Visual Tracking Using Matlab

Shamma Alblooshi
Computer and Electrical Engineering
Khalifa University
100045387@ku.ac.ae / shamma.alblooshi@tii.ae

I. Introduction

Spike coding is a foundational concept in neuromorphic computing, enabling the conversion of traditional sensor signals into streams of spikes for use in spiking neural networks (SNNs). This process is critical in robotics and AI applications where efficient and robust signal representation is required. Spike coding schemes can be broadly categorized into population coding, rate coding, and temporal coding.

Population coding assigns a distribution of responses across a group of neurons, with each neuron representing a specific part of the input signal. Examples include position coding and Gaussian Receptive Fields (GRF), where neurons respond to the input signal based on predefined spatial distributions. This method is effective for encoding continuous signals and supports precise reconstruction when sufficient neurons are available.

Rate coding represents information through the spiking frequency of neurons. Algorithms like Ben's Spike Algorithm (BSA) and the Hough Spike Algorithm (HSA) utilize convolution-based methods to determine spike timings based on signal thresholds. While rate coding is robust to noise and effective for certain tasks, it can be computationally intensive and less efficient due to the high number of spikes generated.

Temporal coding encodes information in the precise timing of spikes, offering greater capacity for representing temporal variations. Algorithms such as the Temporal-Based Representation (TBR) and Step Forward (SF) models rely on signal changes to trigger spikes. The Moving Window (MW) model further incorporates adaptive baselines, improving reconstruction accuracy by adjusting for local signal variations.

Each spike coding scheme balances trade-offs between computational efficiency, signal reconstruction quality, and spiking efficiency. The choice of coding depends on the application, sensor data characteristics, and computational constraints, with each scheme offering unique advantages tailored to different neuromorphic and robotic contexts

In this report, we explore the application of various spike coding methods to convert video streams into discrete event representations, highlighting the strengths and limitations of each approach. The methods considered include population coding, rate coding, and temporal coding, which provide different mechanisms to encode sensor data into sparse spike-based representations suitable for neuromorphic processing.

After converting the video streams into discrete event streams using these spike coding techniques, we reconstruct the original videos from the encoded data. To evaluate the effectiveness of the reconstructed streams, we apply the Kalman filter to track objects in the videos. The Kalman filter, a widely used algorithm for object tracking, enables us to assess the quality and fidelity of the reconstructed videos, comparing their tracking accuracy and robustness to the original videos.

This study provides insights into how different spike coding methods impact discrete event representation and subsequent processing tasks such as tracking. By applying the Kalman filter on the reconstructed videos, we bridge the gap between neuromorphic event-based representations and traditional signal processing methods, demonstrating the practical implications of spike coding in real-world applications.

The objectives of this report are:

- To create a pipeline for the conversion of video data into neuromorphic-inspired event streams.
- To implement Kalman Filtering tracking algorithms using MATLAB for the output from above
- To evaluate the accuracy and robustness of these methods under challenging scenarios, including occlusion and rapid object motion.

A. Kalman Filters

Kalman filtering, or linear quadratic estimation, is a robust process in statistics and control theory that progressively enhances estimates of unknown variables over time, even in the face of statistical noise and measurement errors. Utilizing a sequence of time-stepped observations, the Kalman filter constructs a joint probability distribution among variables, yielding optimal estimates that minimize mean squared error. This recursive filtering technique, devised by Rudolf E. Kálmán, is especially proficient for applications in the guidance, navigation, and control of dynamic systems, including aircraft, spacecraft, and robotic motion.

The Kalman filter works in a two-step process: prediction and update. In the prediction step, it makes an extrapolation of the current state and uncertainty forward in time using a known model of the system dynamics. Upon the availability of a new measurement, the filter then switches to the update step, where the estimate gets updated by a weighted average that combines or reconciles the predicted state with the measurement it has to consider, taking into account their respective uncertainties. The Kalman filter is thus suited for real-time applications because it needs only the most recent estimates of state and uncertainty to make the next prediction if data arrive in a sequential manner.

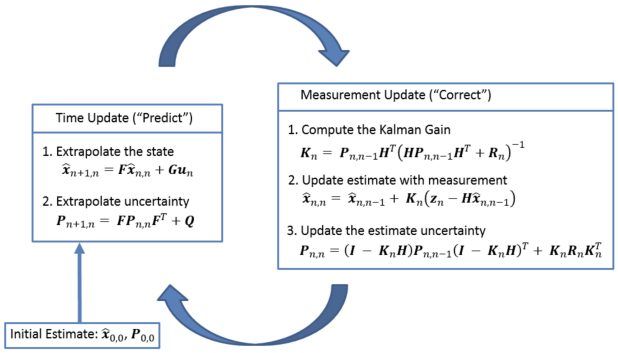

Fig. 1. Kalman filter equations [2]

Kalman filtering assumes the errors to follow a normal (Guassian) distribution , and under this assumption, the Kalman filter is the optimal estimator in a minimum mean-square sense. However, even for some non-Gaussian noise distributions, it can still work quite well. While the algorithm is inherently suited for linear systems, it has been extended to deal with nonlinear models through variations like the Extended Kalman Filter and Unscented Kalman Filter [1]. Figure 1 shows the Kalman filter equations inspired by [2].

Where:

Term	Name
x	State Vector
z	Measurements Vector
F	State Transition Matrix
u	Input Variable
G	Control Matrix
P	Estimate Covariance
Q	Process Noise Covariance
R	Measurement Covariance
w	Process Noise Vector
v	Measurement Noise Vector
H	Observation Matrix
K	Kalman Gain
n	Discrete-Time Index

II. Methodology

This section outlines the methodology for converting video frame data into discrete event streams using spike coding, followed by the implementation of visual tracking using Kalman Filtering. The approach leverages neuromorphic sensing techniques for event encoding and employs MATLAB functions for tracking.

A. Converting Video Image frame data to Discrete Event Video Streams Using Temporal Coding

This subsection uses the temporal coding method, specifically the step forward method (it is explained in detail below). The first part of the implementation is to load the video file (video_122233.MP4) using MATLAB's VideoReader object, which allows frame-by-frame access. We then set a limit of how many frames will be processed at a time by the frameProcessLimit variable. Next, the stepForwardThreshold defines the threshold for detecting changes in pixel intensity when encoding frames into events. The targetFrameRate specifies the desired frame rate for processing the video. Whereas the outputSize determines the resolution of the output 3 videos (original, discrete and reconstructed). In addition, I use the function Videowriter, to create an object named compositeVideoWriter. 'MPEG-4' specifies the format for the video, which is a widely used and efficient video compression standard. This contains original, encoded, and reconstructed frames combined into a single composite frame. The FrameRate property is set to targetFrameRate, which specifies how many frames per second (fps) the video will play. The open function prepares the VideoWriter object for writing data.

```
  % Load video data
videoFile = 'video_122233.MP4';
videoReader = VideoReader(videoFile);
frameProcessLimit = 10; % Number of
    frames to process at a time
stepForwardThreshold = 30; % Encoding
    threshold parameter
targetFrameRate = 10;
outputSize = [1280, 720]; % Desired
    output resolution (Height x Width)
compositeVideoWriter = VideoWriter('
    composite_video.mp4', 'MPEG-4');
compositeVideoWriter.FrameRate =
    targetFrameRate;
open(compositeVideoWriter);
```

After that, I initialize memory buffers for storing. The originalFrames are original grayscale frames. The encodedFrames are the discrete events. The decodedFrames are the reconstructed frames from the encoded data.

```
% Buffers for processing
originalFrames = zeros(videoReader.Height
    , videoReader.Width, frameProcessLimit
    , 'double');
encodedFrames = zeros(videoReader.Height,
     videoReader.Width, frameProcessLimit,
     'double');
decodedFrames = zeros(videoReader.Height,
     videoReader.Width, frameProcessLimit,
     'double');
```

The next step is processing the video in chunks. We first define a processing loop, that has a while loop. This while loop processes the video in chunks of up to the maximum number of frames defined as maxFrames. In addition, currentChunkSize dynamically adjusts if the remaining frames are fewer than frameProcessLimit.

```
while framesProcessed < maxFrames
    % Define the current chunk size
    currentChunkSize = min(
        frameProcessLimit, maxFrames -
        framesProcessed);
```

```
    originalFrames = zeros(videoReader.
        Height, videoReader.Width,
        currentChunkSize, 'double'); %
        Reset for the new chunk
```

After that, we proceed to extract frames. SkipFrames ensures only frames matching the targetFrameRate are processed. In addition, frames are converted to grayscale using rgb2gray to simplify encoding (this was done in the original implementation).

```
    % Extract frames at the target frame
        rate
    for frameIndex = 1:currentChunkSize
        for skipCount = 1:skipFrames
            if hasFrame(videoReader)
                grayscaleFrame = rgb2gray
                    (readFrame(videoReader
                    ));
            end
        end
        originalFrames(:, :, frameIndex)
            = grayscaleFrame;
    end
```

Next, we proceed to the encoding and decoding. This part is also from the original implementation. First, we encode the frames into events. This is done by processing each pixel independently. StepForwardEncoding converts pixel intensity changes into spike events. StepForwardDecoding reconstructs the original pixel intensities from the encoded data. These two functions are from the paper given (both are explained in detail below)

```
    % Encode and decode the frames
    for row = 1:videoReader.Height
        for col = 1:videoReader.Width
            pixelData = squeeze(
                originalFrames(row, col,
                1:currentChunkSize));
            [encodedFrames(row, col, 1:
                currentChunkSize),
                initValue] =
                StepForwardEncoding(
                pixelData,
                stepForwardThreshold);
            decodedFrames(row, col, 1:
                currentChunkSize) =
                StepForwardDecoding(
                encodedFrames(row, col, 1:
                currentChunkSize),
                stepForwardThreshold,
                initValue);
        end
    end
```

The StepForwardEncoding function which is shown below, is used for transforming a series of pixel intensity values into a discrete event stream. The first input to the function is pixelSeries, which is a 1D array representing the intensity values of a pixel over time. The second input is the threshold, which is the value that determines the sensitivity of the encoding process. For this, the changes in pixel intensity must exceed this value to trigger a spike event. The output of this function is encodedData, which is a 1D array of the same size as pixelSeries, where each element represents a spike event: +1 for a positive spike (increase in intensity), -1 for a negative spike (decrease in intensity) and 0 for no significant change. The second output is the startPoint, which is the initial value of the pixel intensity, used as the baseline for the first data point.

First we define the variables startPoint (first value of pixelSeries), numPoints (total number of intensity values in pixelSeries), encodedData (array initialized to zeros, which will store the encoded spike event) , baseline (reference intensity value, initialized to the first pixel intensity ,then is updated during the encoding process). It then loops through the pixel series , but starting from the second value (k = 2) because the first value (startPoint) is used as the baseline.

The positive Spike (+1) is triggered if the current pixel intensity (pixelSeries(k)) exceeds the baseline by more than threshold. The baseline is incremented by threshold to account for the spike. The negative Spike (-1) is triggered if the current pixel intensity is lower than the baseline by more than threshold. The baseline is decremented by threshold to account for the spike. The no Spike (0) is triggered if the intensity change does not exceed the threshold in either direction, hence no spike is emitted, and the baseline remains unchanged. Finally, it returns the spike events (encodedData) and the starting point (startPoint), which serves as the initial baseline.

```
function [encodedData, startPoint] =
    StepForwardEncoding(pixelSeries,
    threshold)
     startPoint = pixelSeries(1);
     numPoints = length(pixelSeries);
     encodedData = zeros(1, numPoints);
     baseline = startPoint;

     for k = 2:numPoints
         if pixelSeries(k) > baseline +
             threshold
              encodedData(k) = 1;
              baseline = baseline +
                  threshold;
         elseif pixelSeries(k) < baseline
             - threshold
              encodedData(k) = -1;
              baseline = baseline -
                  threshold;
         end
     end
end
```

The StepForwardDecoding function reverses the encoding process performed by StepForwardEncoding, reconstructing the original pixel intensity values from the encoded spike events. It takes 3 inputs, encodedData which is the 1D array of spike events (+1, -1, 0) generated during encoding. The next input, threshold is the same threshold value used during encoding, representing the intensity change for a spike event. The startPoint is the initial pixel intensity value used as a baseline for decoding. There is one output, reconstructedData which is 1D array representing the reconstructed pixel intensity values over time. We first initialize the variables numPoints (total number of spike events in encodedData), reconstructedData (array initialized to zeros, where the reconstructed intensity values will be stored) reconstructedData (first intensity value is set to startPoint) , which serves as the baseline for decoding. The for loop starts from the second element (k = 2) because the first value of reconstructedData is already initialized as startPoint. If encodedData(k) is positive, it indicates an increase in pixel intensity. Then the current intensity (reconstructedData(k)) is calculated by adding the threshold to the previous intensity value (reconstructedData(k-1)). If encodedData(k) is negative, it indicates a decrease in pixel intensity. Then the current intensity is calculated by subtracting the threshold from the previous intensity value. If encodedData(k) is zero, the intensity value remains unchanged from the previous value (reconstructedData(k-1)).

```
function reconstructedData =
    StepForwardDecoding(encodedData,
    threshold, startPoint)
     numPoints = length(encodedData);
     reconstructedData = zeros(1,
         numPoints);
     reconstructedData(1) = startPoint;

     for k = 2:numPoints
         if encodedData(k) > 0
             reconstructedData(k) =
                 reconstructedData(k-1) +
                 threshold;
         elseif encodedData(k) < 0
             reconstructedData(k) =
                 reconstructedData(k-1) -
                 threshold;
         else
             reconstructedData(k) =
                 reconstructedData(k-1);
         end
     end
end
```

After that, for visualization and saving we proceed to resizing the frames. This is done by resizing frame to outputSize for consistency in visualization and saving. In addition, spikePlot converts encoded data into a visual format for inspection.

```
    % Save and visualize frames
    for frameIndex = 1:currentChunkSize
        resizedOriginal = imresize(
            originalFrames(:, :,
            frameIndex), outputSize);
        resizedDecoded = imresize(
            decodedFrames(:, :, frameIndex
            ), outputSize);
        encodedVisualization = spikePlot(
            encodedFrames(:, :, frameIndex
            ));
        encodedVisualization = imresize(
            encodedVisualization,
            outputSize);

        % Convert spike image to RGB if
            not already
        if size(encodedVisualization, 3)
            == 1
            encodedVisualization =
                ind2rgb(uint8(
                encodedVisualization *
                255), jet(256));
        end
```

The spikePlot function generates a visual representation of the encoded data (spike events) by mapping positive and negative spikes to colors in an RGB image. It takes as an input encodedData, a 2D matrix of spike events, where each element represents the spike value at a specific pixel. A positive spike is indicated by +1, a negative spike is indicated by -1 and 0 indicates no spike. The output visualization is a 3D RGB matrix where positive spikes (+1) are represented as blue, negative spikes (-1) are represented as red and no spikes (0) are represented as black (no color).

The size of the input matrix encodedData is determined using size, returning the number of rows (rows) and columns (cols). A 3D array visualization is initialized with zeros, representing an RGB image, first dimension is the rows, second dimension is the columns, and third dimension is RGB channels (Red, Green, Blue). It then loops through very pixel in the encodedData matrix, r is the row index and c is the column index. If the spike is positive (+1), the blue channel (visualization(:, :, 3)) is set to 255, making the pixel appear blue. If the spike is negative (-1), the red channel (visualization(:, :, 1)) is set to 255, making the pixel appear red. If there is no Spike (0), the pixel remains black ([0, 0, 0]), as no color is assigned.

```
function visualization = spikePlot(
    encodedData)
     [rows, cols] = size(encodedData);
     visualization = zeros(rows, cols, 3);
     for r = 1:rows
         for c = 1:cols
             if encodedData(r, c) > 0
                 visualization(r, c, 3) =
                     255; % Blue for
```

```
                    positive spikes
            elseif encodedData(r, c) < 0
                visualization(r, c, 1) =
                    255; % Red for
                    negative spikes
            end
        end
    end
end
```

Next, we create a compositeFrame that combines original grayscale frame (left), encoded spike visualization (center). and reconstructed frame (right).

```
        % Create a composite frame
        compositeFrame = uint8(zeros(
            outputSize(1), outputSize(2) *
             3, 3));
        compositeFrame(:, 1:outputSize(2)
            , :) = repmat(uint8(
            resizedOriginal), [1, 1, 3]);
        compositeFrame(:, outputSize(2) +
             1:2 * outputSize(2), :) =
            uint8(255 *
            encodedVisualization);
        compositeFrame(:, 2 * outputSize
            (2) + 1:end, :) = repmat(uint8
            (resizedDecoded), [1, 1, 3]);
```

The composite frames are then written in composite_video.mp4. Finally, the frame count is incremented for the processing of the while loop.

```
        % Write frames to respective
            video files
        writeVideo(compositeVideoWriter,
            compositeFrame);

    % Update processed frame count
    framesProcessed = framesProcessed +
        currentChunkSize;
end
```

B. Converting Video Image frame data to Discrete Event Video Streams Using Rate Coding

This section explains the implementation of ben spike algorithm. I will only be explaining the decoding and encoding functions as it the processing of the videos is the same here, and the only thing that differs is the Ben Spike algorithm encoding and decoding process.

The function first calculates the minimum value (shift) of the input signal and subtracts it from all values in the signal. This ensures that the input is shifted to a non-negative range, simplifying subsequent computations. After that, the length of the input (L) and the FIR filter (F) are determined. A zero vector spikes of the same length as the input is created to store the generated spikes. After that, we have the iterative spike encoding. For each position in the input (up to L-F to avoid exceeding bounds) we calculate two error metrics. The first one is the absolute error between the input segment and the FIR filter values. The second one is the absolute difference between the input values and a baseline (previous input values). If err1 is smaller than err2 multiplied by the threshold, a spike is generated at the current position (spikes(i) = 1). After generating a spike, the FIR filter's influence is subtracted from subsequent values in the input, simulating the effect of the spike on the system.

The resulting spike sequence (spikes) contains sparse binary events indicating when the input signal was approximated by the FIR filter within the threshold bounds. The function returns the spikes array and the shift value, which is needed for decoding and reconstructing the original signal.

```
function [spikes, shift] =
    BenSpikeEncoding(input, fir, threshold
    )
    L = length(input);
    F = length(fir);
    spikes = zeros(1,L);
    shift = min(input);
    input = input - shift;

    for i = 1:(L-F)
        err1 = 0;
        err2 = 0;
        for j = 1:F
            err1 = err1 + abs(input(i+j)
                - fir(j));
            err2 = err2 + abs(input(i+j
                -1));
        end
        if err1 <= err2*threshold
            spikes(i) = 1;
            for j = 1:F
                if i+j+1 <= L
                    input(i+j+1) = input(
                        i+j+1) - fir(j);
                end
            end
        end
    end

end
```

The decoding is then performed. The BenSpikeDecoding function reconstructs a signal from its spike-based representation by using an FIR (finite impulse response) filter and the shift value applied during encoding.

```
function signal = BenSpikeDecoding(spikes
    , fir, shift)
    signal = conv(spikes,fir) + shift;
    signal = signal(1:end-length(fir)+1);
```

```
end
```

C. Converting Video Image frame data to Discrete Event Video Streams Using Population Coding

The GaussianReceptFieldsEncoding function implements a population coding method based on Gaussian receptive fields to encode a 1D input signal into a spike representation. It takes as an input the 1D signal to be encoded and m the number of neurons in the population (determining the resolution of the encoding).

It first finds the length of the input signal (L). Next, it preallocates the spikes matrix to store the encoded spike outputs and a neuron_outputs vector to hold the activation values of the neurons for each input value. After that, we compute the minimum (min_input) and maximum (max_input) of the input signal. These values define the range of the input for mapping to the receptive fields. Next, for each neuron we find its receptive field that is defined by the mean and standard deviation. Furthermore, for each input value, we compute the Gaussian response. We then determine the neuron with the maximum response for the current input value (spikingNeuron) and generate a spike for that neuron in the spikes matrix. The function then returns the spike matrix (spikes) and the range (min_input, max_input) for potential reconstruction.

```
function [spikes,min_input,max_input] =
    GaussianReceptFieldsEncoding(input,m)

    L = length(input);

    spikes = zeros(L,m);
    neuron_outputs = zeros(1,m);

    min_input = min(input);
    max_input = max(input);

    for j = 1:L
        for i = 1:m
            mu = min_input + (2*i-3)/2*(
                max_input - min_input)/(m
                -2);
            sigma = (max_input - min_input
                )/(m-2);
            neuron_outputs(i) = normpdf(
                input(j), mu, sigma)/
                normpdf(mu, mu, sigma);
        end
        [~,spikingNeuron] = max(
            neuron_outputs);
        spikes(j,spikingNeuron) = 1;
    end


end
```

The decoding process converts the sparse spike representation back into a continuous signal using the parameters of the Gaussian receptive fields. For each timestep, the neuron that generated a spike is identified by locating the maximum value in the spike matrix. The index of the spiking neuron is then mapped back to its corresponding input value using the Gaussian receptive field configuration, which is defined by the minimum and maximum input values and the number of neurons. This mapping ensures that each spike directly translates to a specific value in the reconstructed signal, effectively recreating the original input signal from the spike-based representation. The process is efficient and leverages the predefined structure of the receptive fields to maintain fidelity in the reconstruction.

```
function signal =
    GaussianReceptFieldsDecoding(spikes,
    min_input,max_input)

    L = size(spikes,1);
    m = size(spikes,2);
    signal = zeros(1,L);
    for i = 1:L
        [~,index_neuron] = max(spikes(i
            ,:));
        signal(1,i) = min_input + (2*
            index_neuron-3)/2*(max_input -
             min_input)/(m-2);
    end


end
```

III. Kalman Filter

After converting the video to discrete event video stream, we apply Kalman filter on both the original video and the reconstructed videos from each method. We first initialize a vehicle detector using the Aggregate Channel Features (ACF) method pre-trained on front and rear vehicle views. The detector identifies vehicles in video frames and provides their bounding boxes. After that, we read the input video file frame-by-frame using vReader. We then use trackPlayer to display the video frames along with tracking annotations, which also use the position argument that defines the size and location of the display window.

```
detector = vehicleDetectorACF("front-rear
    -view");
vReader = VideoReader("video_122233.MP4")
    ;
%vReader = VideoReader("
    reconstructed_video.mp4");
trackPlayer = vision.VideoPlayer(Position
    =[700 400 700 400]);
```

We then define a multiObjectTracker, which is a MATLAB object for tracking multiple objects across video frames. The

FilterInitializationFcn specifies the Kalman filter initialization function, helperInitDemoFilter . The AssignmentThreshold is the maximum cost for associating detections with existing tracks. The DeletionThreshold is the minimum number of consecutive frames without detections before a track is deleted. The ConfirmationThreshold is the number of frames needed to confirm a track (e.g., 3 detections within 5 frames).

We initialize a Kalman Filter (trackingKF) for each object. This object is defined using a 2D constant velocity model of a state vector: [x, vx, y, vy] (position and velocity in 2D space). In addition, the measurement noise and state covariance are initialized also, but based on the detection.

```
tracker = multiObjectTracker(...
    FilterInitializationFcn=@
        helperInitDemoFilter, ...
    AssignmentThreshold=30, ...
    DeletionThreshold=15, ...
    ConfirmationThreshold=[3 5] ...
);

function filter = helperInitDemoFilter(
    detection)
    state = [detection.Measurement(1); 0;
        detection.Measurement(2); 0];
    stateCov = diag([detection.
        MeasurementNoise(1,1) ...
                    detection.
                        MeasurementNoise
                        (1,1)*100 ...
                    detection.
                        MeasurementNoise
                        (2,2) ...
                    detection.
                        MeasurementNoise
                        (2,2)*100]);

    filter = trackingKF('MotionModel','2D
        Constant Velocity', ...
        'State',state, ...
        'StateCovariance',stateCov, ...
        'MeasurementNoise',detection.
            MeasurementNoise);
end
```

We then define a while loop that reads each frame of the video using readFrame. For vehicle detection, we use detect to identify vehicles in the frame, showing bounding boxes and detection confidence scores. However, it filters out detections with confidence scores below 5%.

```
frameCount = 1;
while hasFrame(vReader)
    % Read new frame and resize it to the
        YOLO v2 detector input size
    frame = readFrame(vReader);

    % Detect vehicles in the frame and
        retain bounding boxes with greater
         than 5% confidence score
    [bboxes,scores] = detect(detector,
        frame);
    bboxes = bboxes(scores>5,:);
```

We then compute the centroids of the detected bounding boxes, which represent the objects' positions. Following that, we wrap each detection in an objectDetection, which includes Timestamp (frameCount) which is the current frame number. The Centroid Position which is the object's position (centroid of bounding box). The MeasurementNoise which is the uncertainty in the detection. Whearas the Object Attributes is additional information like bounding box dimensions.

```
centroids = [bboxes(:,1)+floor(bboxes
    (:,3)/2)bboxes(:,2)+floor(bboxes(:,4)
    /2)];
    % Formulate the detections as a list
        of objectDetection objects
    numDetections = size(centroids,1);
    detections = cell(numDetections,1);
    for i = 1:numDetections
        detections{i} = objectDetection(
            frameCount,centroids(i,:)',
            ...
            MeasurementNoise=
                measurementNoise, ...
            ObjectAttributes=struct(
                BoundingBox=bboxes(i,:)),
                ObjectClassID=1);
    end
```

The tracker then updates the tracks using new detections (detections) and current frame count (frameCount). The Kalman filter estimates the new object positions and maintains consistency across frames. The displayTrackingResults annotates the frame with bounding boxes and track IDs(with "predicted" label if the object wasn't detected in the current frame). It finally displays the annotated frame using videoPlayer.

```
 % Update tracks based on detections
    confirmedTracks = tracker(detections,
        frameCount);

    % Display tracking results and
        increase frame count by 1
    displayTrackingResults(trackPlayer,
        confirmedTracks,frame);
    frameCount = frameCount + 1;
end

function displayTrackingResults(
    videoPlayer,confirmedTracks,frame)
    % Annotates the frame with tracking
        results
```

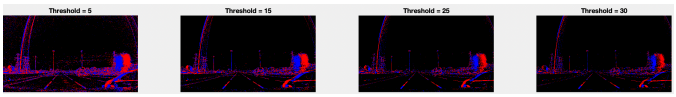

Fig. 2. Kalman filter

```
    % Draws bounding boxes for all
       confirmed tracks
    frame = insertObjectAnnotation(frame
       ,"rectangle",boxes,labels);
    videoPlayer.step(frame);
end
```

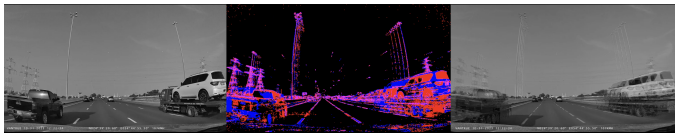

Fig. 3. Encoding/Decoding using Step Forwarding

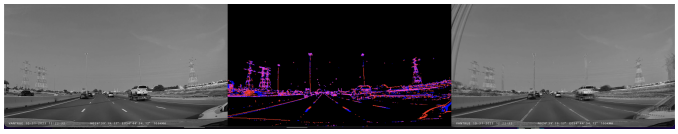

Fig. 4. Encoding/Decoding using Moving Window

IV. Discussion of Conversion to Discrete Event Video Streams

This section will discuss the outputs of the implemented codes.

Figure 2 shows the effect of varying the threshold parameter on the output of the Step Forward Encoding algorithm. As the threshold increases (from left to right: 5, 15, 25, 30), the density of spikes decreases. This indicates that higher thresholds make the algorithm less sensitive to small variations in the input signal, as it requires larger deviations to generate spikes. At a low threshold (e.g., 5), the output is more detailed, capturing finer changes in the scene. However, it also includes more noise. As the threshold increases, the encoding focuses on more prominent changes, filtering out minor details and noise. This demonstrates the trade-off between sensitivity and noise suppression in Step Forward Encoding. Selecting an optimal threshold depends on the desired balance between detail preservation and noise reduction for the specific application.

Figure 3 displays a comparison of three frames: the original grayscale frame, a spike visualization (encoded frame), and the reconstructed frame using the Step Forwarding. The left panel shows the original grayscale video frame captured from the input video. It retains all details of the scene, including textures, shadows, and high-frequency details such as trees and road markings. It serves as a baseline to evaluate the fidelity of the reconstructed frame. We can see that all details are clear and complete, with no artifacts or missing data. The sharpness of edges and the continuity of objects (e.g., cars, road lanes) are noticeable

The middle panel represents the spike encoding of the video frame. The red and blue colors indicate pixel intensity changes: , where blue represents the positive spikes (increase in pixel intensity), the red represents the negative spikes (decrease in pixel intensity) and black represents no significant intensity change. It highlights regions of the frame where significant changes occur, reducing redundant data. The spike visualization emphasizes edges and areas of rapid intensity changes, such as the outlines of vehicles, streetlights, and lane markers. Large uniform areas (e.g., sky and road) have minimal activity, demonstrating efficient data reduction. However, fine details, such as subtle textures, are lost due to the threshold-based encoding, but it is very minimal.

The right panel displays the frame reconstructed from the spike-encoded data. It approximates the original frame using the positive and negative spikes to adjust pixel intensities. This reconstructed data tests how well the event-based encoding preserves the original visual information. It shows that the overall structure of the scene is well-preserved. Larger objects like vehicles, lane markers, and streetlights are clearly identifiable.

The reconstruction output demonstrates noticeable imperfections, but these issues are primarily attributed to the reconstruction algorithm itself rather than the quality of the encoded data. The Step Forward Encoding effectively captures significant variations in the input, producing spike data that is detailed and representative of the original scene. The encoded data, as observed in the visualizations, retains crucial features of the input and exhibits minimal loss of information. However, the reconstruction algorithm introduces minor inaccuracies, likely due to its design or limitations in handling certain edge cases or transitions between spikes. Thus, while the encoding process is robust, the imperfections in the reconstructed frames highlight the need for refining the decoding algorithm to better translate the spike data into accurate reconstructions.

The second output that was analyzed was using the moving window algorithm which is also a temporal coding scheme. Figure 4 shows the output. Similar to the step-forwarding algorithm, the spike visualization accurately captures the dynamic changes in the scene, emphasizing edges and areas of significant contrast. The reconstructed frame on the right closely resembles the original, though some subtle artifacts are noticeable in areas of fine detail (e.g., power lines and distant vehicles). These minor reconstruction issues may stem from the decoding process rather than the encoded data, as the spikes appear to represent the original data effectively.

Figure 5 shows the implementation of the BSE method. The dense blue regions suggest that BSE is capturing numerous dynamic changes, potentially including finer details or noise. This method's emphasis on sensitivity to variations is apparent. The reconstructed frame (right image) demonstrates clear re-

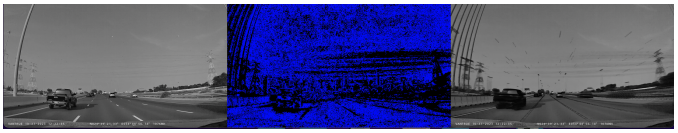

Fig. 5. Encoding/Decoding using BSE

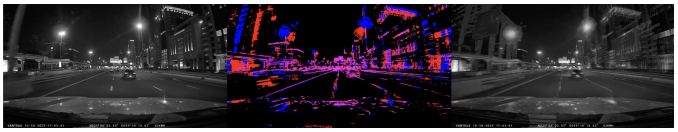

Fig. 6. Encoding/Decoding using SF2 on night video

tention of the overall scene structure, but it exhibits distortions, particularly in areas with fine details or rapid changes. For instance, the power lines and vehicle outlines are slightly skewed or smeared.

The final implementation was done using the GRF method, however the code provided in the toolbox was not working properly, and this was confirmed bythe other people in class. Hence, the comparison was done on all other methods.

Figure 6 shows the implementation of SF encoding applied to a night video sequence. This image demonstrates the application of Step Forward (SF) encoding to a night video. The spike encoding effectively captures significant intensity changes, such as those from streetlights, vehicle headlights, and illuminated buildings, with positive and negative spikes represented in red and blue, respectively. The sparsity of the spikes highlights the method's ability to focus on dynamic changes while ignoring less significant variations, which is particularly effective in low-light conditions. The reconstructed frame retains much of the structural detail from the original, especially in illuminated regions, though minor distortions and loss of finer details are evident in less dynamic areas. These limitations are attributed to the reconstruction algorithm rather than the encoding itself, which demonstrates robust performance in representing night scenes efficiently.

A. Comparitive Analysis of methods

The comparison between Step Forward (SF), Moving Window (MW), and Ben Spike Encoding (BSE) highlights distinct strengths and weaknesses across these methods in terms of spike representation and reconstruction quality.

Step Forward (SF) encoding emerges as the most effective method among the three. Its spike representation is well-distributed and interpretable, with distinct red (positive changes) and blue (negative changes) spikes. This sparse representation captures significant intensity changes efficiently, making it particularly suited for dynamic regions in the video. The reconstruction quality achieved through SF is superior, providing sharp edges and clear details in both static and dynamic areas. The encoding is sensitive to threshold adjustments, which allow it to balance noise and accuracy effectively. SF's simplicity and low computational complexity make it highly practical, but it may miss subtle variations in intensity due to its reliance on fixed thresholds.

Moving Window (MW) encoding, on the other hand, uses a sliding average to detect intensity changes. This method produces denser spike maps, resulting in smoother and less noisy representations compared to SF. However, the averaging effect can oversmooth rapid transitions, leading to a loss of fine details in the spike map. In terms of reconstruction, MW provides smoother but slightly blurred frames, as the averaging inherently sacrifices sharpness to reduce noise. This method is particularly suitable for scenarios with gradual changes in intensity or noisy data. However, it struggles to maintain the crispness and accuracy seen in SF reconstructions, making it less effective in dynamic or high-motion regions.

Ben Spike Encoding (BSE) stands out for its high sensitivity to subtle changes in intensity. The spike maps generated by BSE are dense and detailed, capturing fine-grained variations. While this level of detail is advantageous for precise intensity mapping, the dense representation can sometimes appear overly complex. In terms of reconstruction, BSE faces challenges due to limitations in its decoding process. Although the encoded data is of high quality, the reconstructed frames often suffer from inaccuracies, indicating the need for a more refined reconstruction algorithm. This makes BSE computationally intensive and less efficient compared to SF, despite its ability to capture intricate details.

In conclusion, Step Forward encoding offers the best overall performance, balancing quality, simplicity, and computational efficiency. It excels in scenarios requiring sharp and accurate reconstructions with minimal artifacts. Moving Window encoding is better suited for noisy environments, prioritizing robustness over detail. Meanwhile, Ben Spike Encoding is ideal for tasks requiring high sensitivity to subtle changes but is hindered by reconstruction challenges. These results emphasize that while each method has its niche applications, SF remains the most practical choice for general use.

V. Kalman filter and Discussion

Next, we applied the Kalman filter on the reconstructed video using SF since it performed better than all other methods.

As seen in Figure 6, it is clear that the tracker effectively detects and tracks multiple vehicles with high accuracy due to the richness of visual data in the original video. The reconstructed video is in grayscale, which removes color information. This can reduce detection accuracy for objects that rely on color-based differentiation.

In the case of the accuracy of the Kalman filter on the reconstructed video, bounding boxes are present. Still, they may slightly lag in precision due to potential artifacts or noise introduced during reconstruction. The reconstructed video suffers slight inaccuracies in bounding box placement and struggles with smaller or less distinct objects.

In the case of robustness, the kalman filter in the original video handles object motion and occlusions better because of its detailed information. However, in the reconstructed video it

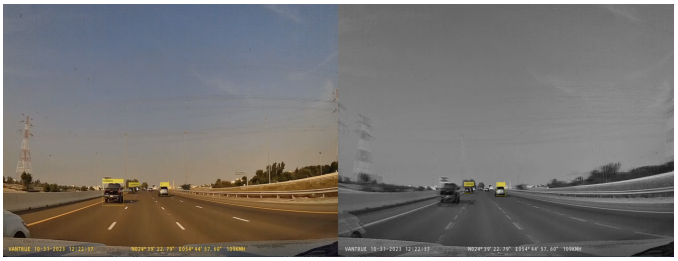

Fig. 7. Kalman filter on original and reconstructed using SF

Fig. 8. Kalman filter on original and reconstructed using SF night video

performs adequately for large, clear objects but struggles with rapid motion or fine details due to artifacts and noise. This is due to the reconstructed video itself rather than the encoding algorithm.

In addition, we performed the Kalman filter on the night video as shown in Figure 8. It illustrates the performance of the Kalman filter on a nighttime driving video. Despite the challenges of low-light conditions, the filter effectively tracks vehicles, albeit with slight inconsistencies in the predicted positions caused by dynamic lighting, reflections, and noise inherent in nighttime scenes. Compared to the original video, the reconstructed view emphasizes the predicted motion patterns while filtering out some of the visual distractions like headlights and reflective surfaces. However, the lack of consistent lighting in the original video sometimes impacts the accuracy of bounding box placement and tracking continuity in the reconstructed frame. Overall, this showcases the robustness of the Kalman filter in processing noisy data, although it demonstrates room for improvement in handling challenging nighttime conditions.

VI. Conclusion

This report has explored the application of various spike coding methods, including Step Forward (SF), Moving Window (MW), and Ben Spike Encoding (BSE), to convert video data into discrete event streams. Each method demonstrated unique strengths and limitations, with SF emerging as the most effective due to its balance of computational simplicity, accurate reconstruction, and efficient encoding of dynamic changes. MW encoding provided robustness in noisy environments but exhibited a trade-off in sharpness and fine detail preservation. BSE showcased high sensitivity to subtle changes, although its dense spike representation and decoding limitations introduced challenges in reconstruction quality.

The integration of the Kalman filter for tracking performance evaluation further highlighted the encoding methods' effectiveness in retaining critical features for visual analysis. The SF-based encoding demonstrated superior results in both daytime and nighttime videos, delivering high fidelity and robust tracking despite minor artifacts in reconstructed frames. Nighttime tracking revealed the limitations of reduced color information and dynamic lighting conditions but underscored the Kalman filter's robustness in managing noisy and complex data.

This study underscores the importance of selecting appropriate spike coding techniques based on application needs and highlights the potential of neuromorphic-inspired methods for real-world visual processing tasks. Future work should focus on refining reconstruction algorithms to fully leverage the potential of encoded spike data and further enhance the practical utility of event-based video processing.

References

[1] Yan Pei, Swarnendu Biswas, Donald S. Fussell, and Keshav Pingali. An elementary introduction to kalman filtering, 2019.

[2] A. Becker. *Kalman Filter from de ground up*. Alex Becker, 2023.