Real-Time Neural Spike Detection

Neuralink, US 2021/0012909 A1, Filed 9/9/2020, Published 1/21/2021

Notebook written by Michael Zhou (mgz2112@columbia.edu)

Background

- **Key information** of neural activity can be identified based on neuron spikes - action potential in neuron
- **Spikes** - characteristic rises in voltage across plasma membrane in cell, caused by depolarization + repolarization in membrane
- **Existing methods** include **total voltage exceeding some threshold** (produces **lots of false positives**) and **machine learning** techniques (**expensive, very power-consuming**).

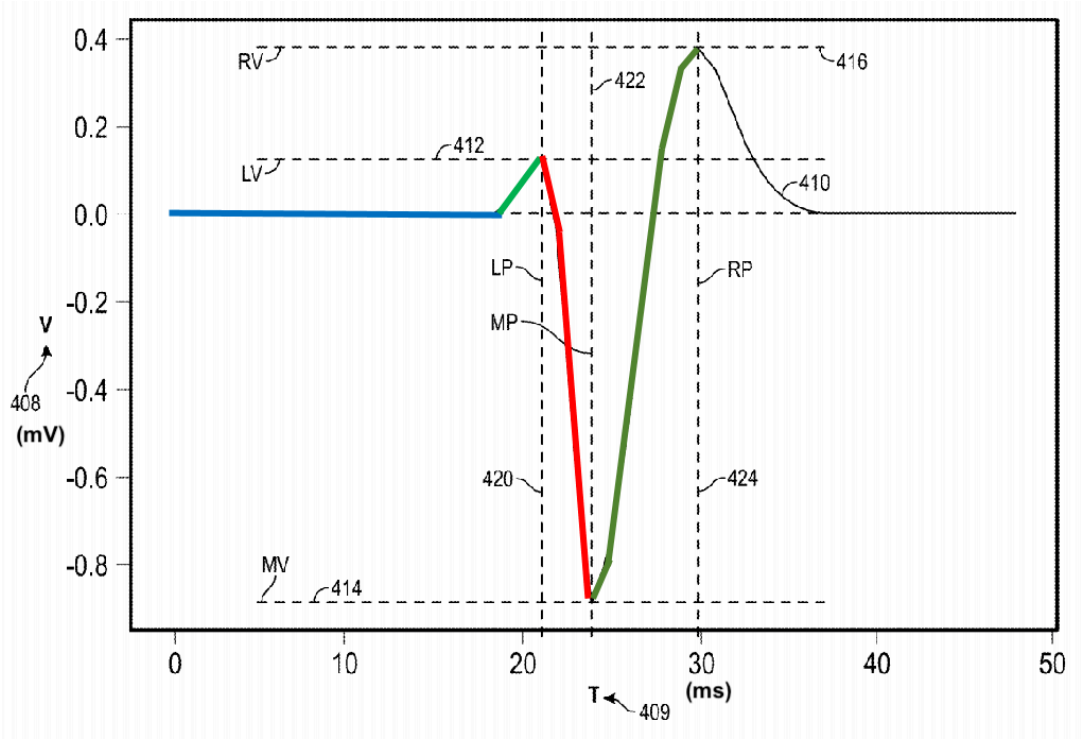
Goal

This notebook provides an implementation of the **neural spike detection algorithm** described in US patent application number **US 2021/0012909 A1** from **Neuralink**, titled **"Real-Time Neural Spike Compression"**.

What is a spike?

A neuron spike is a neural signal with **characteristic rises and falls**, characterized by the following:

- Start out with **initial resting value**
- **First positive change** in amplitude
- **Reduction** in amplitude below initial resting value
- **Second positive change** in amplitude greater than first positive change



Note: This notebook is to be used as a reference only

This notebook is used for **demonstration, education, and reference purposes only**.

Since this implementation involves using **fixed-point** arithmetic, **in practice**, the following steps shown in this patent is **implemented mostly in hardware**. We are just simply displaying these steps in the notebook for demonstration purposes.

Overall Flowchart

Note: May be performed in **some different order** or in **parallel**!

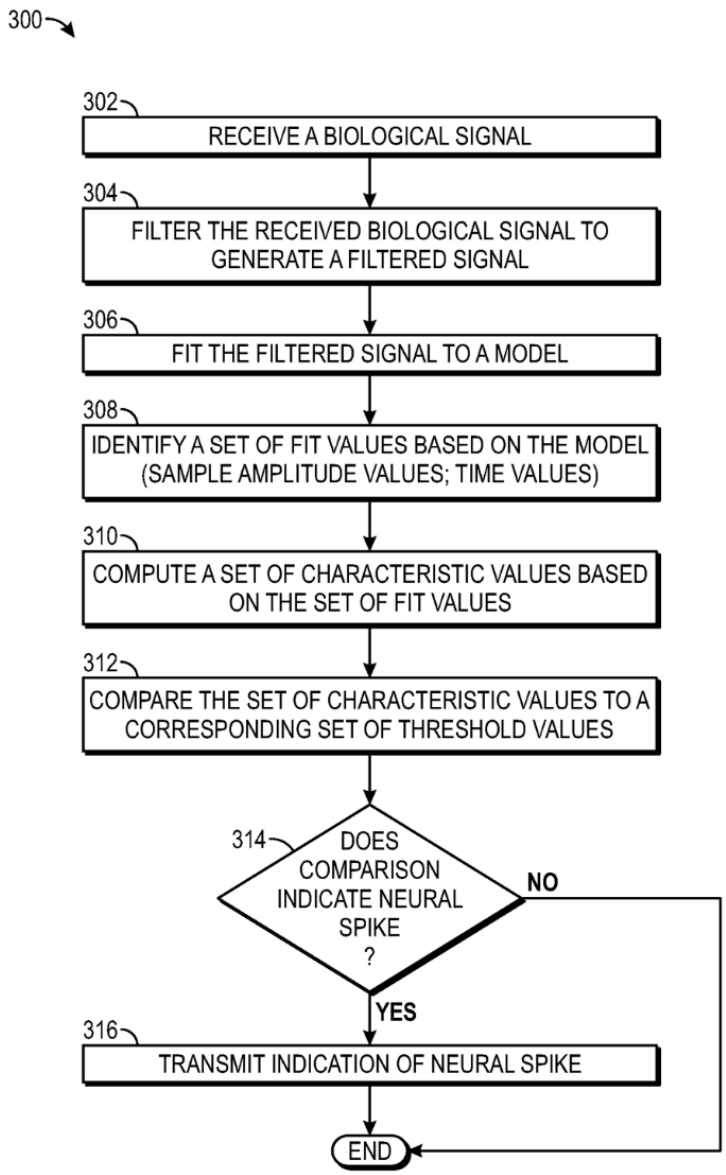


FIG. 3

1. Receive Biological Signal

Receive neurological voltage signal from electrode leads to IC chip

Sample a custom biological signal (pretend that this is received)

Params:

- Sampling frequency (f_s = 20000 samples per second, or 800 samples per window)
- Period (T = 2 seconds)
- Window (blackout period = 40 ms)

In [164…

```
import numpy as np
import scipy
from scipy.signal import find_peaks, argrelextrema, butter, filtfilt, sosfiltfilt, sosfilt, bode, freqz, freqs, sosfreqz, iirfilter
from scipy.fftpack import fft
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import control as ct
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from scipy.optimize import curve_fit
import heapq
import struct
```

In [164…
```
fs = 20000 # sampling frequency (samples per second)
T = 1 # period of interest (seconds)
blackout_period = 0.04 # 40 ms window
t = np.arange(0, T, 1 / fs)
```

In [164…
```
# Converts time t (in seconds) to index in data
def time_to_index(t):
    return int(t * fs)
```

In [164…
```
# Set very small random noise in the range [-epsilon, epsilon]
def small_random_noise(epsilon = 0.00005):
    return epsilon * np.random.randn(len(t))
```

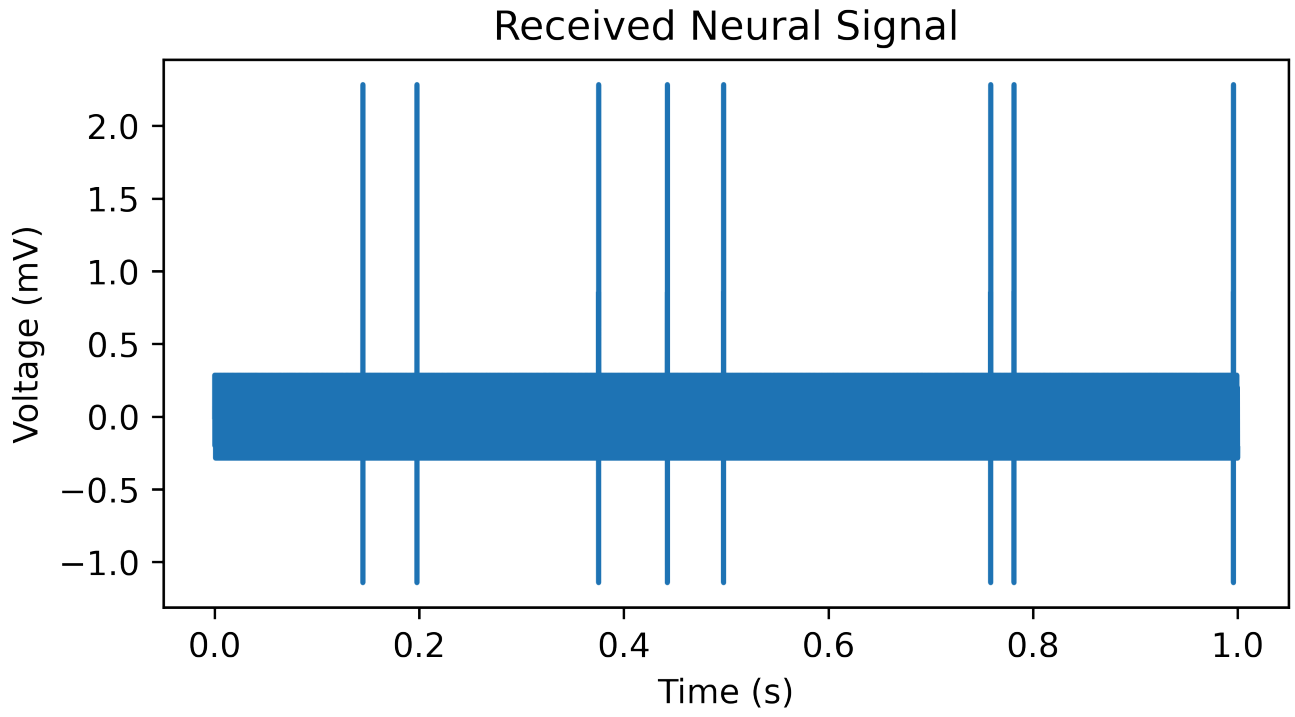
In [164…
```
def sample_signal(epsilon = 0.00005):
    f1 = 2000
    a1 = 0.1
    f2 = 3000
    a2 = 0.2
    return small_random_noise(epsilon=epsilon) + a1 * np.sin(2 * np.pi * f1 * t) + a2 * np.sin(2 * np.pi * f2 * t)
```

In [164…
```
def sample_spikes(x, n_spikes, epsilon = 0.00005):
    ampl = np.max(x)

    offset = 4

    sampled_spike_indices = np.random.choice(np.arange(offset, len(t)), size=n_spikes, replace=False)
    sampled_spike_indices.sort()

    for index in sampled_spike_indices:
        assert(index >= offset)
        x[index - 2 : index - 1] = 3 * ampl
        x[index - 1 : index] = -4 * ampl
        x[index : index + 1] = 8 * ampl

    return sampled_spike_indices
```

In [165…
```
# Sample a signal
data = sample_signal()

# Sample spikes at random timestamps
n_spikes = 8
sampled_spike_indices = sample_spikes(data, n_spikes=n_spikes)

# Ground truth
print('spikes sampled at (s):', t[sampled_spike_indices])
```

```
spikes sampled at (s): [0.14495 0.1978  0.3754  0.44255 0.49755 0.7586  0.7814  0.9959 ]
```

In [165…
```
# Plot the received (sampled) signal
fig, ax = plt.subplots(1, 1, figsize=(6, 3),dpi = 600)

plt.title("Received Neural Signal")
plt.ylabel("Voltage (mV)")
plt.xlabel("Time (s)")

plt.plot(t, data)
plt.show()
```



2. Filter Signal (preprocessing)

Filter - **subset** of signal is **retained**

Preprocessing signal - use **classical control theory**

e.g.,

- **High-pass filter** - select portions of signal that **exceed** some **threshold voltage**
- **Bandpass filter** - select portions of signal in **configured voltage band**
- **Butterworth filter** - type of **bandpass filter** that **selects voltage window**

Here, we use a **bandpass filter**: a **16-bit digital Infinite Impulse Response (IIR) Butterworth 2nd-Order Bandpass Filter**

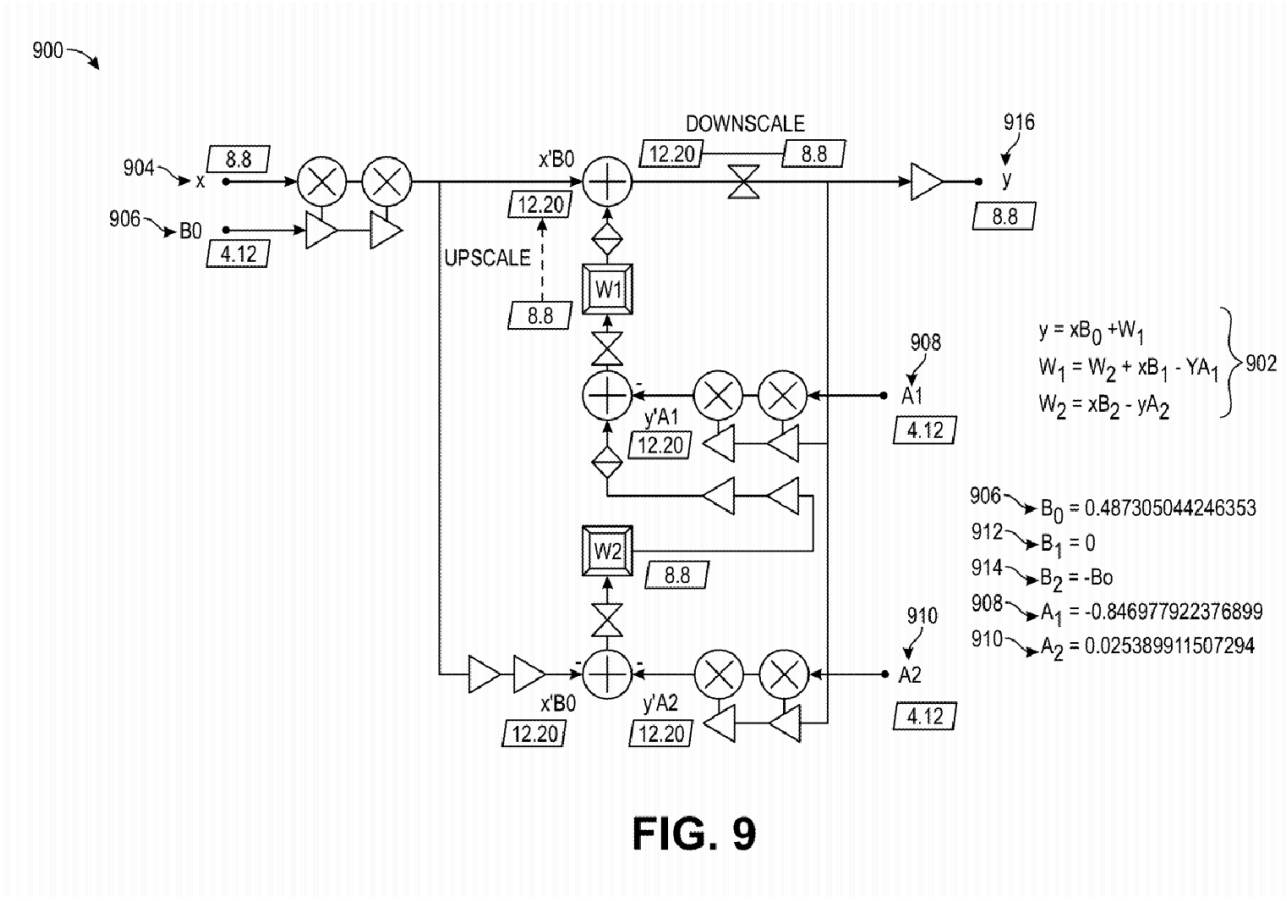


FIG. 9

The equations are as following:

$y = x * b_0 + w_1$

$w_1 = w_2 + x * b_1 - y * a_1$

$w_2 = x * b_2 - y * a_2$

where x - sample for a channel, y - output of filter,

a_1, a_2, b_0, b_1, b_2 - configurable params by user

w_1, w_2 - internal states (16 bits each)

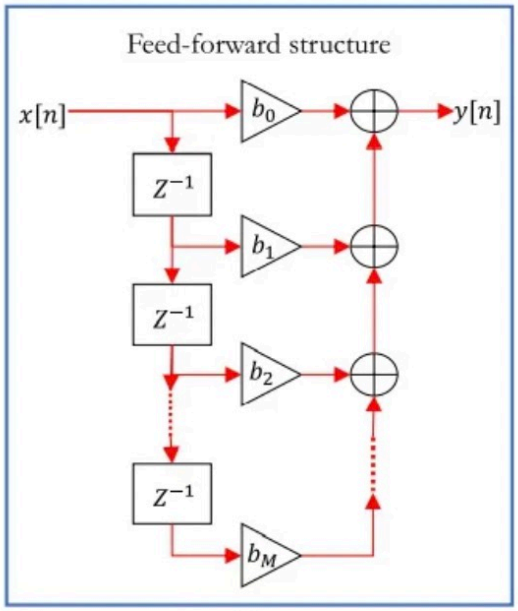
Finite Impulse Response (FIR) vs Infinite Impulse Response (IIR)

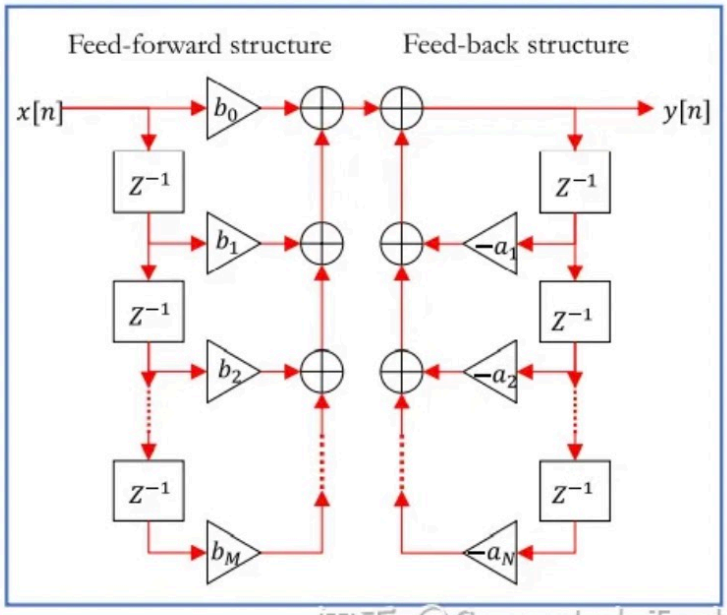
FIR:

- **No feed-back** structure (**only feed-forward**)
- Used in **engineering**
- **Easy to implement** due to **high precision**

IIR:

- Includes **feed-back** structure (**loops**)
- Used in **theoretical research**
- **Very difficult to implement** in **hardware** due to loops (mostly used in **software**)





Finite Impulse Response Filter (FIR) aka Moving Average

What is an impulse response in FIR?

- A FIR filter is a finite duration or finite impulses filter. In time series literature, it is represented by a Moving Average.
- $y_n = \beta_0 x_{k-n} + \beta_1 x_{k-(n-1)} + \cdots + \beta_{n-1} x_k$, where it is not necessary for βs to be positive or sum to one.
- It's the linear phase, so it implies there it would be delayed or lagged by a duration 'n' but no distortion.
- An ideal impulse is of infinitesimally short duration and unit magnitude input. An impulse in real life can be estimated with a unit magnitude with a total duration of 1 second. The energy under the signal is 1, and it is described as $\delta(t) = 0$ for $t \neq 0$. The Energy (area under the curve) is $\int_{-\epsilon}^{\epsilon} \delta(\tau) d\tau = 1$ for $\epsilon > 0$.
- Finally, at all times, unless otherwise stated, the data will be considered as a Stationary Stochastic Process.

Infinite Impulse Response (IIR) Filter

- Starting with an FIR filter (MA(q)), $y_t = \sum_{k=0}^{k=q=\infty} \theta_k \varepsilon_{t-k}$ with $\theta_0 = 1$, if q is replaced by ∞, then it will be called an Infinite Impulse Response (IIR) filter, $y_t = \sum_{k=0}^{k=\infty} \theta_k \varepsilon_{t-k}$.
- However, a filter based on the IIR infinity equation is not practical. Another way to define the IIR is using a recursive method,
$y_t = -\phi_1 y_{t-1} - \phi_2 y_{t-2} - \cdots - \phi_p y_{t-p} + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + \cdots +$
$\theta_q \varepsilon_{t-q} = \sum_{k=0}^{k=q} \theta_k \varepsilon_{t-k} - \sum_{j=0}^{j=p} \phi_j y_{t-j}$, which is an ARMA(p,q) process with $\phi_0 = 0$, $\theta_0 = 1$, zero-mean, and negative coefficients associated with AR part of the process.

θ: b, ϕ: a, ϵ: x (we are not modeling random Gaussian noise, but taking the original signal input and filtering it)

Since we are using a Python Jupyter notebook, an **IIR structure is okay**.

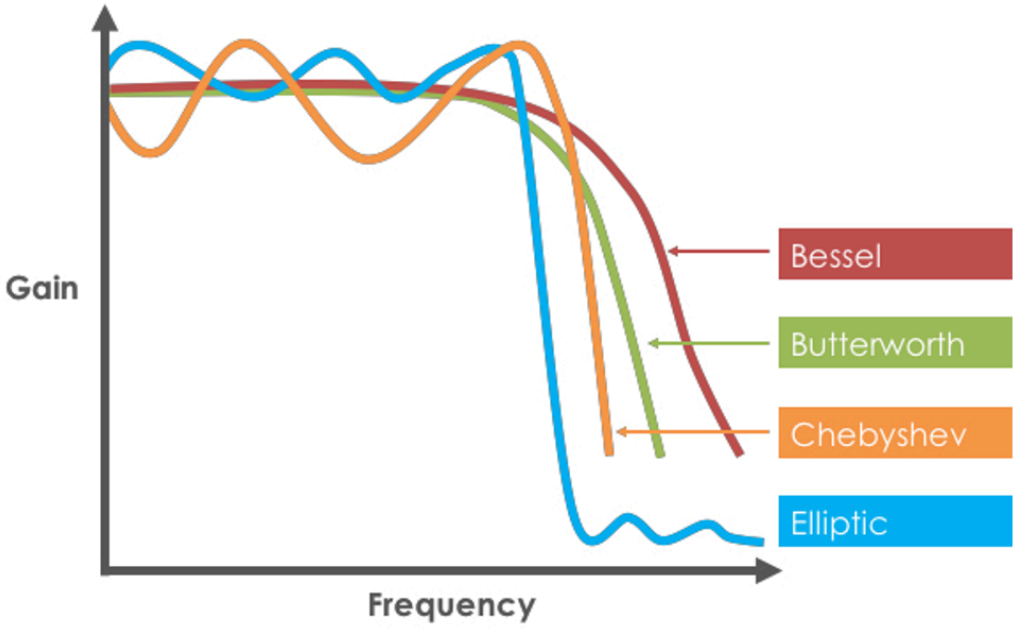
Filter Types: Bessel, Butterworth, Chebyshev, Elliptic

Butterworth – **maximally flat** frequency response in **passband**

Chebyshev – **steeper roll-off** with **ripple** in passband/stopband

Bessel – **maximally flat group delay**

Elliptic – **steeper roll-up** with **ripple** in **passband and stopband**, but **steepest roll-off** achievable in **given order**



To help us save computation, we ideally want a smooth response in the passband to preserve integrity of the signals. The roll-off from passband to stopband does not matter as much. Therefore, the patent chooses a **Butterworth** filter with a **2nd-order** (2 pairs of conjugate poles)

Filter implementation (using same params as mentioned in patent)

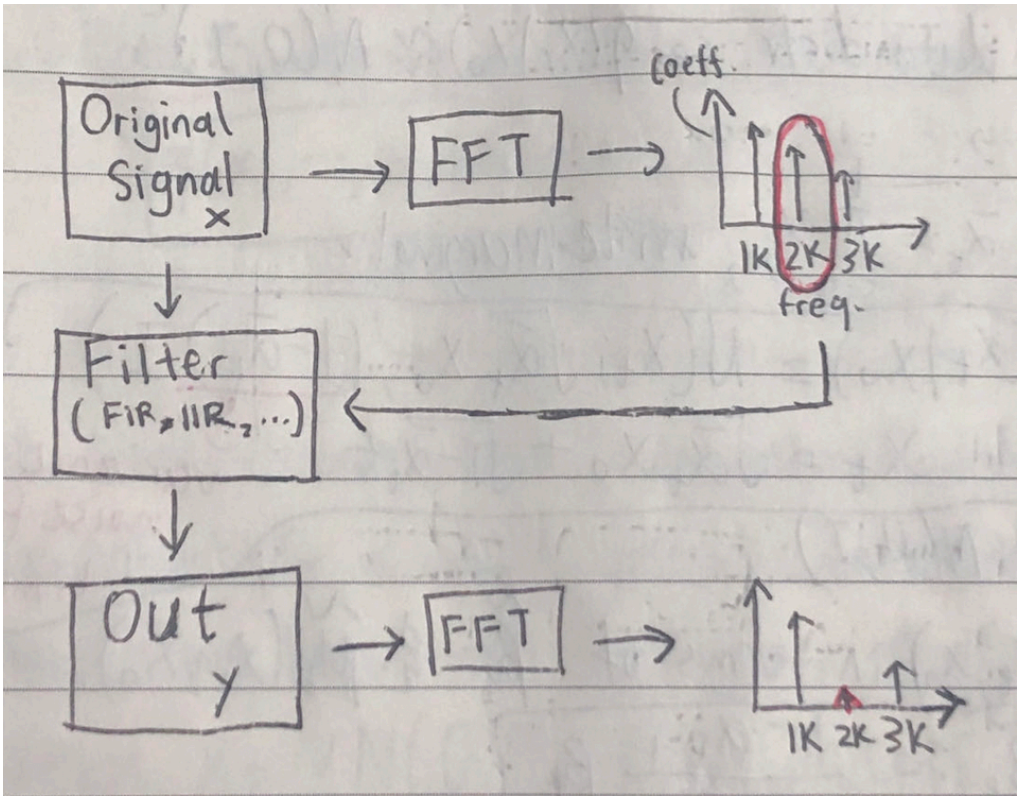
Implement the above IIR structure with exact same param values mentioned in patent

In [165…
```
# Assumes b, a are non-empty!
def iir_impl(b, a, x):
    # Output of filter
    y = []

    # Internal states
    w = np.zeros(len(b) + 1) # w: pad a zero on each end (for code readability)

    y_val = 0

    for i in np.arange(0, len(x)):
        y_val = w[1] + x[i] * b[0] - y_val * a[0]

        for j in np.arange(1, len(b)):
            w[j] = w[j + 1] + x[i] * b[j] - y_val * a[j]

        y.append(y_val)

    y = np.array(y)
    return y
```

In [165…
```
# Configurable params by user (copied from Fig. 9 above)
# These params are used in a transfer function H(z) (described further below)
b0 = 0.487305044246353
b1 = 0
b2 = -b0
a0 = 0
a1 = -0.846977922376899
a2 = 0.025389911507294

# Coeffs b, a for H(s)
b = np.array([b0, b1, b2])  # coeffs for numerator
a = np.array([a0, a1, a2])  # coeffs for denominator

# Sample for each channel
x = data

# Output of IIR filter
y = iir_impl(b, a, x)
```

Usually, when filtering, we follow this procedure:

1. Plot the **original signal**
2. Plot **Fast-Fourier Transform (FFT) of original signal** to get the **configured voltage window (low and high cut frequencies)**, indicated by the **principal components**.
3. **Filter the original signal** with the **configured frequency band** and plot it
4. Plot the **FFT of filtered signal** to see if the interested PCs are kept.

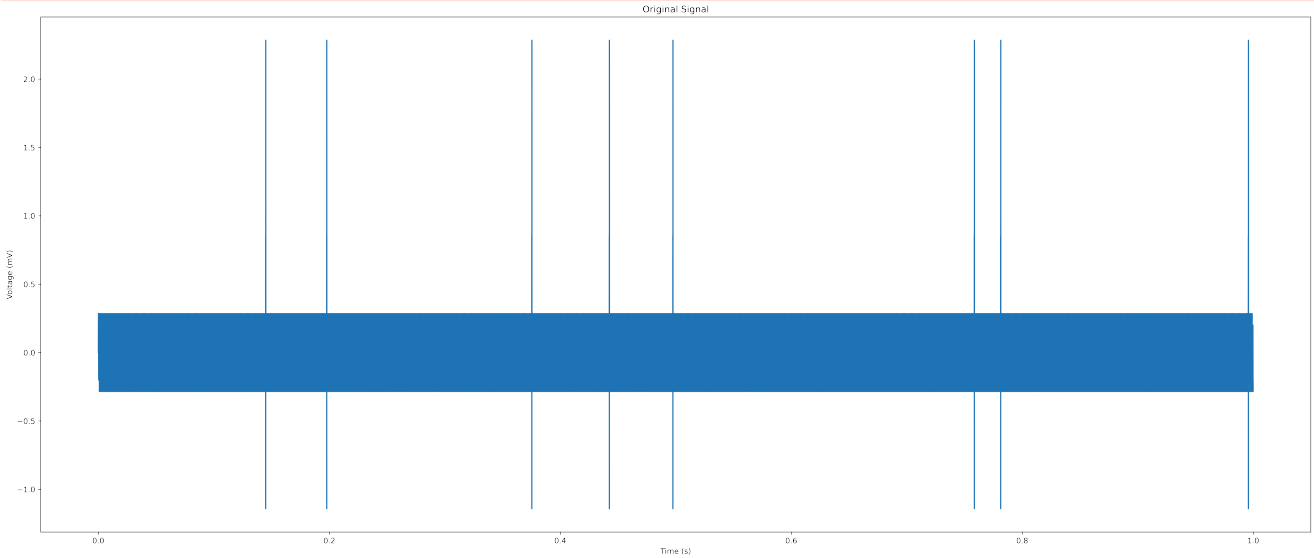


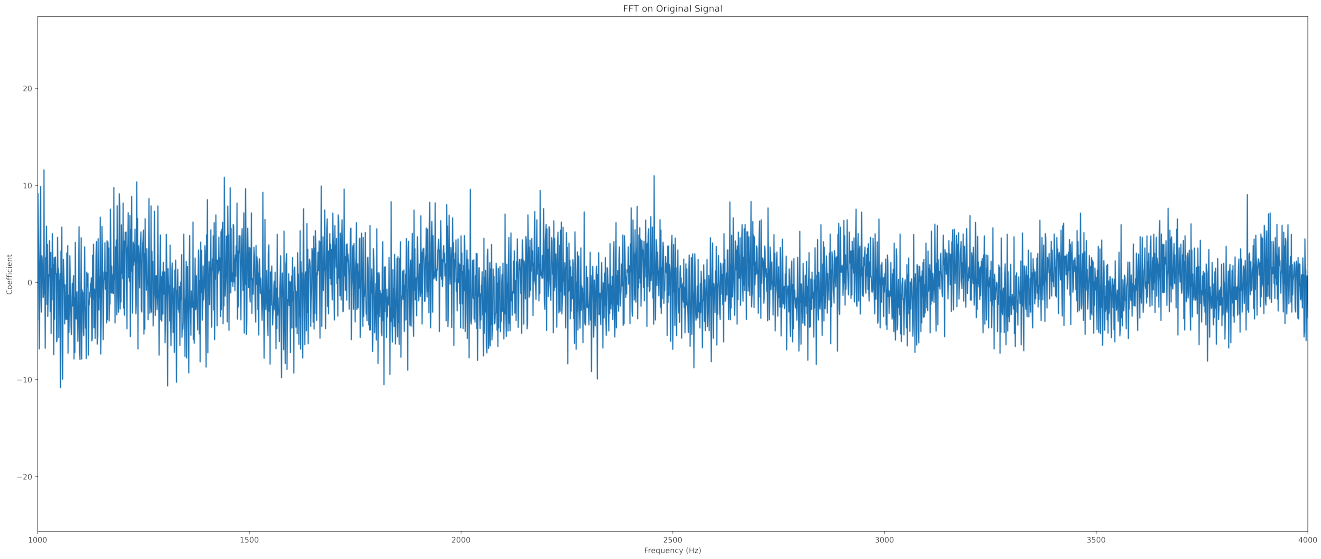
Steps 1 & 2: Original Signal + its FFT

In [165…
```
# Steps 1 & 2
fig, ax = plt.subplots(2, 1, figsize=(30, 30), dpi = 600)
plt.subplots_adjust(hspace=0.4)
```

```
# Step 1: Plot original signal
ax[0].plot(t, x)
ax[0].set_title("Original Signal")
ax[0].set_xlabel("Time (s)")
ax[0].set_ylabel("Voltage (mV)")

# Step 2: Plot FFT of original signal
fft1 = fft(x)
ax[1].plot(t * fs * fs / len(x), fft1)
ax[1].set_title("FFT on Original Signal")
ax[1].set_xlabel("Frequency (Hz)")
ax[1].set_ylabel("Coefficient")
ax[1].set_xlim(1000, 4000)

plt.show()
```

```
/Users/michaelzhou/opt/anaconda3/lib/python3.8/site-packages/matplotlib/cbook/__init__.py:1289: ComplexWarning: Casting complex values to real discards the im
aginary part
  return np.asarray(x, float)
```





Because of a high sampling frequency, it is very difficult to find the particular principal component frequencies that stick out. But for lower sampling frequencies, it is much easier to see them.

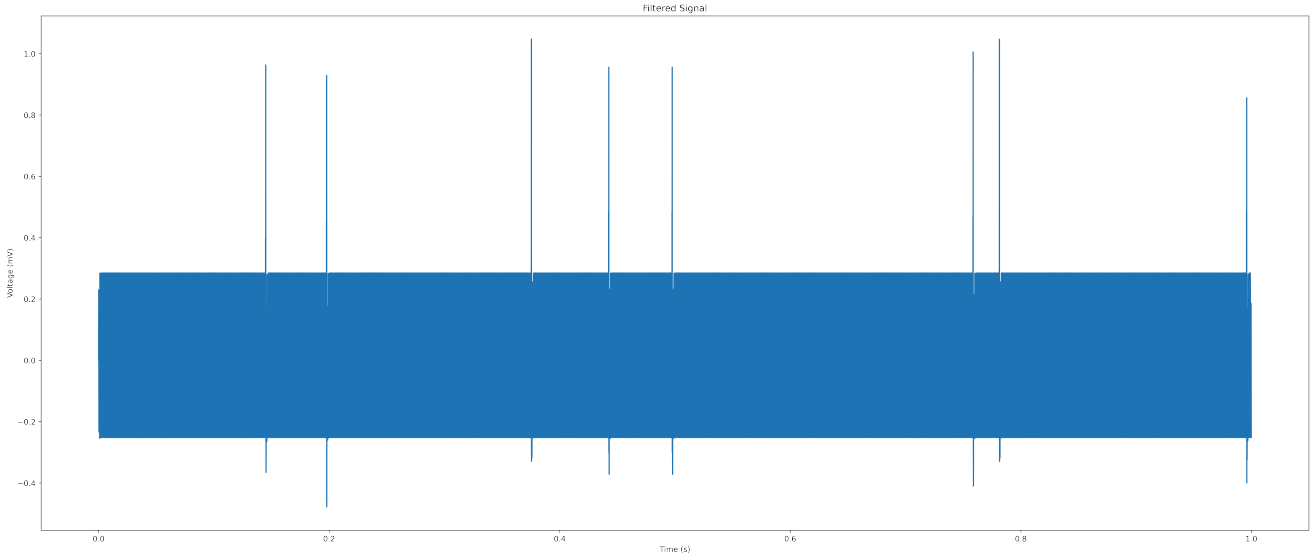
Steps 3 & 4: Filtered Signal + its FFT

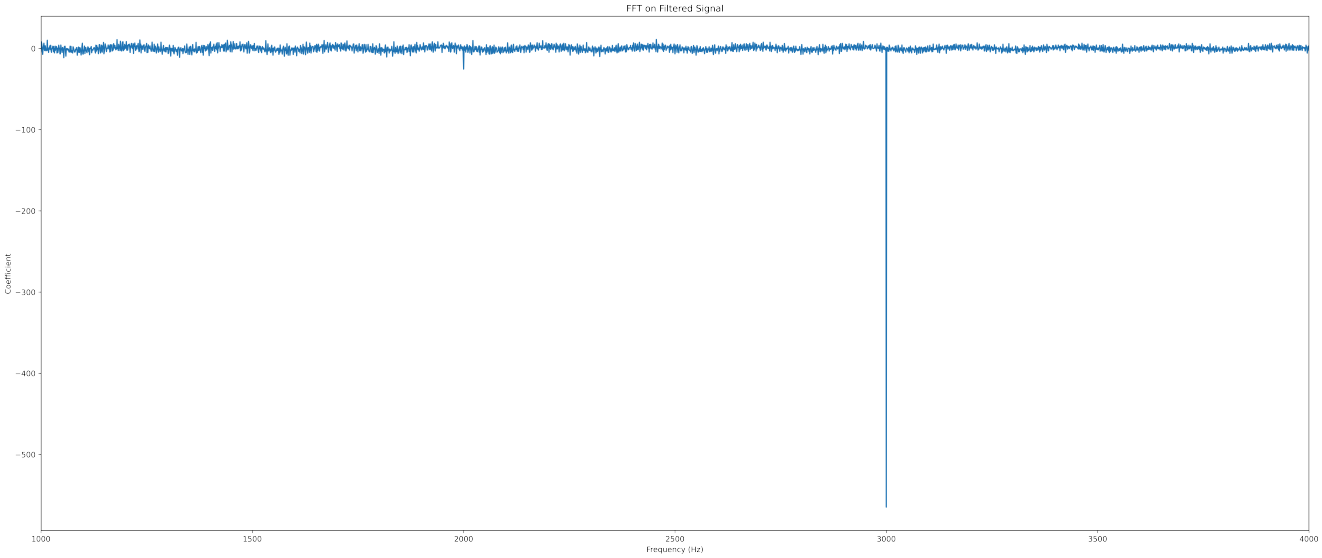
In [165…

```
# Steps 3 & 4
fig, ax = plt.subplots(2, 1, figsize=(30, 30), dpi = 600)
plt.subplots_adjust(hspace=0.4)

# Step 3: Plot filtered signal
ax[0].plot(t, y)
ax[0].set_title("Filtered Signal")
ax[0].set_xlabel("Time (s)")
ax[0].set_ylabel("Voltage (mV)")
```

```
# Step 4: Plot FFT of filtered signal
fft2 = fft(y)
ax[1].plot(t * fs * fs / len(data), fft2)
ax[1].set_title("FFT on Filtered Signal")
ax[1].set_xlabel("Frequency (Hz)")
ax[1].set_ylabel("Coefficient")
ax[1].set_xlim(1000, 4000)

plt.show()
```

```
/Users/michaelzhou/opt/anaconda3/lib/python3.8/site-packages/matplotlib/cbook/__init__.py:1289: ComplexWarning: Casting complex values to real discards the im
aginary part
  return np.asarray(x, float)
```





We can see a clear picture of the locations of the spikes after filtering the signal. But **practically**, you **do NOT want to hard-code hyperparams manually**. But since this algorithm is implemented in hardware, and the patent has given these hyperparameters, we are only using this notebook for **education purposes only**.

Plot the Bode graph of the above filter (using the b, a params)

After filtering the signal, we usually draw a **Bode plot**. A **Bode plot** is a semi-logarithmic coordinate plot of the **transfer function** of a non time-varying time system with respect to frequency, with the **horizontal x axis** representing a **log-frequency scale**. It includes 2 plots, one **amplitude vs frequency** (dB of frequency response gain vs frequency variation) and one **phase vs frequency** (phase of frequency response vs frequency variation).

In order to draw a Bode plot, we need to figure out the **transfer function** $H(s)$, which includes a set of coefficients b in the numerator and another set of coefficients a in the denominator.

The **transfer function formula** is given as:

$$H(s) = \sum_{i=0}^{N} b[N-i]s^i \,/\, \sum_{j=0}^{M} a[M-j]s^j$$

where $b = b_0, b_1, \ldots, b_N$ and $a = a_0, a_1, \ldots, a_M$

In [165…
```
# Print b & a params
print('b:', b)
print('a:', a)

# Write the transfer function H(s) here using the coeffs
ct.tf(b, a)
```

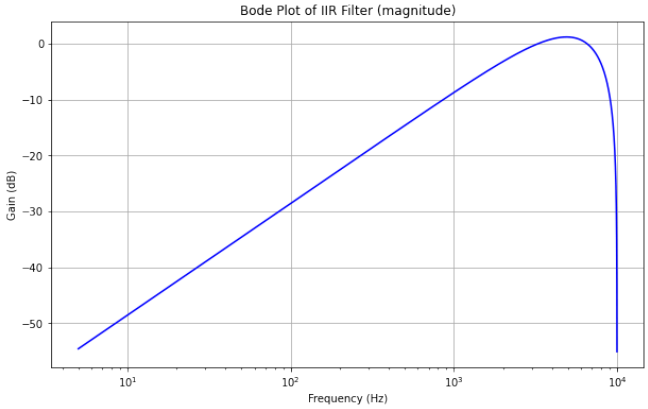
```
b: [ 0.48730504  0.         -0.48730504]
a: [ 0.         -0.84697792  0.02538991]
```

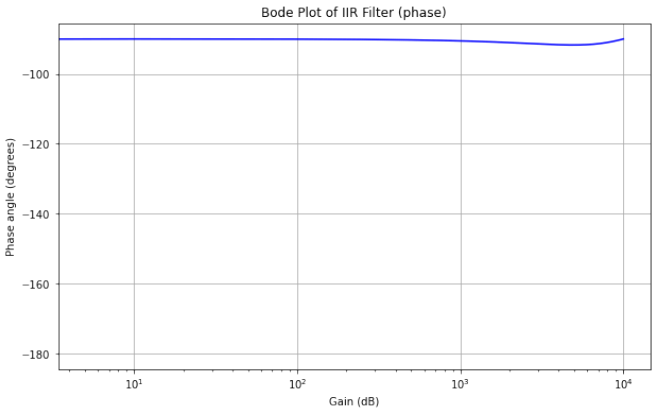
Out[165…

$$\frac{0.4873s^2 - 0.4873}{-0.847s + 0.02539}$$

In [165…
```
# Plot Bode graph here
w, h = freqz(b, a, worN=2000) # obtain w - angular frequency, h - transfer function values

# Calculate magnitude + phase graphs
magnitude = 20*np.log10(np.abs(h))
phase = np.unwrap(np.arctan2(np.imag(h), np.real(h)))*(180/np.pi)

# Bode plot (magnitude)
plt.figure(figsize=(10,6))
plt.semilogx(w*fs/(2*np.pi), magnitude, 'b-')
plt.title('Bode Plot of IIR Filter (magnitude)')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Gain (dB)')
plt.grid()
# plt.xlim(1, 10000)

# Bode plot (phase)
plt.figure(figsize=(10,6))
plt.semilogx(w*fs/(2*np.pi), phase, 'b-')
plt.title('Bode Plot of IIR Filter (phase)')
plt.xlabel('Gain (dB)')
plt.ylabel('Phase angle (degrees)')
plt.grid()

plt.show()
```

```
<ipython-input-1657-e1aaf5ae9f37>:5: RuntimeWarning: divide by zero encountered in log10
  magnitude = 20*np.log10(np.abs(h))
```





In [165…
```
# Compute the high and low cuts of the frequency
frequency_band = w[magnitude >= -3] * fs/(2*np.pi) # Frequencies where gain >= -3 -> keep signals for band-pass
low_cut = frequency_band[0]
high_cut = frequency_band[-1]
print('configured frequency band: [', low_cut, ',', high_cut, '] Hz')
```

```
configured frequency band: [ 2055.0 , 7830.0 ] Hz
```

As we see in the Bode magnitude plot, we see that the **configured frequency range** is **about [2055, 7830] Hz**. According to the patent, this range is about **2000 - 3000 Hz**, where spikes are detected, which is exactly what we sampled.

Usually, the **params** (the **transfer function coefficients** $b = [b_0, b_1, \ldots, b_N]$**,** $a = [a_0, a_1, \ldots, a_M]$) are ***generated by the filter*** rather than manually hard-coded. We are showing this just for demonstration.

About Bode Plots:

- The **Bode plot** has **NOTHING to do** with the **filtered signal** y, since it **only depends on the transfer function** $H(z)$ **(with coefficients** b **and** a**)!!!** So you CAN plot the Bode graph BEFORE filtering the signal!!
- Most of the time, we **only need** look at the **magnitude plot**. But occassionally, when the **filtered signal is faulty** (e.g., when the signal is **diverging**), we would also look at the **phase plot** and look at whether conditions dictate a **self-excitation oscillation** (i.e., whether the signal at the end is *still* at a **certain amplitude and frequency without** adding any **outside input signals**).
- For the **phase plot**, normally, we would need to check whether our **phase range** is good enough for our **chosen frequency range**. This is **dependent on the application**.

3. Smoothing

- Apply **local smoothing function** (**Brown's double exponential smoothing function**)

Smoothing - **Remove** any **system noise** from the data, using **modern control theory**

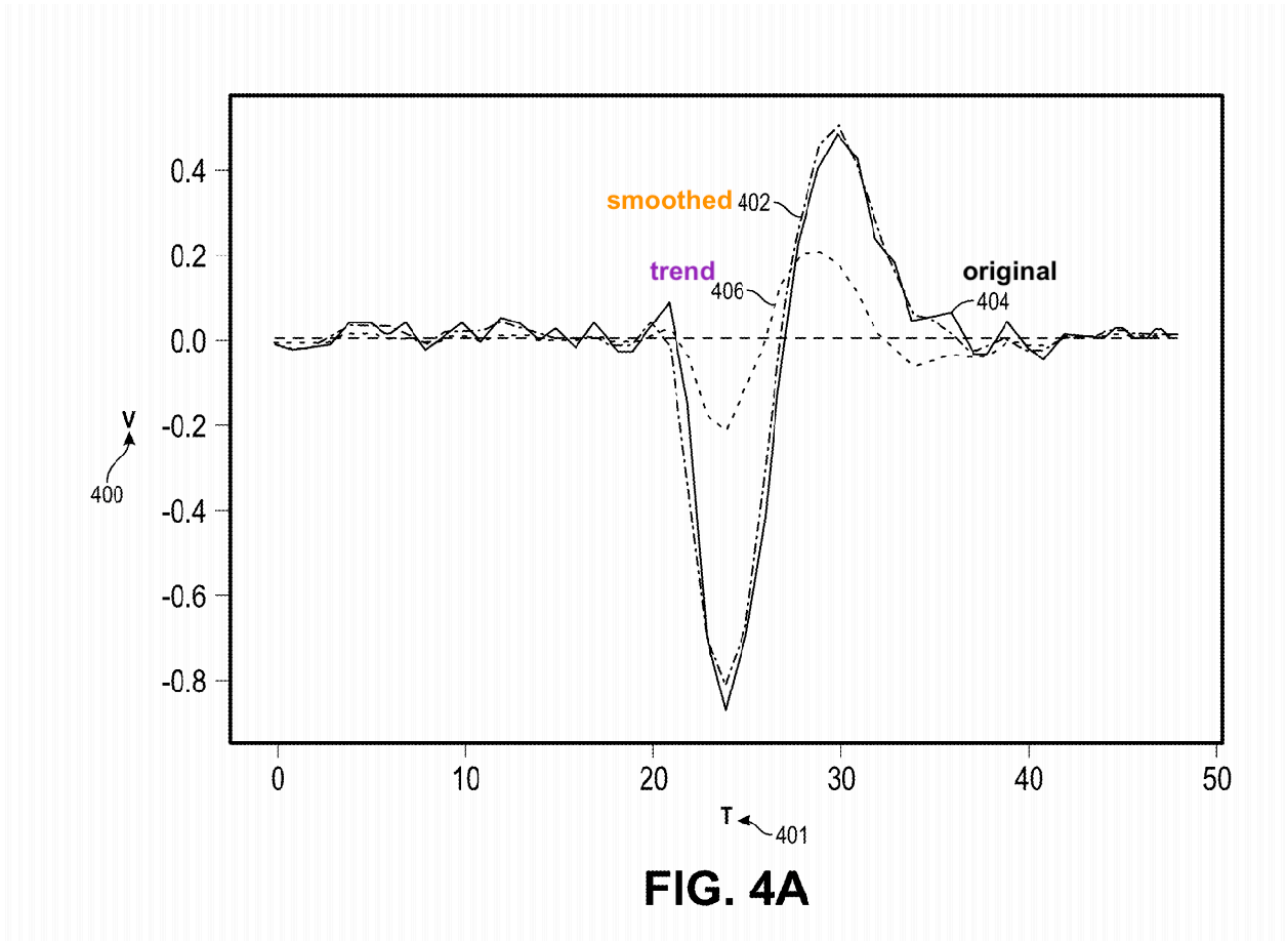
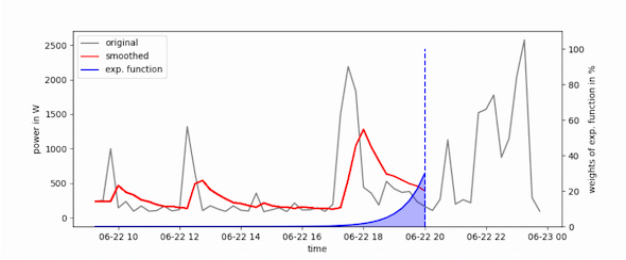


FIG. 4A

Exponential Smoothing

For our smoothing method, we choose **exponential smoothing (ES)**, since it captures **non-linear patterns** (such as **neural spikes**) using an **exponentially decreasing function**, unlike other smoothing methods (moving average, kernel smoothing, etc.).

In ES, **no sliding window** is used, and **more recent data points** are given a **greater weight** than the ones further in the past.

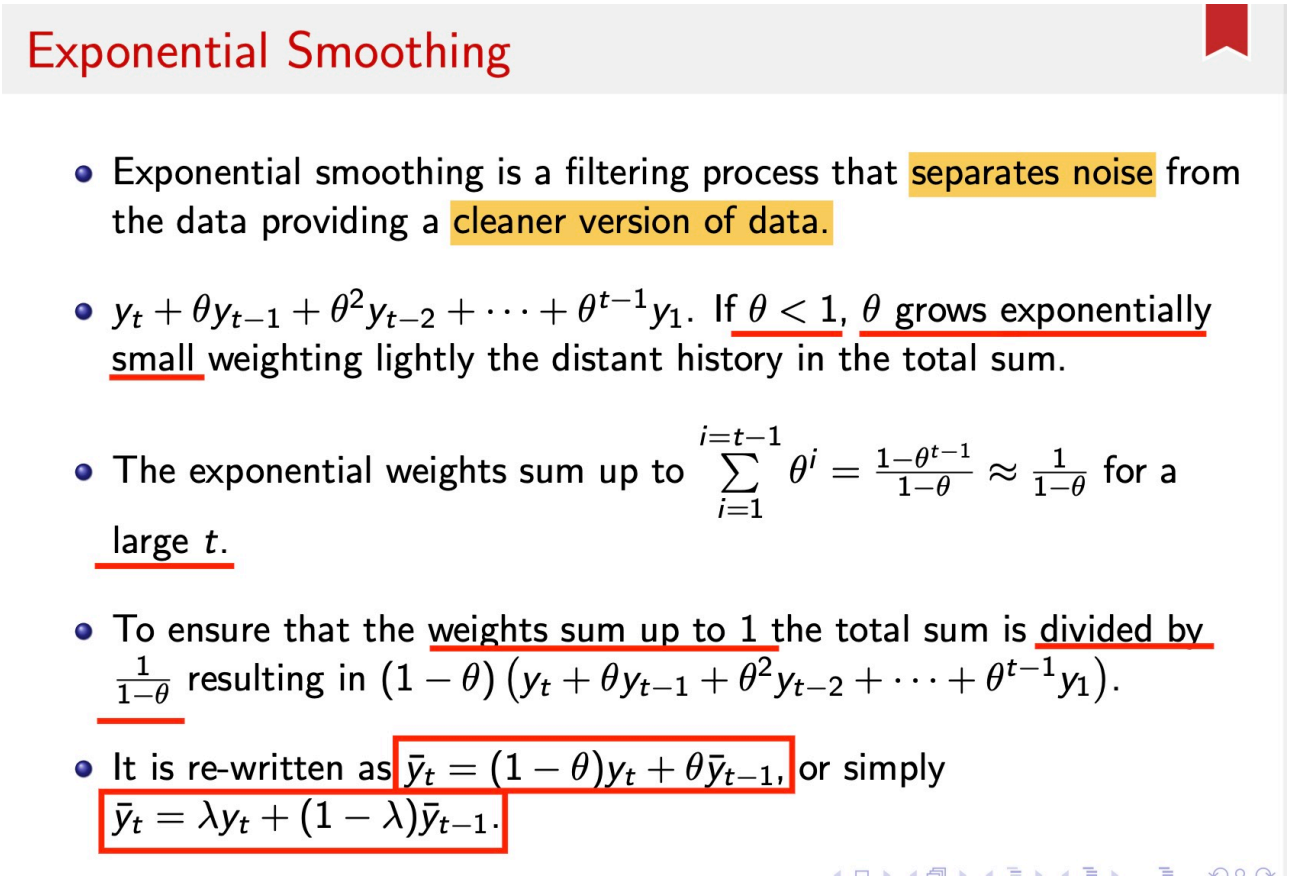


Simple Exponential Smoothing

The formula for (simple) exponential smoothing is given below:

$$\hat{y}_t = \begin{cases} \hat{y}_t = y_t & \text{if } t = 0 \\ \alpha y_t + (1-\alpha)\hat{y}_{t-1} & \text{otherwise} \end{cases}$$

where α is the smoothing factor in the range 0-1 (**bigger** α -- **smaller emphasis** on **past data points**, **smaller smoothing**)



Slide credits from my course *Forecasting: A Real-World Application*, taught by Syed Haider

Double Exponential Smoothing

In double exponential smoothing, we have a **level** (**current estimate** of series' underlying level or **average**; **smoothed value** at currrent time point) and the **trend** (how data is **progressing** over time).

Normally, we have **two (2)** hyperparameters (both in the range 0-1): α and β, which controls the level and the trend respectively. The **trend** can be **additive (linear trend)** or **multiplicative (exponential trend)**. We must identify whether the trend is linear or exponential.

The most common double exponential smoothing method is **Holt's Double ES** with an **additive (linear) trend**, and is represented by:

$$\hat{y}_t = l_t + r_t$$

$$l_t = \begin{cases} y_t & \text{if } t = 0 \\ \alpha y_t + (1-\alpha)(l_{t-1} + r_{t-1}) & \text{otherwise} \end{cases}$$

$$r_t = \begin{cases} y_{t-1} - y_t & \text{if } t = 0 \\ \beta(l_t - l_{t-1}) + (1-\beta)r_{t-1}) & \text{otherwise} \end{cases}$$

where l is the level and r is the trend.

Triple Exponential Smoothing

In triple exponential smoothing, we have the **seasonality** (**regular, predictable** changes over time), in addition to the **level** and **trend** in double ES.

Normally, we have **three (3)** hyperparameters (all in the range 0-1): α, β, and γ, which control the level, trend, and seasonality respectively. Similar to the trend, the **seasonality** can be **additive (linear seasonality)** or **multiplicative (exponential seasonality)**. We must identify whether the trend and seasonality is linear or exponential.

The most common double exponential smoothing method is **Holt-Winters ES** with an **additive (linear) trend** and **additive (linear) seasonality**, and is represented by:

$$\hat{y}_{t+k} = l_t + kr_t + s_{t+k-M}$$

$$l_t = \alpha(y_t - s_{t-M}) + (1-\alpha)(l_{t-1} + r_{t-1})$$

$$r_t = \begin{cases} \frac{\sum_{i=1}^{M} y_{M+i} - y_i}{M} & \text{if } t = 0 \\ \beta(l_t - l_{t-1}) + (1-\beta)r_{t-1}) & \text{otherwise} \end{cases}$$

$$s_t = \gamma(y_t - l_t) + (1-\gamma)s_{t-M}$$

where l_t is the level estimate for time t, k is the number of forecasts into the future, r_t is the trend estimate at time t, s_t is the seasonal estimate at time t, and M is the number of seasons.

Brown's Double Exponential Smoothing

In this patent, we use **Brown's double ES**, since we can find both the **level** AND the **trend** using only **one (1) hyperparameter (α)**, instead of two (in Holt). Note that our spikes are **randomly sampled**, and can happen at **ANY** time, so there is **no seasonality involved**.

The formula is expressed as the following:

Single exponential smoothing: $S'(t) = \alpha Y(t) + (1 - \alpha)S'(t-1)$

Double exponential smoothing: $S''(t) = \alpha S'(t) + (1 - \alpha)S''(t-1)$

Forecast: $\hat{y}_{t+1} = l_t + r_t$

where $l_t = 2S'(t) - S''(t)$ (estimated level at time t)

$r_t = \frac{\alpha}{1-\alpha}(S'(t) - S''(t))$ (estimated trend at time t)

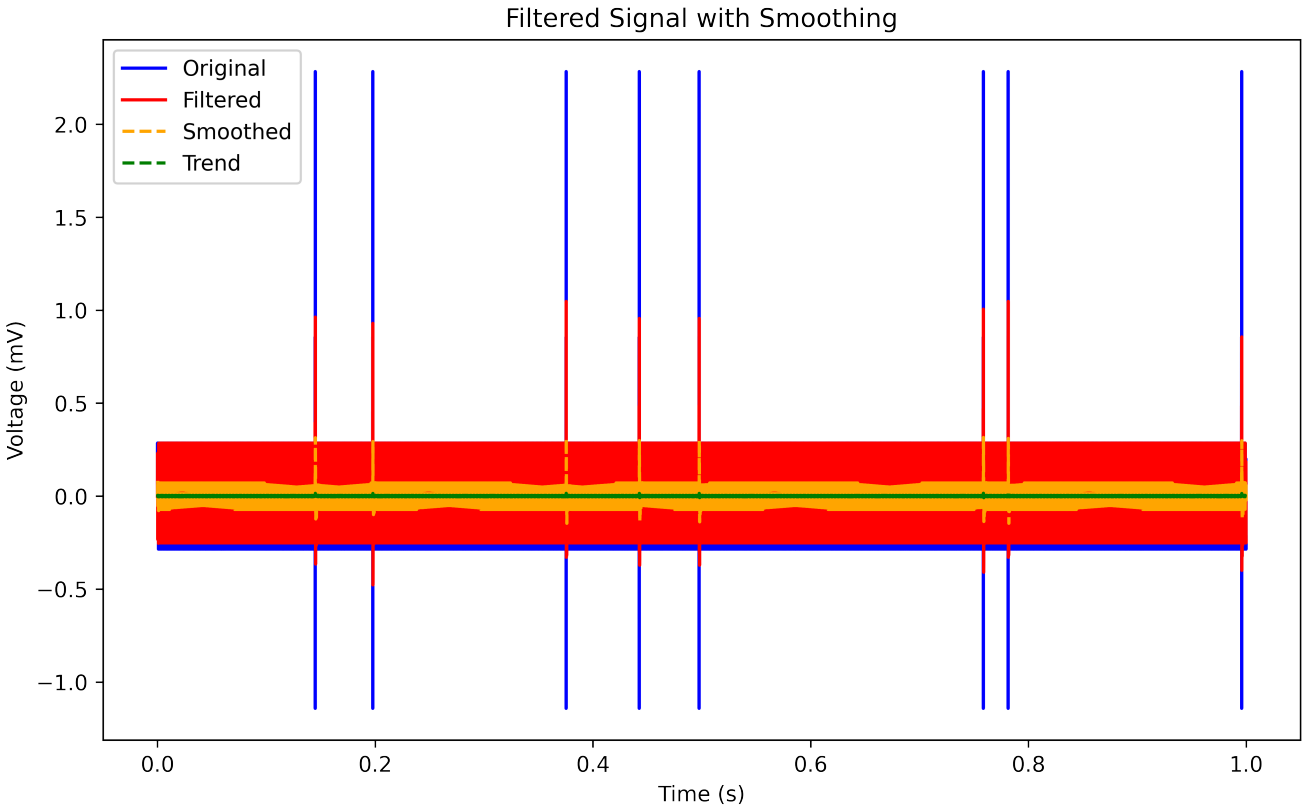
In [165…
```python
def brown_double_es(y, alpha=0.1):

    single = np.zeros(y.shape)
    double = np.zeros(y.shape)

    level = np.zeros(y.shape)
    trend = np.zeros(y.shape)

    y_hat = np.zeros(len(y) + 1)

    for i in np.arange(0, len(x) - 1):
        single[i] = alpha * y[i] + (1 - alpha) * single[i - 1]
        double[i] = alpha * single[i] + (1 - alpha) * double[i - 1]

        level[i] = 2 * single[i] - double[i]
        trend[i] = (alpha / (1 - alpha)) * (single[i] - double[i])

        y_hat[i + 1] = level[i] + trend[i]

    return np.array(y_hat[:len(y_hat) - 1]), np.array(trend)
```

In [166…
```python
# Perform smoothing
forecast, trend = brown_double_es(y, alpha=0.1) # Choose between {0.1, 0.2, ..., 0.9} with smallest MSE
mean_squared_error(forecast, y)
```

Out[166…
```
0.02848645757631062
```

In [166…
```python
plt.figure()
fig, ax = plt.subplots(1, 1, figsize=(10, 6), dpi = 600)
plt.subplots_adjust(hspace=0.4)

# Plot signal + smoothed signal
plt.plot(t, x, color='blue', linestyle='-', label='Original')
plt.plot(t, y, color='red', linestyle='-', label='Filtered')
plt.plot(t, forecast, color='orange', linestyle='--', label='Smoothed')
plt.plot(t, trend, color='green', linestyle='--', label='Trend')
plt.title("Filtered Signal with Smoothing")
plt.xlabel("Time (s)")
plt.ylabel("Voltage (mV)")
plt.legend()
plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```



4. Identify Fit Values

- Identify **local maxima and minima** of **filtered signal** in **each time window**

Note: Optionally, you could plot a polynomial function to identify the fit values, but since we already have the local maxima and minima, we do not need to do it here.

The fit values are given as lv (left voltage), mv (middle voltage), rv (right voltage), lp (left timestamp), mp (middle timestamp), and rp (right timestamp), as shown in the graph below.

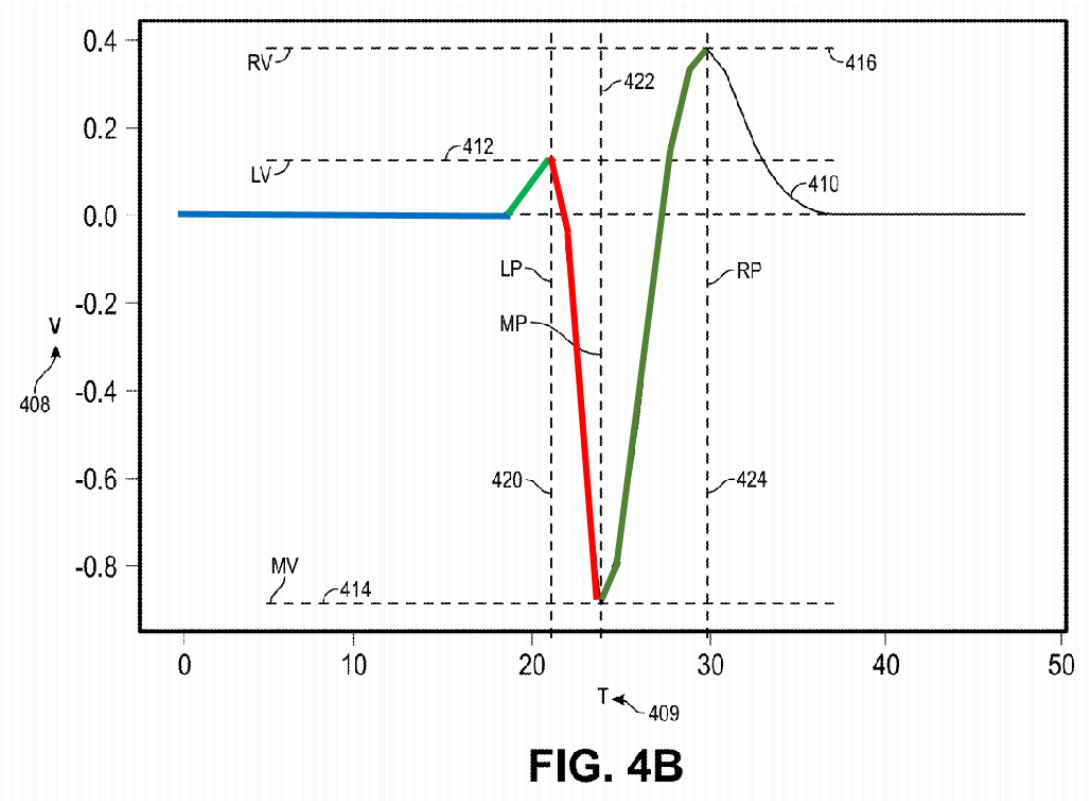


FIG. 4B

Find local maxima and minima in filtered signal for each time window:

Use the **time window length (blackout period)** set in the very beginning:

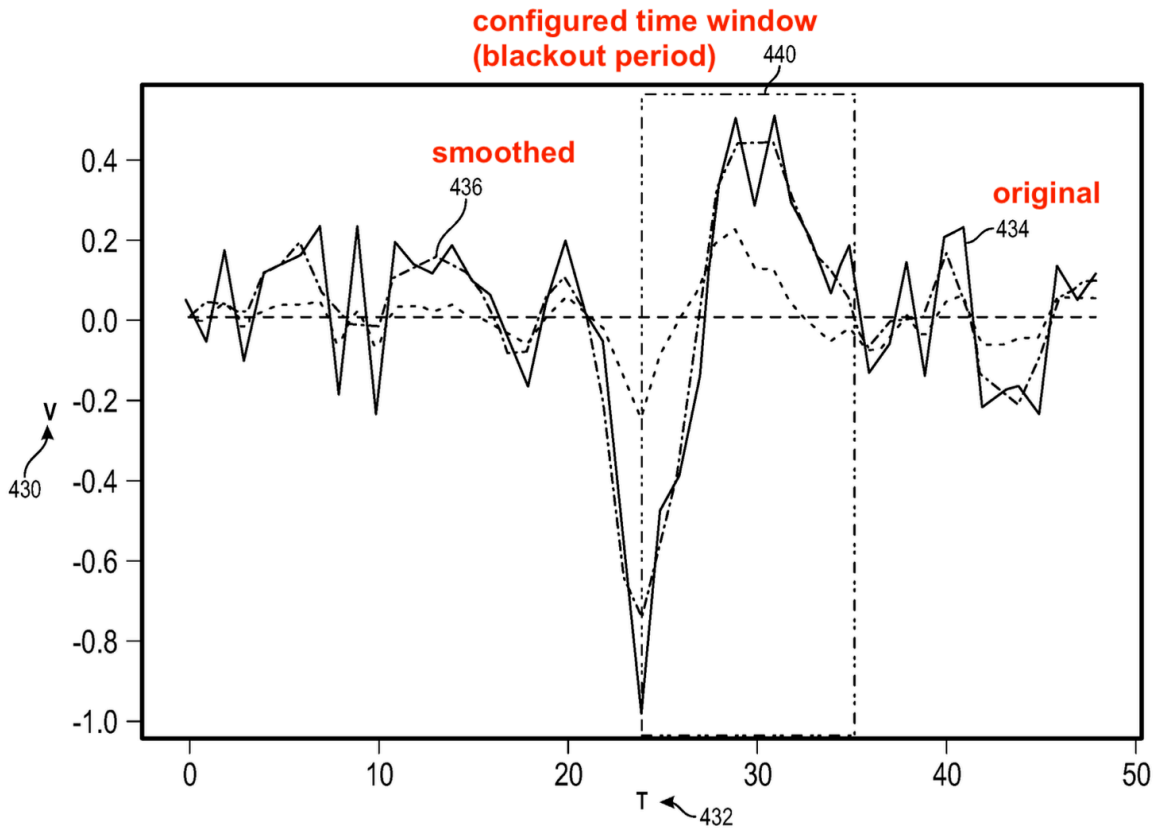


FIG. 4C

In [166…

```python
# Post-processing: For any consecutive [min, min, min, ...] or [max, max, max, ...] pattern, ONLY keep the most extreme extremum!
def post_processing(local_extrema):
    index = 0
    while index < len(local_extrema) - 1:

        # Find the next extrema with high value
        vals = []
        j = index + 1
        while j < len(local_extrema) and local_extrema[j][1] == local_extrema[index][1]:
            vals.append(y[int(local_extrema[j][0])])
            j += 1

        #             i   j=i+1     i   j=i+1
        # If indeed [min, max]  or [max, min], skip
        if j == index + 1:
            index = j
            continue

        # Otherwise, keep only the most extreme extremum
        else:
            type_ = local_extrema[index][1]
            assert(type_ == local_extrema[j - 1][1])
            if type_ == 'min':
                offset = np.argmin(vals)
                local_extrema[index : j] = local_extrema[index + offset]
            else:
                offset = np.argmax(vals)
                local_extrema[index : j] = local_extrema[index + offset]

        index = j

    # Remove all repeated values (keep only 1 of them)
    local_extrema_unique = []
    index = 0

    while index < len(local_extrema):
        j = index + 1

        while j < len(local_extrema) and local_extrema[j][0] == local_extrema[index][0]:
            j += 1

        local_extrema_unique.append(tuple(local_extrema[index]))
        index = j

    return local_extrema_unique
```

In [166…

```python
# Get the indices for each window of length `blackout_period`
def get_local_extrema(blackout_period = 0.04):
    index = 0

    local_extrema = []

    # Convert window time to indices
    blackout_period_in_indices = time_to_index(blackout_period)

    while index < len(y) - blackout_period_in_indices:
        i1 = index
        i2 = min(len(y), index + blackout_period_in_indices)

        # Find the local min and max of signals
        local_min = index + np.argmin(y[i1 : i2])
        local_max = index + np.argmax(y[i1 : i2])

        local_extrema.append((int(local_min), 'min'))
        local_extrema.append((int(local_max), 'max'))

        # Move to next window
        index += 1

    # Sort the local extrema
    local_extrema.sort()

    # Perform post-processing on local_extrema
    local_extrema = np.array(local_extrema)
    local_extrema = post_processing(local_extrema)

    # Add the local min and max accordingly
    local_minima = []
    local_maxima = []
    for tup in local_extrema:
        if tup[1] == 'min':
            local_minima.append(int(tup[0]))
        else:
            local_maxima.append(int(tup[0]))

    # Post-processing: remove first local extrema if it's a minima (we don't need it!)
    if local_minima[0] < local_maxima[0]:
        local_minima = local_minima[1:]

    return local_minima, local_maxima
```

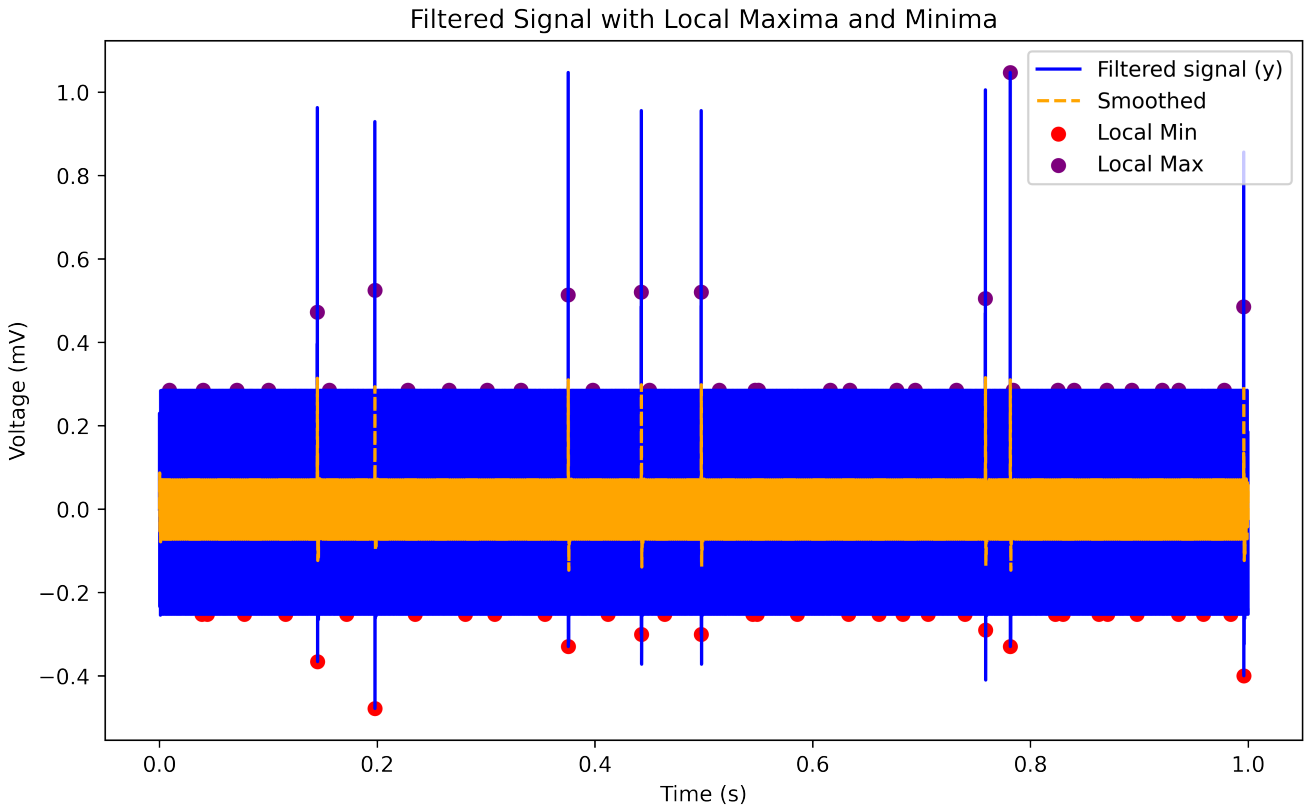
In [166…

```python
# Get the local extrema (min and max) per window
local_minima, local_maxima = get_local_extrema(blackout_period = blackout_period)
```

In [166…

```python
plt.figure()
fig, ax = plt.subplots(1, 1, figsize=(10, 6), dpi = 600)
plt.subplots_adjust(hspace=0.4)

# Plot filtered signal (for determining thresholds)
plt.plot(t, y, color='blue', linestyle='-', label='Filtered signal (y)')
plt.plot(t, forecast, color='orange', linestyle='--', label='Smoothed')
plt.scatter(t[local_minima], y[local_minima], color='red', label='Local Min')
plt.scatter(t[local_maxima], y[local_maxima], color='purple', label='Local Max')

plt.title("Filtered Signal with Local Maxima and Minima")
plt.xlabel("Time (s)")
plt.ylabel("Voltage (mV)")
plt.legend()
# plt.xlim(0, 5)

plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```



Identify fit values

In [166…

```
# Determine the fit values between a signal
# idx: index of local max/min
def fit_values(local_minima, local_maxima, idx):

    # Get left, middle, right timestamps
    lp = t[local_maxima[idx]]
    mp = t[local_minima[idx]]
    rp = t[local_maxima[idx + 1]] # aka end time of window

    # Get left, middle, right voltages
    lv = y[local_maxima[idx]]
    mv = y[local_minima[idx]]
    rv = y[local_maxima[idx + 1]]

    return lp, mp, rp, lv, mv, rv
```

5. Compute Characteristic Values

The **mean absolute deviation (MAD) estimation** function is given as:

$$\hat{m} = m + \alpha(|x| - m),$$

where m is the mean value across the **smoothed** signal in a given time window, $alpha$ is a **user-configured parameter** for the update multiplier (set as **0.0002**), and $|x|$ is the **value of the original signal**.

Note that we are taking the MAD **for each time window** $[t_1, t_2]$, which entails **accumulating the absolute deviation of the signal (**$m + \alpha(|x| - m)$**)** to the **smoothed function** within the time window, and then **dividing it by the magnitude of the negative deflection**. So the actual formula is as follows:

$\hat{m} = \frac{\sum_{t=t_1}^{t_2} m+\alpha(|x_t|-m)}{\sum_{t=t_1}^{t_2} |x_t|-m}$, where m is the mean of the **smoothed** signal calculated **over** t_1 **and** t_2.

The **threshold** $thres$ is given as:

$thres = \beta * m$, where β is a **scale factor constant** given in the **range from 0 to 7.875** (given in **Q3.3 format - 3 bits for integer, 3 bits for fraction**; must be a **multiple of 0.125**)

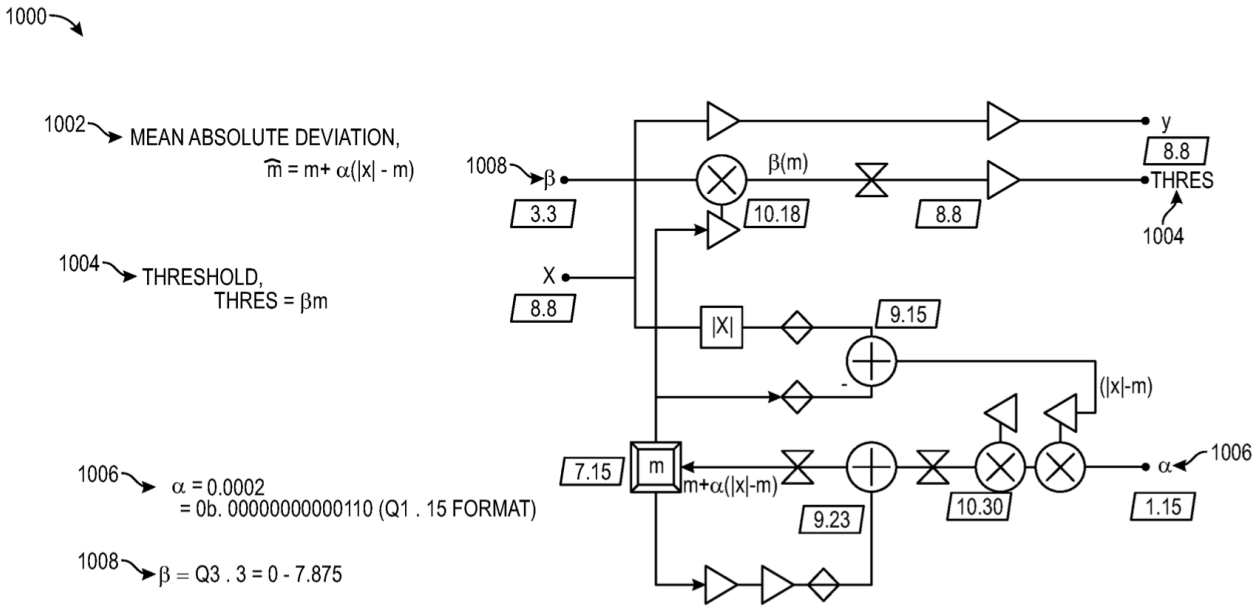


FIG. 10

In [166…
```python
# Calculate the mean absolute deviation (MAD) of a signal
# MAD = sum(m + alpha * (|x| - m)) / sum(|x| - m),
# where m is the mean of the smoothed signal over a time period, and x is an original signal value
# alpha - user-configurable multiplicative constant, a small number (0.0002)
def mean_absolute_deviation(smoothed_data, original_data, i1, i2, alpha):
    mean = np.mean(smoothed_data[i1:i2])
    dev = np.abs(original_data[i1:i2]) - mean
    negative_deflection = np.sum(dev)
    mad = np.abs(np.sum(mean + alpha * dev) / negative_deflection)
    return mad
```

In [166…
```python
# Calculate the threshold
# thres = beta * m, where m is the mean of the signal over a time period
def calulate_threshold(smoothed_signal, i1, i2, beta):
    return beta * np.mean(smoothed_signal[i1:i2])
```

The **characteristic values** (in a **configured time window**) are shown as:

$ldist = mp - lp$ (time elapsed between first local maximum and local minimum)

$rdist = rp - mp$ (time elapsed between local minimum and second local maximum)

$ratio = \frac{rv}{|mv|}$ (relative voltage ratio of second local maximum and local minimum voltage magnitude)

$asym = \frac{lv}{rv}$ (meausre of asymmetric property - $asym = 1$ means the left & right local maxima are symmetric)

$cost = \frac{esterr}{|mv|}$ (ratio between second local maximum voltage and local minimum voltage magnitude)

where lp, mp, rp are the **left, middle, and right timestamps** and lv, mv, rv are the **left, middle, and right voltage values (shown in Fig. 4B above)**.

$esterr$ is given as the **mean absolute deviation (MAD) estimate**.

Note that the patent says to use the same **fixed** configurable time window for iterating over the fit values (local minima and maxima) to compute characteristic values + MAD. We are doing something *slightly different* here: Instead of sampling a **fixed** time window, we sample $[lp, rp]$ as our **NON-fixed time window**, since we already have the local minima and maxima; i.e., the interval that mean m is calculated over is **NOT fixed**, unlike in the patent. We can do this, as long as the computation intervals are consistent.

In [166…
```python
# Get characteristic values on the intervals i1, i2, based on previously computed fit values (lp, mp, rp, lv, mv, rv)
# alpha: hyperparam for MAD calculation
# beta: hyperparam for threshold calculation
def characteristic_values(lp, mp, rp, lv, mv, rv, i1, i2, alpha, beta):
    # Compute characteristic values
    ldist = mp - lp
    rdist = rp - mp
    ratio = rv / abs(mv)
    asym = lv / rv
    esterr = mean_absolute_deviation(forecast, x, i1, i2, alpha) # esterr - MAD of signal within time window
    cost = esterr / abs(mv)

    # Compute threshold
    thres = calulate_threshold(forecast, i1, i2, beta)

    return ldist, rdist, ratio, asym, esterr, cost, thres
```

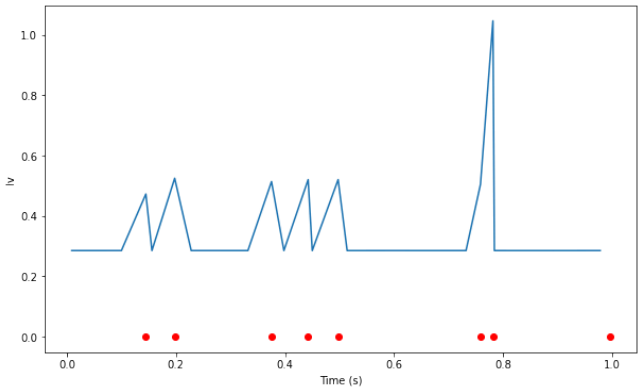
6. Determining Thresholds

These following cells in this section will help you determine the optimal thresholds.

In [167…
```
# Set alpha (for MAD calculation) and beta (for threshold calculation) as follows:
alpha = 0.0002
beta = 3.75
```

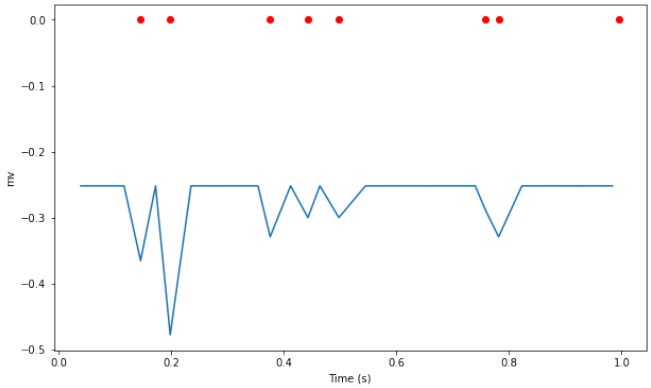
In [167…
```
def plot_characteristic(local_minima, local_maxima, characteristic, alpha, beta):
    output = []
    times = []
    spike_times = []
    spike_output = []

    i_pointer = 0

    MIN_WINDOW_BOUND_TOLERANCE_TIME = 0.005

    plt.figure(figsize=(10, 6))

    for idx in np.arange(0, len(local_maxima) - 1):
        if idx < len(local_minima): # bounds check
            # Compute fit values
            lp, mp, rp, lv, mv, rv = fit_values(local_minima, local_maxima, idx)

            # Compute characteristic values
            ldist, rdist, ratio, asym, esterr, cost, thres = characteristic_values(lp, mp, rp, lv, mv, rv, local_maxima[idx], local_maxima[idx + 1], alpha, be

            # Check if in range
            spike_in_range = (i_pointer < len(sampled_spike_indices)) and (t[sampled_spike_indices][i_pointer] >= lp or np.abs(t[sampled_spike_indices][i_poi

            if characteristic == 'asym':
                output.append(asym)
                times.append(lp)
                if spike_in_range > 0:
                    spike_times.append(lp)
                    spike_output.append(asym)
                    i_pointer += 1
            elif characteristic == 'ratio':
                output.append(ratio)
                times.append(rp)
                if spike_in_range:
                    spike_times.append(rp)
                    spike_output.append(ratio)
                    i_pointer += 1
            elif characteristic == 'cost':
                output.append(cost)
                times.append(rp)
                if spike_in_range:
                    spike_times.append(rp)
                    spike_output.append(cost)
                    i_pointer += 1
            elif characteristic == 'thres':
                output.append(thres)
                times.append(mp)
                if spike_in_range:
                    spike_times.append(mp)
                    spike_output.append(thres)
                    i_pointer += 1
            elif characteristic == 'ldist':
                output.append(ldist)
                times.append(mp)
                if spike_in_range:
                    spike_times.append(mp)
                    spike_output.append(ldist)
                    i_pointer += 1
            elif characteristic == 'rdist':
                output.append(rdist)
                times.append(mp)
                if spike_in_range:
                    spike_times.append(mp)
                    spike_output.append(rdist)
                    i_pointer += 1
            elif characteristic == 'lv':
                output.append(lv)
                times.append(lp)
                if spike_in_range:
                    spike_times.append(lp)
                    spike_output.append(lv)
                    i_pointer += 1
            elif characteristic == 'rv':
                output.append(rv)
                times.append(rp)
                if spike_in_range:
                    spike_times.append(rp)
                    spike_output.append(rv)
                    i_pointer += 1
            elif characteristic == 'mv':
                output.append(mv)
                times.append(mp)
                if spike_in_range:
                    spike_times.append(mp)
                    spike_output.append(mv)
                    i_pointer += 1

    plt.plot(times, output)
    plt.xlabel('Time (s)')
    plt.ylabel(characteristic)
    plt.scatter(t[sampled_spike_indices], 0.00004 + np.zeros(sampled_spike_indices.shape), color='r')
    plt.show()

    return np.array(spike_times), np.array(times), np.array(spike_output), np.array(output)
```

In [167…
```
# Plot lv
spike_times, times, spike_output, output = plot_characteristic(local_minima, local_maxima, characteristic='lv', alpha=alpha, beta=beta)
print('times where spikes detected:', spike_times)
print('times where spikes sampled:', t[sampled_spike_indices])
print('lv of spikes:', spike_output)
print('max lv of spikes:', np.max(spike_output))
```



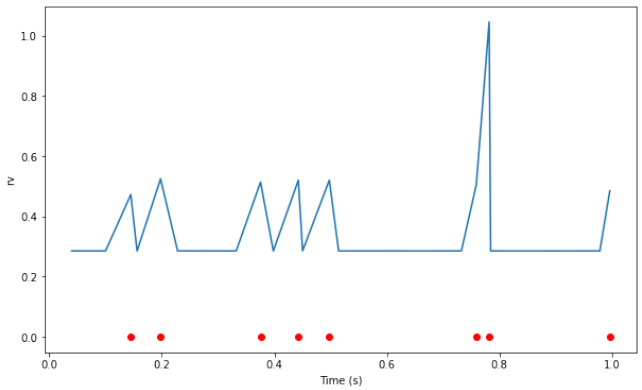

```
times where spikes detected: [0.1001 0.1561 0.3321 0.3981 0.4501 0.7321 0.7586 0.9781]
times where spikes sampled: [0.14495 0.1978  0.3754  0.44255 0.49755 0.7586  0.7814  0.9959 ]
lv of spikes: [0.28520591 0.28518013 0.28518639 0.28519456 0.28517769 0.2851732
 0.50493251 0.28520713]
max lv of spikes: 0.5049325062312797
```

In [167…
```
# Plot mv
spike_times, times, spike_output, output = plot_characteristic(local_minima, local_maxima, characteristic='mv', alpha=alpha, beta=beta)
print('times where spikes detected:', spike_times)
print('times where spikes sampled:', t[sampled_spike_indices])
print('mv of spikes:', spike_output)
print('min mv of spikes:', np.min(spike_output))
```



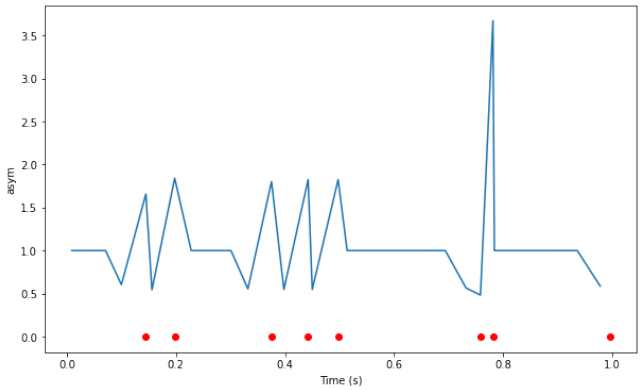

```
times where spikes detected: [0.1159  0.1719  0.3539  0.4119  0.4639  0.7399  0.75885 0.9839 ]
times where spikes sampled: [0.14495 0.1978  0.3754  0.44255 0.49755 0.7586  0.7814  0.9959 ]
mv of spikes: [-0.25212553 -0.2521511  -0.25211024 -0.25210188 -0.25211701 -0.25208399
 -0.2900306  -0.25216951]
min mv of spikes: -0.29003059604054715
```

In [167…
```
# Plot rv
spike_times, times, spike_output, output = plot_characteristic(local_minima, local_maxima, characteristic='rv', alpha=alpha, beta=beta)
print('times where spikes detected:', spike_times)
print('times where spikes sampled:', t[sampled_spike_indices])
print('rv of spikes:', spike_output)
print('min rv of spikes:', np.min(spike_output))
```



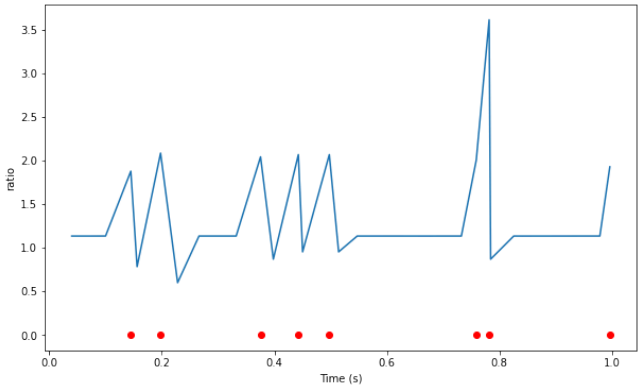

```
times where spikes detected: [0.14495 0.1978  0.3754  0.44255 0.49755 0.7586  0.78145 0.9958 ]
times where spikes sampled: [0.14495 0.1978  0.3754  0.44255 0.49755 0.7586  0.7814  0.9959 ]
rv of spikes: [0.47266696 0.5250145  0.51407666 0.52053878 0.52054226 0.50493251
 1.04720866 0.48538842]
min rv of spikes: 0.47266696281174525
```

In [167…
```
# Plot asym per lp
spike_times, times, spike_output, output = plot_characteristic(local_minima, local_maxima, characteristic='asym', alpha=alpha, beta=beta)
print('times where spikes detected:', spike_times)
print('times where spikes sampled:', t[sampled_spike_indices])
print('asym of spikes:', spike_output)
print('max asym of spikes:', np.max(spike_output))
```



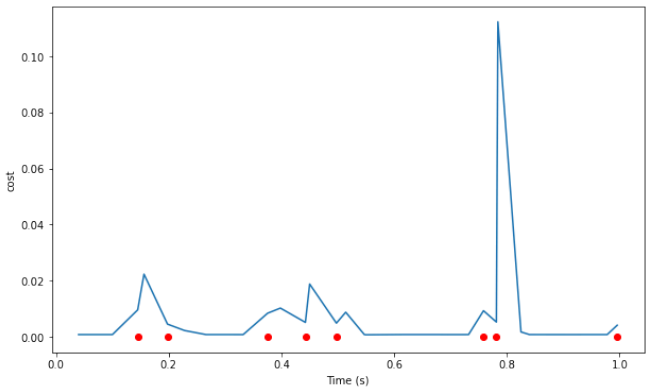

```
times where spikes detected: [0.1001 0.1561 0.3321 0.3981 0.4501 0.7321 0.7586 0.9781]
times where spikes sampled: [0.14495 0.1978  0.3754  0.44255 0.49755 0.7586  0.7814  0.9959 ]
asym of spikes: [0.60339718 0.54318525 0.55475459 0.54788341 0.54784734 0.56477489
 0.48216991 0.58758537]
max asym of spikes: 0.603397184337803
```

In [167…
```
# Plot ratio per rp
spike_times, times, spike_output, output = plot_characteristic(local_minima, local_maxima, characteristic='ratio', alpha=alpha, beta=beta)
print('times where spikes detected:', spike_times)
print('times where spikes sampled:', t[sampled_spike_indices])
print('ratio of spikes:', spike_output)
print('min ratio of spikes:', np.min(spike_output[spike_output > 0]))
```



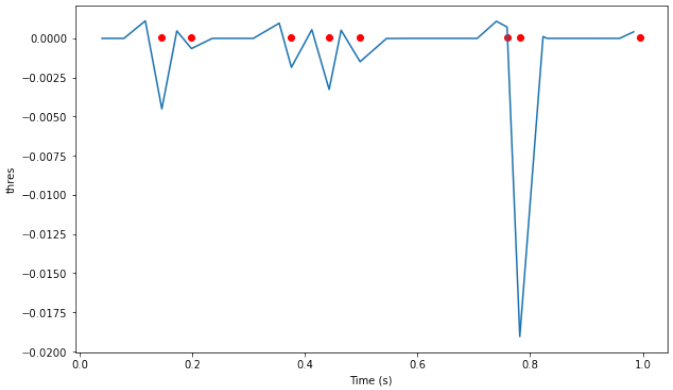

```
times where spikes detected: [0.14495 0.1978  0.3754  0.44255 0.49755 0.7586  0.78145 0.9958 ]
times where spikes sampled: [0.14495 0.1978  0.3754  0.44255 0.49755 0.7586  0.7814  0.9959 ]
ratio of spikes: [1.87472868 2.08214241 2.0390947  2.06479531 2.06468522 2.0030328
 3.61068342 1.92484977]
min ratio of spikes: 1.874728682921318
```

In [167…
```
# Plot cost per rp
spike_times, times, spike_output, output = plot_characteristic(local_minima, local_maxima, characteristic='cost', alpha=alpha, beta=beta)
print('times where spikes detected:', spike_times)
print('times where spikes sampled:', t[sampled_spike_indices])
print('cost of spikes:', spike_output)
print('max cost of spikes:', np.max(spike_output))
```



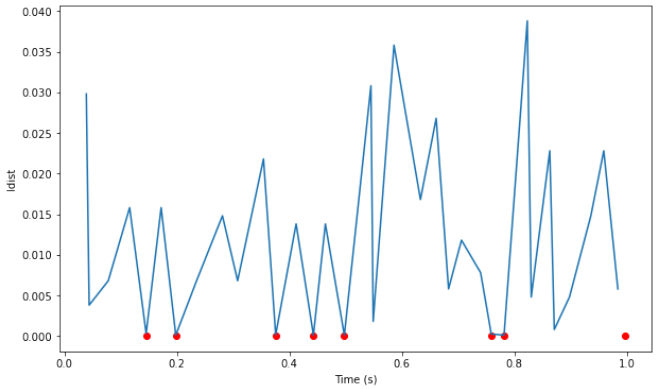

```
times where spikes detected: [0.14495 0.1978  0.3754  0.44255 0.49755 0.7586  0.78145 0.9958 ]
times where spikes sampled: [0.14495 0.1978  0.3754  0.44255 0.49755 0.7586  0.7814  0.9959 ]
cost of spikes: [0.00960776 0.00452327 0.00839692 0.00513683 0.00487107 0.0092961
 0.00526012 0.00412251]
max cost of spikes: 0.009607759628933473
```

In [167…
```
# Plot thres
spike_times, times, spike_output, output = plot_characteristic(local_minima, local_maxima, characteristic='thres', alpha=alpha, beta=beta)
print('times where spikes detected:', spike_times)
print('times where spikes sampled:', t[sampled_spike_indices])
print('thres of spikes:', spike_output)
print('min thres of spikes:', np.min(spike_output))
print('max thres of spikes:', np.max(spike_output))
```



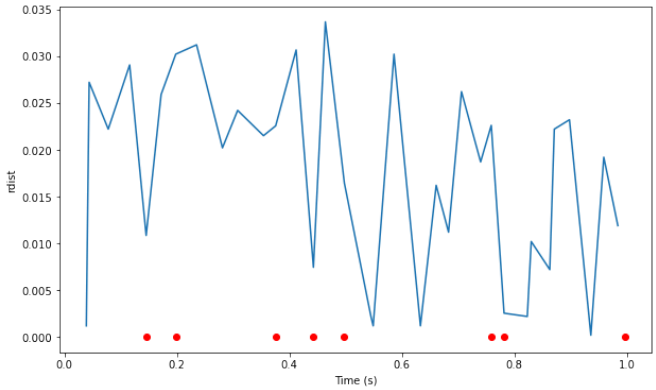

```
times where spikes detected: [0.1159  0.1719  0.3539  0.4119  0.4639  0.7399  0.75885 0.9839 ]
times where spikes sampled: [0.14495 0.1978  0.3754  0.44255 0.49755 0.7586  0.7814  0.9959 ]
thres of spikes: [0.00111947 0.00047468 0.00096956 0.00055362 0.00051933 0.00109312
 0.00072903 0.0004169 ]
min thres of spikes: 0.00041690318720667496
max thres of spikes: 0.0011194661342500355
```

In [167…
```
# Plot ldist
spike_times, times, spike_output, output = plot_characteristic(local_minima, local_maxima, characteristic='ldist', alpha=alpha, beta=beta)
print('times where spikes detected:', spike_times)
print('times where spikes sampled:', t[sampled_spike_indices])
print('ldist of spikes:', spike_output)
print('max ldist of spikes:', np.max(spike_output))
```




```
times where spikes detected: [0.1159  0.1719  0.3539  0.4119  0.4639  0.7399  0.75885 0.9839 ]
times where spikes sampled: [0.14495 0.1978  0.3754  0.44255 0.49755 0.7586  0.7814  0.9959 ]
ldist of spikes: [0.0158  0.0158  0.0218  0.0138  0.0138  0.0078  0.00025 0.0058 ]
max ldist of spikes: 0.021799999999999986
```

In [168…
```
# Plot rdist
spike_times, times, spike_output, output = plot_characteristic(local_minima, local_maxima, characteristic='rdist', alpha=alpha, beta=beta)
print('times where spikes detected:', spike_times)
print('times where spikes sampled:', t[sampled_spike_indices])
print('rdist of spikes:', spike_output)
print('min rdist of spikes:', np.min(spike_output))
print('max rdist of spikes:', np.max(spike_output))
```




```
times where spikes detected: [0.1159  0.1719  0.3539  0.4119  0.4639  0.7399  0.75885 0.9839 ]
times where spikes sampled: [0.14495 0.1978  0.3754  0.44255 0.49755 0.7586  0.7814  0.9959 ]
rdist of spikes: [0.02905 0.0259  0.0215  0.03065 0.03365 0.0187  0.0226  0.0119 ]
min rdist of spikes: 0.011900000000000022
max rdist of spikes: 0.03365000000000001
```

Based on the above trials, α, β params, as well as more manual tuning, I set the thresholds to the following:

In [168…
```
# Set threshold values here
MAX_ASYM = 4
MIN_RATIO = 1.25
MAX_COST = 0.1
MIN_THRES = -0.004
MAX_THRES = 0.002
```

```
MAX_LDIST = 0.1
MIN_RDIST = 5e-5
MAX_RDIST = 0.1
```

7. Determine Whether Comparison indicates Spike + Timestamp of detection

The overall algorithm for determining spikes is listed below:

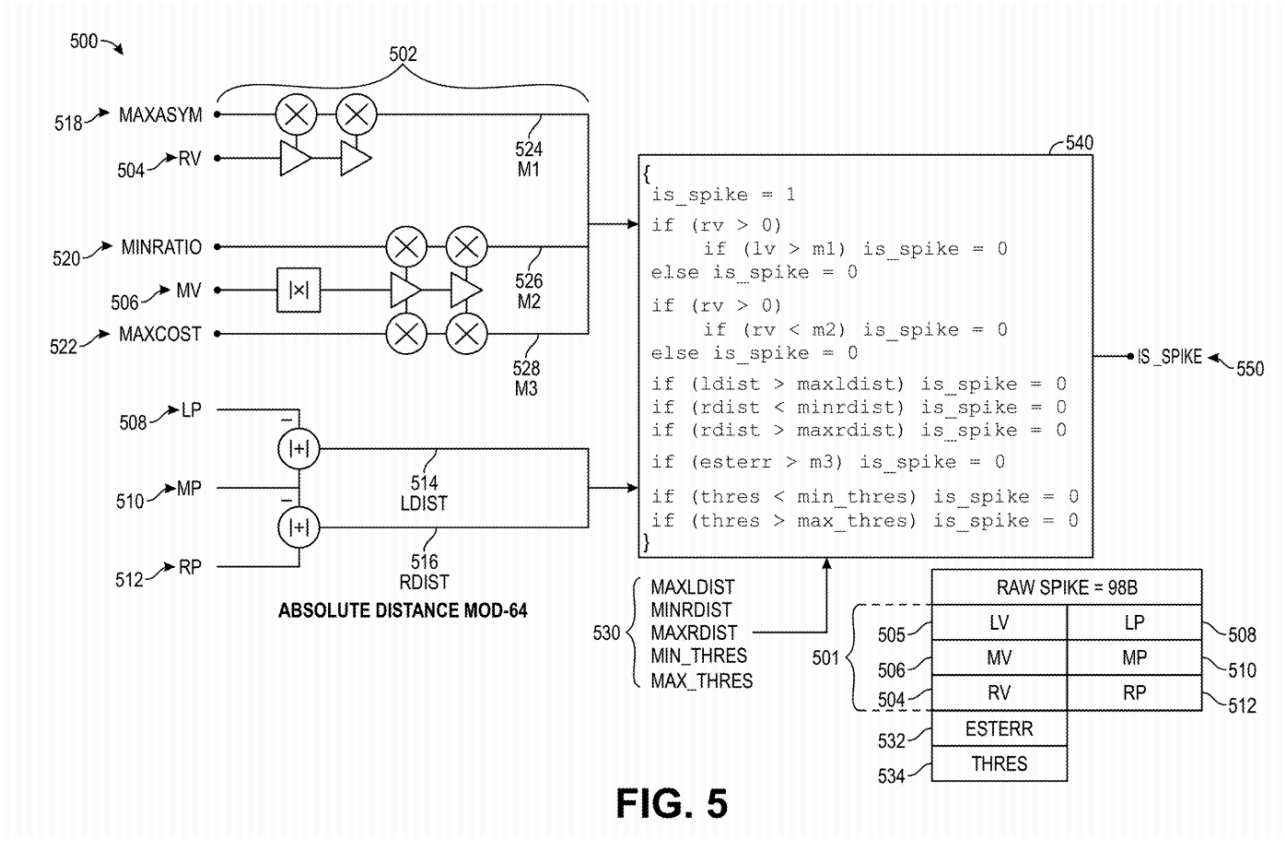


FIG. 5

In [168…

```
# Compare characteristic values with threshold
# idx: index of the local maxima/minima
def is_spike(idx, alpha=0.0002, beta=3.75, MAX_ASYM=0.3, MIN_RATIO=4/3, MAX_COST=0.4, MIN_THRES=0.2, MAX_THRES=0.7, MAX_LDIST=0.02, MIN_RDIST=0.0001, MAX_RDIS
    # assert beta in [0, 7.875] range (Q3.3)
    assert(beta >= 0 and beta <= 7.875)

    # Compute fit values
    lp, mp, rp, lv, mv, rv = fit_values(local_minima, local_maxima, idx)

    # Compute characteristic values
    ldist, rdist, ratio, asym, esterr, cost, thres = characteristic_values(lp, mp, rp, lv, mv, rv, local_maxima[idx], local_maxima[idx + 1], alpha, beta)

    m1 = rv * MAX_ASYM # compare with lv
    m2 = abs(mv) * MIN_RATIO # compare with rv
    m3 = abs(mv) * MAX_COST # compare with esterr

    # Determine whether signal represents a spike
    # spike: rv MUST be positive!!
    if rv > 0:
        # lv must be under a certain value (m1)
        if lv > m1:
            return 0
        # rv must be at least a certain value (m2)
        if rv < m2:
            return 0
    else:
        return 0

    # threshold thres must be in specified range [MIN_THRES, MAX_THRES]
    if thres < MIN_THRES:
        return 0

    if thres > MAX_THRES:
        return 0

    # ldist cannot exceed MAX_LDIST
    if ldist > MAX_LDIST:
        return 0

    # rdist must be in specified range [MIN_RDIST, MAX_RDIST]
    if rdist < MIN_RDIST:
        return 0

    if rdist > MAX_RDIST:
        return 0

    # MAD must be under certain value (m3)
    if esterr > m3:
```

```
        return 0

    spike_payload = {
        'lp': lp,
        'mp': mp,
        'rp': rp,
        'lv': lv,
        'mv': mv,
        'rv': rv,
        'ldist': ldist,
        'rdist': rdist,
        'ratio': ratio,
        'asym': asym,
        'cost': cost,
        'thres': thres
    }

    return (1, spike_payload)
```

In [168…
```
# Sample each window (based on the local max-min-max values)
idx = 0
spikes = []

while idx < len(local_maxima) - 1:
    if idx < len(local_minima): # bounds check
        assert(local_maxima[idx] < local_minima[idx])
        assert(local_minima[idx] < local_maxima[idx + 1])

        # Determine whether a spike is detected
        spike = is_spike(idx, alpha=alpha, beta=beta, MAX_ASYM=MAX_ASYM, MIN_RATIO=MIN_RATIO, MAX_COST=MAX_COST, MIN_THRES=MIN_THRES, MAX_THRES=MAX_THRES, MA

        if spike:
            spikes.append(spike)
            idx += 2
            continue

        idx += 1
```

8. Transmit Indication of Spike (if detected)

- Transmit **timestamp** of corresponding spike and **value "1"** that spike was identified
- Can be transmitted via wired or wireless connection

In [168…
```
spikes
```

Out[168…

```
[(1,
  {'lp': 0.10010000000000001,
   'mp': 0.1159,
   'rp': 0.14495,
   'lv': 0.2852059144901081,
   'mv': -0.2521255300128051,
   'rv': 0.47266696281174525,
   'ldist': 0.015799999999999995,
   'rdist': 0.029049999999999992,
   'ratio': 1.874728682921318,
   'asym': 0.603397184337803,
   'cost': 0.009607759628933473,
   'thres': 0.0011194661342500355}),
 (1,
  {'lp': 0.15610000000000002,
   'mp': 0.1719,
   'rp': 0.1978,
   'lv': 0.2851801349344209,
   'mv': -0.25215110277674246,
   'rv': 0.5250145045427902,
   'ldist': 0.01579999999999998,
   'rdist': 0.025900000000000006,
   'ratio': 2.082142408901714,
   'asym': 0.5431852500585113,
   'cost': 0.004523270988724369,
   'thres': 0.0004746814564677536}),
 (1,
  {'lp': 0.3321,
   'mp': 0.3539,
   'rp': 0.3754,
   'lv': 0.2851863858213671,
   'mv': -0.252110242501527,
   'rv': 0.5140766593201013,
   'ldist': 0.021799999999999986,
   'rdist': 0.02150000000000002,
   'ratio': 2.0390947000774373,
   'asym': 0.5547545889333783,
   'cost': 0.008396923503133213,
   'thres': 0.0009695607027081244}),
 (1,
  {'lp': 0.3981,
   'mp': 0.41190000000000004,
   'rp': 0.44255,
   'lv': 0.28519456418378586,
   'mv': -0.2521018807336064,
   'rv': 0.5205387809797631,
   'ldist': 0.013800000000000034,
   'rdist': 0.030649999999999955,
   'ratio': 2.064795309995372,
   'asym': 0.5478834135028131,
   'cost': 0.005136826710944883,
   'thres': 0.0005536206253423961}),
 (1,
  {'lp': 0.4501,
   'mp': 0.46390000000000003,
   'rp': 0.49755000000000005,
   'lv': 0.2851776928997619,
   'mv': -0.25211700624411704,
   'rv': 0.5205422561699117,
   'ldist': 0.013800000000000034,
   'rdist': 0.03365000000000001,
   'ratio': 2.0646852186793256,
   'asym': 0.5478473448016797,
   'cost': 0.0048710704251438315,
   'thres': 0.0005193300009530326}),
 (1,
  {'lp': 0.7321000000000001,
   'mp': 0.7399,
   'rp': 0.7586,
   'lv': 0.2851732002158608,
   'mv': -0.2520839929346945,
   'rv': 0.5049325062312797,
   'ldist': 0.007799999999999918,
   'rdist': 0.018700000000000005,
   'ratio': 2.0030328001115434,
   'asym': 0.5647748891120902,
   'cost': 0.009296096593837497,
   'thres': 0.0010931243099576 46}),
 (1,
  {'lp': 0.9781000000000001,
   'mp': 0.9839,
   'rp': 0.9958,
   'lv': 0.28520713435383743,
   'mv': -0.2521695080571091,
   'rv': 0.48538842048395736,
   'ldist': 0.005799999999999916,
   'rdist': 0.011900000000000022,
   'ratio': 1.924849773565926,
   'asym': 0.5875853694026553,
   'cost': 0.004122506629797255,
   'thres': 0.0004169031872066749 6})]
```

In [168…

```
def get_spikes_df(spikes):
    spike_payloads = [spike[1] for spike in spikes]
    df = pd.DataFrame(spike_payloads)
    return df
```

In [168…

```
df = get_spikes_df(spikes)
df
```

Out[168…

	lp	mp	rp	lv	mv	rv	ldist	rdist	ratio	asym	cost	thres
0	0.1001	0.1159	0.14495	0.285206	-0.252126	0.472667	0.0158	0.02905	1.874729	0.603397	0.009608	0.001119
1	0.1561	0.1719	0.19780	0.285180	-0.252151	0.525015	0.0158	0.02590	2.082142	0.543185	0.004523	0.000475
2	0.3321	0.3539	0.37540	0.285186	-0.252110	0.514077	0.0218	0.02150	2.039095	0.554755	0.008397	0.000970
3	0.3981	0.4119	0.44255	0.285195	-0.252102	0.520539	0.0138	0.03065	2.064795	0.547883	0.005137	0.000554
4	0.4501	0.4639	0.49755	0.285178	-0.252117	0.520542	0.0138	0.03365	2.064685	0.547847	0.004871	0.000519
5	0.7321	0.7399	0.75860	0.285173	-0.252084	0.504933	0.0078	0.01870	2.003033	0.564775	0.009296	0.001093
6	0.9781	0.9839	0.99580	0.285207	-0.252170	0.485388	0.0058	0.01190	1.924850	0.587585	0.004123	0.000417

In [168…

```
# Convert float to binary (in hexadecimal format - 8 bits per word)
def float_to_binary_hex(num):
    return "{:08x}".format(struct.unpack('<I', struct.pack('<f', num))[0])
```

In [168…

```
bitstream = ''
for val in df['rp']:
    bin_hex = float_to_binary_hex(val)
    bitstream += '1' + str(bin_hex) + ', '
print('bitstream to transmit:', bitstream)
```

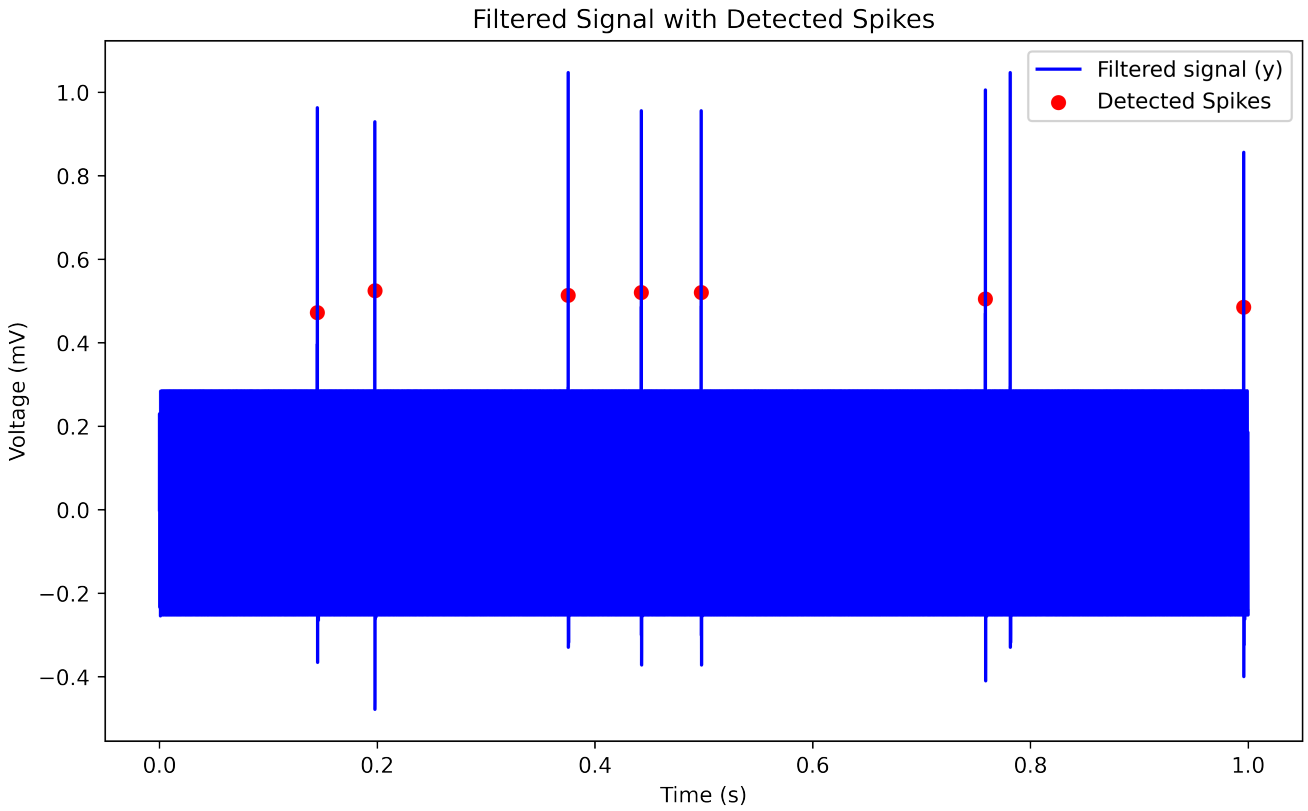
```
bitstream to transmit: 13e146dc6, 13e4a8c15, 13ec0346e, 13ee295ea, 13efebee0, 13f42339c, 13f7eecc0,
```

Plot the detected spikes

In [168…

```
plt.figure()
fig, ax = plt.subplots(1, 1, figsize=(10, 6), dpi = 600)
plt.subplots_adjust(hspace=0.4)

# Plot signals for spikes
plt.plot(t, y, color='blue', linestyle='-', label='Filtered signal (y)', zorder=1)
plt.scatter(df['rp'], df['rv'], color='red', label='Detected Spikes')

plt.title("Filtered Signal with Detected Spikes")
plt.xlabel("Time (s)")
plt.ylabel("Voltage (mV)")
plt.legend()

plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```



For the most part, we can see the **spikes are detected correctly**. For spikes that are relatively close to each other, only one of them is detected. Every spike that is spread out is detected.

Note that **if the local maximum in the first time window happens to be a spike, it is ignored**, as normally it would be **detected in a previous window before time 0** (since neural signals are sampled in ***real-time***).

Additional: Classify Spikes into Bins

In practice, there are **billions** of neural signals / spikes being sampled at the same time, so much more sophisticated plots are needed. But here, since we do not deal with a lot of spikes, we can just simply plot a histogram for each of them.

For example, we can **give each bin a value (0, 1, 2, 3, ...)**, and transmit it along with a **timestamp of when the spike is detected (`rp`)**.

Classification of spikes is advantageous, as **two spikes** with **two different values** could indicate **two different actions** (e.g., one neuron means **moving a hand up**, another neuron means **moving a hand down**).

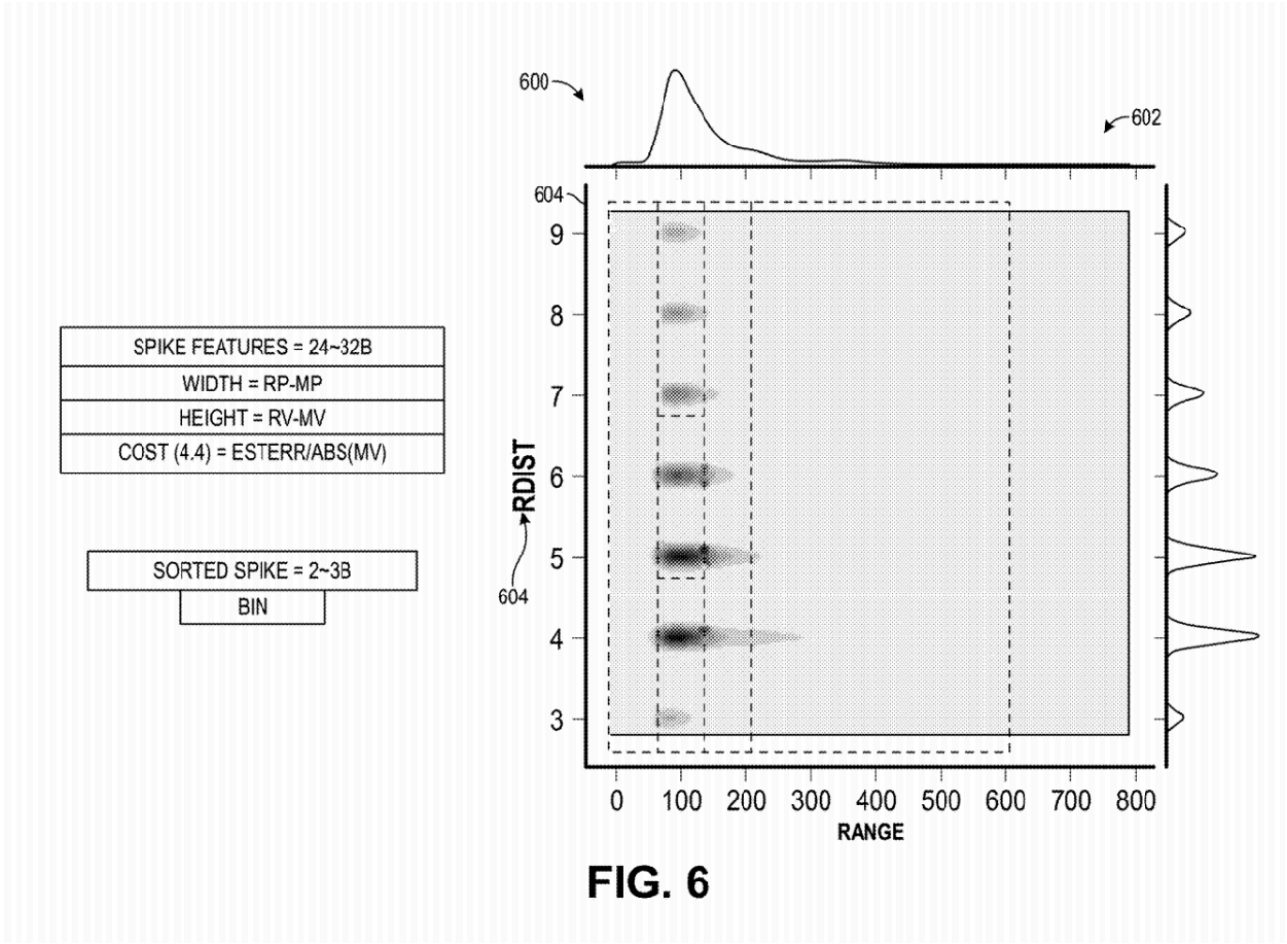


FIG. 6

In [169…
```python
# Define a function to calculate histogram plot
def calculate_histogram(data, bins):
    hist, bin_edges = np.histogram(data, bins=bins)
    return hist, bin_edges

# Classify each point into bins
def classify_points(data, hist, bin_edges):
    # Initialization: zeros
    classification = np.zeros(len(data), dtype=int)

    # Iterate each point, then find the range and then its corresponding bin classification
    for i, point in enumerate(data):
        for j, (lower_bound, upper_bound) in enumerate(zip(bin_edges, bin_edges[1:])):
            if lower_bound <= point <= upper_bound:
                classification[i] = j
                break
    return classification
```

In [169…
```python
# New dataframe - for binning
df2 = pd.DataFrame(df['rp'])
df2.rename(columns={'rp': 'timestamp of spike (s)'}, inplace=True)
```

In [169…
```python
# Set # of bins here
n_bins = 5
```

Time between min & max voltage (rdist)

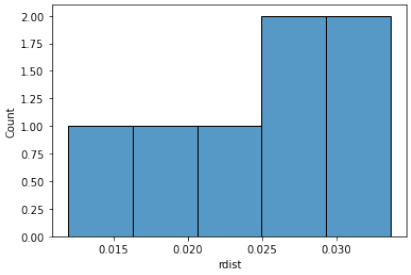
In [169…
```python
# Calculate histogram
hist, bin_edges = calculate_histogram(df['rdist'], bins=n_bins)

# Classify points
classification = classify_points(df['rdist'], hist, bin_edges)

# Print result
df2['rdist'] = df['rdist']
df2['bin'] = classification
df2
```

Out[169…

	timestamp of spike (s)	rdist	bin
0	0.14495	0.02905	3
1	0.19780	0.02590	3
2	0.37540	0.02150	2
3	0.44255	0.03065	4
4	0.49755	0.03365	4
5	0.75860	0.01870	1
6	0.99580	0.01190	0

In [169…

```
# Plot histogram
sns.histplot(df['rdist'], bins=n_bins)
plt.show()
```



Time between last max and min voltage (ldist)

In [169…

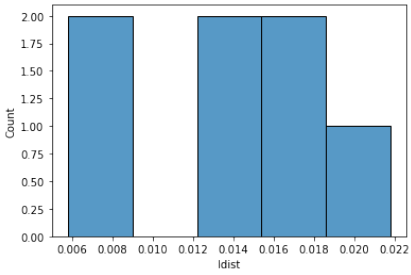
```
# Calculate histogram
hist, bin_edges = calculate_histogram(df['ldist'], bins=n_bins)

# Classify points
classification = classify_points(df['ldist'], hist, bin_edges)

# Print result
df2 = df2.rename(columns={'rdist': 'ldist'})
df2['ldist'] = df['ldist']
df2['bin'] = classification
df2
```

Out[169…

	timestamp of spike (s)	ldist	bin
0	0.14495	0.0158	3
1	0.19780	0.0158	3
2	0.37540	0.0218	4
3	0.44255	0.0138	2
4	0.49755	0.0138	2
5	0.75860	0.0078	0
6	0.99580	0.0058	0

In [169…

```
# Plot histogram
sns.histplot(df['ldist'], bins=n_bins)
plt.show()
```



Maximum voltage values (rv)

In [169…

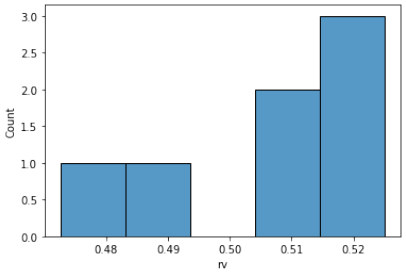
```
# Calculate histogram
hist, bin_edges = calculate_histogram(df['rv'], bins=n_bins)

# Classify points
classification = classify_points(df['rv'], hist, bin_edges)

# Print result
df2 = df2.rename(columns={'ldist': 'rv'})
df2['rv'] = df['rv']
df2['bin'] = classification
df2
```

Out[169…

	timestamp of spike (s)	rv	bin
0	0.14495	0.472667	0
1	0.19780	0.525015	4
2	0.37540	0.514077	3
3	0.44255	0.520539	4
4	0.49755	0.520542	4
5	0.75860	0.504933	3
6	0.99580	0.485388	1

In [169…

```
# Plot histogram
sns.histplot(df['rv'], bins=n_bins)
plt.show()
```



Magnitude of left hump (lv)

In [169…

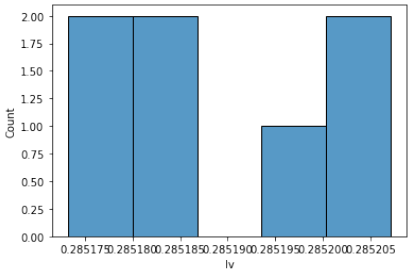
```
# Calculate histogram
hist, bin_edges = calculate_histogram(df['lv'], bins=n_bins)

# Classify points
classification = classify_points(df['lv'], hist, bin_edges)

# Print result
df2 = df2.rename(columns={'rv': 'lv'})
df2['lv'] = df['lv']
df2['bin'] = classification
df2
```

Out[169…

	timestamp of spike (s)	lv	bin
0	0.14495	0.285206	4
1	0.19780	0.285180	1
2	0.37540	0.285186	1
3	0.44255	0.285195	3
4	0.49755	0.285178	0
5	0.75860	0.285173	0
6	0.99580	0.285207	4

In [170…

```
# Plot histogram
sns.histplot(df['lv'], bins=n_bins)
plt.show()
```



Relative magnitude of left and right voltages (asym)

In [170…

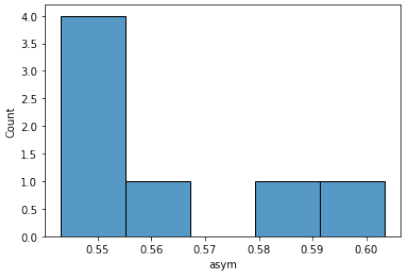
```
# Calculate histogram
hist, bin_edges = calculate_histogram(df['asym'], bins=n_bins)

# Classify points
classification = classify_points(df['asym'], hist, bin_edges)

# Print result
df2 = df2.rename(columns={'lv': 'asym'})
df2['asym'] = df['asym']
df2['bin'] = classification
df2
```

Out[170…

	timestamp of spike (s)	asym	bin
0	0.14495	0.603397	4
1	0.19780	0.543185	0
2	0.37540	0.554755	0
3	0.44255	0.547883	0
4	0.49755	0.547847	0
5	0.75860	0.564775	1
6	0.99580	0.587585	3

In [170…

```
# Plot histogram
sns.histplot(df['asym'], bins=n_bins)
plt.show()
```



Relative (absolute) magnitude between right and middle voltages (ratio)

In [170…

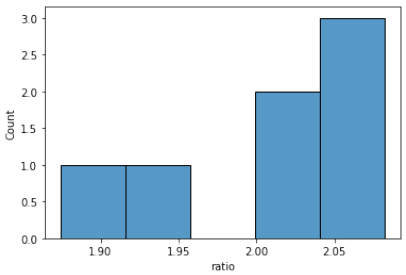
```
# Calculate histogram
hist, bin_edges = calculate_histogram(df['ratio'], bins=n_bins)

# Classify points
classification = classify_points(df['ratio'], hist, bin_edges)

# Print result
df2 = df2.rename(columns={'asym': 'ratio'})
df2['ratio'] = df['ratio']
df2['bin'] = classification
df2
```

Out[170…

	timestamp of spike (s)	ratio	bin
0	0.14495	1.874729	0
1	0.19780	2.082142	4
2	0.37540	2.039095	3
3	0.44255	2.064795	4
4	0.49755	2.064685	4
5	0.75860	2.003033	3
6	0.99580	1.924850	1

In [170…

```
# Plot histogram
sns.histplot(df['ratio'], bins=n_bins)
plt.show()
```



Ratio of estimated MAD to absolute value of second amplitude (cost)

In [170…

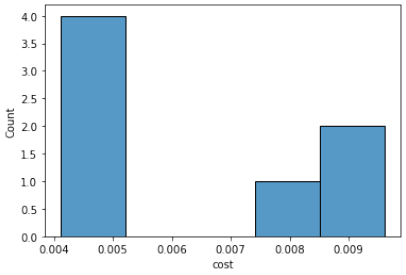
```
# Calculate histogram
hist, bin_edges = calculate_histogram(df['cost'], bins=n_bins)

# Classify points
classification = classify_points(df['cost'], hist, bin_edges)

# Print result
df2 = df2.rename(columns={'ratio': 'cost'})
df2['cost'] = df['cost']
df2['bin'] = classification
df2
```

Out[170…

	timestamp of spike (s)	cost	bin
0	0.14495	0.009608	4
1	0.19780	0.004523	0
2	0.37540	0.008397	3
3	0.44255	0.005137	0
4	0.49755	0.004871	0
5	0.75860	0.009296	4
6	0.99580	0.004123	0

In [170…

```
# Plot histogram
sns.histplot(df['cost'], bins=n_bins)
plt.show()
```



Advantages of this system

- **Significant reduction in size** of signal (while **keeping useful information**)
- **Reduce received waveforms** by **3 orders of magnitude** for **low-power wireless transmission** (for 1000 channels, reduction from *gigabits* per second to *megabits* per second)
- Can be performed **much faster** with **lower memory & storage** requirements
- **Near real-time** detecion (approximately **1 microsecond** from receiving signal to spike detection)
- **Less power consumption** (1000 channels - consumes 1-2 milli-Watts per channel or less; 256 channels - under 50 milli-Watts, which is 10x more reduction than in prior systems)

Hardware Implementation

As mentioned earlier above, the actual algorithm is **implemented in hardware**, since all variables here use **fixed-point computation**, as it **preserves resources while maintaining precision**.

An example schematic of this implementation (showing MAD and threshold calculations) is shown here:

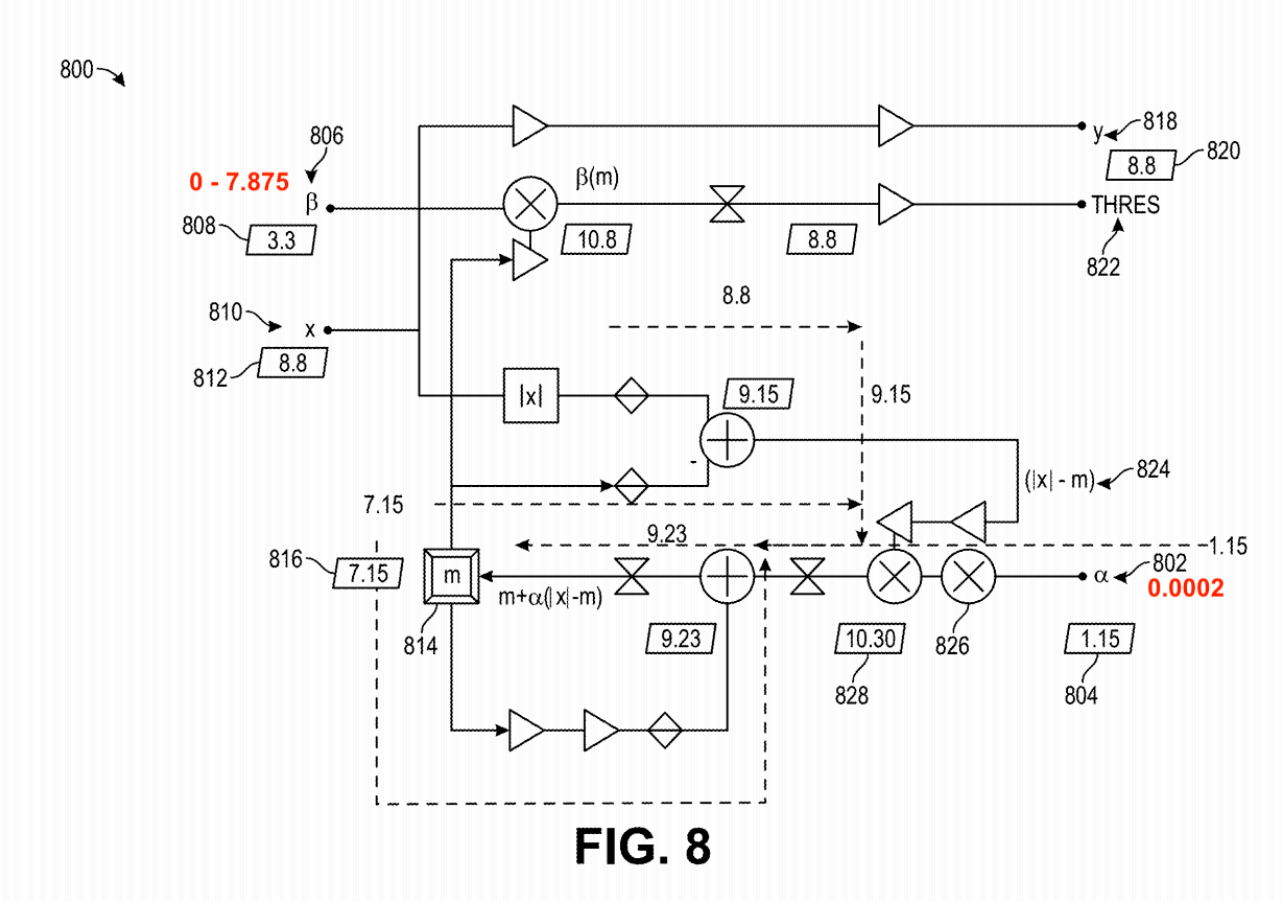

800
806
0 - 7.875
β
808
3.3
β(m)
10.8
8.8
y
818
8.8
820
THRES
822
810
x
812
8.8
8.8
|x|
9.15
9.15
-
(|x| - m)
824
7.15
9.23
1.15
816
7.15
m
m+α(|x|-m)
814
9.23
10.30
828
826
1.15
804
α
802
0.0002

FIG. 8

In [ ]: