Neural Signal Compression for Brain Computer Interfaces

Neuralink, US Patent No. 2023/0284982 A1, Filed 3/9/2022, Published 9/14/2023

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

Background

- **Neural signal compression** is a technique used for **transmitting and processing neural data**
- Gathering/transmitting data as a whole wirelessly leads to **excess information overflow**, which **increases latency/delay** of signal transmission.
- **Data compression** solves this problem by preventing overflow/latency issues, reduces size and space, improving overall system performance.

Goal

This notebook provides an implementation of the 4 neural signal compression algorithms described in US patent application number US 2023/0284982 by Neuralink, titled "Neural Signal Compression for Brain Computer Interfaces".

The 4 compression algorithms include:

- **Lossless** compression
- **Lossy** compression
- **Sparse** compression: **Binned-Spikes** mode
- **Sparse** compression: **Spike-Band** mode

In [1]:
```
import numpy as np
import math
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import scipy.stats as stats
import pandas as pd
from tqdm import tqdm
import zlib
import struct
import pywt
import array
import fractions
import random
from fractions import Fraction
from functools import reduce
```

```
/Users/michaelzhou/opt/anaconda3/lib/python3.8/site-packages/scipy/__init__.py:138: UserWarning: A NumPy version >=1.16.5 and <1.23.0 is required for this ver
sion of SciPy (detected version 1.24.4)
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion} is required for this version of "
```

Neural Transmitter - Overall Signal Flowchart

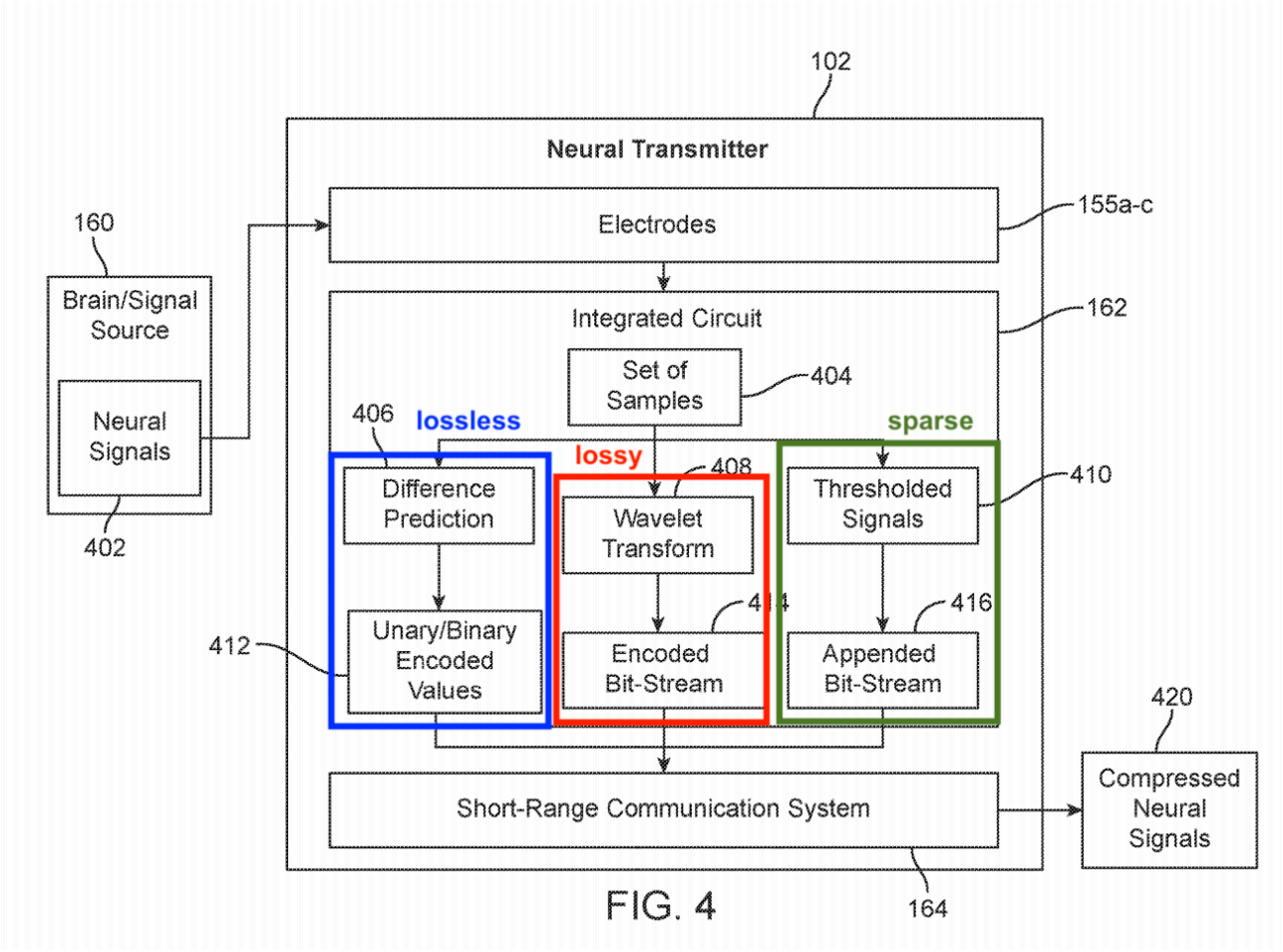


FIG. 4

Lossless Compression

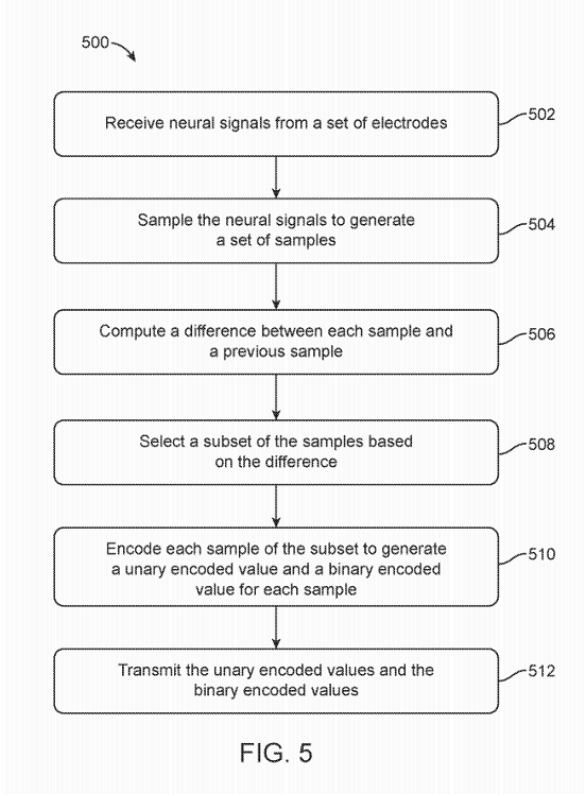


FIG. 5

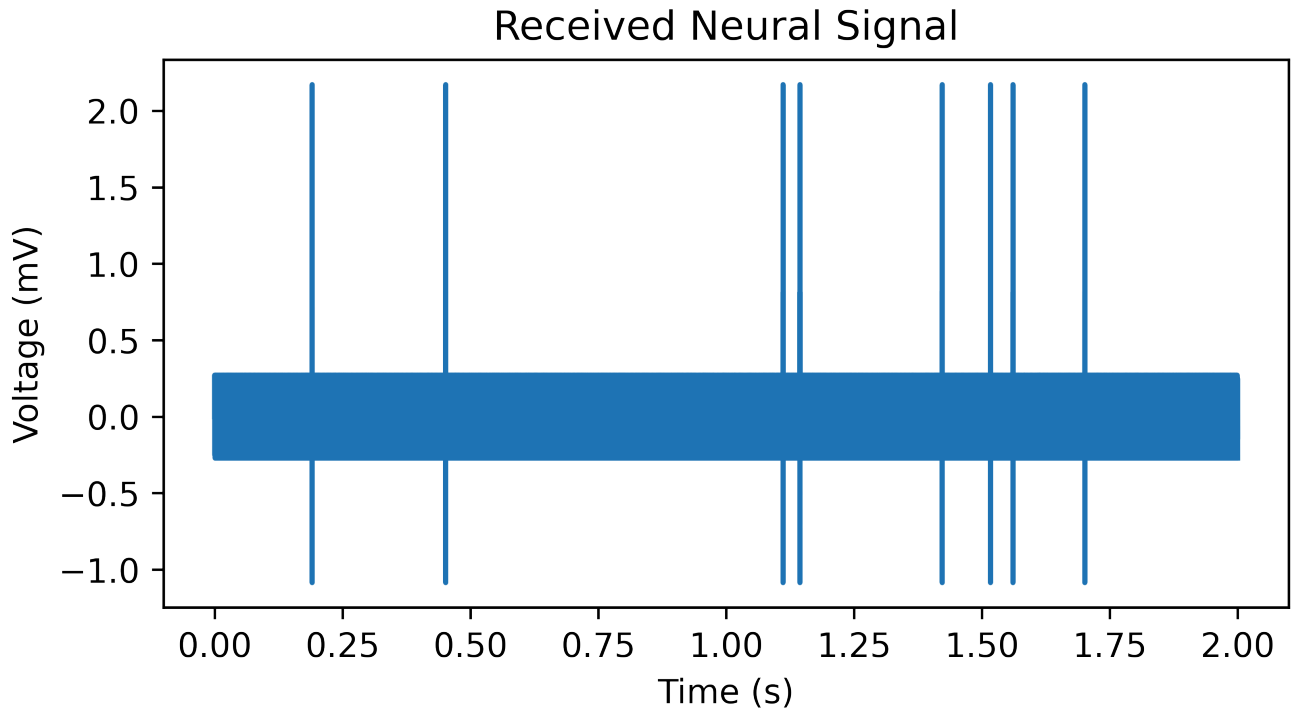
1. Receive Signal

Receive neural signals from electrodes.

Since we do not have the actual equipment, let's generate a (simulated) neural signal.

In [2]:
```
fs = 20000
T = 2
```

In [3]:
```
%run sampler.ipynb

sampler = Sampler(fs, T, n_spikes=8)
t = sampler.t
x = sampler.sample()
```



2. Generate Set of Samples (that correspond to spikes)

Detect spikes in the signal.

Refer to the notebook for US patent 2021/0012909 A1, "Real-Time Neural Spike Detection".

The patent mentions **two operation modes**: **broadband** and **spike-snippets**. To save computational resources and memory, let's use the **spike-snippets** mode (only take the samples that revolve around a spike).

Detect spikes:

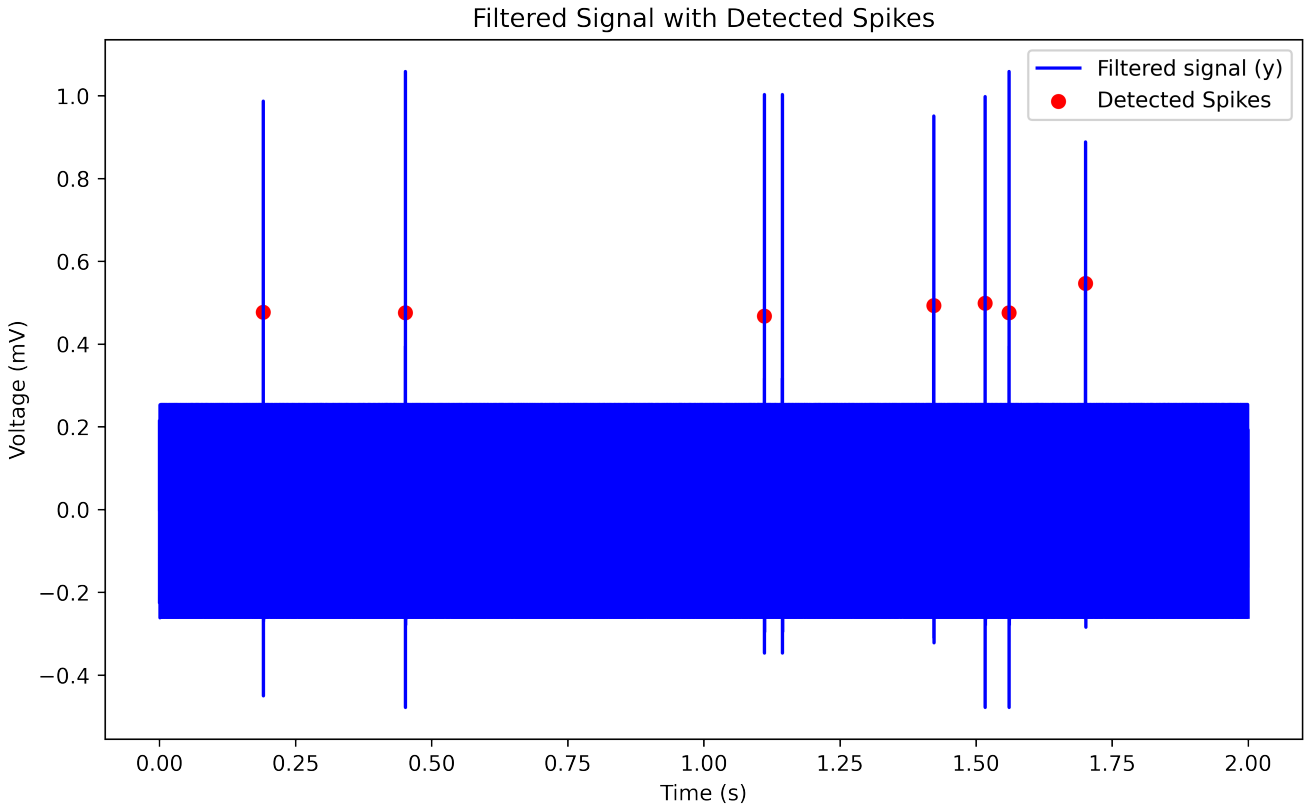
In [4]:
```
blackout_period = 0.04
```

In [5]:
```
# Used for MAD + threshold calculations
alpha=0.0002
beta=3.75
```

In [6]:
```
# Set threshold values here
MAX_ASYM = 4
MIN_RATIO = 1.01
MAX_COST = 2
MIN_THRES = -0.004
MAX_THRES = 0.002
MAX_LDIST = 0.1
MIN_RDIST = 5e-5
MAX_RDIST = 0.1
```

In [7]:
```
%run spike_detector.ipynb

spike_detector = SpikeDetector(x, t,
                               fs,
                               blackout_period = blackout_period,
                               alpha = alpha,
                               beta = beta,
                               MIN_RATIO = MIN_RATIO,
                               MAX_ASYM = MAX_ASYM,
                               MAX_COST = MAX_COST,
                               MIN_THRES = MIN_THRES,
                               MAX_THRES = MAX_THRES,
                               MAX_LDIST = MAX_LDIST,
                               MIN_RDIST = MIN_RDIST,
                               MAX_RDIST = MAX_RDIST
                              )
spikes = spike_detector.detect()
spike_detector.plot_spikes(spikes)
df = spike_detector.spikes_dataframe(spikes)
```

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



Spike-Snippet Mode: Take Snippets of Samples that Revolve around Spikes

In [8]:
```python
x = spike_detector.x
t = spike_detector.t
spike_times = np.array([spike[1]['rp'] for spike in spikes])
```

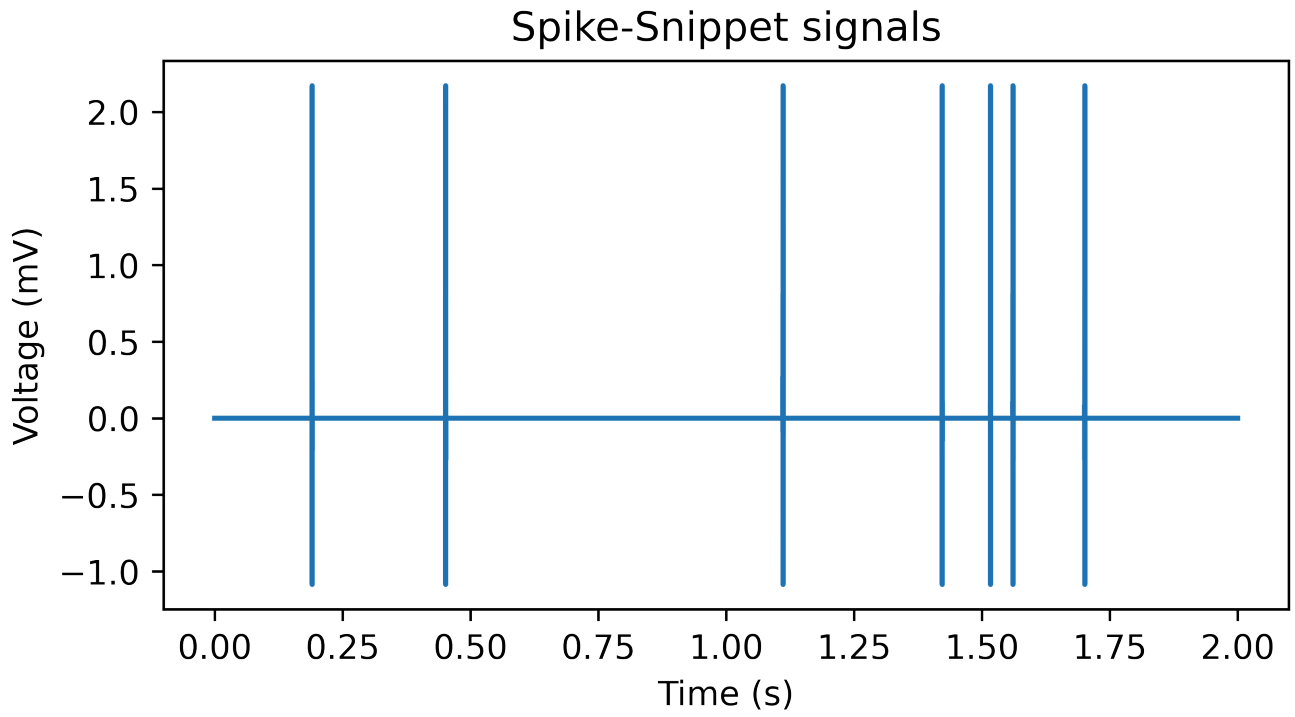
In [9]:
```python
# Converts time t (in seconds) to index in data
def time_to_index(t):
    return int(t * fs)
```

In [10]:
```python
def get_signal_snippets(x, t, spike_times, window_size=5):
    """
    Get snippets of the original signal centered around spike_times with a specific window size

    params:
    - signal: original signal
    - spike_times: times of each spike
    - window_size: length of window from point center

    Return:
    - snippets: snippets of the original signal
    """
    # Convert detected spike times to indices
    f = np.vectorize(time_to_index)
    spike_indices = f(spike_times)

    # Return the snippets centered around the spike indices with intervals: [point - window_size, point + window_size]
    snippets = np.array([x[point - window_size : point + window_size] for point in spike_indices])
    snippet_times = np.array([t[point - window_size : point + window_size] for point in spike_indices])
    return snippets, snippet_times

snippets, snippet_times = get_signal_snippets(x, t, spike_times, window_size=5)
```

In [11]:
```python
# Convert snippet times to indices
f = np.vectorize(time_to_index)
snippet_indices = f(snippet_times).ravel()
```

In [12]:
```python
# Make a copy of x first
x_snippets = np.copy(x)

# For all signals that are not part of the spike-snippets, set its value to zero.
for i in range(len(t)):
    if i not in snippet_indices:
        x_snippets[i] = 0
```

In [13]:
```python
# Plot the spike-snippet signals
fig, ax = plt.subplots(1, 1, figsize=(6, 3),dpi = 600)

plt.title("Spike-Snippet signals")
plt.ylabel("Voltage (mV)")
plt.xlabel("Time (s)")

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



3. Compute Differences between Samples

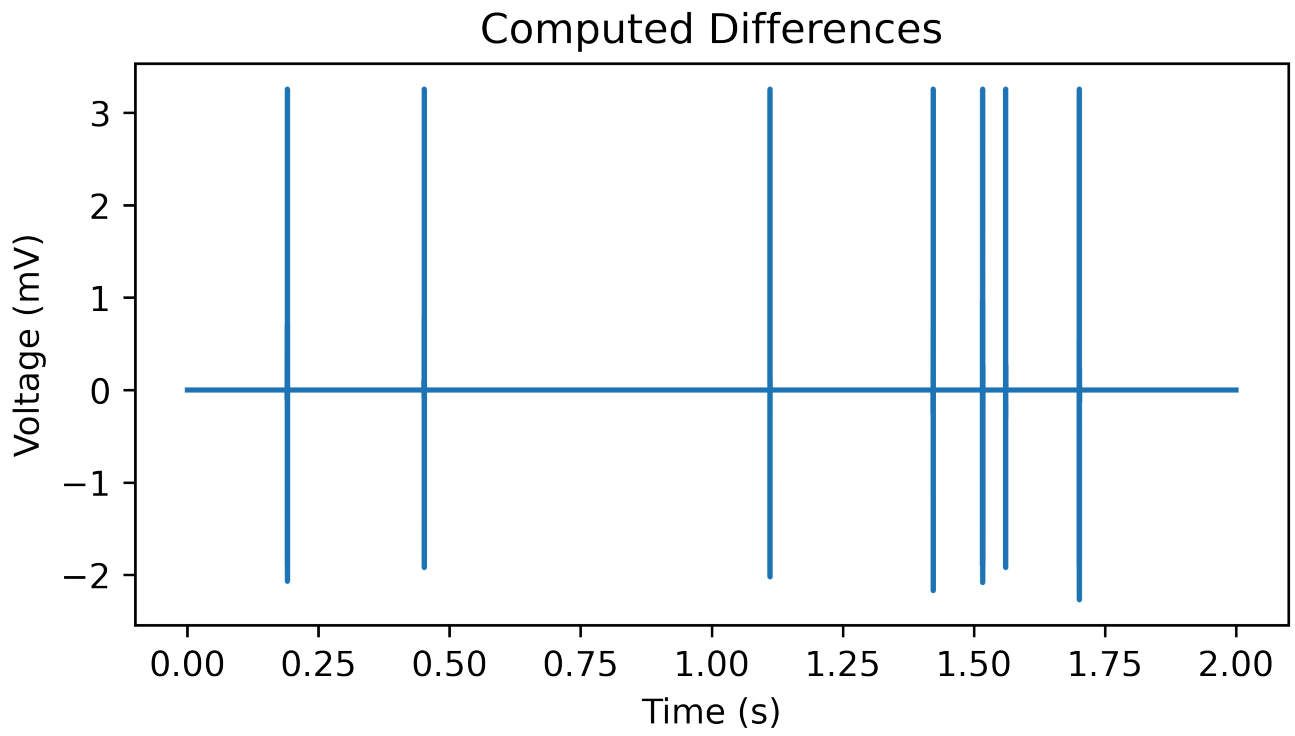
The differences between samples can be represented as:

$$y[n] = x[n] - x[n-1],$$

where n is the index of a signal, and x is the original signal.

In [14]:
```
# Compute differences between samples
diff = np.diff(x_snippets)
```

In [15]:
```
# Plot the computed differences
fig, ax = plt.subplots(1, 1, figsize=(6, 3),dpi = 600)

plt.title("Computed Differences")
plt.ylabel("Voltage (mV)")
plt.xlabel("Time (s)")

plt.plot(t[1:], diff)
plt.show()
```



4. Identify Subset of Values (with Non-Zero Difference)

To reduce duplicative data, as well as the size per symbol, we only sample a ***subset*** of samples, specifically those that have **non-zero difference**.

In [16]:
```
# Take the subset with non-zero differences
subset_indices = np.where(diff > 0)[0]
diff_subset = diff[subset_indices]
t_subset = t[subset_indices]
x_subset = x_snippets[subset_indices]
```

In [17]:
```
# Do not use scientific notation
np.set_printoptions(suppress=True)
```

In [18]:
```
print('Subset of values with non-zero difference:\n', x_subset)
```

```
Subset of values with non-zero difference:
 [-0.1093372  -0.00004628  0.10932375 -1.08527244  0.10007955 -0.21254671
  0.          0.00001592  0.02241741 -1.08527244 -0.27103547  0.
 -0.08661614 -1.08527244 -0.1094108  -0.24906722 -0.10005749  0.13144988
 -1.08527244 -0.00001528 -0.14848736  0.         -0.14847141 -1.08527244
 -0.27112135 -0.00000676  0.02239139 -1.08527244 -0.27106474  0.
 -0.27113508 -0.00001855  0.27114207 -1.08527244 -0.10002866]
```

5. Generate Unary & Binary-Encoded Value (via Rice-Golomb Encoding)

Compute a **factor** (e.g., entropy measure of samples) that **minimizes total size** of each **encoded sample**.

To achieve a **high-compression ratio** with **low computational cost** (**minimal space & power usage**), we choose **Rice-Golomb encoding** as our encoding method.

Rice-Golomb encoding:

Based on the computed factor, compute a **quotient (**Q**, unary)** and **remainder (**R**, binary)**

$S = Q \times M + R$

$Q = [\frac{X}{M}]$ (encoded by **unary encoding**; i.e., Q '1's followed by a '0')

$R = X - QM$ (encoded via **truncated binary encoding**; i.e., encode remainder in binary using $M - 1$ bits)

M - selected via **Bernoulli process** to **predict M** value (**minimum # of bits** to **encode**) to **minimize total size of encoded sample** (selected based on the **maximum absolute value** to be encoded)

We know the X, the original signal, is given in **Q8.8** encoding (8 bits integer, 8 bits fraction)

In [19]:
```
def determine_optimal_m(max_value):
    # Find the optimal value for m based on the maximum value encountered in the data
    m = 1
    while 2 ** m < max_value:
        m += 1
    return m

def rice_golomb_encode(value, m):
    sign = 0 if value >= 0 else 1
    abs_value = abs(value)
    quotient = abs_value // m # q = int(x / m)
    remainder = abs_value % m # r = x - qm
    unary_code = '1' * quotient + '0'
```

```
    binary_code = format(remainder, '0{}b'.format(m-1))
    return str(sign) + unary_code + binary_code
```

In [20]:
```
# x is in Q8.8 format (8 bits integer, 8 bits fraction)
values = x_subset
int_bits = 8  # Integer part bits
frac_bits = 8  # Fractional part bits

# Find the maximum magnitude (absolute) value in the data
max_value_q88 = max(abs(int(value * (2**frac_bits))) for value in values)

# Determine the optimal value for 'm'
m = determine_optimal_m(max_value_q88)

print("Optimal value for 'm':", m)

encoded_values = []

# Encode and decode the values using the dynamically determined 'm'
for value in values:
    value_q88 = int(value * (2**frac_bits))
    encoded_value = rice_golomb_encode(value_q88, m)
    encoded_values.append(encoded_value)
```

```
Optimal value for 'm': 9
```

6. Transmit Unary and Binary Values to Bitstream

Use **'00'** as the prefix to denote **"lossless"** compression.

In [21]:
```
bitstream = ''

for value in encoded_values:
    bitstream += value

compression_prefix = '00' # Use '00' to denote "lossless" compression
bitstream = compression_prefix + bitstream
bitstream
```

Out[21]:
```
'00111100000000000000000000011100000000011111111111111111111111111111111110000001110110000001111111110000000000000000000000000000000010111111111111111111111
11111111110000001111111111110000001100000000000111000000100111111111111111111111111111111110000001111111100000000111111111000000000111000000111011100000001101111
11111111111111111111111111111110000001100000000000111110000000100000000000111110000000101111111111111111111111111111111110000001111111111000000110000000000000000
00101111111111111111111111111111111111000000111111111110000001100000000000111111110000001100000000000011111110000001101111111111111111111111111111111110000001111
11000000111'
```

Print compression ratio

In [22]:
```
def bitstream_to_bytes(bitstream):
    """Convert a binary string (bitstream) to bytes."""
    bytes_data = bytearray()
    for i in range(0, len(bitstream), 8):
        byte = bitstream[i:i+8]
        bytes_data.append(int(byte, 2))
    return bytes(bytes_data)
```

In [23]:
```
# Print the compression ratio
original_size = x_subset.nbytes
compressed_size = len(bitstream_to_bytes(bitstream[2:]))
compression_ratio = original_size / compressed_size
print('compression ratio:', compression_ratio)
```

```
compression ratio: 3.5
```

We see the compression ratio for **lossless** operation is **between 3:1 and 4:1**. According to the patent, the compression ratio is **about 3:1**. Our ratio is close enough.

Additional: Decode Bitstream

In [24]:
```
def rice_golomb_decode(encoded_value, m):
    sign = int(encoded_value[0])
    unary_length = encoded_value[1:].index('0') + 2
    quotient = unary_length - 1
    binary_code = encoded_value[unary_length:unary_length+m-1]
    remainder = int(binary_code, 2)
    abs_value = quotient * m + remainder
    return abs_value if sign == 0 else -abs_value
```

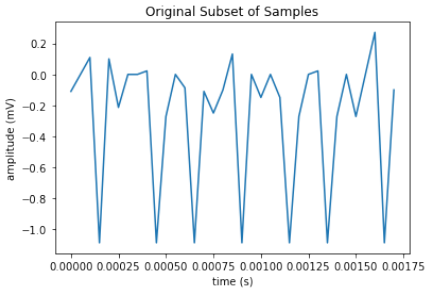
In [25]:
```
index = 2
decoded_values = []

while index < len(bitstream):
    substr_end = index + bitstream[index+1:].index('0') + m + 1
    decoded_value_q88 = rice_golomb_decode(bitstream[index : substr_end], m)
    decoded_value = decoded_value_q88 / (2**frac_bits)
    decoded_values.append(decoded_value)
    index = substr_end
```

In [26]:
```
print('Subset of values with non-zero difference:\n', x_subset)
print('Decoded values:\n', decoded_values)
```

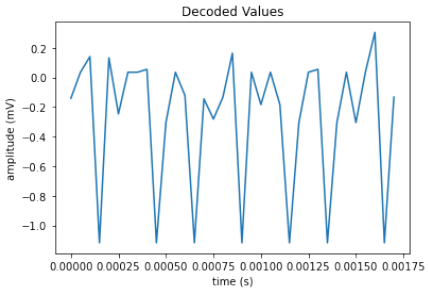
```
Subset of values with non-zero difference:
 [-0.1093372  -0.00004628  0.10932375 -1.08527244  0.10007955 -0.21254671
  0.          0.00001592  0.02241741 -1.08527244 -0.27103547  0.
 -0.08661614 -1.08527244 -0.1094108  -0.24906722 -0.10005749  0.13144988
 -1.08527244 -0.00001528 -0.14848736  0.         -0.14847141 -1.08527244
 -0.27112135 -0.00000676  0.02239139 -1.08527244 -0.27106474  0.
 -0.27113508 -0.00001855  0.27114207 -1.08527244 -0.10002866]
Decoded values:
 [-0.140625, 0.03515625, 0.140625, -1.1171875, 0.1328125, -0.24609375, 0.03515625, 0.03515625, 0.0546875, -1.1171875, -0.3046875, 0.03515625, -0.12109375, -1.
1171875, -0.14453125, -0.28125, -0.1328125, 0.1640625, -1.1171875, 0.03515625, -0.18359375, 0.03515625, -0.18359375, -1.1171875, -0.3046875, 0.03515625, 0.054
6875, -1.1171875, -0.3046875, 0.03515625, -0.3046875, 0.03515625, 0.3046875, -1.1171875, -0.1328125]
```

In [27]:
```
# Plot original subset of samples
plt.title("Original Subset of Samples")
```

```
plt.plot(t[np.arange(0, len(x_subset))], x_subset)
plt.xlabel('time (s)')
plt.ylabel('amplitude (mV)')
plt.show()
```



In [28]:
```
# Plot decoded values (from the Rice-Golomb encoded values)
plt.title("Decoded Values")
plt.plot(t[np.arange(0, len(decoded_values))], decoded_values)
plt.xlabel('time (s)')
plt.ylabel('amplitude (mV)')
plt.show()
```



You can see here that the decoded values and the original subset of samples (ground truth) are identical. This is expected, because we are doing a "lossless" compression. Our encoding/decoding works properly.

Lossy Compression

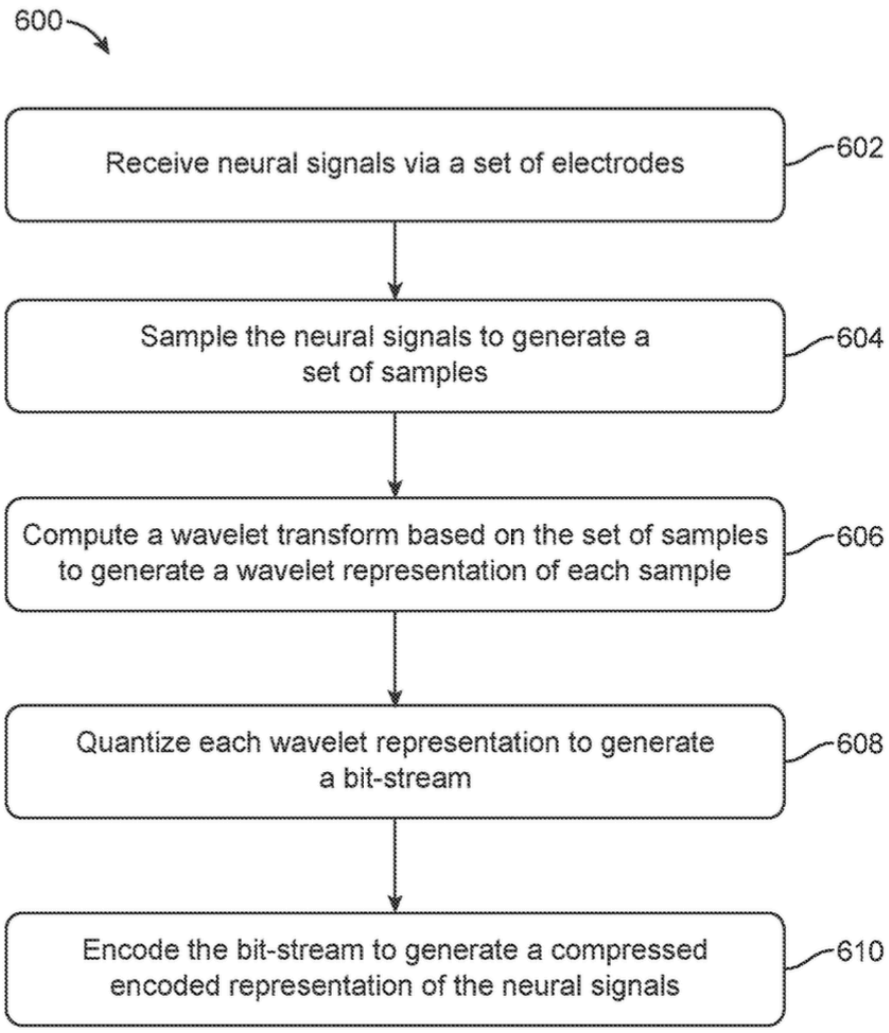


FIG. 6

Attempt 1: Analyze original spike-snippets (without characterization)

Use the **approximation coefficients** for PMF quantization.

1. Receive Signal

Receive neural signals from electrodes.

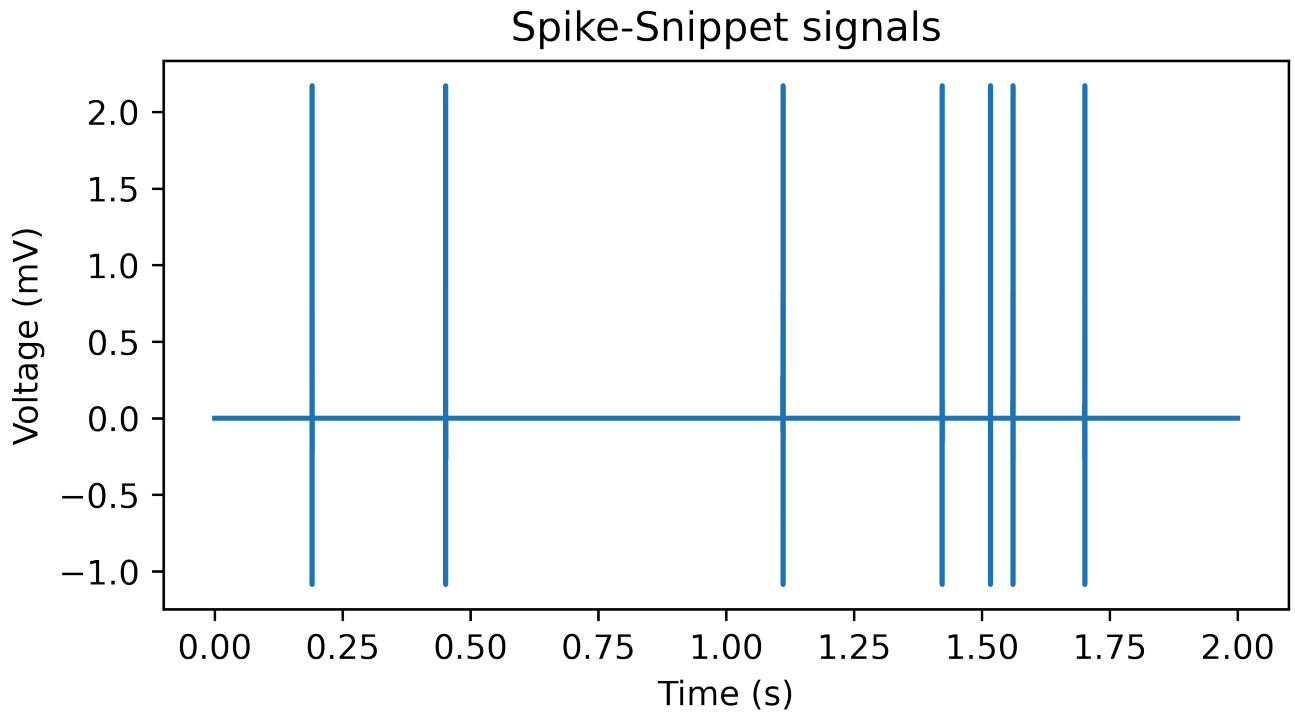
2. Generate Set of Samples (that correspond to spikes)

Detect spikes in the signal.

Refer to the notebook for US patent 2021/0012909 A1, "Real-Time Neural Spike Detection".

Let's **reuse the results from the first two steps** from the last part (Lossless Compression Algorithm), since they are **identical**.

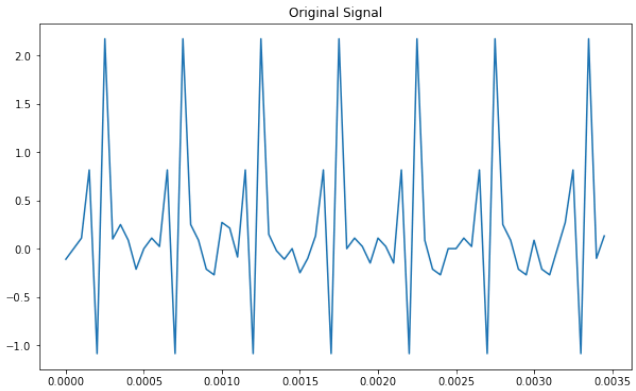
In [29]:
```
# Plot the spike-snippet signals
fig, ax = plt.subplots(1, 1, figsize=(6, 3),dpi = 600)

plt.title("Spike-Snippet signals")
plt.ylabel("Voltage (mV)")
plt.xlabel("Time (s)")

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



We want to pick the **subset** of signals that correspond to **spikes**.

In [31]:
```
# Get all non-zero signals of the spike snippets
nonzero_indices = np.where(x_snippets != 0)[0]
x_snippets_nonzero = x_snippets[nonzero_indices]
```

In [32]:
```
# Plot the non-zero signals of the spike snippets
plt.figure(figsize=(10, 6))
plt.plot(t[np.arange(0, len(x_snippets_nonzero))], x_snippets_nonzero, label='Original Signal')
plt.title('Original Signal')
plt.show()
```



3. Determine Wavelet Transform (Decompose into Wavelets)

Determine the wavelet representations for each sample by performing a **wavelet transform** w.r.t. the set of samples, such as a **discrete wavelet transform (DWT)**, which decomposes the signal into a number of sets of time series of coefficients describing the time evolution in corresponding frequency band.

Essentially, what we are doing here is **decomposing the original spike-snippets signal** (where each spike looks like an individual **wavelet**) into different components.

There are multiple types of wavelet transforms suitable for our application, including **Biorthogonal 2.2 (Bior 2.2)**, **Haar transform**, sliding Discrete Fourier Transform (sliding DFT), Goertzel algorithm, and sliding Goertzel algorithm.

Limitations of Fast-Fourier Transforms

Fast-Fourier transforms (FFTs), though widely used in practice, **cannot capture non-stationary signals (e.g., spikes)** effectively, since **every position is treated equally**. FFTs are also **ineffective** when **signal and noise are mixed together**, since they **rely on signal and nose separation**. We could use a **short-term FFT**, but its **window length is *fixed****. *What if we want a ***variable* window length**?

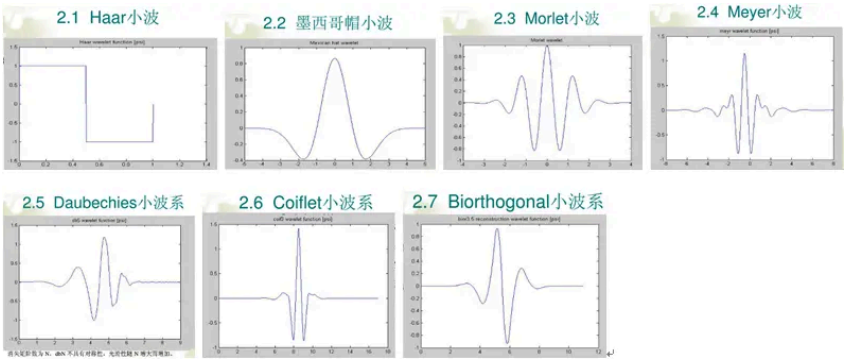
Discrete Wavelet Transform (DWT)

This is where **discrete wavelet transform** comes in.

Wavelets are **rapidly decaying** wave signals with **limited energy** (signal is **nonzero** over a ***finite* interval**) and **relatively concentrated local areas**. They can effectively capture **short-term, non-stationary impulses (e.g., spikes)** in the signal.

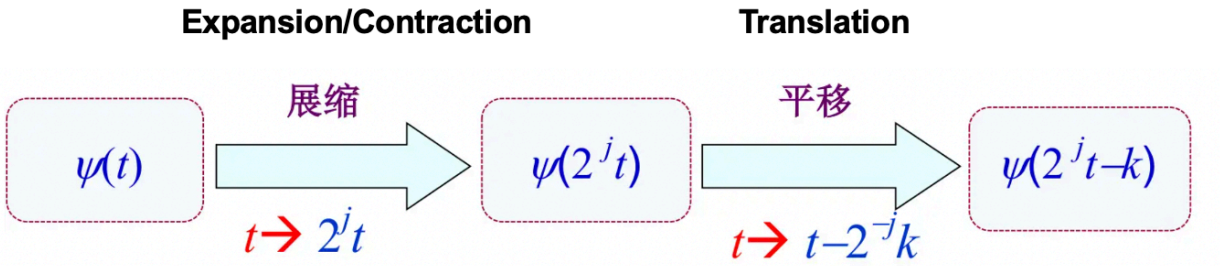
Nominally, DWT is applied to **"discrete"** sequences or **sampled** data, using **"discrete" time and scale steps**, **"discrete" filter banks** (low-pass, high-pass filters), and produces **discrete wavelet coefficients**. Compared continuous wavelet transforms (CWT), DWT allows for **efficient processing** of data, making it **suitable for real-world applications** involving digital signals and images, through applications such as **signal decomposition**.

Here are some common wavelet functions used in practice:

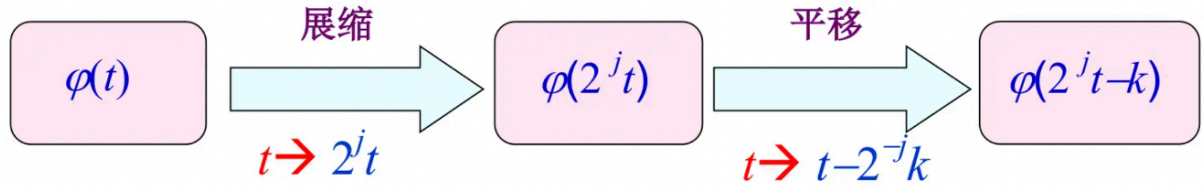


Wavelet transforms are made up of two components: the **wavelet function** (ψ) and **scaling function** (ϕ):

Wavelet function (ψ):



Scaling function (φ):



We can define the overall signal $x(t)$ as:

$$x(t) = \sum_n c_0[k]\varphi_{0,k}(t) + \sum_n d_0[k]\psi_{0,k}(t) + \sum_n d_1[k]\psi_{1,k}(t) + \cdots$$

Approximation coefficients — **Scaling** functions — **Detail** coefficients — **Wavelet** functions

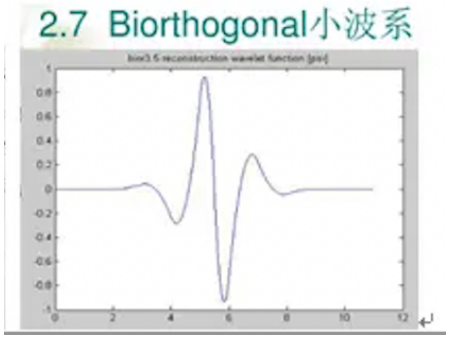
where **most information** are stored in the **approximation coefficients (lower frequency bands)**, while the **detail coefficients (higher frequency bands)** contain the *details* of the signal.

DWT is especially effective for **lossy** compression for the following reasons:

- **Localization of time and frequency domains** (can effectively capture *localized regions*, such as spikes)
- **Energy compaction** (**most energy concentrated in *low-frequency (approximation) components***, for better data reduction while preserving important information)
- **Multi-resolution analysis** (capture **both high-frequency and low-frequency components**)
- **Sparse representation** (**many wavelet coefficients close to zero** - can be discarded with minimal loss of information)
- **Scalability** (can **adjust level of decomposition and thresholds** applied to coefficients to **control** the kilter between **compression ratio and signal quality**)

Biorthogonal 2.2 (Bior 2.2)

We choose to use a **Biorthogonal (Bior)** wavelet. Biorthogonal wavelets are **symmetric**, which is useful for **preserving the phase and structure** of the original signal. A Bior wavelet contains two sets of wavelets: one used for **decomposition (analysis)**, another used for **reconstruction (synthesis)**.



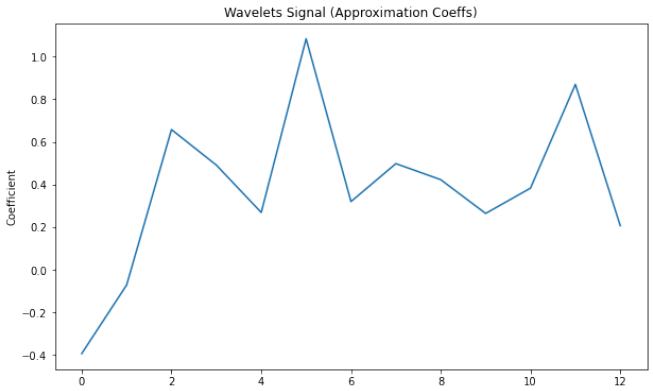
Specifically, we choose a **Bior 2.2** wavelet, which stands for an **order 2 wavelet function** and an **order 2 scaling function** (both wavelet and scaling contain **2 vanishing moments/coefficients**). It is good for **compact support** (nonzero signal over *finite* interval), which is **computationally efficient**.

Here, we can see that there is **1 set of approximation coefficients** and **3 sets of detail coefficients (level = 3)**.

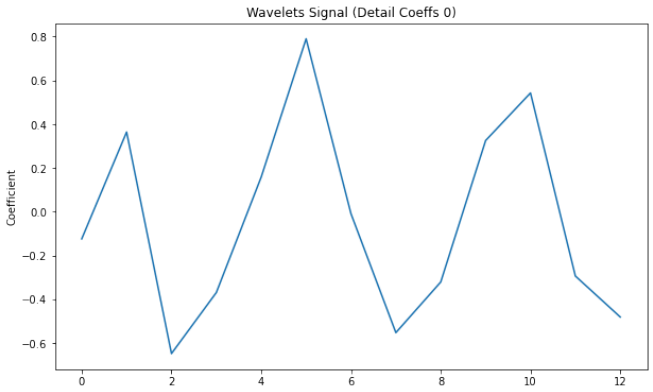
In [33]:
```
def perform_wavelet_decomposition(signal, wavelet='bior2.2', level=3):
    """Perform wavelet decomposition and retain only the approximation coefficients."""
    coefficients = pywt.wavedec(signal, wavelet, level=level)

    return coefficients[0], coefficients[1:]
```

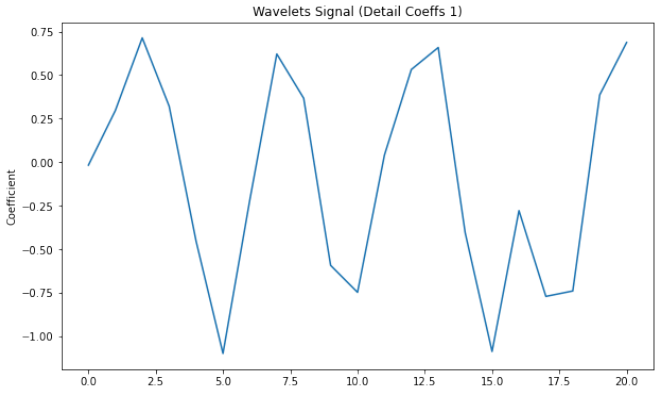
In [34]:
```
# Perform wavelet decomposition and retain both approximation and detail coefficients
approximation_coeffs, detail_coeffs = perform_wavelet_decomposition(x_snippets_nonzero)
```

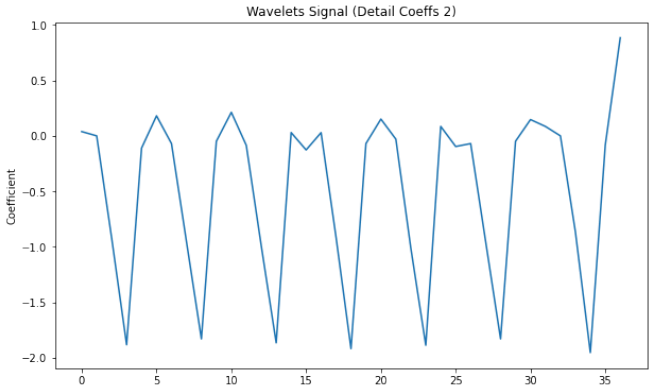
In [85]:
```
# Plot approximation coeffs
plt.figure(figsize=(10, 6))
plt.plot(approximation_coeffs)
plt.title('Wavelets Signal (Approximation Coeffs)')
plt.ylabel('Coefficient')
plt.show()
```



In [86]:
```
# Plot detail coeffs
i = 0
for coeffs in detail_coeffs:
    plt.figure(figsize=(10, 6))
    plt.plot(detail_coeffs[i])
    plt.title('Wavelets Signal (Detail Coeffs ' + str(i) + ')')
    plt.ylabel('Coefficient')
    plt.show()
    i += 1
```







4. Quantize Wavelets to Bitstream

Quantizing the wavelets **restricts each wavelet representation** into **discrete (unary/binary) values**, using techniques such as **probability mass function (PMF) quantization**. This **restricts** each wavelet representation into **discrete values** (unary and/or binary values).

We effectively **reduce the amount of bits or computational resources per detail**, with the **most important information** stored in **lower frequencies** (with higher frequency close to little or no information).

PMF Quantization

- **Approximate a PMF** using finite number of discrete levels/bins. $P[X = x_i]$ is the probability of the random variable X taking a certain value x_i.
- **Quantization levels** - probabilities of different outcomes (bin ranges) in the original data distribution. Let Q be the number of quantization levels or bins, and $q_1, q_2, \ldots, q_Q$ be the quantization levels.
- **Quantize the PMF** by **reducing the amount of data** needed to represent the distribution, while **keeping important statistical properties**.
- **Reconstructed PMF**: $P_q(q_j)$ is the probability mass assigned to quantization level q_j.
- Useful for **lossy data compression** and **signal processing**

In [37]:
```python
def estimate_pmf(data, num_bins=7):
    """Estimate the probability mass function (PMF) based on the data."""
    hist, bins = np.histogram(data, bins=num_bins, density=True)
    pmf = hist / np.sum(hist)
    return pmf, bins

def define_quantization_levels(pmf, bins):
    """Define quantization levels based on the PMF."""
    num_levels = len(pmf)

    # Adaptive quantization based on the PMF
    quantization_levels = np.zeros(num_levels)
    cumulative_pmf = np.cumsum(pmf) # Cumulative sum over the PMF probabilities
    for i in range(num_levels):
        idx = np.argmin(np.abs(cumulative_pmf - (i + 0.5) / num_levels))
        quantization_levels[i] = bins[idx + 1]
    return quantization_levels

def quantize_coeffs(approximation_coeffs, quantization_levels):
    """Quantize the approximation coefficients based on the specified quantization levels."""
    quantized_coeffs = np.zeros_like(approximation_coeffs, dtype=int)
    for i, coeff in enumerate(approximation_coeffs):
        # Find the closest quantization level for each coefficient
        quantized_coeffs[i] = np.argmin(np.abs(quantization_levels - coeff))
    return quantized_coeffs
```

In [38]:
```python
# Estimate PMF based on the approximation coefficients
pmf, bins = estimate_pmf(approximation_coeffs)

# Define quantization levels based on the PMF
quantization_levels = define_quantization_levels(pmf, bins)

# Apply quantization (bin categorization wrt quantization levels) to the approximation coefficients
quantized_approx_coeffs = quantize_coeffs(approximation_coeffs, quantization_levels)
```

```
# Apply quantization to the detail coefficients
quantized_detail_coeffs = [quantize_coeffs(coeff, quantization_levels) for coeff in detail_coeffs]
```

In [39]:
```
def convert_probabilities_to_integers(probabilities):
    # Convert probabilities to fractions to avoid floating point inaccuracies
    fractions = [Fraction(prob).limit_denominator() for prob in probabilities]

    # Extract the denominators
    denominators = [frac.denominator for frac in fractions]

    # Compute the least common multiple of the denominators
    def lcm(a, b):
        return abs(a * b) // math.gcd(a, b)

    overall_lcm = reduce(lcm, denominators)

    # Multiply each fraction by the overall LCM to get integer counts
    integer_counts = np.array([int(prob * overall_lcm) for prob in probabilities])

    return integer_counts
```

5. Encode Bitstream

Append the **details** from the wavelet representation into the bit stream.

Use an application of **range Asymmetric Numerical Systems (rANS)** to encode bitstream, such as **fast rANS (frANS)**, to **relax some constraints (e.g., PMF quantization)**, causing a **marginal loss of data ("lossy")**, while keeping the **accuracy loss negligible**.

Note that since rANS is an **entropy-coding method**, it relies on **accurately classifying spikes in each bin based on spike categories**, as it **increases compression ratio** from 8:1 (without categorization) to 23:1 (with categorization), according to the patent.

Fast-rANS Encoding:

- Use **adaptive models** to **adjust** quantized probabilities.
- **Relax PMF constraints** by allowing **PMF approximation** (instead of exact PMFs)
- Suitable for practical purposes, espeically when exact PMF is difficult to obtain, as we **increase the flexibility and efficiency**
- **No pre-defined codebook** is needed (unlike in Huffman encoding) - can simply use **frequency counts of each symbol**
- **Great for parallelization**, **large samples of data**, as well as **highly complex statistical structures**

Some notation:

Let $S = (s_1, s_2, \ldots, s_n)$ be an input string of symbols of size n

Let $A = \{a_1, a_2, \ldots, a_k\}$ be an alphabet of size k

Let $F = \{F_{a_1}, F_{a_2}, \ldots, F_{a_k}\}$ be the frequency counts of each alphabet symbol a_i.

Let $M = \sum_{i=1}^{k} F_i$. Then, probability $p_i = \frac{F_{a_i}}{M}$.

Cumulative frequency counts $C_{a_i} = \sum_{j=1}^{i-1} F_{a_j}$ be the cumulative distribution function (CDF)

Encoding:

Let X_t be the integer state of fast-rANS after t input symbols

Let $F_{inv_{s_t}}$ be the inverse-frequency count of symbol s_t (used for **lookup**)

Initial state: $X_0 = 0$

Update state:

$X_t = X_{t-1}$, if $F_{s_t} = 0$ (no state change)

$X_t = X_{t-1} * F_{inv_{s_t}} + C_{s_t}$, otherwise

Final state: X_n - representation using $\lceil log(X_n) \rceil$ bits

Decoding:

$s_t =$ first index greater than X_t in C_{s_t}

$X_{t-1} = \lfloor \frac{X_t}{M} \rfloor * F_{s_t} + slot - C_{s_t}$, where $slot = mod(X_t, M)$

In [40]:
```
class FastRANS:
    def __init__(self, symbol_counts):
        self.total_counts = np.sum(symbol_counts) # M
        self.cumul_counts = np.insert(np.cumsum(symbol_counts), 0, 0)
        self.inverse_counts = 1 / (symbol_counts + 1e-15)  # Precompute inverses
        self.symbol_counts = symbol_counts  # Store symbol counts

    def encode(self, s, state):
        # If symbol count is zero, just return the same state
        if self.symbol_counts[int(s)] == 0:
            return state

        s_count = self.inverse_counts[int(s)]  # Lookup inverse instead of division
        next_state = (state * s_count) + self.cumul_counts[int(s)]  # Replace division with multiplication
        return next_state

    def decode(self, state):
        slot = state % self.total_counts
        s = np.searchsorted(self.cumul_counts, slot, side='right') - 1 # first index greater than state in cumulative symbol counts
        prev_state = (state // self.total_counts) * self.symbol_counts[int(s)] + slot - self.cumul_counts[int(s)]
```

```
        return s, prev_state

def rANS_encode_relaxed_pmf(quantized_coeffs, pmf):
    symbol_counts = convert_probabilities_to_integers(pmf)
    encoder = FastRANS(symbol_counts)
    encoded_data = bytearray()
    for coeff in quantized_coeffs:
        for level in coeff:
            symbol = encoder.encode(level, 0)
            encoded_data.extend(int(symbol).to_bytes(2, byteorder='big'))  # Use 2 bytes for the state
    compressed_data = zlib.compress(encoded_data)
    return compressed_data
```

Print compression ratio:

In [41]:
```
# Encode quantized detail coefficients using fast rANS encoding
compressed_data = rANS_encode_relaxed_pmf(quantized_detail_coeffs, pmf)

# Calculate compression ratio
original_size = x_snippets_nonzero.nbytes
compressed_size = len(compressed_data)
compression_ratio = original_size / compressed_size

print("Compression ratio:", compression_ratio)
```

```
Compression ratio: 12.444444444444445
```

We see the compression ratio for **lossy** operation is **between 10:1 and 15:1**. According to the patent, since we did not split based on characteristics (but rather on the approximation values), the compression ratio is **about 8:1**. This may depend on the type of waveform we use, as well as our specific implementation of the fast-rANS encoding algorithm.

Encoded Bitstream:

Use **'01'** as the prefix to denote **"lossy"** compression.

In [42]:
```
def bytes_to_bitstream(bytes_data):
    """Convert bytes data to a binary string."""
    return ''.join(format(byte, '08b') for byte in bytes_data)

# Convert compressed data to bitstream
compression_prefix = '01' # Use '01' to denote "lossy" compression
bitstream = compression_prefix + bytes_to_bitstream(compressed_data)

# Sample usage
print("Compressed bitstream:", bitstream)
```

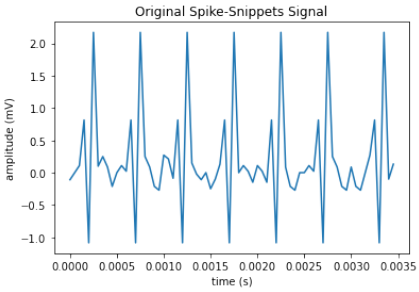
```
Compressed bitstream: 0101111000100111000110001101100000011000000110000001100110000000000000001010001100000011001101110000001100000011000101000100010110001100111001110100000100110000000001000011000000100011010010011010001011000100000100010110101000110000010110010000110001101000111011000101010001010101001000000011011000000010010001111000110110000000000010000000001110000000000110100
```

Additional: Decode Bitstream

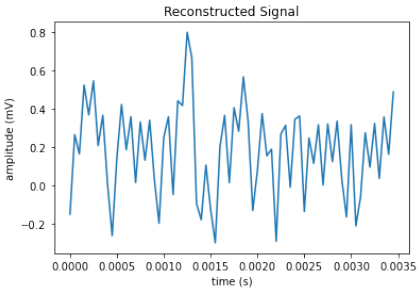
Reconstruct the original signal.

In [43]:
```
def rANS_decode_relaxed_pmf(compressed_data, pmf, quantized_detail_coeffs_lengths):
    decompressed_data = zlib.decompress(compressed_data)
    symbol_counts = convert_probabilities_to_integers(pmf)
    encoder = FastRANS(symbol_counts)
    quantized_coeffs = []

    state_index = 0
    for i in range(len(quantized_detail_coeffs_lengths)):
        arr = []
        for j in range(quantized_detail_coeffs_lengths[i]):
            state = int.from_bytes(decompressed_data[state_index:state_index+2], byteorder='big')
            symbol, prev_state = encoder.decode(state)
            arr.append(symbol)
            state_index += 2
        quantized_coeffs.append(np.array(arr))

    return quantized_coeffs

def dequantize_coeffs(quantized_coeffs, quantization_levels, wavelet='bior2.2'):
    dequantized_coeffs = np.zeros_like(quantized_coeffs, dtype=float)
    for i, q in enumerate(quantized_coeffs):
        dequantized_coeffs[i] = quantization_levels[q]
    return dequantized_coeffs

def reconstruct_signal(approximation_coeffs, detail_coeffs, wavelet='bior2.2'):
    coefficients = [approximation_coeffs] + detail_coeffs
    reconstructed_signal = pywt.waverec(coefficients, wavelet=wavelet)
    return reconstructed_signal

# Decode bitstream + compressed data
quantized_detail_coeffs_lengths = [len(quantized_detail_coeffs[i]) for i in range(len(quantized_detail_coeffs))]
decoded_compressed_data = bitstream_to_bytes(bitstream[2:])
decoded_detail_coeffs = rANS_decode_relaxed_pmf(decoded_compressed_data, pmf, quantized_detail_coeffs_lengths)

# Dequantize the detail coefficients
dequantized_detail_coeffs = [dequantize_coeffs(coeff, quantization_levels) for coeff in decoded_detail_coeffs]

# Reconstruct the signal using the approximation and detail coefficients
reconstructed_signal = reconstruct_signal(approximation_coeffs, dequantized_detail_coeffs)
```

In [44]:
```
# Plot original subset of spike-snippets
plt.title("Original Spike-Snippets Signal")
plt.plot(t[np.arange(0, len(x_snippets_nonzero))], x_snippets_nonzero)
plt.xlabel('time (s)')
plt.ylabel('amplitude (mV)')
plt.show()
```



In [45]:
```
# Plot reconstructed signal
plt.title("Reconstructed Signal")
plt.plot(t[np.arange(0, len(reconstructed_signal))], reconstructed_signal)
plt.xlabel('time (s)')
plt.ylabel('amplitude (mV)')
plt.show()
```



Since we are in "lossy" compression mode, we expect the original and reconstructed signals to be slightly different. This loss of information is due to decomposing the signal into wavelets and relaxing the PMF constraints during the fast-RANS encoding.

Attempt 2: Analyze based on spike characteristics (with characterization)

Redo the above steps, but split the spikes into bins **based on a specific characteristic (e.g., `rdist`, `ldist`, `rv`, etc.)**, and use this for PMF quantization (instead of approximation coefficients). The wavelet decomposition should still be the same.

In [84]:
```
df
```

Out[84]:

	lp	mp	rp	lv	mv	rv	ldist	rdist	ratio	asym	cost	thres
0	0.1451	0.17795	0.19070	0.253885	-0.260473	0.477109	0.03285	0.01275	1.831698	0.532134	0.004895	5.424962e-04
1	0.4171	0.44695	0.45175	0.253916	-0.260469	0.475899	0.02985	0.00480	1.827090	0.533550	0.006815	7.966344e-04
2	1.0681	1.07195	1.11130	0.253872	-0.260469	0.467648	0.00385	0.03935	1.795409	0.542869	0.006602	7.683142e-04
3	1.3541	1.37495	1.42245	0.253894	-0.260451	0.492909	0.02085	0.04750	1.892523	0.515093	0.002576	2.372343e-04
4	1.4511	1.47895	1.51680	0.253893	-0.260486	0.498458	0.02785	0.03785	1.913568	0.509356	0.003307	3.325406e-04
5	1.5221	1.54495	1.56075	0.253898	-0.260462	0.475914	0.02285	0.01580	1.827194	0.533495	0.006204	7.150779e-04
6	1.6211	1.65195	1.70110	0.253897	-0.260464	0.546861	0.03085	0.04915	2.099560	0.464282	0.000767	-7.794417e-08

In [48]:
```
# Choose characteristic
characteristic = 'rdist'
```

In [49]:
```
def estimate_pmf_characteristic(data, characteristic, num_bins=7):
    hist, bins = np.histogram(data[characteristic], bins=num_bins, density=True)
    pmf = hist / np.sum(hist)
    return pmf, bins

# Estimate the PMF based on the chosen characteristic
pmf, bins = estimate_pmf_characteristic(df, characteristic)

# Define quantization levels based on the PMF
quantization_levels = define_quantization_levels(pmf, bins)

# Apply quantization (bin categorization wrt quantization levels) to the approximation coefficients
quantized_approx_coeffs = quantize_coeffs(approximation_coeffs, quantization_levels)

# Apply quantization to the detail coefficients
quantized_detail_coeffs = [quantize_coeffs(coeff, quantization_levels) for coeff in detail_coeffs]

# Encode quantized detail coefficients using fast rANS encoding
compressed_data = rANS_encode_relaxed_pmf(quantized_detail_coeffs, pmf)

# Convert compressed data to bitstream
compression_prefix = '01' # Use '01' to denote "lossy" compression
bitstream = compression_prefix + bytes_to_bitstream(compressed_data)

print("Compressed bitstream:", bitstream)
```

```
Compressed bitstream: 0101111000100111000110001101100000011000000110000001100101000000000000000010101011000101000100011011100111011000010001011101000101000000
0011101000001111000100110000010110101001100010110000101110111111101101011000000000000100001111110010000000001101010
```

In [50]:
```
# Calculate compression ratio
original_size = x_snippets_nonzero.nbytes
compressed_size = len(compressed_data)
compression_ratio = original_size / compressed_size
```

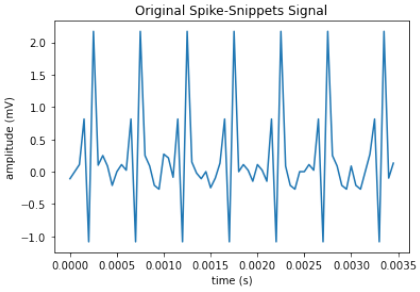
```
print("Compression ratio:", compression_ratio)
```

```
Compression ratio: 18.06451612903226
```

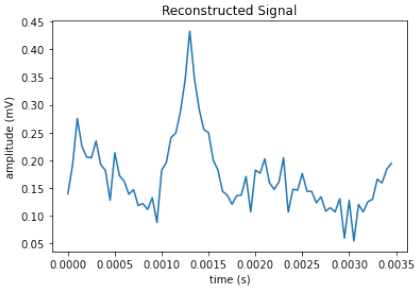
As we see, **with characterization**, the **compression ratio increases respectably**. According to the patent, the ratios should increase **from 8:1 to 23:1** if we add characterization to the spikes. This may also **depend on the specific implementation of the fast-rANS encoding** as well as the **wavelet transform function**.

In [51]:
```
# Decode bitstream + compressed data
quantized_detail_coeffs_lengths = [len(quantized_detail_coeffs[i]) for i in range(len(quantized_detail_coeffs))]
decoded_compressed_data = bitstream_to_bytes(bitstream[2:])
decoded_detail_coeffs = rANS_decode_relaxed_pmf(decoded_compressed_data, pmf, quantized_detail_coeffs_lengths)

# Dequantize the detail coefficients
dequantized_detail_coeffs = [dequantize_coeffs(coeff, quantization_levels) for coeff in decoded_detail_coeffs]

# Reconstruct the signal using the approximation and detail coefficients
reconstructed_signal = reconstruct_signal(approximation_coeffs, dequantized_detail_coeffs)
```

In [52]:
```
# Plot original subset of spike-snippets
plt.title("Original Spike-Snippets Signal")
plt.plot(t[np.arange(0, len(x_snippets_nonzero))], x_snippets_nonzero)
plt.xlabel('time (s)')
plt.ylabel('amplitude (mV)')
plt.show()
```



In [53]:
```
# Plot reconstructed signal
plt.title("Reconstructed Signal")
plt.plot(t[np.arange(0, len(reconstructed_signal))], reconstructed_signal)
plt.xlabel('time (s)')
plt.ylabel('amplitude (mV)')
plt.show()
```



We can see that with **'rdist'** as our characteristic, the **spike locations** now coincide with **local minima and maxima**. Since we are using a **"lossy"** compression, we do not expect the reconstructed signal to be the same as the original (unlike in "lossless" mode). Note that using **different characteristics** may result in **different reconstruction signal shapes** (as well as **different compression ratios**), due to the **inherent underlying pattern** of the **characteristic** used.

Sparse Compression: Binned-Spikes

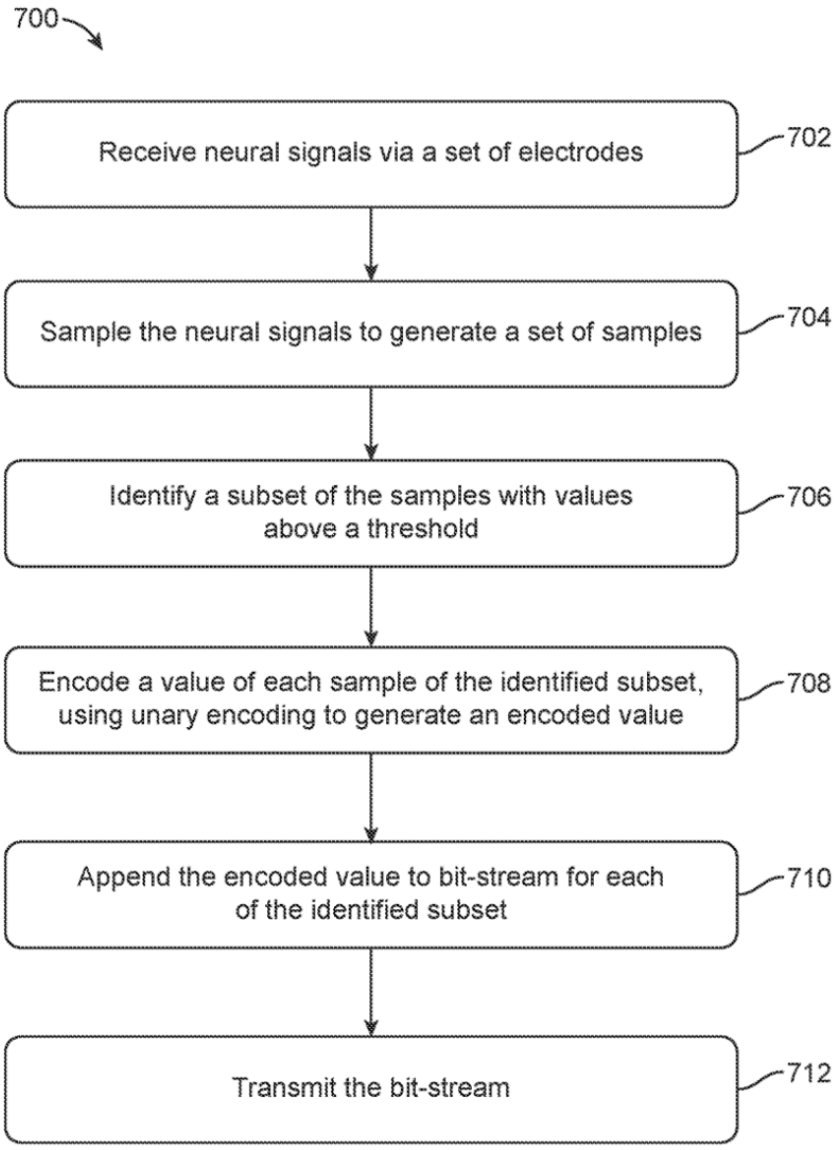


FIG. 7

1. Receive Signal

Receive neural signals from electrodes.

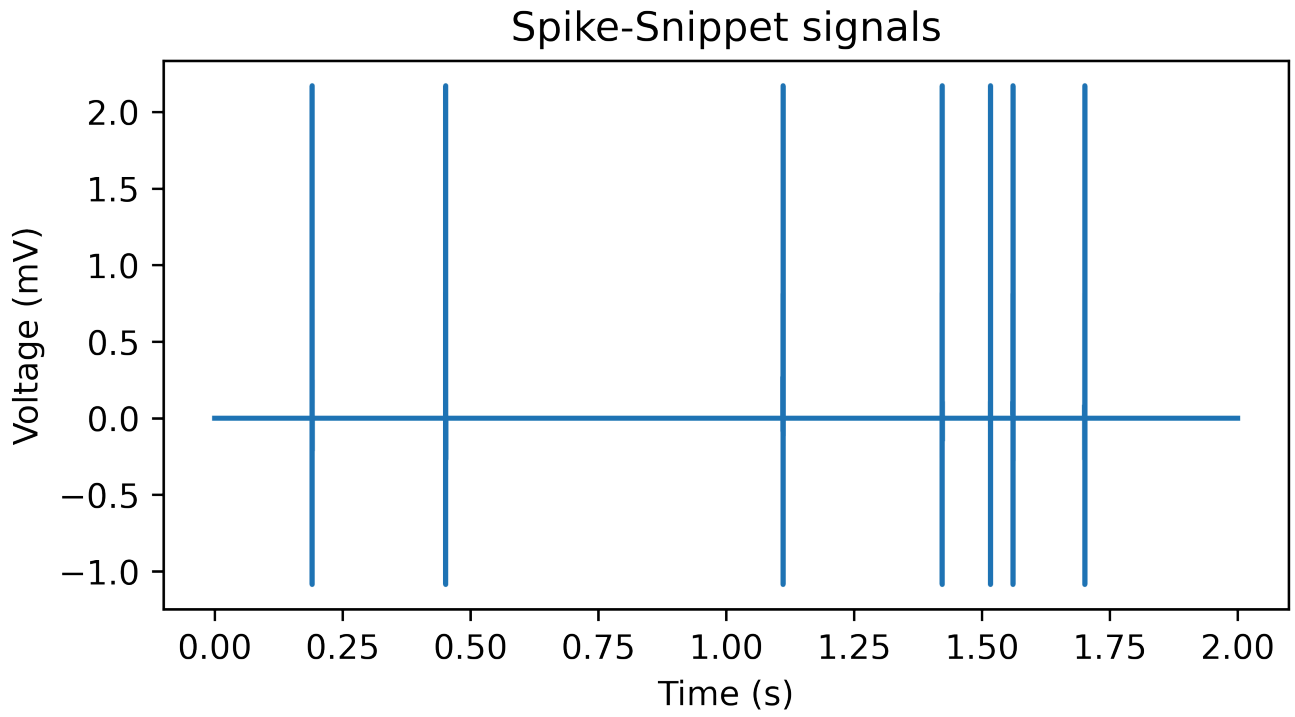
2. Generate Set of Samples (that correspond to spikes)

Detect spikes in the signal.

Refer to the notebook for US patent 2021/0012909 A1, "Real-Time Neural Spike Detection".

Let's **reuse the results from the first two steps** from the last part (Lossless Compression Algorithm), since they are **identical**.

In [54]:
```
# Plot the spike-snippet signals
fig, ax = plt.subplots(1, 1, figsize=(6, 3),dpi = 600)

plt.title("Spike-Snippet signals")
plt.ylabel("Voltage (mV)")
plt.xlabel("Time (s)")

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



3. Identify non-zero subset of samples

Filter out all values that equal to zero to leverage the sparse nature of neural signals. **DO NOT transmit if all signals are zero.**

In [55]:
```
nonzero_indices = np.where(np.abs(x_snippets) > 0)[0]
x_nonzero = np.abs(x_snippets[nonzero_indices])
t_nonzero = t[nonzero_indices]
```

In [56]:
```
# Transmit the signal only when there are 1 or more non-zero values
transmit = len(x_nonzero) != 0
transmit
```

Out[56]:
```
True
```

4. Encode non-zero values via unary uncoding

Use the same procedure as described in "Lossless Compression", except that we are ignoring the remainder (binary encoding).

In [57]:
```
def unary_encoding(value, m):
    abs_value = abs(value)
    quotient = abs_value // m # q = int(x / m)
    unary_code = '1' * quotient + '0'
    return unary_code
```

In [58]:
```
values_q88 = (x_nonzero * (2**frac_bits)).astype(int)
unary = np.vectorize(unary_encoding)
encoded_values = unary(values_q88, m)
```

5. Append encoded values to bitstream

If all values are zero or close to zero -> append "0" to bit stream

Otherwise, if 1 or more values are non-zero -> append "1" + each encoded value to bit stream

Use **'10'** as the prefix to denote **"binned-spikes"** compression.

In [59]:
```
if transmit:
    compression_prefix = '10' # Use '10' to denote "binned-spikes" compression

    # append "1" + each encoded value to bit stream
    bitstream = compression_prefix + '1'

    for value in encoded_values:
        bitstream += value
else:
    # append "0" to bit stream
    bitstream = '0'
```

6. Transmit Bitstream

Transmit bitstream to external device.

Note: Do NOT transmit bitstream if all samples are zero

In [60]:
```
bitstream
```

```
Out[60]: '1011110011101111111111111111111111110111111111111111111111111111111111101111111111111111111111111111111111111111111111111111111111111111111101101111111011011111110011100
11111111111111111111111101111111111111111111111111111111111111111011111111111111111111111111111111111111111111111111111111111101111111011011111110111111101111111011111
0110111111111111111111111111011111111111111111111111111111111111111011111111111111111111111111111111111111111111111111111111111111101111001110011111110110111011111111111
1111111111111011111111111111111111111111111111111101111111111111111111111111111111111111111111111111111111111111111001110011110111001111011111111111111111111111111011111
11111111111111111111111111011111111111111111111111111111111111111111111111111111111111111111111111101101111110111111100011100111111111111111111111111110111111111111111111111
111111111011111111111111111111111111111111111111111111111111111111111111111011111110110111111101101111111011111111011011111101111111001111111011111111111111111111111110111111
11111111111111111111111011111111111111111111111111111111111111111111111111111111111111111111111111101101110'
```

Print compression ratio

```python
In [61]: # Print the compression ratio
original_size = x_nonzero.nbytes
compressed_size = len(bitstream_to_bytes(bitstream[2:]))
compression_ratio = original_size / compressed_size
print('compression ratio:', compression_ratio)
```

```
compression ratio: 4.341085271317829
```

We see the compression ratio for **binned-spikes** operation is **between 4:1 and 5:1**. According to the patent, the compression ratio is **about 5:1**. Our ratio is close enough.

Additional: Decode bitstream

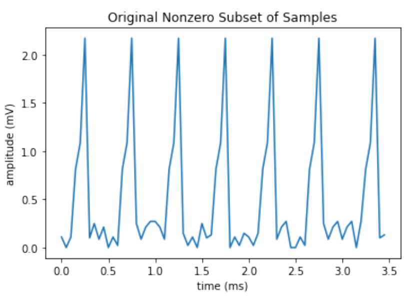
External device decodes bitstream for further analysis

Reverse all or some steps from above process - **reconstruct samples**

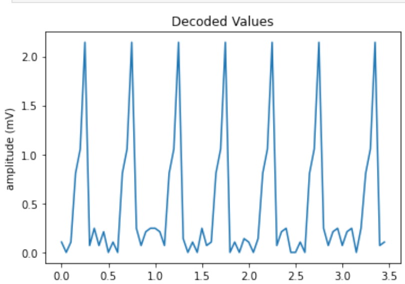
```python
In [62]: def unary_decoding(encoded_value, m):
    quotient = encoded_value.index('0')
    abs_value = quotient * m
    return abs_value
```

```python
In [63]: decoded_values = []

if transmit:
    index = 3 # ignore the compression prefix ('10') and 'transmit' bit ('1')
    while index < len(bitstream):
        substr_end = index + bitstream[index:].index('0') + 1
        decoded_value_q88 = unary_decoding(bitstream[index : substr_end], m)
        decoded_value = decoded_value_q88 / (2**frac_bits)
        decoded_values.append(decoded_value)
        index = substr_end
```

```python
In [64]: # Plot original nonzero subset of samples
plt.title("Original Nonzero Subset of Samples")
plt.plot(1000 * t[np.arange(0, len(x_nonzero))], x_nonzero)
plt.xlabel('time (ms)')
plt.ylabel('amplitude (mV)')
plt.show()
```




```python
In [65]: # Plot decoded values
plt.title("Decoded Values")
plt.plot(1000 * t[np.arange(0, len(decoded_values))], decoded_values)
plt.xlabel('time (ms)')
plt.ylabel('amplitude (mV)')
plt.show()
```



You can see here that the decoded values and the original subset of samples (ground truth) are very similar in plot shape, as their values scale similarly. That means our encoding/decoding works properly.

Sparse Compression: Spike-Band

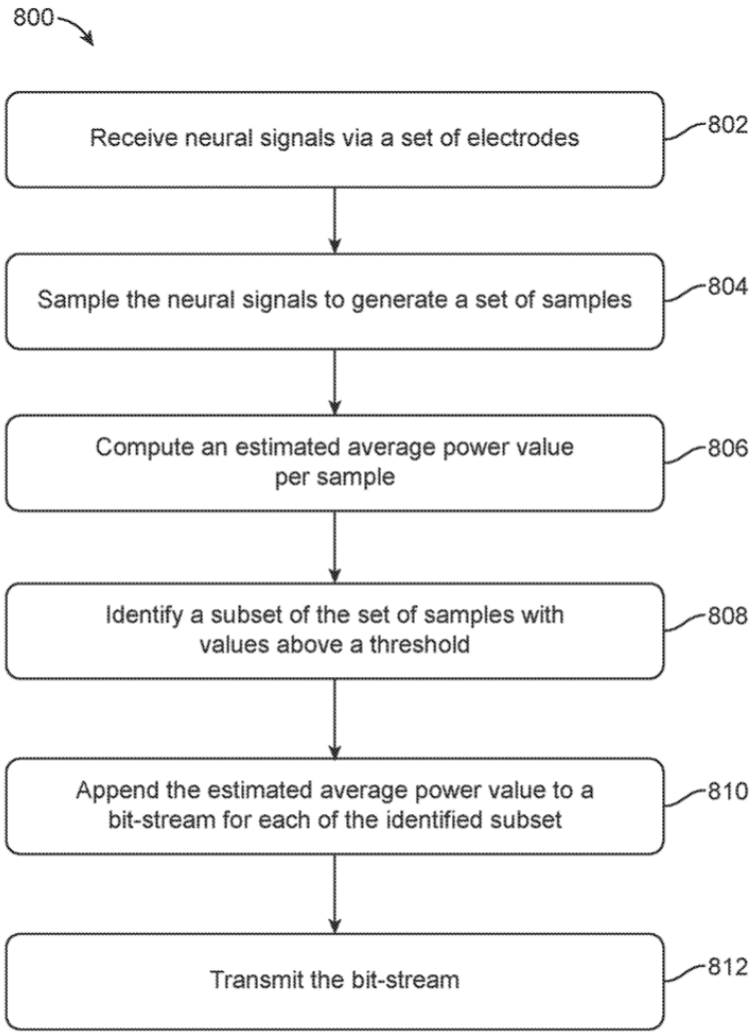


FIG. 8

1. Receive Signal + Detect Spikes

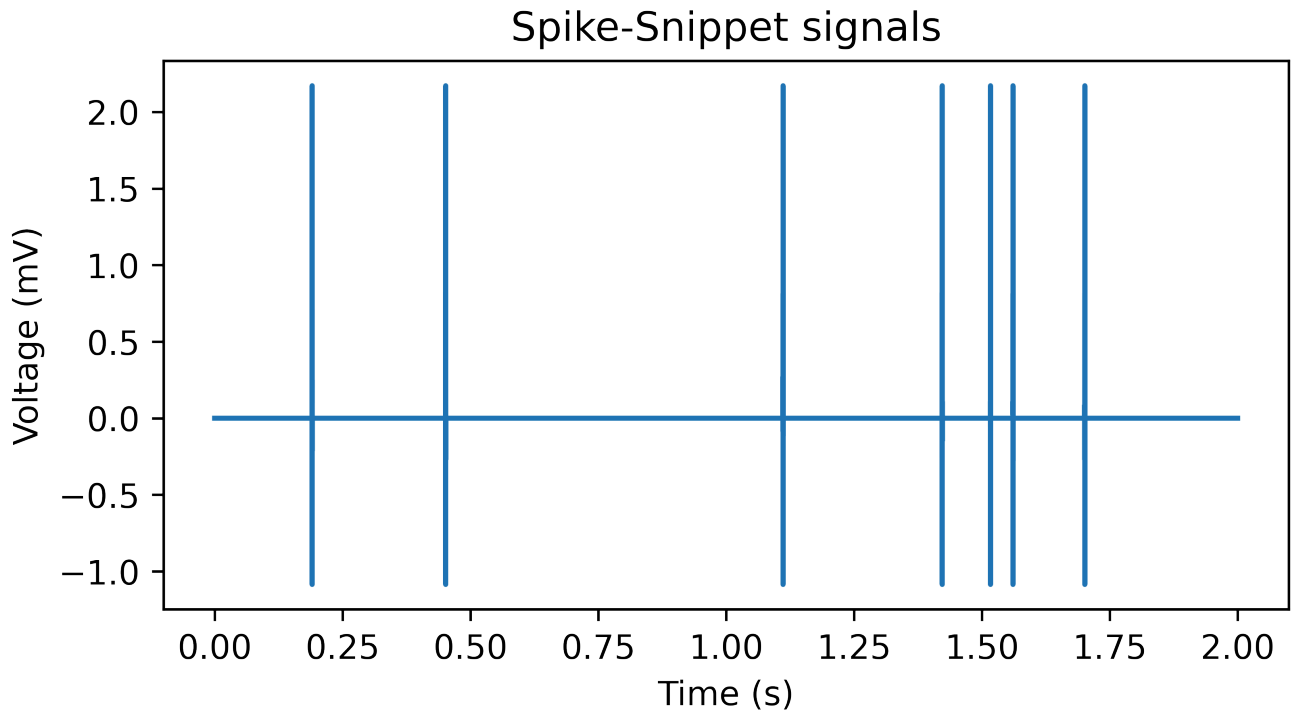
Receive neural signals from electrodes, and only keep the signals corresponding to spikes.

Then apply a filter to this signal.

Refer to the notebook for US patent 2021/0012909 A1, "Real-Time Neural Spike Detection".

Let's **reuse the results from the first two steps** from the last part (Lossless Compression Algorithm), since they are **identical**.

In [66]:
```
# Plot the spike-snippet signals
fig, ax = plt.subplots(1, 1, figsize=(6, 3),dpi = 600)

plt.title("Spike-Snippet signals")
plt.ylabel("Voltage (mV)")
plt.xlabel("Time (s)")

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



Filter the spike-snippets signal (need it for power computation):

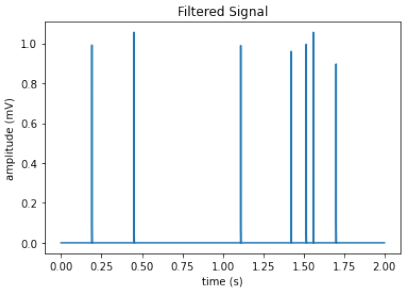
In [67]:
```
# Apply a filter to the spike-snippets signal
filtered_signal = spike_detector.filter_signal(x_snippets)
```

2. Generate Set of Samples

Compute an absolute value of each sampled channel. Each sample represents a channel.

In [68]:
```
filtered_signal = np.abs(filtered_signal)
```

In [69]:
```
# Plot the filtered signal (absolute values)

# Plot original nonzero subset of samples
plt.title("Filtered Signal")
plt.plot(t, filtered_signal)
plt.xlabel('time (s)')
plt.ylabel('amplitude (mV)')
plt.show()
```



3. Calculate average power (i.e., average amplitude)

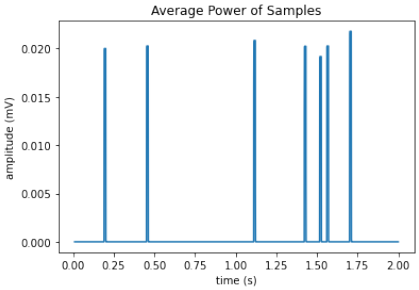
Compute average power for each fixed time window.

In [70]:
```
def average_val(signal, i1, i2):
    return np.mean(signal[i1 : i2])
```

In [71]:
```
def average_power(signal, blackout_period=0.01):
    index = 0
    duration = time_to_index(blackout_period)
    mean = []

    while index < len(signal) - duration:
        avg = average_val(signal, index, index + duration)
        mean.append(avg)
        index += 1

    return np.array(mean)
```

In [72]:
```
avg_power = average_power(filtered_signal, blackout_period=0.01) # use a 10ms time window
```

In [73]:
```
# Plot average power
plt.title("Average Power of Samples")
plt.plot(t[time_to_index(0.01):], avg_power)
plt.xlabel('time (s)')
plt.ylabel('amplitude (mV)')
plt.show()
```



4. Identify subset with nonzero average power

Identify subset of samples with nonzero average power

In [74]:
```
# Identify subset with non-zero average power
nonzero_indices = np.where(avg_power > 1e-8)[0]
avg_power_nonzero = avg_power[nonzero_indices]
```

In [75]:
```
# Transmit the signal only when there are 1 or more non-zero values
transmit = len(avg_power_nonzero) != 0
transmit
```

Out[75]:
```
True
```

5. Append encoded values to bitstream

Encode to unary as following:

- If average power value is nonzero, append "1" + average values to bitstream
- If average power value is zero, append "0" to bitstream

Use **'11'** as the prefix to denote **"spike-band"** compression.

In [76]:
```
# Find the maximum value in the data
max_value_q88 = max(abs(int(value * (2**frac_bits))) for value in avg_power_nonzero)

# Determine the optimal value for 'm'
m = determine_optimal_m(max_value_q88)

print("optimal 'm':", m)
```

```
optimal 'm': 3
```

In [77]:
```
values_q88 = (avg_power_nonzero * (2**frac_bits)).astype(int)
encoded_values = unary(values_q88, m)
```

In [78]:
```
if transmit:
    compression_prefix = '11' # Use '11' to denote "spike-band" compression

    # append "1" + each encoded value to bit stream
    bitstream = compression_prefix + '1'

    for value in encoded_values:
        bitstream += value
else:
    # append "0" to bit stream
    bitstream = '0'
```

6. Transmit Bitstream

Transmit bitstream to external device.

Note: Do NOT transmit bitstream if all samples are zero

In [79]:
```
bitstream
```

Out[79]:

```
'11100000001010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010
1010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101
010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101000000000000000000000000000000000000000000000000000000000000000
0000000000000101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010
1010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101
010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101000000000000000000000000000000000000000000000000000000000000000
0000000000000101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010
1010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101
010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101000000000000000000000000000000000000000000000000000000000000000
0000000000000010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101
0101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010
1010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101000000000000000000000000000000000000000000000000000000000
0000000000000001010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010
1010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101
01010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101000000000000000000000000000000000000000000000000000000
0000000000000000101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101
0101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010
1010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101000000000000000000000000000000000000000000000000000000000
0000000000000001010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010
1010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101
01010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101000000000000000000000000000000000000000000000000000000
00000000000000'
```

Print compression ratio

In [80]:

```
# Print the compression ratio
original_size = avg_power_nonzero.nbytes
compressed_size = len(bitstream_to_bytes(bitstream[2:]))
compression_ratio = original_size / compressed_size
print('compression ratio:', compression_ratio) # Should be close to 30:1
```

```
compression ratio: 36.94685990338164
```

We see the compression ratio for spike-band operation is **about 37:1**. According to the patent, the compression ratio is **about 30:1**. Our ratio is close enough.

Additional: Decode bitstream

External device decodes bitstream for further analysis

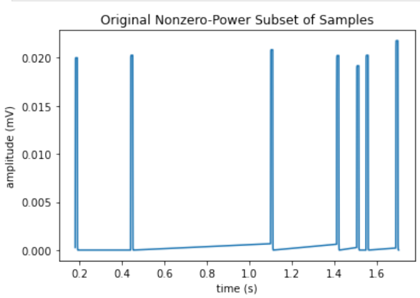
Reverse all or some steps from above process - **reconstruct average power vector**

In [81]:

```
decoded_values = []

if transmit:
    index = 3 # ignore the compression prefix ('11') and 'transmit' bit ('1')
    while index < len(bitstream):
        substr_end = index + bitstream[index:].index('0') + 1
        decoded_value_q88 = unary_decoding(bitstream[index : substr_end], m)
        decoded_value = decoded_value_q88 / (2**frac_bits)
        decoded_values.append(decoded_value)
        index = substr_end
```

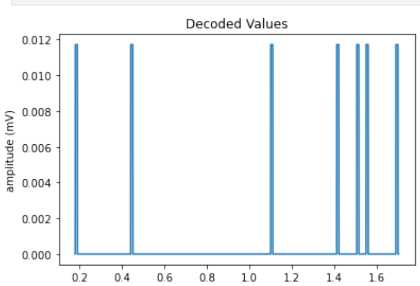
In [82]:

```
# Plot original nonzero subset of samples
plt.title("Original Nonzero-Power Subset of Samples")
plt.plot(t[nonzero_indices], avg_power_nonzero)
plt.xlabel('time (s)')
plt.ylabel('amplitude (mV)')
plt.show()
```



In [83]:

```
# Plot decoded values
plt.title("Decoded Values")
plt.plot(t[nonzero_indices], decoded_values)
plt.xlabel('time (s)')
plt.ylabel('amplitude (mV)')
plt.show()
```



We can see that there are **different amplitude magnitudes** between the two signals, which can be attributed to the **unary encoding scheme**, where we take the *quotient* of each encoded value, rather than the value itself.

Yet despite different values and scaling, we can see that the two plots (original nonzero power subset & decoded values) are similar in that **spikes are visible in both plots**, since spike-band mode **only cares about** signals that are **spikes**.

This shows that our encoding/decoding method works well enough for our application.

Usage of Machine Learning Models in Compression Operations

The 4 compresssion algorithms mentioned above can also **leverage machine learning models** to **determine the intent of action** of the user (e.g., raise left hand, raise right hand, move right foot, etc.), which can be transmitted off the neural transmitter without any extraneous transmission. For example, a **high bandwidth (high # of channels simultaneously transmitted) operation mode** may be chosen, such as **binned-spikes** or **spike-band**. Initially, the **compression data** is **configured to an external device** while the user performs mental actions. The **external device generates a model** to compute the **probabilities of a certain action being executed** by the user. The **model** is then **uploaded to the neural transmitter**, which **directly translates the neural activity into user actions**, achieving a **compression ratio** on the order of **100:1**.

Conclusion

We have introduced 4 different compression techniques: lossless, lossy, binned-spikes, and spike-band compression operations. For each specific dataset, it is to be determined which of the four compression techniques is the most feasible.

We can encode which of the 4 compression techniques is being used by including a **prefix** ('00' - lossless, '01' - lossy, '10' - binned-spikes, '11' - spike-band) at the front of the bitstream, as shown above.