CS7.505 - Mid-Semester Exam

Avneesh Mishra
avneesh.mishra@research.iiit.ac.in *

March 5, 2022 to March 6, 2022

Contents

List of Figures

Listings

*M.S. by research - CSE, IIIT Hyderabad, Roll No: 2021701032

1 Questions

I formulated the following questions for the exam.

1. Enunciate any two methods of keypoint detection and description. Can these be used in an ensemble? If yes, then how?

2. Describe and derive the Fundamental Matrix, Essential Matrix, and Homography Matrix (for the case of pure rotation). When there are many correspondences between two images, can these methods be used to filter out the best correspondences?

3. You've been provided with an image taken from a self-driving car that shows another car in front. A camera has been placed on top of the car, 1.65 m from the ground. The camera intrinsic matrix K is provided. Your task is to draw a 3D-bounding box around the car in front. Your approach should be to place eight points in the 3D world such that they surround all the corners of the car, then project them onto the image and connect the projected image points using lines. Make a python program for this.

   Assume that the image plane is perfectly perpendicular to the ground. You might have to apply a small 5° rotation about the vertical axis to align the box perfectly. Rough car dimensions - h: 1.38 m, w: 1.51, l: 4.10. Also, estimate the approximate translation vector to the mid-point of the two rear wheels of the car in the camera frame.

1.1 Reasons

The reasons why I feel these questions are worthy and interesting

- The set has the perfect balance of theory, math, and critical thinking
  - Question 1 is theoretical and involves reading papers.
  - Question 2 is towards practical mathematics.
  - Question 3 involves programming.
- The questions can have concrete answers and are not vague.

Q1: Keypoint detection and description

Question

> Enunciate any two methods of keypoint detection and description. Can these be used in an ensemble? If yes, then how?

Reason This question promotes a deeper dive into the reading material for the theoretical methods taught in class. It should serve as a quick and good reference for traditional keypoint detection and description methods. The aim is to have a good information archive, created through reading the original text, well summarized, in one place.

Ensemble techniques have recently caught steam, especially in the age of deep learning. Exploring such options for traditional methods could yield stronger baselines for traditional feature detection and description methods.

Q2: Relating matrices and RANSAC

Question

> Describe and derive the Fundamental Matrix, Essential Matrix, and Homography Matrix (for the case of pure rotation). When there are many correspondences between two images, can these methods be used to filter out the best correspondences?

Reason This question is to lay a foundation for matrices relating to two images. Though the derivation is not traditionally important, it is good to have the theoretical backbone in one place. Random Sample Consensus (RANSAC) is a very popular method to boost the performance of a correspondence matching algorithm. This question aims to derive the theory behind it and also give a direction on how it can be applied using the knowledge of these basic matrices in computer vision. It concludes with examples and references to a python package that can perform RANSAC using the matrices.

Q3: Bounding Box

Question

> You've been provided with an image taken from a self-driving car that shows another car in front. A camera has been placed on top of the car, 1.65 m from the ground. The camera intrinsic matrix K is provided. Your task is to draw a 3D-bounding box around the car in front. Your approach should be to place eight points in the 3D world such that they surround all the corners of the car, then project them onto the image and connect the projected image points using lines. Make a python program for this.
>
> Assume that the image plane is perfectly perpendicular to the ground. You might have to apply a small 5° rotation about the vertical axis to align the box perfectly. Rough car dimensions - h: 1.38 m, w: 1.51, l: 4.10. Also, estimate the approximate translation vector to the mid-point of the two rear wheels of the car in the camera frame.

Reason The question is to implement the camera model and transformations in Python to solve a real-world problem. The question can test the understanding of the camera model if solved correctly. Plus, it will be something that can be extended and is the only interactive part of this exam.

2 Q1: Keypoint Detection and Description

Question

> Enunciate any two methods of keypoint detection and description. Can these be used in an ensemble? If yes, then how?

Keypoints are points of *interest* and are useful in image matching and description. Keypoints have to be *detected* (location found in an image) by a detector, and they have to be described by a *descriptor* (for some unique identification).

The answer is described in the subsections below.

2.1 SIFT

Scale Invariant Feature Transform (SIFT) is an image feature generation method introduced by David G. Lowe in [Low99]. A more explained iteration was presented in [Low04] with some revisions to the model. The primary contribution of the author was exploring the features in multiple scales (through an image pyramid), which makes the keypoint descriptors *scale* invariant.

Detector The keypoint detector has the following basic steps

1. Construction of DoG (Difference of Gaussian) image pyramid: The input image resolution is increased (scaled up) by a factor of two using bilinear interpolation. Then two successive gaussian blurs are applied, yielding image `A` (less blurred) and `B` (more blurred). Subtracting image `B` from `A` given the Difference of Gaussian image. This is repeated for scales of 1.5 in each direction (up and down). This scaling factor was later revised to 2.

2. Achieve keypoint locations: The local extrema in the DoG image is a keypoint. First, eight neighbor comparisons are made at the same scale. Then, if the point is in extrema (maximum or minimum), comparisons are made at higher scales (position interpolation to maintain scale).

3. Extract Keypoint orientations: The image `A` is used to compute the gradient magnitudes and orientations. The magnitudes are thresholded to 0.1 times the maximum gradient value (to reduce illumination effects). A histogram of gradient orientations in the local neighborhood of keypoints is created. The weight of the orientations is the thresholded gradient values. This histogram (containing 36 bins covering $0°$ to $360°$) is smoothened, and the peak is chosen as the gradient orientation.

In the end, the keypoint locations (on the image) and orientations are obtained. The direction is used to achieve rotation-invariant features.

Descriptor The keypoint descriptor (as presented in the original work in [Low99]) has the following basic steps

1. Reorient the local region around the keypoint. This is basically to set the local orientation (of keypoint) as a reference. This is done by simple subtraction of gradient orientations in later steps.

2. Subdivide the local region: The local region (within the radius of 8 pixels) is sub-divided into a 4×4 sub-array, with each sub-array having an 8-bin gradient histogram.

3. Run the same on a larger scale version: On one scale higher in the pyramid, perform the above step but with a 2×2 sub-array (still 8 bins in the histogram).

Note that the gradient directions for the histogram are not just the gradients at the center pixel but are interpolated in the $n \times n$ grid (sub-array with $n = 4$ or 2). The total number of SIFT descriptors (length) for a keypoint is $8 \times 4 \times 4 + 8 \times 2 \times 2 = 160$.

In the revised edition [Low04], a 16×16 local region is sub-divided into 4×4 grid, with each grid having 8 orientation bins (from histogram). Therefore, the new descriptor length becomes $8 \times 4 \times 4 = 128$. It is found that this is much faster to compute and doesn't have a large compromise on performance.

2.2 SURF

Speeded-Up Robust Features (SURF) is another feature detection and description algorithm proposed by Herbert Bay, et al. in [BTG06]. This was also described in a more illustrated manner in [Bay+08]. The primary contribution of the authors were exploiting the idea of integral image (which Voila and Jones originally proposed in [VJ01]), to speed up calculation of an approximated second order Gaussian (the Hessian matrix). The authors also proposed robust methods for descriptor extraction.

Detector The keypoint detector has the following basic steps

1. Estimate the integral image for the input image, using the equation below.

$$I_{\sum}(\mathbf{x}) = \sum_{i=0}^{i \leq x} \sum_{j=0}^{j \leq y} I(i, j)$$

2. Approximate the terms in the hessian matrix: The determinant of hessian matrix is used as a proxy for feature points. Instead of using the Difference of Gaussian to estimate the terms in matrix, a second order approximation with simple terms is used.

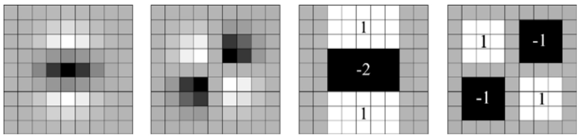


Figure 1: Approximation of second order derivatives
Left to right: Instead of L_{yy} and L_{xy} (first two), we use D_{yy} and D_{xy}.

This approximation requires yields feature value as $\det(H_{approx}) = D_{xx}D_{yy} - (0.9D_{xy})^2$. This is much faster to compute, but is still single scale

3. Multiple scales: Instead of resizing the image, we can resize these kernels (as shown in the figure above). The standard sizes (to preserve center pixel) are 9×9, 15×15, 27×27, ... An octave consists of a series of filter response maps obtained using convolution with different filter sizes (set of four usually successive sizes).

4. Non-Maximum Suppression: The maxima values in a $3 \times 3 \times 3$ neighborhood are retained and these are interpolated to their true scale and position on image.

This yields the position of the keypoints; not just on the image, but also the particular scale of detection (s).

Orientation alignment We need to obtain orientations before getting the descriptors. This is done in the following steps

1. Calculate the *Haar-wavelets* in the local region around the keypoint: We approximate the dx and dy filter (gradient in X and Y direction respectively) with a kernel containing $+1$ and -1 values. This is run on the $6s \times 6s$ neighborhood of the keypoint to get the gradients of neighboring points.

2. Apply a gaussian weight with $\sigma = 2s$

3. Represent each point in the neighborhood as a point in a 2D scatter plot with X and Y values being the weighed dx and dy values.

4. On this 2D scatter plot, run a window in polar form with the angle being $\pi/3$. Get the window with maximum sum (of weights of the points in the window).

5. For this window, the orientation is calculated by summing the X and the Y values of the points (separately) and then getting the angle.

We now have the orientation of each keypoint (thereby allowing us to get rotation robust descriptors). This orientation is also linked to the same scaling factor in which the keypoint was detected.

Descriptor The descriptor is calculated in the following steps

1. Get a $20s \times 20s$ oriented square patch around the keypoint (centered at the local feature). Calculate the integral image for this oriented patch, and estimate the Haar-wavelets (similar to the orientation alignment part) for dx and dy values for each pixel in this patch.

2. Split this patch into 4×4 sub-regions, with each sub-region having 5×5 samples (actually, $5s \times 5s$ pixels).

3. For each sub-region, calculate $\mathbf{v} = [\sum d_x, \sum |d_x|, \sum d_y, \sum |d_y|]$, a 4-dimensional descriptor of the particular sub-region.

4. Stacking these 4-dimensional descriptors for every sub-region into a column vector gives the SURF descriptor. Invariance to contrast is achieved through normalizing them.

The traditional SURF algorithm therefore gives a descriptor of length $4 \times 4 \times 4 = 64$.

Despite being of smaller length, the descriptor (along with the matching method described in section 4.3 of [Bay+08]) seems to give more robust correspondences than most other then-state-of-the-art methods. The authors demonstrate 3D reconstruction from un-calibrated cameras in section 5.2 of [Bay+08].

2.3 Ensemble

TL; DR It depends on the application. Let us take the application of *finding feature correspondences between two images* as an example.

Example The aim is to match the identical features in two images. Feature description plays an essential role here.

Traditionally, the descriptors are uniquely defined for each method (SIFT and SURF, for Example, have different descriptor formats). They, therefore, cannot be concatenated or merged in any easy way.

However, we can apply some tricks to get an ensemble of correspondences. One of them is to apply descriptor matching (using techniques like the mutual nearest neighbor, cosine distance, Euclidean distance, or Mahalanobis distance) for the *individual* methods (separately). Then, obtain the keypoints (again, separately) and then concatenate the obtained keypoints. We now have point correspondences from both methods.

Such methods can boost correspondences between two images by a significant margin.

3 Q2: Relating Matrices and RANSAC

Question

> Describe and derive the Fundamental Matrix, Essential Matrix, and Homography Matrix (for the case of pure rotation). When there are many correspondences between two images, can these methods be used to filter out the best correspondences?

Before answering the questions, it is essential to brief on Epipolar Geometry.

3.1 Epipolar Geometry

Consider the image below where two cameras are capturing the images of the world.

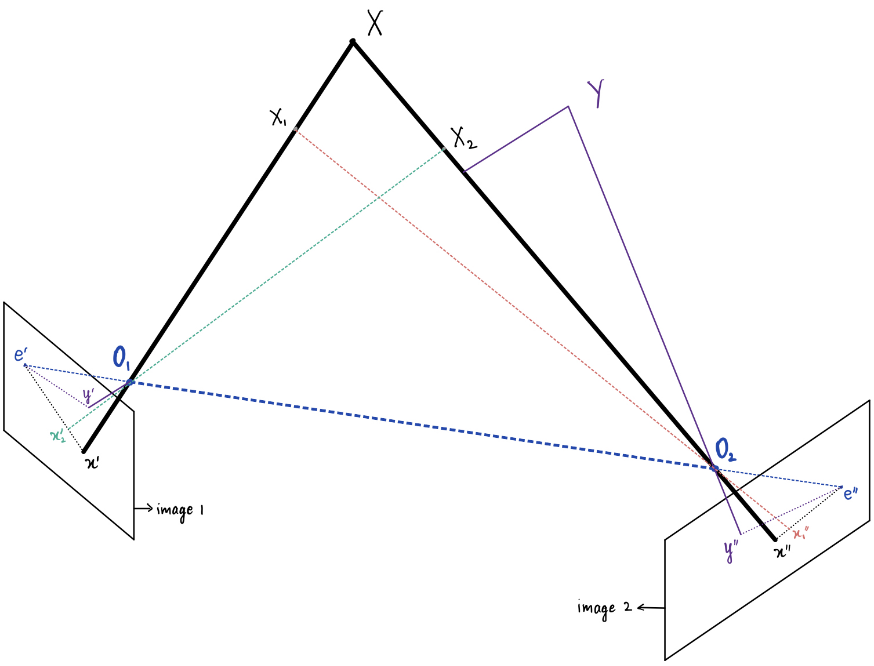


Figure 2: Epipolar Geometry
Points X and Y are points in the real world whose image falls at (pixel locations) x' and y' in image 1 and x'' and y'' in image 2. The origins of these cameras are located at O_1 and O_2, respectively.
As a convention, points in the first image have a single hyphen, whereas points in the second image have two hyphens.

Epipolar Axis The line joining O_1 and O_2 is called the **epipolar axis**. It intersects the images at e' and e'' respectively.

Epipolar Plane We know O_1O_2X form a plane (they are three points in the world). This plane is called the **epipolar plane**.

Epipolar Line It is clear that the image of X_1 in camera 1 will also fall on x' (same line), similarly the image of X_2 in camera 2 will also fall on x''. However, the image of X_1 in camera 2 will fall at x''_1, which is on the line joining x'' and e''. This line is called the **epipolar line**. Similarly, the image of X_2 in camera 1 will fall at x'_2 (which is also on the line joining e' and x').

When X_1 moves along $\overline{XO_1}$, its image x''_1 traces a line in the second image (the *epoipolar line*). Same can be said for X_2 and the first image.

3.2 Fundamental Matrix

Say we have three vectors $\vec{a}$, $\vec{b}$, and $\vec{c}$. Their triple product $\left\langle \vec{a}\ \vec{b}\ \vec{c} \right\rangle = \vec{a} \cdot \left(\vec{b} \times \vec{c}\right)$ is the volume of the parallelepiped formed by the three vectors.

Since the three vectors $\overrightarrow{O_1X}$, $\overrightarrow{O_1O_2}$, and $\overrightarrow{O_2X}$, all lie on the same plane, their triple product will be zero. That is $\langle O_1X \;\; O_1O_2 \;\; O_2X\rangle = \mathbf{0}$.

From camera projection properties, we can write

$$x' = \mathbf{K}'\mathbf{R}'\left[\mathbf{I} \mid -\mathbf{X}_{O'}\right] X \qquad\qquad x'' = \mathbf{K}''\mathbf{R}''\left[\mathbf{I} \mid -\mathbf{X}_{O''}\right] X \tag{1}$$

Where $\mathbf{K}'$, $\mathbf{R}'\left[\mathbf{I} \mid -\mathbf{X}_{O'}\right]$ and $\mathbf{K}''$, $\mathbf{R}''\left[\mathbf{I} \mid -\mathbf{X}_{O''}\right]$ are camera intrinsic and extrinsic parameters (for camera 1 and camera 2) respectively. Note that all the above terms are in *homogeneous coordinates*. The vector X can be assumed to be unit-scale (last term - the scaling factor - is 1).

We know that $\overrightarrow{O_1X} = X - X_{O'} \equiv \mathbf{R}'^{-1}\mathbf{K}'^{-1}x'$ and $\overrightarrow{O_2X} = X - X_{O''} \equiv \mathbf{R}''^{-1}\mathbf{K}''^{-1}x''$. Another reduction is $\overrightarrow{O_1O_2} = b$ for the baseline vector (joining the two camera centers).

Therefore, the triple product constraint mentioned above can be reduced to

$$\langle O_1X \;\; O_1O_2 \;\; O_2X\rangle = \mathbf{0} \Rightarrow (X - X_{O'}) \cdot (b \times (X - X_{O''})) \equiv \left(\mathbf{R}'^{-1}\mathbf{K}'^{-1}x'\right) \cdot \left(b \times \left(\mathbf{R}''^{-1}\mathbf{K}''^{-1}x''\right)\right) = 0$$

Using $a \cdot b = a^\top b$ and $a \times b = [a]_\times\, b$ (where $[a]_\times$ is the cross product skew symmetric matrix), we can reduce the above equation to

$$\begin{aligned}\langle O_1X \;\; O_1O_2 \;\; O_2X\rangle &= \left(\mathbf{R}'^{-1}\mathbf{K}'^{-1}x'\right) \cdot \left(b \times \left(\mathbf{R}''^{-1}\mathbf{K}''^{-1}x''\right)\right) = \left(\mathbf{R}'^{-1}\mathbf{K}'^{-1}x'\right)^\top [b]_\times \left(\mathbf{R}''^{-1}\mathbf{K}''^{-1}x''\right) \\ &= x'^\top \left(\mathbf{K}'^{-\top}\mathbf{R}'^{-\top}\,[b]_\times\,\mathbf{R}''^{-1}\mathbf{K}''^{-1}\right) x'' = x'^\top \mathbf{F} x'' = 0\end{aligned} \tag{2}$$

The equation 2 is the basis for two points (in different images) to be projected to the same point in the 3D world. If the points correspond in the 3D world, they must satisfy the equation. However, the converse is not necessarily valid, as we will see later. Let us get the intuition of the epipolar line and the epipoles through the fundamental matrix.

Epipolar line

Say we have a point x' in one image, and we want to find the corresponding point x'' in a second image. Assume that we have $\mathbf{F}$ (the fundamental matrix) relating the two images.

Referring to figure 2, our job would become much easier if we know the *epipolar line* of x' in the second image (the line $e''x''$). Let us call this line l'' (since it's in the second image).

For a true x'' to lie on l'', it must satisfy $x'' \cdot l'' = x''^\top l'' = l''^\top x'' = 0$. From equation 2, we know that $x'\mathbf{F}x'' = 0$.

Matching the two results, we get $l''^\top = x'\mathbf{F} \Rightarrow l'' = \mathbf{F}^\top x'$ as the equation of the epipolar line in the second image (of the point x' in the first image). Now, a search along this line in the second image has higher chances of yielding the true x''.

Epipoles

We know that the epipolar line in the second image (of a point x' in the first image) is given by $l'' = \mathbf{F}^\top x'$.

We know that the epipole e'' (in the second image) is the projection of O_1 in the second image. That is $e'' = \mathbf{P}'' X_{O''}$ (where $\mathbf{P}'' = \mathbf{K}''\mathbf{R}''\left[\mathbf{I} \mid -\mathbf{X}_{O''}\right]$ is the second camera's projection matrix). We also know that the epipole e'' lies on line l'', since all epipolar lines intersect at the epipoles (this is seen by considering another epipolar plane with a 3D point Y in figure 2). We therefore have $l''^\top e'' = 0$.

For any point x' in the first image, there will be a unique epipolar line l'' in the second image. We therefore have

$$l''^\top e'' = \left(\mathbf{F}^\top x'\right)^\top e'' = x'^\top \mathbf{F} e'' = (\mathbf{F}e'')^\top x' = 0 \tag{3}$$

We have two conditions: x' can be any valid point in image 1 (in homogeneous coordinates) *and* equation 3 always has to hold true. The only possibility where both these conditions hold true is when $(\mathbf{F}e'')^\top = \mathbf{0}^\top$ (it is a row of three zeros). We therefore have $\mathbf{F}e'' = \mathbf{0}$.

In other words, the epipole e'' is the null space of the fundamental matrix $\mathbf{F}$. We can obtain the epipoles from $\mathbf{F}$ through eigendecomposition (by obtaining the eigenvector with the least - ideally zero - eigenvalue).

Transpose Relation

Equation 2 gives the fundamental matrix relating image 1 to image 2 (simply because point x' in image 1 comes before point x'' which is in image 2). Transposing it gives

$$(x'^\top)_{1,3}\,\mathbf{F}_{3,3}\,(x'')_{3,1} = 0 \Rightarrow \left(x'^\top \mathbf{F} x''\right)^\top = 0 \Rightarrow x''^\top \mathbf{F}^\top x' = 0 \tag{4}$$

Therefore, the fundamental matrix relating the second image to the first is given by the *transpose*. That is, if ${}^1_2\mathbf{F} = \mathbf{F}$, then ${}^2_1\mathbf{F} = \mathbf{F}^\top$.

3.3 Essential Matrix

The fundamental matrix is used for uncalibrated cameras where we are not interested in knowing the intrinsic parameters of the two cameras. The **essential matrix** is used for relating images from two *calibrated* cameras; that is, we know the intrinsic matrices.

We can get the projected rays (in 3D space) for the pixel homogeneous coordinates (in both the images) as ${}^kX' = \mathbf{K}'^{-1}x'$ and ${}^kX'' = \mathbf{K}''^{-1}x''$. Say that the pixels correspond, that is, these rays intersect each other. The equation 2 for the fundamental matrix can then be written as

$$\begin{aligned}
&\rightarrow x'^\top \left(\mathbf{K}'^{-\top}\mathbf{R}'^{-\top}\,[b]_\times\,\mathbf{R}''^{-1}\mathbf{K}''^{-1}\right) x'' = \left(\mathbf{K}'^{-1}x'\right)^\top \left(\mathbf{R}'^{-\top}\,[b]_\times\,\mathbf{R}''^{-1}\right)\left(\mathbf{K}''^{-1}x''\right) = 0\\
&\Rightarrow {}^kX'^\top\ \left(\mathbf{R}'^{-\top}\,[b]_\times\,\mathbf{R}''^{-1}\right)\ {}^kX'' = {}^kX'^\top\ \mathbf{E}\ {}^kX'' = 0 \Rightarrow \mathbf{E} = \mathbf{R}'^{-\top}\,[b]_\times\,\mathbf{R}''^{-1}
\end{aligned} \tag{5}$$

The essential matrix relates images from calibrated cameras (whose projected rays can be resolved). Also, correlating equations 2 and 5 we get

$$\mathbf{F} = \left(\mathbf{K}'^{-\top}\mathbf{R}'^{-\top}\,[b]_\times\,\mathbf{R}''^{-1}\mathbf{K}''^{-1}\right) = \mathbf{K}'^{-\top}\left(\mathbf{R}'^{-\top}\,[b]_\times\,\mathbf{R}''^{-1}\right)\mathbf{K}''^{-1} = \mathbf{K}'^{-\top}\,\mathbf{E}\,\mathbf{K}''^{-1} \tag{6}$$

The above equation derives the relation between fundamental matrix $\mathbf{F}$ and essential matrix $\mathbf{E}$.

Note that the fundamental matrix has 7 degrees of freedom, whereas the essential matrix has 5 degrees of freedom.

3.4 Pure Rotation Homography

A camera's projection equation can be given as

$$\mathbf{x} = \mathbf{K}\mathbf{R}[\mathbf{I} \mid -\mathbf{X}_O]\,\mathbf{X} \tag{7}$$

Where

- $\mathbf{K}$ is the camera intrinsic matrix
- $\mathbf{R}$ is the rotation matrix of the camera (expressed in the real world coordinates)
- $\mathbf{I}$ is the 3×3 identity matrix
- $\mathbf{X}_O$ is the origin of the camera's projection center in the world (expressed as 3D world coordinates)
- $\mathbf{X}$ is the point in scene (which is being projected) expressed in homogeneous coordinates

The point $\mathbf{x}$ is the location of the point in the image plane (also in homogeneous coordinates).

Note that since there is a dimension lost ($\mathbf{x}$ is $\mathbb{P}^2$ whereas $\mathbf{X}$ is $\mathbb{P}^3$), we cannot truly recover the point $\mathbf{X}$ from just a pixel location $\mathbf{x}$ in the image.

However, we can recover the *line* passing through the camera center that yields the point $\mathbf{x}$ (for any point on that line). This line can be rotated and projected back as a pixel.

Assume that the initial frame of the camera is given by $\{1\}$ (image pixels represented by $\mathbf{x}'$), the world frame is given by $\{0\}$ and the new camera frame (after *strict* rotation) is given by $\{2\}$ (image pixels represented by $\mathbf{x}''$).

Writing the projection equations, we get

$$\mathbf{x}' = \mathbf{K}\,{}^1_0\mathbf{R}[\mathbf{I} \mid -{}_0\mathbf{X}_O]\,{}_0\mathbf{X} \qquad\qquad \mathbf{x}'' = \mathbf{K}\,{}^2_0\mathbf{R}[\mathbf{I} \mid -{}_0\mathbf{X}_O]\,{}_0\mathbf{X} \tag{8}$$

Note that the camera center and the point (${}_0\mathbf{X}_O$ and ${}_0\mathbf{X}$ in $\mathbb{P}^3$) are represented in the world frame (frame $\{0\}$); and are unchanged. Also, note that the two poses of the camera are related as

$$ {}^0_2\mathbf{R} =^0_1 \mathbf{R}\,{}^1_2\mathbf{R} \Rightarrow^2_0 \mathbf{R} =^0_2 \mathbf{R}^\top =^1_2 \mathbf{R}^\top\,{}^0_1\mathbf{R}^\top \Rightarrow^2_0 \mathbf{R} = \,{}^2_1\mathbf{R}\,{}^1_0\mathbf{R} \tag{9} $$

Where ${}^2_1\mathbf{R}$ is $\{1\}$'s orientation expressed in $\{2\}$. Substituting the result of equation 9 in equation 8, and noting that we're dealing with homogeneous coordinates here (uniformly scaled values are the same), we get

$$ \begin{aligned} &\rightarrow \mathbf{x}' = \mathbf{K}\,{}^1_0\mathbf{R}[\mathbf{I} \mid -{}_0\mathbf{X}_O]\;{}_0\mathbf{X} \Rightarrow \mathbf{K}^{-1}\mathbf{x}' \equiv \;{}^1_0\mathbf{R}[\mathbf{I} \mid -{}_0\mathbf{X}_O]\;{}_0\mathbf{X} \\ &\rightarrow \mathbf{x}'' = \mathbf{K}\,{}^2_0\mathbf{R}[\mathbf{I} \mid -{}_0\mathbf{X}_O]\;{}_0\mathbf{X} \Rightarrow \mathbf{x}'' = \mathbf{K}\,{}^2_1\mathbf{R}\left({}^1_0\mathbf{R}\,[\mathbf{I} \mid -{}_0\mathbf{X}_O]\;{}_0\mathbf{X}\right) \\ &\Rightarrow \mathbf{x}'' = \;\mathbf{K}\,{}^2_1\mathbf{R}\,\mathbf{K}^{-1}\mathbf{x}' = \mathbf{H}\,\mathbf{x}' \qquad \text{where } \mathbf{H} = \mathbf{K}\,{}^2_1\mathbf{R}\,\mathbf{K}^{-1} \end{aligned} \tag{10} $$

The equation 10 gives the resulting homography $\mathbf{H}$ (for pure rotation), relating pixels $\mathbf{x}'$ in first image to pixels $\mathbf{x}''$ in the second image.

3.5 RANSAC

Ransom Sample Consensus (RANSAC) is a method to find the best set of inliers from a large collection of samples. It basically is picking the minimum number of required points randomly from a large collection of points; running the estimation and checking algorithm; giving the chosen set a score; and moving on to the next cycle.

Let us say that we have n points with $e\%$ of them being outliers. Our model requires s points (at minimum) to fit / estimate a relation (which holds true for inliers). We want to estimate the inliers in our data (the n points) with probability p (call this probability of success).

Let us calculate the maximum number of cycles T that will be required.

The probability that we pick an inlier from our data is $1 - e$.

The probability that we pick s inliers from our data (each selection is independent), is therefore $(1-e)^s$. Therefore, the probability that we pick *at least* one outlier is $1 - (1-e)^s$.

The probability that we pick at least one outlier T times is $(1-(1-e)^s)^T$. However, the experiment should end with T trials and if we pick at least one outlier every trial, then we've essentially failed. The probability of failure is $1 - p$. These two should be equal. We therefore have

$$ \begin{aligned} 1 - p = (1-(1-e)^s)^T &\Rightarrow \log(1-p) = T\log(1-(1-e)^s) \\ &\Rightarrow T = \frac{\log(1-p)}{\log(1-(1-e)^s)} \end{aligned} \tag{11} $$

The equation 11 can be used to estimate the maximum number of random samplings the RANSAC process should need.

Python package - `pydegensac`

The python package `pydegensac` allows us to run such RANSAC procedures quickly. Official repository can be found on GitHub.

The following can be used for pure homographies (cases like $\mathbf{x}'' = \mathbf{H}\,\mathbf{x}'$)

```
H, mask = pydegensac.findHomography(src_pts, dst_pts, 4.0, 0.99, 5000)
```

Here, the `src_pts` and `dst_pts` are the $n, 2$ correspondences (this could be a numpy array). We accept a pixel threshold of 4.0 pixel distance (correspondences within these distances, re-projected from the model are considered inliers). The confidence is 0.99, with 5000 max iterations (cycles).

Sometimes, there is a viewpoint change. In such cases, doing RANSAC for the estimation of fundamental matrix and finding the inlier mask becomes helpful. The following can be used for fundamental matrix (cases like $x'^\top\mathbf{F}x'' = 0$)

```
F, mask = pydegensac.findFundamentalMatrix(src_pts, dst_pts, 4.0, 0.999, 10000,
enable_degeneracy_check= True)
```

All argument (except the last) retain their previous meanings. The argument `enable_degeneracy_check` allows the checking of the case when the points are degenerate (the fundamental matrix cannot be calculated under these conditions). This usually happens when the linear equation in $\mathbf{F}$ loses rank.

4 Q3: Bounding Box

Question

You've been provided with an image taken from a self-driving car that shows another car in front. A camera has been placed on top of the car, 1.65 m from the ground. The camera intrinsic matrix K is provided. Your task is to draw a 3D-bounding box around the car in front. Your approach should be to place eight points in the 3D world such that they surround all the corners of the car, then project them onto the image and connect the projected image points using lines. Make a python program for this.

Assume that the image plane is perfectly perpendicular to the ground. You might have to apply a small 5° rotation about the vertical axis to align the box perfectly. Rough car dimensions - h: 1.38 m, w: 1.51, l: 4.10. Also, estimate the approximate translation vector to the mid-point of the two rear wheels of the car in the camera frame.

The image given is

The camera projection matrix is

K = [7.2153e+02,0,6.0955e+02;0,7.2153e+02,1.7285e+02;0,0,1]

4.1 Theory

For the context, refer to the image below

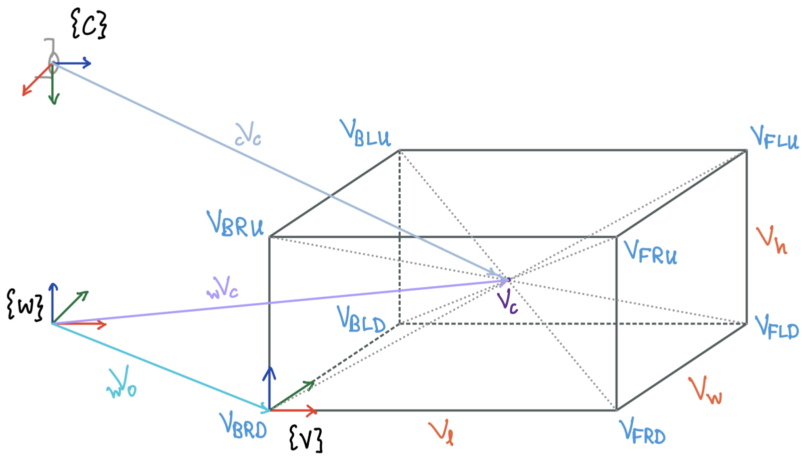


Figure 3: System Model

Frames The following frames are described in the figure 3

- **World frame** {W}: The frame directly below the camera, fixed to the vehicle. This frame could be the odometry frame (whose transform to all sensors on the vehicle is known).

- **Vehicle frame** {V}: The vehicle frame. This is attached to the *rear bottom right* of the vehicle, which is being localized.
- **Camera frame** {C}: The camera frame. The Z-axis looks out, Y goes down, and X to the right. It is assumed that the transform between this and the world frame {W} is known.

Vectors The following vectors are important

- The vector ${}_{\mathrm{W}}\mathrm{V}_{\mathrm{O}}$ is the point $\mathrm{V}_{\mathrm{BRD}}$ represented in {W}
- The vector ${}_{\mathrm{V}}\mathrm{V}_{\mathrm{C}}$ is the vector of the centroid of the bounding box in {V}. This is not shown above, but is implicitly assumed. The vectors ${}_{\mathrm{W}}\mathrm{V}_{\mathrm{C}}$ and ${}_{\mathrm{C}}\mathrm{V}_{\mathrm{C}}$ are the projections (transforms) of this vector in {W} and {C} respectively.
- The vector ${}_{\mathrm{W}}\mathrm{C}_{\mathrm{O}}$ is the origin of {C} expressed in {W}

Symbols The following symbols are used in the derivation hereon

- v_x and v_y are the X and Y coordinates of ${}_{\mathrm{W}}\mathrm{V}_{\mathrm{O}}$ (the Z coordinate is 0). These have to be computed (they are unknowns).
- v'_{cx} and v'_{cy} are the *pixel coordinates* (X and Y) of the image of ${}_{\mathrm{C}}\mathrm{V}_{\mathrm{C}}$ in the camera.

  The user can pick this, but it can also be retrieved through a detection algorithm (through the center of bounding boxes, maybe). These are, therefore, known.
- v_l, v_w, and v_h are the length, width, and height of the bounding box of the vehicle (in the real world measurements). These are known.
- V_θ or v_θ is the yaw (rotation about Z in radians) of {V} in {W}. This is also given in the statement.
- C_h is the height of the camera above ground

Important Notes The following points are important

- The edges/corners of the bounding box are in blue. Each corner is named (subscript) according to position on X-axis (`B`ack or `F`ront), followed by position on Y-axis (`L`eft or `R`ight) followed by position on Z axis (`U`p or `D`own).

  These axes are of {V}, and the points are the 3D corners of the bounding box. The origin of {V} is at $\mathrm{V}_{\mathrm{BRD}}$, and the problem entails finding this point in the XY plane of {W}. If we find this, we can automatically find every other point on the vehicle (they're all rigid).
- The point V_{C} is the centroid of the vehicle (in front). A nice property of projection homography (transform) is that it *preserves line intersections*. The centroid is the intersection of 4 lines (body diagonals of the cuboid).

  Usually, we have an object detection algorithm that gives us the *bounding box* of the car ahead (which is a rectangle in pixel coordinates). The center of this bounding box can be projected outwards (as a line, if we know the camera intrinsic matrix $\mathbf{K}$) to intersect with the actual center of the cuboid (described by points above). This is a **critical** assumption of this method.

4.1.1 Theoretical Solution

With respect to {V} and {W}, we know the following

$$
{}_{\mathrm{V}}\mathrm{V}_{\mathrm{O}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \qquad
{}_{\mathrm{W}}\mathrm{V}_{\mathrm{O}} = \begin{bmatrix} v_x \\ v_y \\ 0 \\ 1 \end{bmatrix} \qquad
{}_{\mathrm{V}}\mathrm{V}_{\mathrm{C}} = \begin{bmatrix} v_l/2 \\ v_w/2 \\ v_h/2 \\ 1 \end{bmatrix} \qquad
{}^{\mathrm{W}}_{\mathrm{V}}\mathbf{T} = \begin{bmatrix} \cos(v_\theta) & -\sin(v_\theta) & 0 & v_x \\ \sin(v_\theta) & \cos(v_\theta) & 0 & v_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
\rightarrow \ {}_{\mathrm{W}}\mathrm{V}_{\mathrm{C}} = {}^{\mathrm{W}}_{\mathrm{V}}\mathbf{T} \ {}_{\mathrm{V}}\mathrm{V}_{\mathrm{C}}
$$

With respect to {C} and {W}, we know the following

$$
{}_{\mathrm{W}}\mathrm{C}_{\mathrm{O}} = \begin{bmatrix} 0 \\ 0 \\ C_h \\ 1 \end{bmatrix} \qquad {}^{\mathrm{W}}_{\mathrm{C}}\mathbf{T} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & C_h \\ 0 & 0 & 0 & 1 \end{bmatrix} \rightarrow {}^{\mathrm{C}}_{\mathrm{W}}\mathbf{T} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & C_h \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad {}^{\mathrm{C}}_{\mathrm{V}}\mathbf{T} = {}^{\mathrm{C}}_{\mathrm{W}}\mathbf{T}\, {}^{\mathrm{W}}_{\mathrm{V}}\mathbf{T}
$$

Now, we get the vehicle center in the camera frame

$$
{}_{\mathrm{C}}\mathrm{V}_{\mathrm{C}} = {}^{\mathrm{C}}_{\mathrm{V}}\mathbf{T}\; {}_{\mathrm{V}}\mathrm{V}_{\mathrm{C}} \quad \rightarrow {}_{\mathrm{C}}\mathrm{V}_{\mathrm{C}_h} = \left(\frac{{}_{\mathrm{C}}\mathrm{V}_{\mathrm{C}}\,[1:3]}{{}_{\mathrm{C}}\mathrm{V}_{\mathrm{C}}\,[4]} \right)_{3,1}
$$

We then de-homogenize it (scale it to unit scaling factor, and remove the last element). This is done using the camera projection matrix (note that the point is already in the camera frame), we get

$$
\mathrm{v}'_{\mathrm{C}} = \begin{bmatrix} v'_{cx} \\ v'_{cy} \\ 1 \end{bmatrix} \qquad\qquad \mathrm{v}'_{\mathrm{C}} \equiv \mathbf{K}\; {}_{\mathrm{C}}\mathrm{V}_{\mathrm{C}_h} \Rightarrow \mathbf{K}^{-1}\; \mathrm{v}'_{\mathrm{C}} \equiv {}_{\mathrm{C}}\mathrm{V}_{\mathrm{C}_h}
$$

Since this is a homogeneous relationship, we can set the last element of the vectors on both sides to 1 (fix the scaling) and then use the other two equations to solve for the two unknown variables v_x and v_y.

Since the equations are long, but in the form of two variable and two simple equations, this task can be given to sympy solvers. Something like this can be used

```
    eq_s = sp.Eq(lhs_eqn, rhs_eqn)    # Equality to solve
    sols = sp.solvers.solve(eq_s, [vx, vy]) # Solutions to the equality
```

Once we know the point in 3D, we can create a bounding box through normal camera projection (apply the camera model to the 3D points to obtain the image points).

4.2 Solution

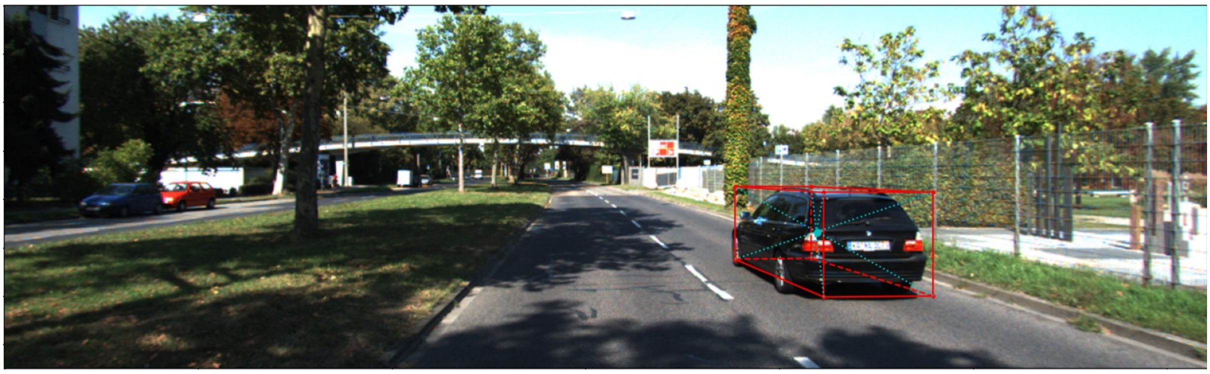
The code for solving the theory equations and yielding the results is presented in Appendix 5.1. The images can be seen in Figure 4.

The main snippet that does the last part of theory is shown below

```
    # %% Camera projection equations (Main solution)
    lhs_eq = K_sp.inv() * vimg  # Image projected to the world
    rhs_eq = sp.Matrix([    # Vehicle center in camera frame [X;Y;Z]
        [vc_c[0]/vc_c[3]],
        [vc_c[1]/vc_c[3]],
        [vc_c[2]/vc_c[3]]])
    # The last value of LHS is 1 (projection), set the same of for RHS
    rhs_eqn = rhs_eq / rhs_eq[2] # Last value corresponds
    lhs_eqn = lhs_eq / lhs_eq[2] # Last value corresponds
    eq_s = sp.Eq(lhs_eqn, rhs_eqn)    # Equality to solve
    sols = sp.solvers.solve(eq_s, [vx, vy]) # Solutions to the equality
    vx_sol = sols[vx]
    vy_sol = sols[vy]
```

The variables `vx_sol` and `vy_sol` are the equations for finding v_x and v_y respectively. These are currently in symbolic form and the actual values are later substituted (to get floating point results).

(a) Chosen center point



(b) Bounding box

(c) Bounding box with the center of rear axle

Figure 4: Program output
In a, the red cross is the center pixel
In b, the bounding box is shown
In c, the bounding box (in blue), along with the center of the rear axle (in green) is shown

A part of the program output answering the question is shown below

Center pixel is: 839, 234

Vehicle BRD at (X, Y): 9.3510, -4.5330

Rear axle in camera frame is (X, Y, Z): [3.70936019 1.65 10.10205256]

Rear axle in image at (x, y): 874, 290

5 Appendix

5.1 Q3: Code for Bounding Box

The code for creating the bounding box is shown below

```python
# %% Import everything
import sympy as sp
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
import os

# %% Read image and get the center pixel
# Function to select a pixel
# A function that takes an image and returns the marked indices
def get_clicked_points(img, img_winname = "Point Picker",
    dmesg = False):
    """
        Get the clicked locations as a list of [x, y] points on image
        given as 'img'. Note that the origin is at the top left corner
        with X to the right and Y downwards.

        Parameters:
        - img: np.ndarray   shape: N, M, C
            An image, should be handled by OpenCV or be a numpy array
            (height, width, channels). The passed image is not altered
            by the function.
        - img_winname: str  default: "Point Picker"
            Window name (to be used by OpenCV)
        - dmesg: bool or str    default: False
            If True (or `str` type) a string for debug is printed. If
            the type is `str`, then the string is prepended to the
            debug message.

        Returns:
        - img_points: list
            A list of [x, y] points clicked on the image
        - _img: np.ndarray  shape: N, M, C
            The same image, but annotated with points clicked. Random
            colors are assigned to each point.
    """
    img_points = [] # A list of [x, y] points (clicked points)
    _img: np.ndarray = img.copy()   # Don't alter img

    def img_win_event(event, x, y, flags, params):
        if event == cv.EVENT_LBUTTONUP:
            # Print the debug message (if True or 'str')
            if dmesg == True or type(dmesg) == str:
                db_msg = f"Clicked on point (x, y): {x}, {y}"
                if type(dmesg) == str:
                    db_msg = dmesg + db_msg
                print(db_msg)
            # Record point
            img_points.append([x, y])   # Record observation
            # -- Put marker on _img for the point --
            # Random OpenCV BGR color as tuple
            _col = tuple(map(int, np.random.randint(0, 255, 3)))
            # Add circle
            cv.circle(_img, (x, y), 10, _col, -1)
            # Add text
            cv.putText(_img, f"{len(img_points)}", (x, y-15),
                cv.FONT_HERSHEY_SIMPLEX, 0.8, _col, 2, cv.LINE_AA)

    # Create GUI Window
    cv.namedWindow(img_winname, cv.WINDOW_NORMAL)
    cv.resizeWindow(img_winname, 1242, 375)  # Window (width, height)
    cv.setMouseCallback(img_winname, img_win_event)
    # Main loop
    while True:
        cv.imshow(img_winname, _img)
        k = cv.waitKey(1)
        if k == ord('q'):
```

```
            break
    cv.destroyWindow(img_winname)
    # Return results
    return img_points, _img


img_location = "./image.png"
img_location = os.path.realpath(os.path.expanduser(img_location))
assert os.path.isfile(img_location)
car_img = cv.imread(img_location)
# Click center pixel
cent_px, _ = get_clicked_points(car_img, "Pick Centroid Pixel")
cpx, cpy = cent_px[0]   # Resolve as pixel values
print(f"Center pixel is: {cpx}, {cpy}")

# %% Symbols
# -- Known parameters (as floats) --
vl_val, vw_val, vh_val = 4.10, 1.51, 1.38   # L, W, H in m
vth_val = np.deg2rad(5) # Angle (in rad)
ch_val = 1.65   # Cam height in m
# vcx_val, vcy_val = 839, 234 # Camera pixel of vehicle center
vcx_val, vcy_val = cpx, cpy # Camera pixel of vehicle center
K_val = [   # Camera intrinsic parameter matrix
    [7.2153e+02,0,6.0955e+02],
    [0,7.2153e+02,1.7285e+02],
    [0,0,1]]
K_np = np.array(K_val, float)   # As numpy
# - The above will only be used in the end -

# -- Known Parameters (as symbols) --
# Vehicle properties
vl, vw, vh = sp.symbols(r"V_l, V_w, V_h")   # dimensions (L, W, H)
vth = sp.symbols(r"V_\theta")   # Z rotation for vehicle (in rad)
# Camera properties
ch = sp.symbols(r"C_h") # Camera height (from ground)
# Camera projection matrix
K_11, K_12, K_13 = sp.symbols(r"k_{11}, k_{12}, k_{13}")
K_22, K_23, K_33 = sp.symbols(r"k_{22}, k_{23}, k_{33}")
K_sp = sp.Matrix([[K_11, K_12, K_13], [0, K_22, K_23], [0, 0, K_33]])
vcx, vcy = sp.symbols(r"V'_{c_x}, V'_{c_y}")    # Pixel of car center

# -- Unknown parameters --
# Vehicle parameters
vx, vy = sp.symbols(r"V_x, V_y")    # Vehicle X and Y from {world}

# %% Prior to main work
# Image point (homogeneous coordinates)
vimg = sp.Matrix([vcx, vcy, 1])
# -- Homogeneous Transformations --
# - TF {vehicle} in {world} -
# Rotation for {vehicle} in {world}
R_w_v = sp.Matrix([ # Rot(Z, vth)
    [sp.cos(vth), -sp.sin(vth), 0],
    [sp.sin(vth), sp.cos(vth), 0],
    [0, 0, 1]])
# Vehicle origin (in {world} - homogeneous coordinates)
vorg_w = sp.Matrix([vx, vy, 0, 1])
# Homogeneous Transformation matrix ({vehicle} in {world})
tf_w_v = sp.Matrix.hstack(  # Stacking R_w_v and vorg_w
    sp.Matrix.vstack(R_w_v, sp.Matrix([[0, 0, 0]])), vorg_w)
# - TF {camera} in {world} -
# Rotation from world to camera
R_w_c = sp.Matrix([ # Z out of cam, Y down, X to right
    [0, 0, 1],
    [-1, 0, 0],
    [0, -1, 0]])
# Camera origin (in {world} - homogeneous coordinates)
corg_w = sp.Matrix([0, 0, ch, 1])
# Homogeneous Transformation matrix ({camera} in {world})
tf_w_c = sp.Matrix.hstack(  # Stacking R_w_v and vorg_w
    sp.Matrix.vstack(R_w_c, sp.Matrix([[0, 0, 0]])), corg_w)
# - TF {world} in {camera} -
tf_c_w = sp.Matrix.hstack(
```

```python
    sp.Matrix.vstack(R_w_c.T, sp.Matrix([[0, 0, 0]])),
    sp.Matrix.vstack(
        -R_w_c.T * sp.Matrix(corg_w[0:3]), sp.Matrix([[1]]))
    )   # Invert the transformation matrix

# %% Equation for resolving points
# Vehicle center in {vehicle}
vc_v = sp.Matrix([vl/2, vw/2, vh/2, 1])
# Vehicle center in {world}
vc_w = tf_w_v * vc_v
# Vehicle center in {camera}
vc_c = tf_c_w * vc_w

# %% Camera projection equations (Main solution)
lhs_eq = K_sp.inv() * vimg  # Image projected to the world
rhs_eq = sp.Matrix([    # Vehicle center in camera frame [X;Y;Z]
    [vc_c[0]/vc_c[3]],
    [vc_c[1]/vc_c[3]],
    [vc_c[2]/vc_c[3]]])
# The last value of LHS is 1 (projection), set the same of for RHS
rhs_eqn = rhs_eq / rhs_eq[2] # Last value corresponds
lhs_eqn = lhs_eq / lhs_eq[2] # Last value corresponds
eq_s = sp.Eq(lhs_eqn, rhs_eqn)    # Equality to solve
sols = sp.solvers.solve(eq_s, [vx, vy]) # Solutions to the equality
vx_sol = sols[vx]
vy_sol = sols[vy]

# %% Solution for vehicle positions
# Substitution values
val_subs = {
    vl: vl_val,
    vw: vw_val,
    vh: vh_val,
    ch: ch_val,
    vth: vth_val,
    vcx: vcx_val,
    vcy: vcy_val,
    K_11: K_np[0, 0], K_12: K_np[0, 1], K_13: K_np[0, 2],
    K_22: K_np[1, 1], K_23: K_np[1, 2], K_33: K_np[2, 2]
}
vx_res = float(vx_sol.subs(val_subs))
vy_res = float(vy_sol.subs(val_subs))
print(f"Vehicle BRD at (X, Y): {vx_res:.4f}, {vy_res:.4f}")

# %% Show car with center pixel
car_img_plt = cv.cvtColor(car_img.copy(), cv.COLOR_BGR2RGB)
plt.figure(figsize=(15, 10))
plt.imshow(car_img_plt)
plt.title("Car with center pixel")
plt.plot(vcx_val, vcy_val, 'rx')
plt.savefig("./fig1.png")
plt.show()

# %%
# Transform solution (for vehicle to camera frame) as floats
vx_w_sol = vx_res
vy_w_sol = vy_res
tf_c_v_sp = tf_c_w * tf_w_v
tf_c_v = tf_c_v_sp.subs(val_subs).subs({vx: vx_w_sol, vy: vy_w_sol})
tf_c_v = np.array(tf_c_v, float)    # As numpy floats
print(f"Transformation from vehicle to camera frame is: \n{tf_c_v}")
# Camera projection matrix (in numpy)
K_np = np.array(K_sp.subs(val_subs), float)
print(f"Camera projection matrix is: \n{K_np}")

# %% Project the 3D points to the camera frame
points_v = np.array([   # Points in [X, Y, Z], in {Vehicle}
    [0, 0, 0],  # V_BRD
    [vl_val, 0, 0], # V_FRD
    [vl_val, vw_val, 0],    # V_FLD
    [0, vw_val, 0],     # V_BLD
    [0, 0, vh_val],     # V_BRU
    [vl_val, 0, vh_val],    # V_FRU
```

```
    [vl_val, vw_val, vh_val],   # V_FLU
    [0, vw_val, vh_val]     # V_BLU
])
# Convert to homogeneous coordinates
corners_v = np.vstack((points_v.T, np.ones((1, points_v.shape[0]))))
# Corners in camera frame
corners_c = tf_c_v @ corners_v
corners_c = corners_c[0:3, :]   # Loose the last row
# Convert to the camera coordinates
corners_img = K_np @ corners_c
# Scale to 1 (for pixel representations)
corners_img_px = corners_img / corners_img[2]

# %% Show bounding box
# Show the results
c_img = corners_img_px.astype(int)[0:2, :].T
plt.figure(figsize=(20, 10))
plt.imshow(car_img_plt)
plt.title("Bounding Box")
# Center
plt.plot(vcx_val, vcy_val, 'co')
# All bounding boxes
plt.plot(c_img[:, 0], c_img[:, 1], 'r.')
# Make lines
plt.plot(c_img[[0, 1, 2], 0], c_img[[0, 1, 2], 1], 'r--')
plt.plot(c_img[[2, 3, 0], 0], c_img[[2, 3, 0], 1], 'r-')
plt.plot(c_img[4:8, 0], c_img[4:8, 1], 'r-')
plt.plot(c_img[[7, 4, 0], 0], c_img[[7, 4, 0], 1], 'r-')
plt.plot(c_img[[1, 5], 0], c_img[[1, 5], 1], 'r--')
plt.plot(c_img[[2, 6], 0], c_img[[2, 6], 1], 'r-')
plt.plot(c_img[[3, 7], 0], c_img[[3, 7], 1], 'r-')
# Dotted diagonal lines
plt.plot(c_img[[0, 6], 0], c_img[[0, 6], 1], 'c:')
plt.plot(c_img[[1, 7], 0], c_img[[1, 7], 1], 'c:')
plt.plot(c_img[[2, 4], 0], c_img[[2, 4], 1], 'c:')
plt.plot(c_img[[3, 5], 0], c_img[[3, 5], 1], 'c:')
plt.savefig("./fig2.png")
plt.show()

# %% Rear axle part
# Rear axle in vehicle frame
rear_axle_v = np.array([0.2*vl_val, 0.5*vw_val, 0, 1])  # Homogeneous
# Rear axle in camera frame
rear_axle_c = tf_c_v @ rear_axle_v
raxle_c = rear_axle_c[:3]
print(f"Rear axle in camera frame is (X, Y, Z): {raxle_c}")
# Image point
raxle_img = K_np @ raxle_c
rax_i = raxle_img / raxle_img[2]
rax_i = rax_i.astype(int)
print(f"Rear axle in image at (x, y): {rax_i[0]}, {rax_i[1]}")

# %% Show bounding box with rear axle
# Show the results (with axle)
plt.figure(figsize=(20, 10))
plt.imshow(car_img_plt)
plt.title("Bounding Box with Axle")
# Center
plt.plot(vcx_val, vcy_val, 'co')
# All bounding boxes
plt.plot(c_img[:, 0], c_img[:, 1], 'r.')
# Make lines
plt.plot(c_img[[0, 1, 2], 0], c_img[[0, 1, 2], 1], 'r--')
plt.plot(c_img[[2, 3, 0], 0], c_img[[2, 3, 0], 1], 'r-')
plt.plot(c_img[4:8, 0], c_img[4:8, 1], 'r-')
plt.plot(c_img[[7, 4, 0], 0], c_img[[7, 4, 0], 1], 'r-')
plt.plot(c_img[[1, 5], 0], c_img[[1, 5], 1], 'r--')
plt.plot(c_img[[2, 6], 0], c_img[[2, 6], 1], 'r-')
plt.plot(c_img[[3, 7], 0], c_img[[3, 7], 1], 'r-')
# Dotted diagonal lines
plt.plot(c_img[[0, 6], 0], c_img[[0, 6], 1], 'c:')
plt.plot(c_img[[1, 7], 0], c_img[[1, 7], 1], 'c:')
plt.plot(c_img[[2, 4], 0], c_img[[2, 4], 1], 'c:')
```

```
plt.plot(c_img[[3, 5], 0], c_img[[3, 5], 1], 'c:')
# Rear axle
plt.plot(rax_i[0], rax_i[1], 'go')
plt.savefig("./fig3.png")
plt.show()

# %%
```

The output of the program is as follows

Center pixel is: 839, 234

Vehicle BRD at (X, Y): 9.3510, -4.5330

Transformation from vehicle to camera frame is: [[-0.08715574 -0.9961947 0. 4.5329549] [0. 0. -1. 1.65] [0.9961947 -0.08715574 0. 9.35097549] [0. 0. 0. 1.]]

Camera projection matrix is: [[721.53 0. 609.55] [0. 721.53 172.85] [0. 0. 1.]]

Rear axle in camera frame is (X, Y, Z): [3.70936019 1.65 10.10205256]

Rear axle in image at (x, y): 874, 290

References

[Low99] David G Lowe. “Object recognition from local scale-invariant features”. In: *Proceedings of the seventh IEEE international conference on computer vision*. Vol. 2. Ieee. 1999, pp. 1150–1157.

[VJ01] Paul Viola and Michael Jones. “Rapid object detection using a boosted cascade of simple features”. In: *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*. Vol. 1. Ieee. 2001, pp. I–I.

[Low04] David G Lowe. “Distinctive image features from scale-invariant keypoints”. In: *International journal of computer vision* 60.2 (2004), pp. 91–110.

[BTG06] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. “Surf: Speeded up robust features”. In: *European conference on computer vision*. Springer. 2006, pp. 404–417.

[Bay+08] Herbert Bay et al. “Speeded-up robust features (SURF)”. In: *Computer vision and image understanding* 110.3 (2008), pp. 346–359.