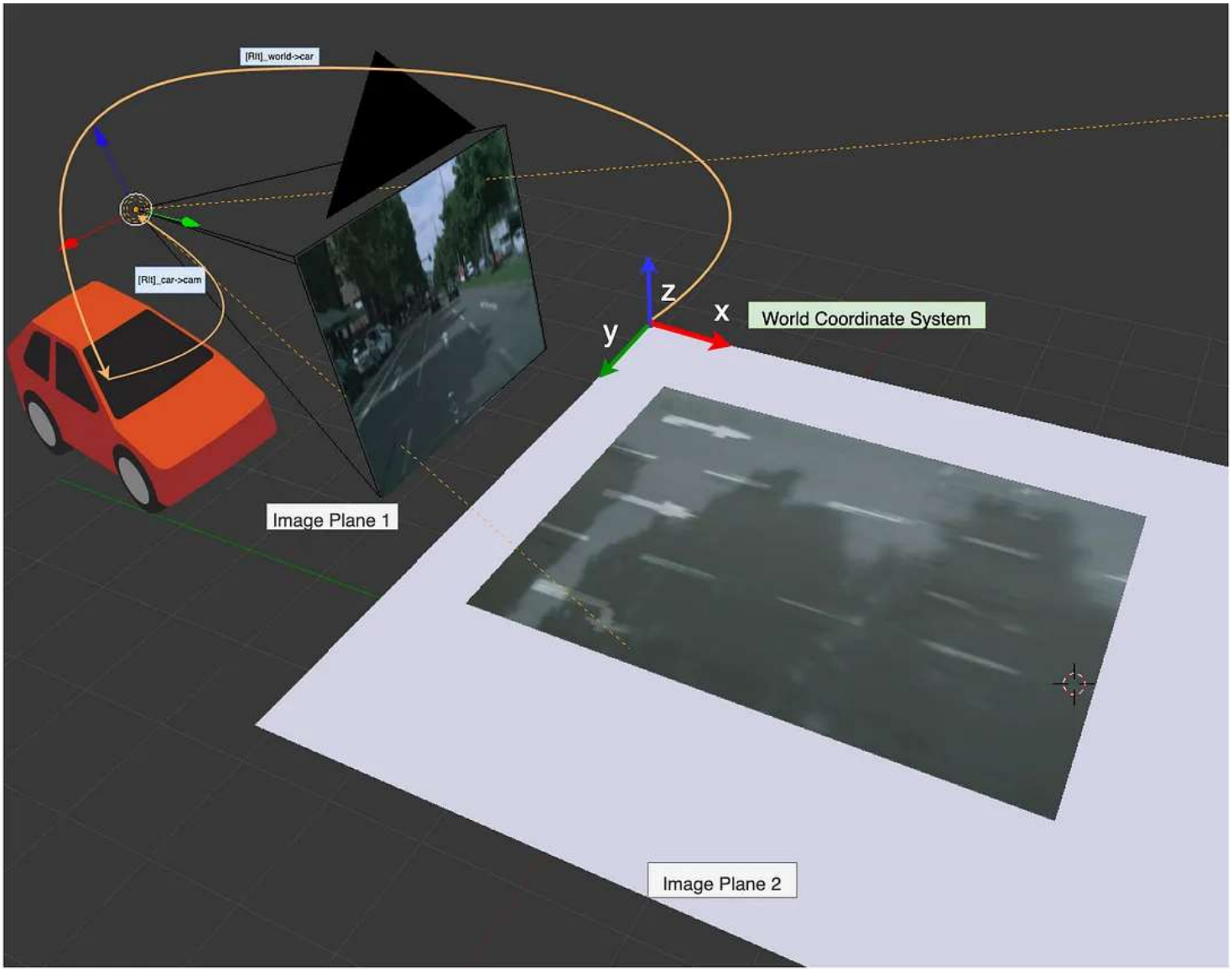


Figure 6. 3D view of the scene. Only the road is modelled

3. Viewing this forward projection as a homography

As mention, we assumed that the road is flat on the ground. Therefore, `Z = 0` for every points lying on the road. Which effectively transform the problem into planar homography. This is one method that can be done in OpenCV to perform the image warping.

Implementation Details

`ipm_from_parameters`

The approach I have taken to tackle the problem is as follows:

1. Slice out a region of the road that we wish to view in BEV `plane`. For this region, define the pixel resolution, absolute distance per pixel (scale) and pose (position and orientation).
2. Apply the perspective projection `perspective` for all 3D points (X, Y, Z=0) in the region using camera projection model to pixel coordinate.
3. Resample from the front view image the corresponding pixels and map it back to image plane 2. Some form of interpolation is needed to prevent holes and aliasing effect. I use bilinear interpolation `bilinear_sampler`.

Lets now go through each point mention above.

1. Defining the plane on the ground

As the vehicle moves, we want the viewing region to be consistent relative to it. Hence the plane is defined with respect to the vehicle.

The origin of the plane is located at the top left corner. The viewable region is illuminated and depends on the field of view of the camera. That is why unobservable pixel (black) is on the image after IPM. I have defined the plane with the following properties (figure 7)

- Size of the region in pixels: 500 X 500
- Resolution: 0.1m per pixel.
- The camera is located and aligned with the midpoint of the plane's y-axis.

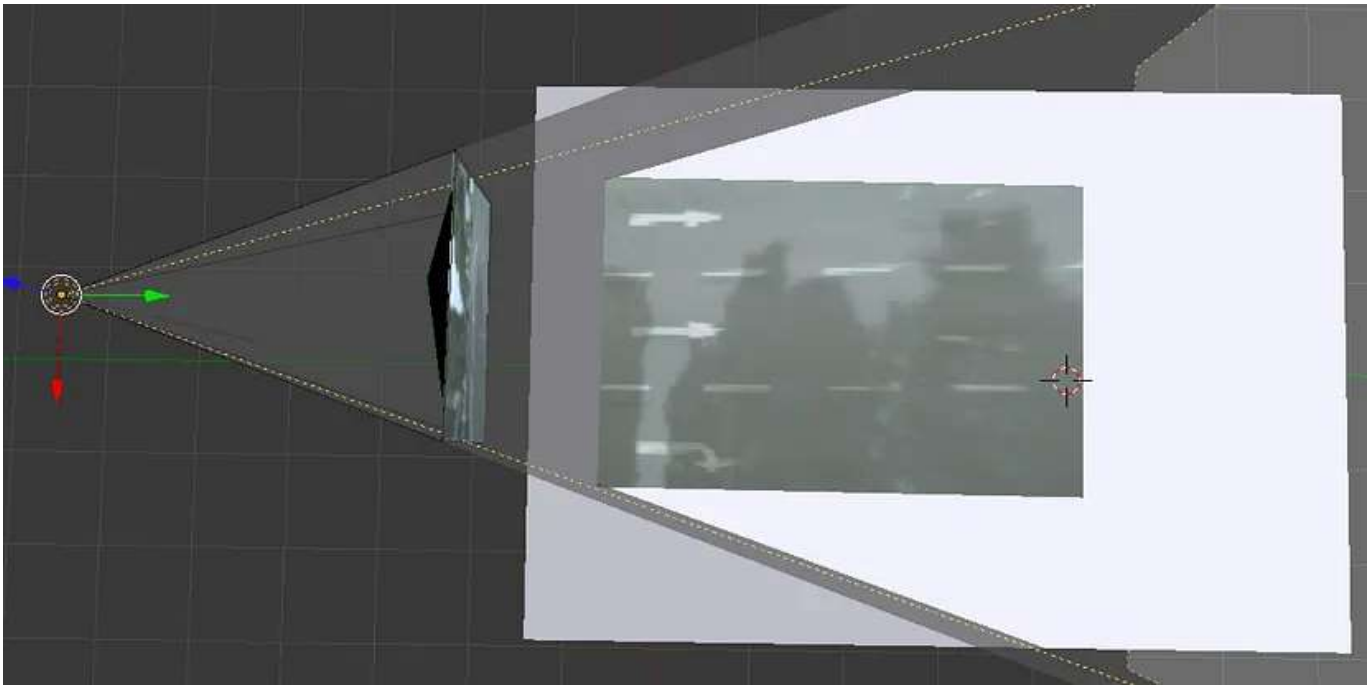


Figure 7. The plane is defined by the grey region. The visible area is illuminated denoting the camera's field of view

2. Deriving and Applying Perspective projection

The camera parameters are given in `camera.json` and given with respect to the ego-vehicle. Since the projection model requires us to define the scene with respect to the camera, some manipulation is required to reverse the transformation.

```
# Notably
np.linag.inv(T) = - T
np.linag.inv(R) = np.transpose(R) # property of orthogonal matrix
```

The extrinsic and intrinsic parameters are constructed in this function `load_camera_params`.

The projective transformation is then applied after transforming the points into homogenous representation. The pixels are scale normalize. i.e. effective focal length = 1. This is done in `perspective`.

3. Resampling pixels

As we move further away from the camera, sparsity becomes more observable as a consequence of perspective. Therefore, I use bilinear interpolation as an approximation.

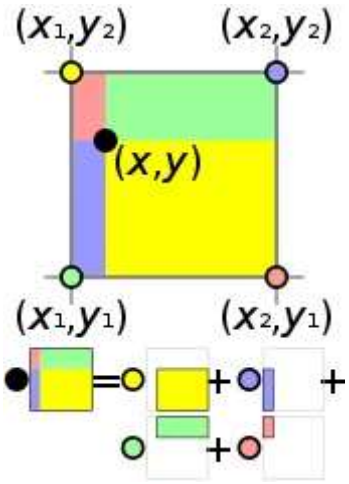


Figure 8. Bilinear interpolation

This is done by taking the weighted (by distance) sum of the 4 corners

```
(px, py) = w0 * im0 + w1 * im1 + w2 * im2 + w3 * im3
```


Now that we understand how to perform the perspective warping. Let's see how all the above steps can be easily performed in OpenCV. This is done by formulating the problem as a plane to plane transformation.

OpenCV does this by solving for the homography matrix **H** internally using point correspondence. **H** now is equivalent to the camera matrix that I have

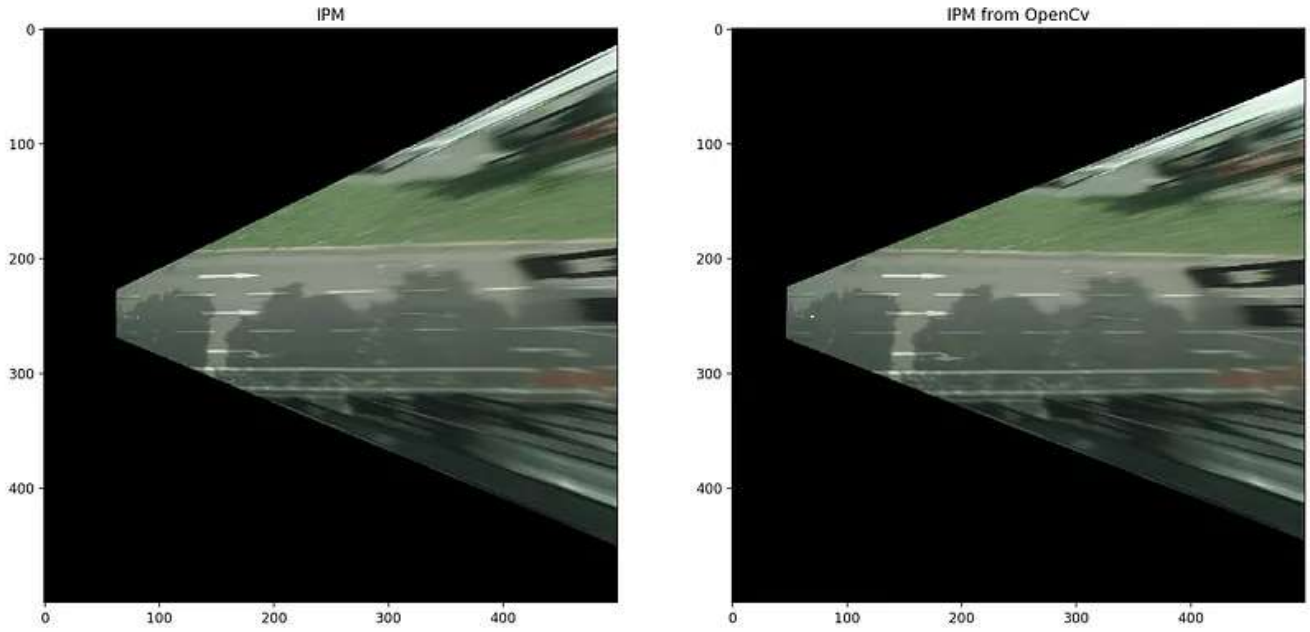
discussed above. A minimum of 4 points is required to estimate H via least-squares method.

There are only 2 steps to perform the transformation

- Pick at least 4 salient features in the image and define its new location in the target image. Pass these correspondences into `cv2.getPerspectiveTransform` to obtain projection matrix.
- Apply the projection matrix to the image using `cv2.warpPerspective`. This will warp it to a bird's eye view and at the same time, applying the desire interpolation method that you have chosen.

The most convenient features to choose are the lane lines. Since we know for sure they are going to be parallel in the target image (BEV).

To show that it produces approximately the same result as the derived method discuss previously, I picked out the points denoting the green polygon vertices as shown in figure 4.



Ending Off

And there we have it. I hope you will find this post useful if you are working on computer vision.

darylclimb/cvml_project

Projects and application using computer vision and machine learning - darylclimb/cvml_project

github.com

Mathematics Computer Vision Machine Learning

Written by Daryl Tan

463 Followers · Writer for Towards Data Science

AV Machine Learning Engineer

Follow

More from Daryl Tan and Towards Data Science