AA 274A: Principles of Robot Autonomy I
Problem Set 3

Name: Li Quan Khoo
SUID: lqkhoo (06154100)

October 21, 2020

Problem 1: Camera Calibration

(i) (code)

(ii) (code)

(iii) (code)

(iv) (code)

(v) (code)

Problem 2: Line Extraction

(i) (code)

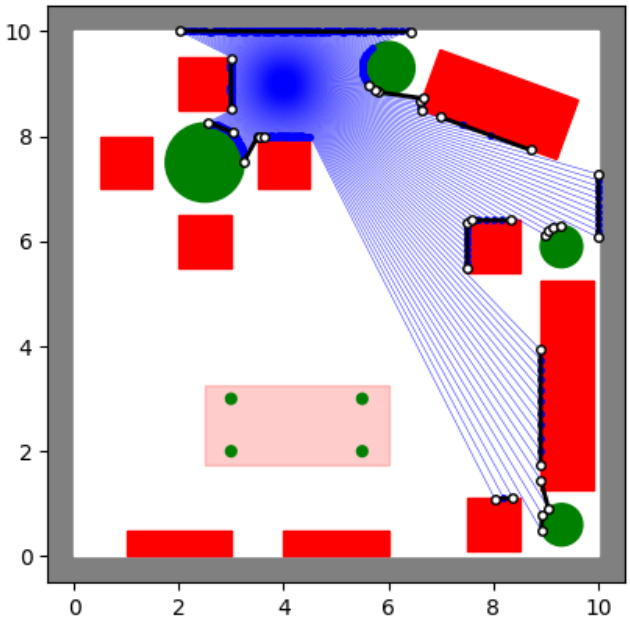
(ii) With parameters

(a) MIN_SEG_LENGTH = 0.05

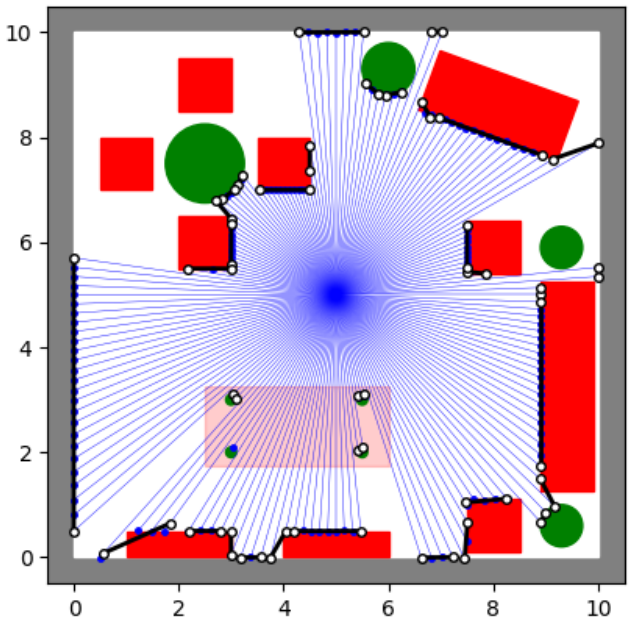
(b) LINE_POINT_DIST_THRESHOLD = 0.02

(c) MIN_POINTS_PER_SEGMENT = 2

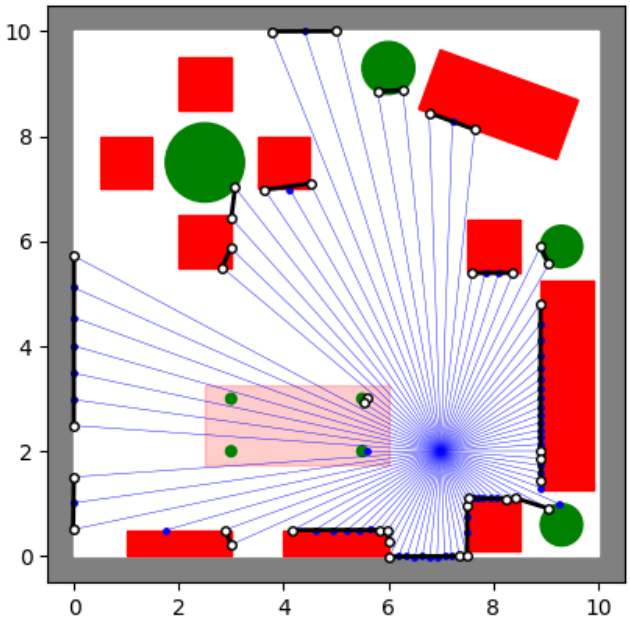
(d) MAX_P2P_DIST = 1



Split-and-merge on `rangeData_4_9_360`



Split-and-merge on `rangeData_5_5_180`



Split-and-merge on `rangeData_7_2_90`

Problem 3: Linear Filtering

(i) We'll just show the workings for the elements at $(1, 1)$. Where

$$I = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \tag{1}$$

(a)

$$F = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad , \quad F \otimes I(1,1) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad , \quad F \otimes I = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \tag{2}$$

(b)

$$F = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad , \quad F \otimes I(1,1) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 6 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad , \quad F \otimes I = \begin{bmatrix} 2 & 3 & 0 \\ 5 & 6 & 0 \\ 8 & 9 & 0 \end{bmatrix} \tag{3}$$

(c) This kernel is doing discrete difference (differentiation) of the image along the horizontal axis. It could be used to perform edge detection tasks.

$$F = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad , \quad F \otimes I(1,1) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 6 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad , \quad F \otimes I = \begin{bmatrix} 7 & 4 & -7 \\ 15 & 6 & -15 \\ 13 & 4 & -13 \end{bmatrix} \tag{4}$$

(d) This is an isotropic, normalized Gaussian kernel performing blurring on the image. It could be used to filter out high frequency information on either axis.

$$F = \frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix} \quad , \quad F \otimes I(1,1) = \frac{1}{16}\begin{bmatrix}0 & 0 & 0\\ 0 & 80 & 0\\ 0 & 0 & 0\end{bmatrix} \quad , \quad F \otimes I = \frac{1}{16}\begin{bmatrix}21 & 36 & 33\\ 52 & 80 & 68\\ 57 & 84 & 69\end{bmatrix} \tag{5}$$

(ii) I'm assuming that the question is alluding to the fact that performing correlation (or convolution) over an input of depth d means we end up summing over d individual correlations or convolutions performed on individual input channels.

This is saying that:

$$G(i,j) = \sum_w \left(\sum_u \sum_v F_w(u,v) \cdot I_w(i+u, j+v) \right) \tag{6}$$

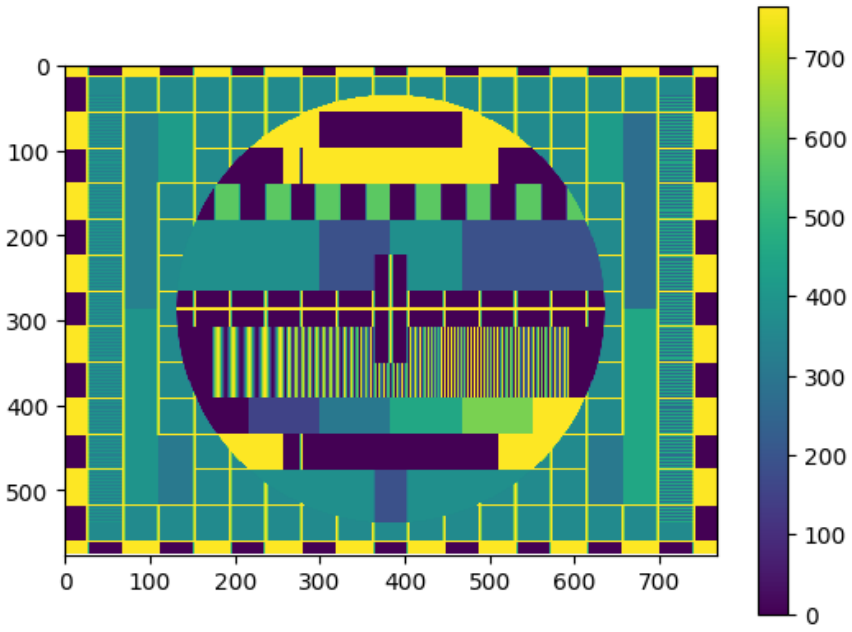
where $G(i,j)$ is a correlation operation at position i,j of the image, and F and I are flattened vector representations of the kernel and current image patch (including padding) that we are operating over. Therefore, writing out the matrices explicitly:

$$G(i,j) = \sum_w \begin{bmatrix} - & F_w^\mathsf{T}(u,v) & - \end{bmatrix} \begin{bmatrix} | \\ I_w \\ | \end{bmatrix} \tag{7}$$

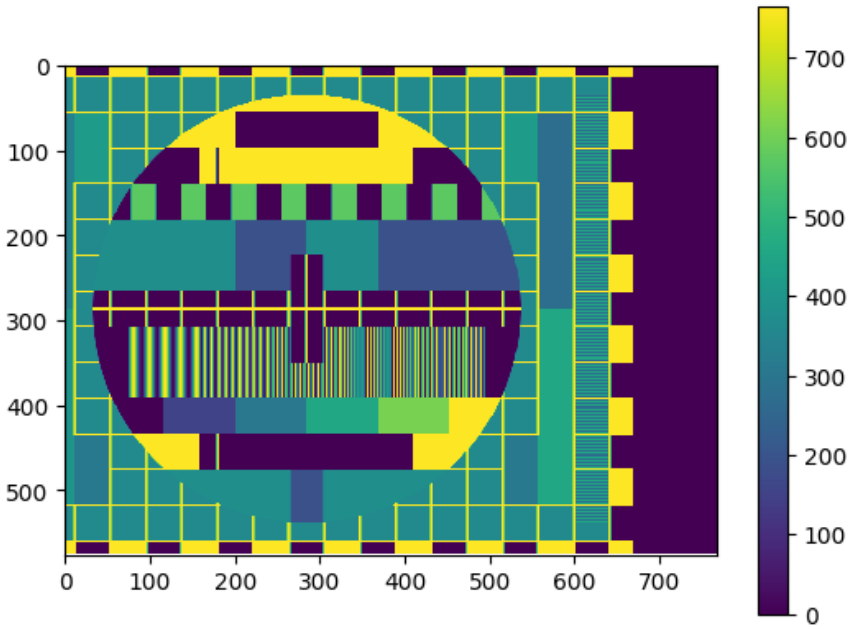
which is of course equal to taking the dot product of f, which is a single big vector f of length $u \cdot v \cdot w$ with a single big vector $t(i,j)$ of length $u \cdot v \cdot w$ as expressed below:

$$G(i,j) = \begin{bmatrix} F_1^\mathsf{T} & \dots & F_w^\mathsf{T} \end{bmatrix} \begin{bmatrix} I_1 \\ \vdots \\ I_w \end{bmatrix} = f^\mathsf{T} t_{i,j} \tag{8}$$

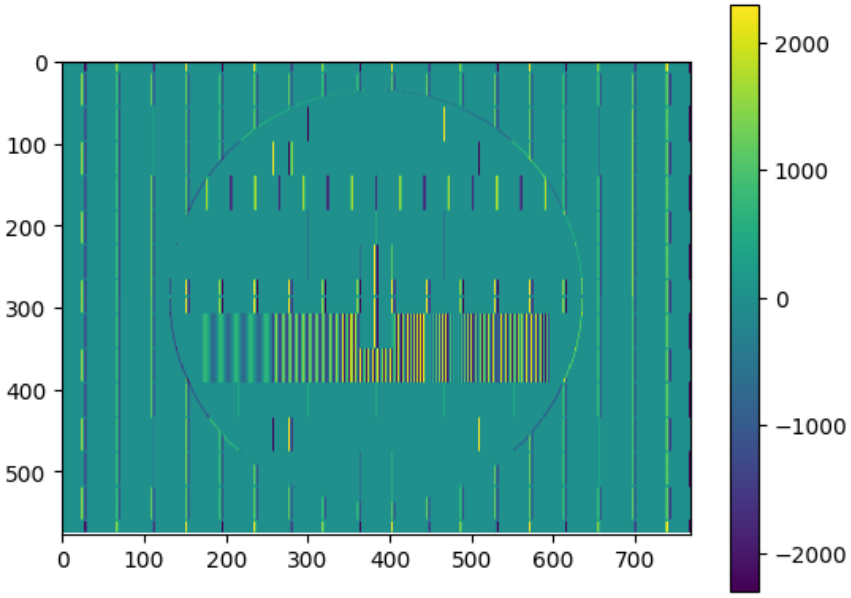
(iii) (code)



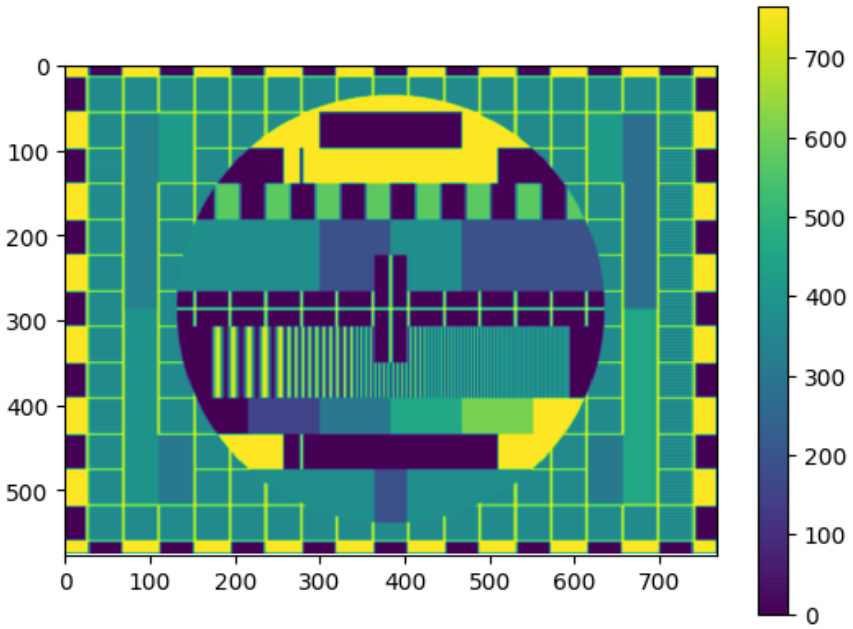
Correlation with `filt0`



Correlation with `filt1`



Correlation with `filt2`



Correlation with `filt3`

(iv) Naive implementation as above using a loop: Runtimes are 0.79, 1.36, 0.77, 0.80 seconds.

Vectorized implementation batching all image pixels: Runtimes are 0.13, 7.55, 0.12, 0.34 seconds.

(a) To answer the first hint, no, G does not have to run sequentially pixel by pixel. For a mono-channel image, each individual patch could be flattened and stacked into a $u \cdot v$ by $h \cdot w$ array and processed in parallel.

(b) To answer the second hint, the total number of addmul operations are $u \cdot v \cdot w \cdot h$ as we are applying a filter with a receptive field of u by v over a single-channel input of size w by h. If the filter could be expressed as an outer product, the total cost would be $(u + v) \cdot w \cdot h$.

(c) We could implement Winograd's minimal filtering algorithm that pre-computes intermediate values that depend only on kernel weights with the motivation of saving redundant computation.

(d) For large inputs, instead of direct convolution, we can first transform our input and kernel into the frequency domain via FFT (with suitable windowing), multiply the two signals, and then translate back into our original spatial domains via inverse DFT. This approach is an approximation, the accuracy of which depends on the Fourier window, but for large inputs, the performance gains are significant.

(v) **Observation:** any $m \times n$ matrix of rank 1 could be expressed as a vector outer product uv^T, because the column rank of any vector u has to be equal to 1. Let our real matrix be $F = uv^\mathsf{T}$.

Case: Most specific case where $u = v$.

To find the vectors, consider the LU decomposition of F:

Existence and uniqueness: Since $F = uu^\mathsf{T}$, by inspection F also has to be symmetric, and therefore square. If F is square, there exists an LU decomposition where $F = PLU$, but because F is rank-deficient, it cannot be positive definite, which means this decomposition is not unique.

Proposition: Because F is rank 1 and U is upper-triangular, the only nonzero row of U is its top row. Let b be this vector, and let a be the first column of the matrix PL. Then we have $F = ab^\mathsf{T}$. Each entry u_i in the vector u is then simply $\sqrt{a_i b_i}$. So $F = uu^\mathsf{T} = k\hat{f}\hat{f}^\mathsf{T}$ where k is a constant and $\hat{f}$ are unit vectors.

Example: (via `scipy.linalg.lu`)

$$
\begin{aligned}
F = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} &= \underbrace{\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{P} \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.5 & 0 & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 2 & 4 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}}_{U} = \underbrace{\begin{bmatrix} 0.5 & 1 & 0 \\ 1 & 0 & 0 \\ 0.5 & 0 & 1 \end{bmatrix}}_{PL} \underbrace{\begin{bmatrix} 2 & 4 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}}_{U} \\
&= \begin{bmatrix} 0.5 \\ 1 \\ 0.5 \end{bmatrix} \begin{bmatrix} 2 & 4 & 2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} = uu^{\mathsf{T}}
\end{aligned} \tag{9}
$$

Uniqueness: The above proves a slightly more general result than we set out to show. We have seen that if F is square and rank 1, it admits an LU decomposition such that U is also rank 1. Therefore for such F, there exists a unique decomposition $F = k\hat{f}\hat{f}^{\mathsf{T}}$.

Case: F not square, so $u \neq v$.

Observation: Since F is not square, it does not admit an LU decomposition or eigendecomposition. We approach this from $F = U\Sigma V^{\mathsf{T}}$ i.e. its singular value decomposition. It is known that for any real matrix, its SVD exists and is unique (up to sign change of U and V).

Since F is rank 1, there is only one nonzero eigenvalue λ in Σ. Let i be the position of λ down the diagonal of Σ. Let u be the i-th column of U and let v be the i-th row of V. Then $F = \lambda uv^{\mathsf{T}}$.

Example: (via `np.linalg.svd`)

$$
\begin{aligned}
F &= \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 6 & 8 \\ 3 & 6 & 9 & 12 \end{bmatrix} \\
&= \begin{bmatrix} -0.26 & -0.05 & 0.96 \\ -0.53 & -0.82 & -0.19 \\ -0.80 & -0.56 & -0.19 \end{bmatrix} \begin{bmatrix} 20.49 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} -0.18 & -0.36 & -0.55 & -0.73 \\ 0.97 & -0.03 & -0.22 & -0.07 \\ 0.12 & -0.63 & 0.72 & -0.26 \\ -0.06 & -0.69 & -0.36 & 0.63 \end{bmatrix} \\
&= 20.49 \begin{bmatrix} -0.26 \\ -0.53 \\ -0.80 \end{bmatrix} \begin{bmatrix} -0.18 & -0.36 & -0.55 & -0.73 \end{bmatrix} = \lambda uv^{\mathsf{T}}
\end{aligned} \tag{10}
$$

(vi) (code)

(vii) Convolution with a flipped filter in all its dimensions would produce the same output as correlation with an unmodified filter.

In other words,

$$
\begin{aligned}
G(i,j) &= \sum_{u=1}^{k} \sum_{v=1}^{l} F(u,v) \cdot I(i-u, j-v) \\
&= \sum_{u=1}^{k} \sum_{v=1}^{l} F(k-u, l-v) \cdot I(i+u, j+v)
\end{aligned} \tag{11}
$$

Problem 4: Template Matching

(i) (code)

(ii) (code)

(iii) Taking a page from data augmentation, we can:

(a) Alter the color of the filter to improve matching for color variations.

(b) Perform rotation on the filter to account for rotated candidates.

However, unlike a neural net implementation that learns and summarizes individual filters, our runtime would increase linearly with the number of filters we are applying on a given image. The upside is that this process is embarrasingly parallel.

Problem 5: Stop Sign Detection and FSM in ROS

(i) `supervisor.py` publishes two messages on the following prefixes:

(a) `/cmd_pose`, type `geometry_msgs.msg.Pose2D`

(b) `/cmd_vel`, type `geometry_msgs.msg.Twist`

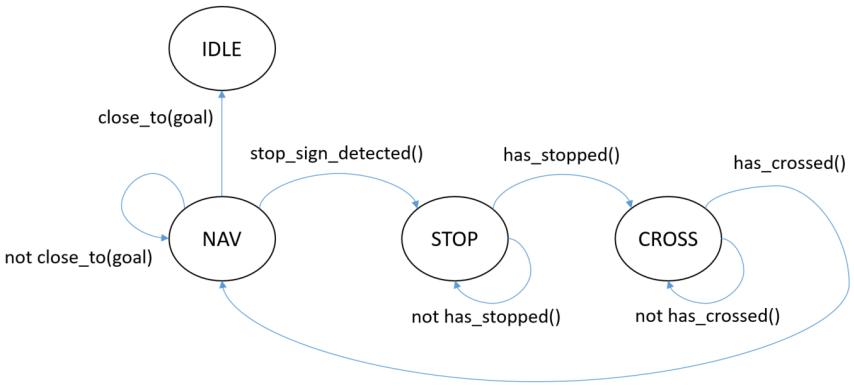
(ii) (code)

(iii) (code)

(iv) (code)

(v) null

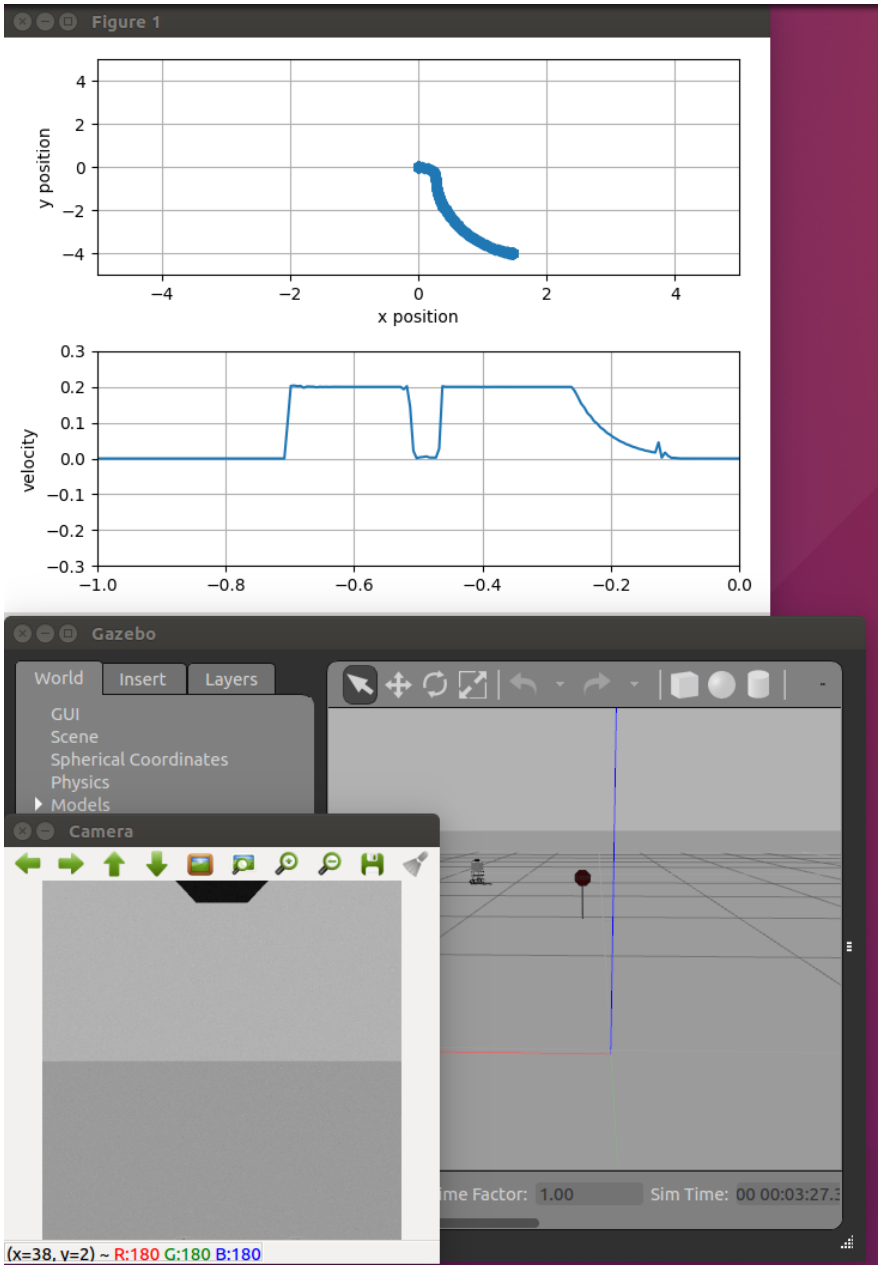
(vi)



FSM of `supervisor.py`

(vii) (code)

(viii)

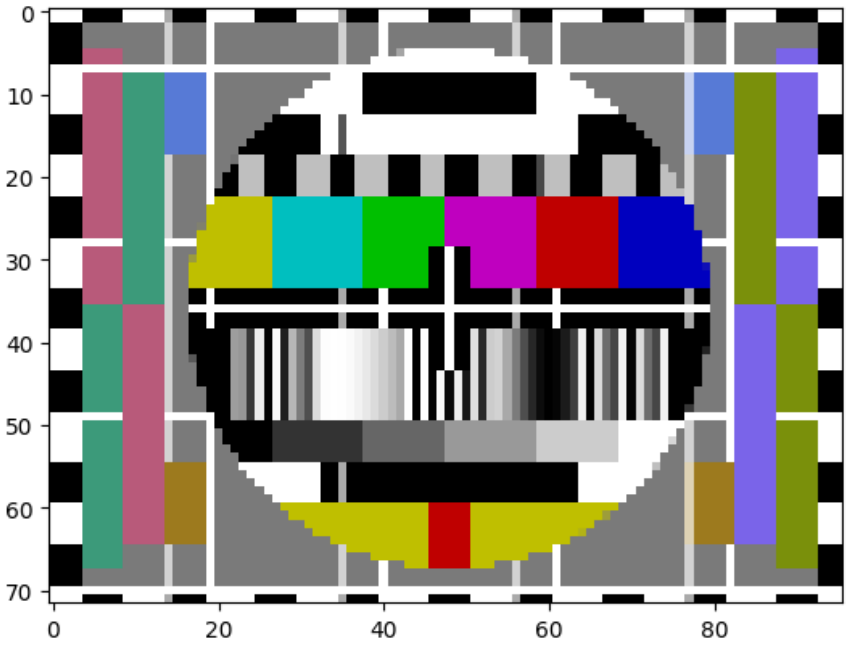


Velocity profile of Turtlebot model 'burger' under state machine policy.

Extra Problem: Image Pyramids

(i) (code)

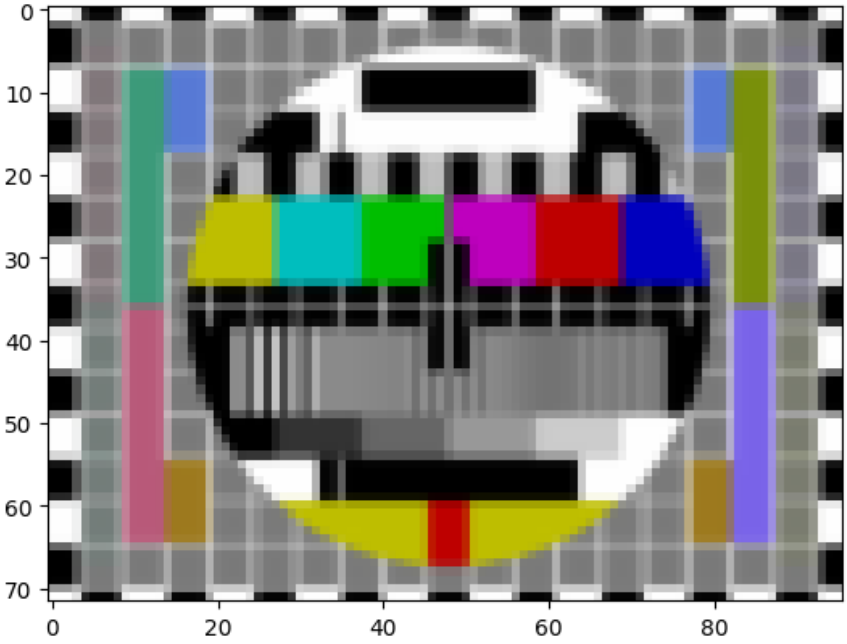
(ii) We have lost high-frequency information as a by-product of our sampling process, specifically the signal or gradient between each adjacent pixel. Notice that some of the vertical lines are completely missing.



1/8-sized image by discarding every other row and column 3 times.

(iii) (code)

(iv) The Gaussian blur 'expanded' high frequency information along the x and y axes, so we preserve some component of those signals when sampling the image.



1/8-sized image with Gaussian blur (5x5 kernel) applied before discarding data.

(v) (code)

(vi) We have simply expanded each pixel by 8x along each dimension. To look at it another way, we have created 8x8 identical subpixels within each original pixel.

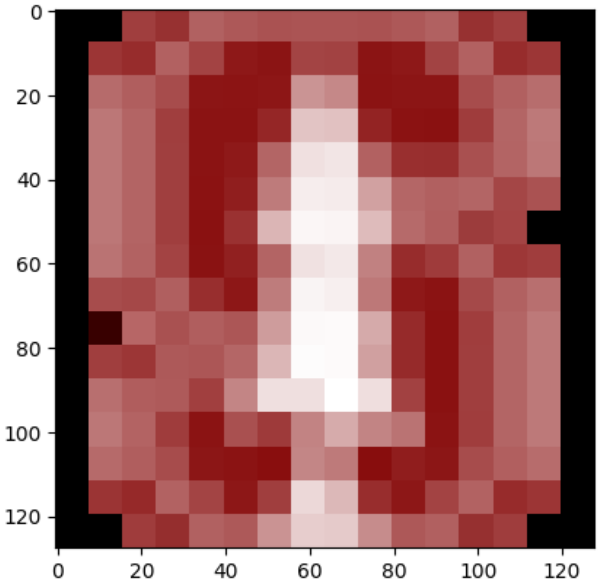


Image expanded 8 times without bilinear interpolation.

(vii) (code).

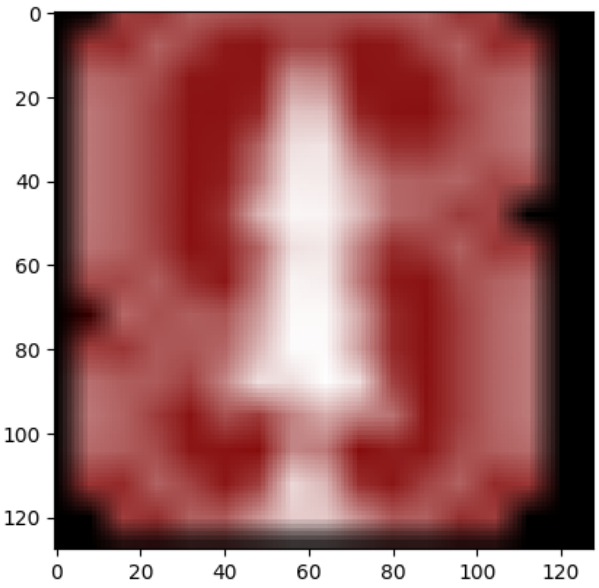


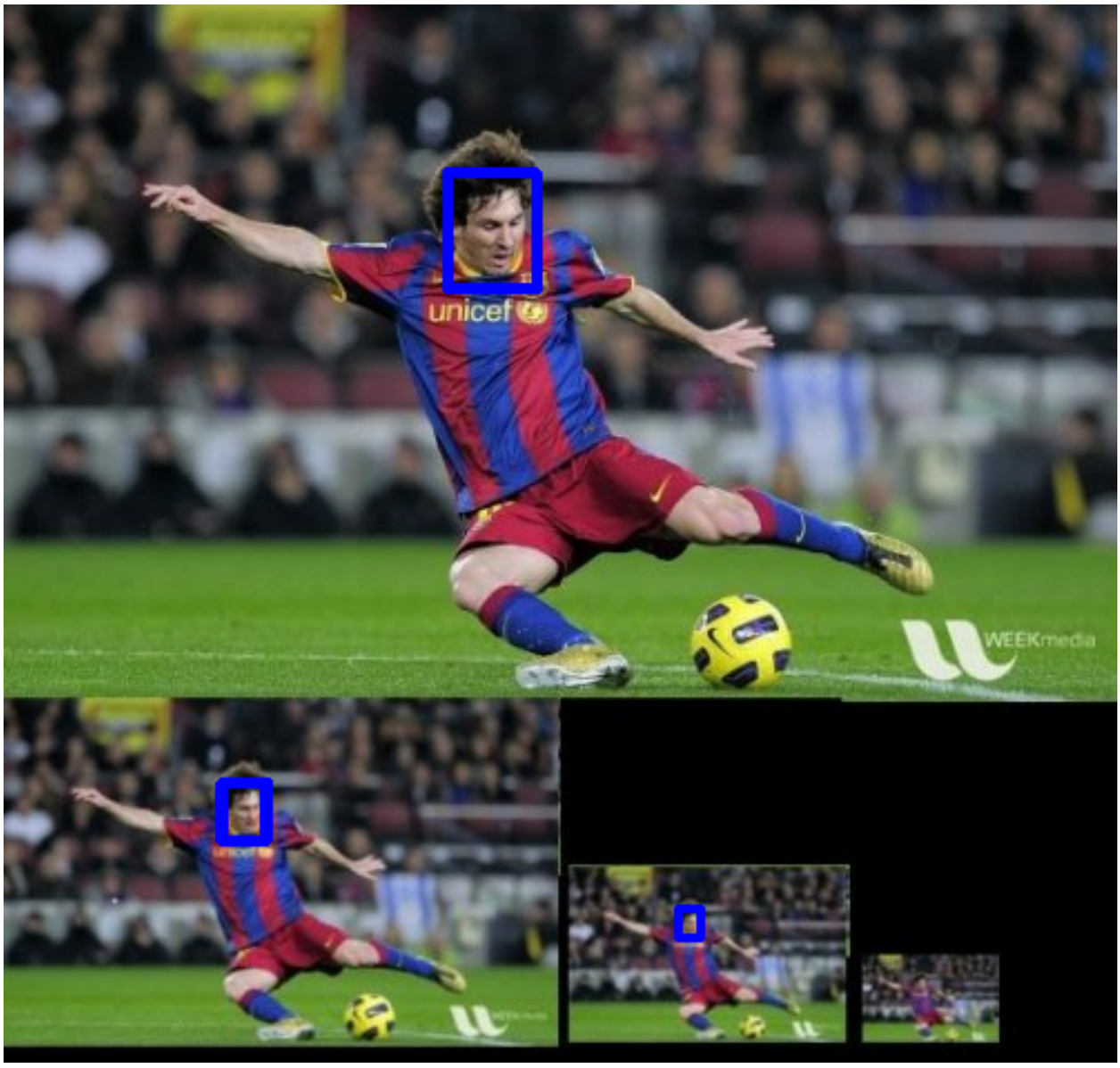
Image expanded 8 times with bilinear interpolation.

Comparing the images: the intensity differences between the pixel regions are less abrupt.

Informal explanation: I'm going to explain this in terms of a 2x upscaling. From the way we've expanded the image, and how OpenCV zero-pads the boundaries, we have 'islands' of pixels spread

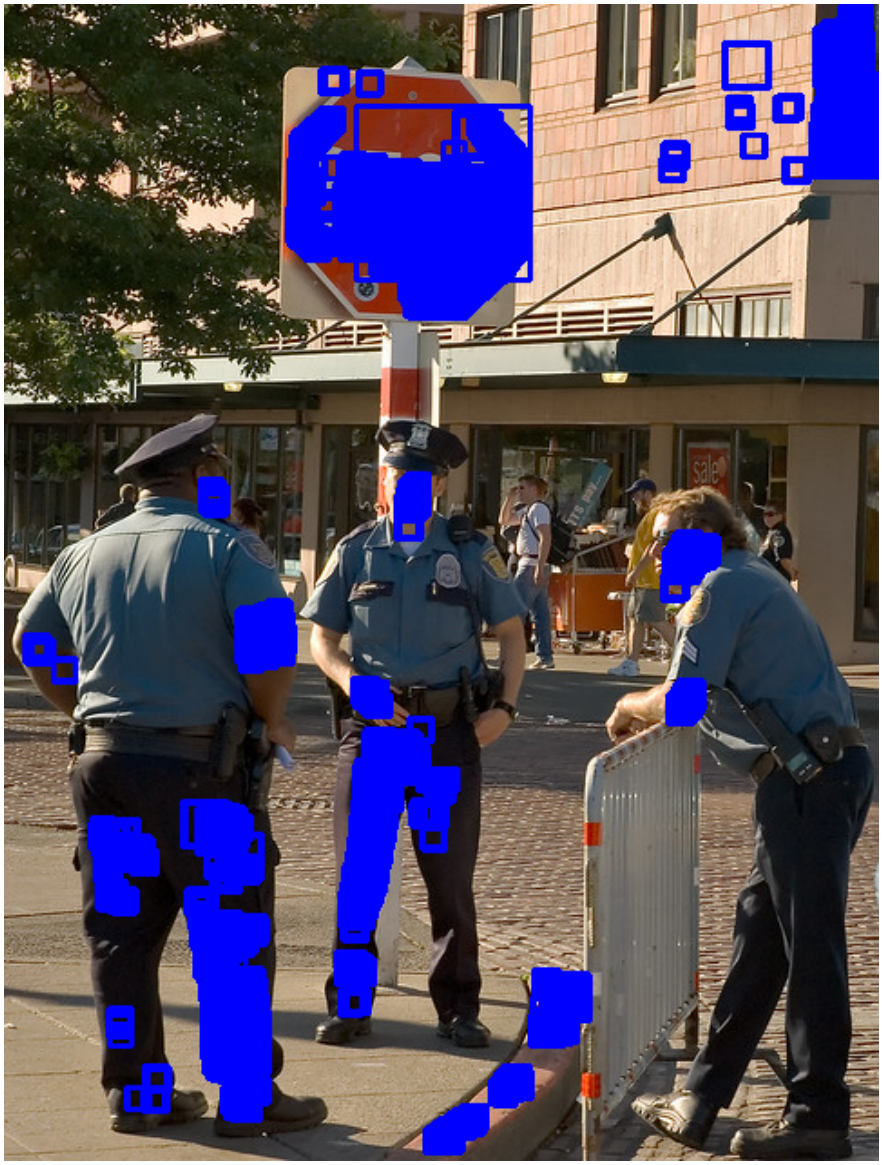
apart by exactly the same distance which we want to expand the image by. Also, normalization works out as a sanity check: The kernel for 2x2 upscaling sums to 4. 'Striding' or expanding the image as described for a 2x2 upscale effectively divides each region by 4. When viewed along these 'islands' of original pixel values, the tent function is doing discrete linear interpolation of values in between each island. Hence bilinear interpolation, because we have two axes x and y. Without loss of generalization, this applies to any upscaling which is a factor or 2.

(viii) (code).



Bounding boxes with threshold = 0.93

(ix) The template only matches the original image exactly, which is the second stop sign; it is not robust to intra-class variation. The template also returns many false positives; its specificity is low.



Bounding boxes at default threshold on `stop1`.

Bounding boxes at default threshold on `stop2`.

Bounding boxes at default threshold on `stop3`.

Bounding boxes at default threshold on `stop4`.

Bounding boxes at default threshold on `stop5`.