Programming Exercise

Array Signal and Multichannel Processing

Professor: Dr. Marc Oispuu

Presented by:

- Enrique Niebles - 50225947

Description

Experimental System: Acoustic Crow's Nest Array

- Volumetric Microphone Array with 16 randomly distributed elements
- Condensator Microphones, frequency range: 5 Hz-100 kHz, sample frequency: 48 kHz
- Optimized for stationary or mobile operation on an unmanned ground vehicle
- Possible Applications: speech, shots, ground-based and airborne platforms

Flight Experiment in Wachtberg

- Crow's Nest Array measuring the sound emitted by an Unmanned Ground Vehicle (UAV) in flight
- **Challenge:** Determine DOA of the sound emitted by the UAV

Consider the measurements from a flight experiment delivered from the described volumetric Crow's Nest Array with 16 elements. Channel 9 was not in operation during the experiment and is therefore missing from the measurement. In the folder `measurements` you find the pre-processed measurements of the 15 channels. The measurements contain the recording of a drone flight over the microphone array of about 100 seconds duration. The data are band-pass filtered with cut-off frequencies from 2 kHz to 20 kHz. In addition to the pre-processed data, a file containing the raw data of the first channel is included in the data set.

In [1]:
```
import librosa
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import glob
from scipy.spatial.distance import pdist
```

```
from typing import Tuple, List, Dict, Any
import warnings

warnings.filterwarnings("ignore")
```

In [2]:
```
SAMPLE_RATE = 48000
SOUND_VELOCITY = 343  # m/s
N_FFT_BEAMFORMING = 400
N_FFT_SPECTROGRAM = 1024
ANGULAR_STEP_DEG = 2
N_FREQ_BINS = N_FFT_BEAMFORMING // 2
```

In [3]:
```
def load_audio_data() -> Tuple[List[np.ndarray], np.ndarray, Dict[str, Any]]:
    """
    Load preprocessed audio channels and the raw data file.

    Returns:
        channels_data: List of all 15 channel data arrays.
        raw_audio: Raw audio data from the first channel.
        metadata: Dictionary containing audio metadata.
    """
    channel_audio_files = sorted(glob.glob("measurements/AudioData_*.wav"))
    channels_data = []

    for file_path in channel_audio_files:
        y, _ = librosa.load(file_path, sr=SAMPLE_RATE)
        channels_data.append(y)

    raw_audio, _ = librosa.load("measurements/RawData.wav", sr=SAMPLE_RATE)

    duration = len(channels_data[0]) / SAMPLE_RATE
    num_samples = len(channels_data[0])

    metadata = {
        "num_channels": len(channels_data),
        "sample_rate": SAMPLE_RATE,
        "duration": duration,
        "num_samples": num_samples
    }

    return channels_data, raw_audio, metadata
```

In [4]:
```
def load_element_positions() -> np.ndarray:
    """
    Load array element positions from CSV, convert to meters, and handle the
```

```
    missing channel. Channel 9 (index 8) is excluded as it was not operational.

    Returns:
        element_positions: Array element coordinates in meters.
    """
    locations = pd.read_csv('elements_positions.csv', sep=',', header=0)
    # The element coordinates are given in centimeters and must be converted to meters.
    element_positions = locations[['x_m', 'y_m', 'z_m']].values / 100

    # Remove channel 9 (index 8)
    element_positions = np.delete(element_positions, 8, axis=0)

    return element_positions
```

In [5]:
```
channel_data, raw_audio, metadata = load_audio_data()
element_coords = load_element_positions()
```

Task 1: (10 points)

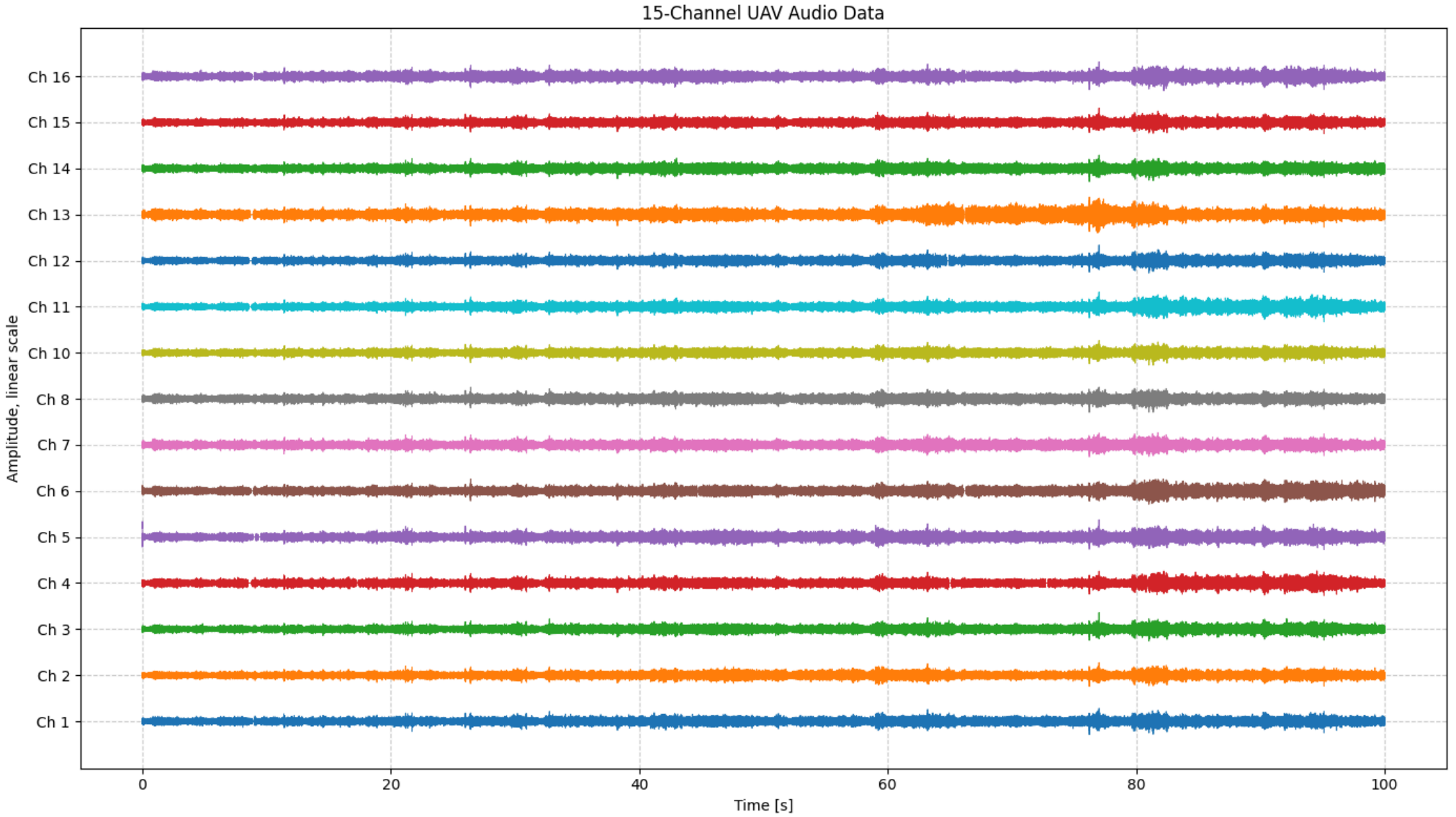
Consider the 15 preprocessed audio channels containing the UAV measurements:

- Read in the prepocessed audio data from all 15 channels. If possible, use a built-in function in your programming environment, e.g. `audioread` in MATLAB.
- Plot all 15 channels one above the other over a time-axis in one figure.

In [6]:
```
def plot_all_channels(channel_data: List[np.ndarray], metadata: Dict[str, Any]):
    """
    Plot all 15 channels over time.

    Args:
        channel_data: List of all 15 channel data arrays.
        metadata: Dictionary containing audio metadata.

    Returns:
        None
    """
    time_axis = np.linspace(0, metadata["duration"], metadata["num_samples"])
    offset = np.max(np.abs(channel_data)) * 2.5
    _, ax = plt.subplots(figsize=(14, 8))
    for i, data in enumerate(channel_data):
        ax.plot(time_axis, data + (i * offset), linewidth=0.7)
    ax.set_yticks([(i * offset) for i in range(metadata["num_channels"])])
    ax.set_yticklabels(
        [f'Ch {i+1 if i < 8 else i+2}' for i in range(metadata["num_channels"])])
```

```
    ax.set_xlabel('Time [s]')
    ax.set_ylabel('Amplitude, linear scale')
    ax.set_title('15-Channel UAV Audio Data')
    ax.grid(True, linestyle='--', alpha=0.6)
    plt.tight_layout()
    plt.show()
```

In [7]:
```
plot_all_channels(channel_data, metadata)
```



Task 2: (14 points)

For the first audio channel, consider the pre-processed and raw data and perform the following steps for both signals:

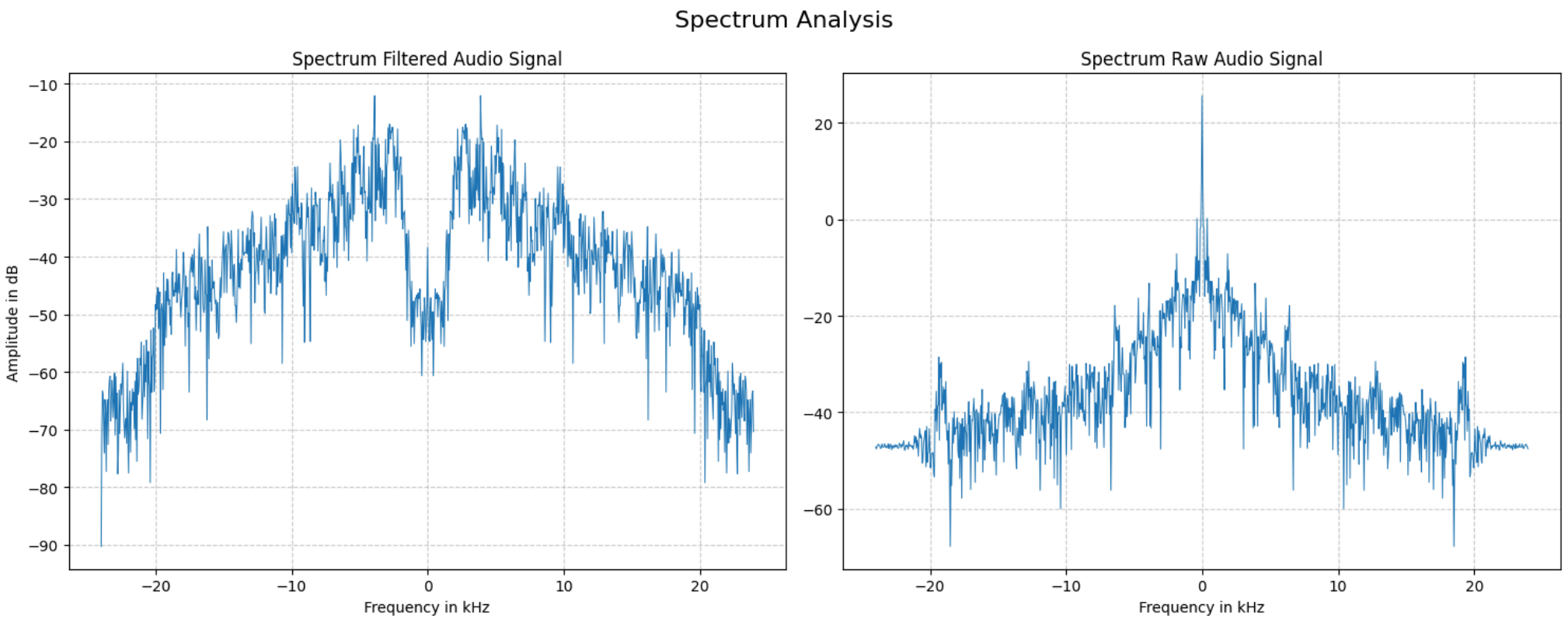
- Transform the signal to the frequency domain via Fast Fourier Transformation with $N = 1024$. If possible, use a built-in function in your programming environment, e.g. `fft` in MATLAB.
- Plot the spectra of the signals in dB over a frequency axis and compare both spectra.

In [8]:
```python
def linear_to_db(signal: np.ndarray) -> np.ndarray:
    """
    Convert a linear signal to dB.

    Args:
        signal: The signal to convert to dB.

    Returns:
        The signal in dB.
    """
    return 20 * np.log10(np.abs(signal) + np.finfo(float).eps)


def compute_fft(signal: np.ndarray, n_fft: int) -> Tuple[np.ndarray, np.ndarray]:
    """
    Compute the FFT of a signal and return frequencies and magnitude.

    Args:
        signal: The signal to compute the FFT of.
        n_fft: The number of points to use for the FFT.

    Returns:
        frequencies: The frequencies of the FFT.
        fft_output: The FFT of the signal.
    """
    fft_output = np.fft.fft(signal, n=n_fft)
    frequencies = np.fft.fftfreq(n=n_fft, d=1/SAMPLE_RATE)
    return np.fft.fftshift(frequencies), np.fft.fftshift(fft_output)


def compute_fft_db(signal: np.ndarray, n_fft: int) -> Tuple[np.ndarray, np.ndarray]:
    """
    Compute the FFT of a signal and return frequencies and magnitude in dB.

    Args:
        signal: The audio signal to compute the FFT of.
        n_fft: The number of points to use for the FFT.

    Returns:
```

```
        frequencies: The frequencies of the FFT.
        magnitude_spectrum_db: The magnitude of the FFT in dB.
    """
    frequencies, fft_output = compute_fft(signal, n_fft)
    magnitude_spectrum_db = linear_to_db(fft_output)
    return frequencies, magnitude_spectrum_db
```

In [9]:
```
def plot_spectrum_comparison(filtered_signal: np.ndarray, raw_signal: np.ndarray):
    """
    Plot and compare spectra of filtered and raw signals.

    Args:
        filtered_signal: Filtered audio signal.
        raw_signal: Raw audio signal.
    """
    fig, axes = plt.subplots(1, 2, figsize=(15, 6))
    freqs_filt, db_filt = compute_fft_db(
        filtered_signal, n_fft=N_FFT_SPECTROGRAM)
    freqs_raw, db_raw = compute_fft_db(raw_signal, n_fft=N_FFT_SPECTROGRAM)

    axes[0].plot(freqs_filt / 1e3, db_filt, linewidth=0.7)
    axes[0].set_title("Spectrum Filtered Audio Signal")
    axes[0].set_xlabel("Frequency in kHz")
    axes[0].set_ylabel("Amplitude in dB")
    axes[0].grid(True, linestyle='--', alpha=0.6)

    axes[1].plot(freqs_raw / 1e3, db_raw, linewidth=0.7)
    axes[1].set_title("Spectrum Raw Audio Signal")
    axes[1].set_xlabel("Frequency in kHz")
    axes[1].grid(True, linestyle='--', alpha=0.6)

    fig.suptitle("Spectrum Analysis", fontsize=16)
    plt.tight_layout()
    plt.show()
```

In [10]:
```
first_audio_channel = channel_data[0]
plot_spectrum_comparison(first_audio_channel, raw_audio)
```



Task 3: (20 points)

Examine the element geometry of the considered Crow's Nest Array. Attention: The element coordinates are given in centimeters! Assume a sound velocity of $c = 343 \frac{m}{s}$.

- Compute the array factor $AF(\mathbf{u})$ for the highest considered frequency of 20 kHz.
- Plot the results in uv-coordinates with $u^2 + v^2 = 1$.
- Does the smallest element spacing fulfil the requirements for unambiguous DF results? If not, is this a problem?
- Calculate the maximum frequency from which the DF for the smallest element distance becomes unambiguous. Plot the array factor for this frequency.

In [11]:
```python
def compute_steering_vector(direction_vector: np.ndarray, frequency: float) -> np.ndarray:
    """
    Calculates the steering vector a_0(u) for a given direction and frequency.

    Args:
        direction_vector: The direction vector.
        frequency: The frequency.
```

```
    Returns:
        The steering vector.
    """
    wavelength = SOUND_VELOCITY / frequency
    # This is the dot product e^T(u) * d_m for all m
    phase_delays = (2 * np.pi / wavelength) * \
        (element_coords @ direction_vector)
    return np.exp(1j * phase_delays)


def compute_array_factor(direction_vector: np.ndarray, look_direction_vector: np.ndarray, frequency: float) -> complex:
    """
    Calculates the Array Factor.

    Args:
        direction_vector: The direction vector.
        look_direction_vector: The look direction vector.

    Returns:
        The Array Factor.
    """
    a0_u = compute_steering_vector(
        direction_vector, frequency)
    a0_u0 = compute_steering_vector(look_direction_vector, frequency)
    # np.vdot computes the conjugate dot product, equivalent to a_0^H * a_0
    return np.vdot(a0_u0, a0_u)


def compute_direction_vector(azimuth_angle: float, elevation_angle: float) -> np.ndarray:
    """
    Compute the direction vector for a given azimuth and elevation angle.

    Args:
        azimuth_angle: The azimuth angle in radians.
        elevation_angle: The elevation angle in radians.

    Returns:
        The direction vector.
    """
    return np.array([
        np.sin(azimuth_angle) * np.sin(elevation_angle),
        np.cos(azimuth_angle) * np.sin(elevation_angle),
        np.cos(elevation_angle)
    ])
```

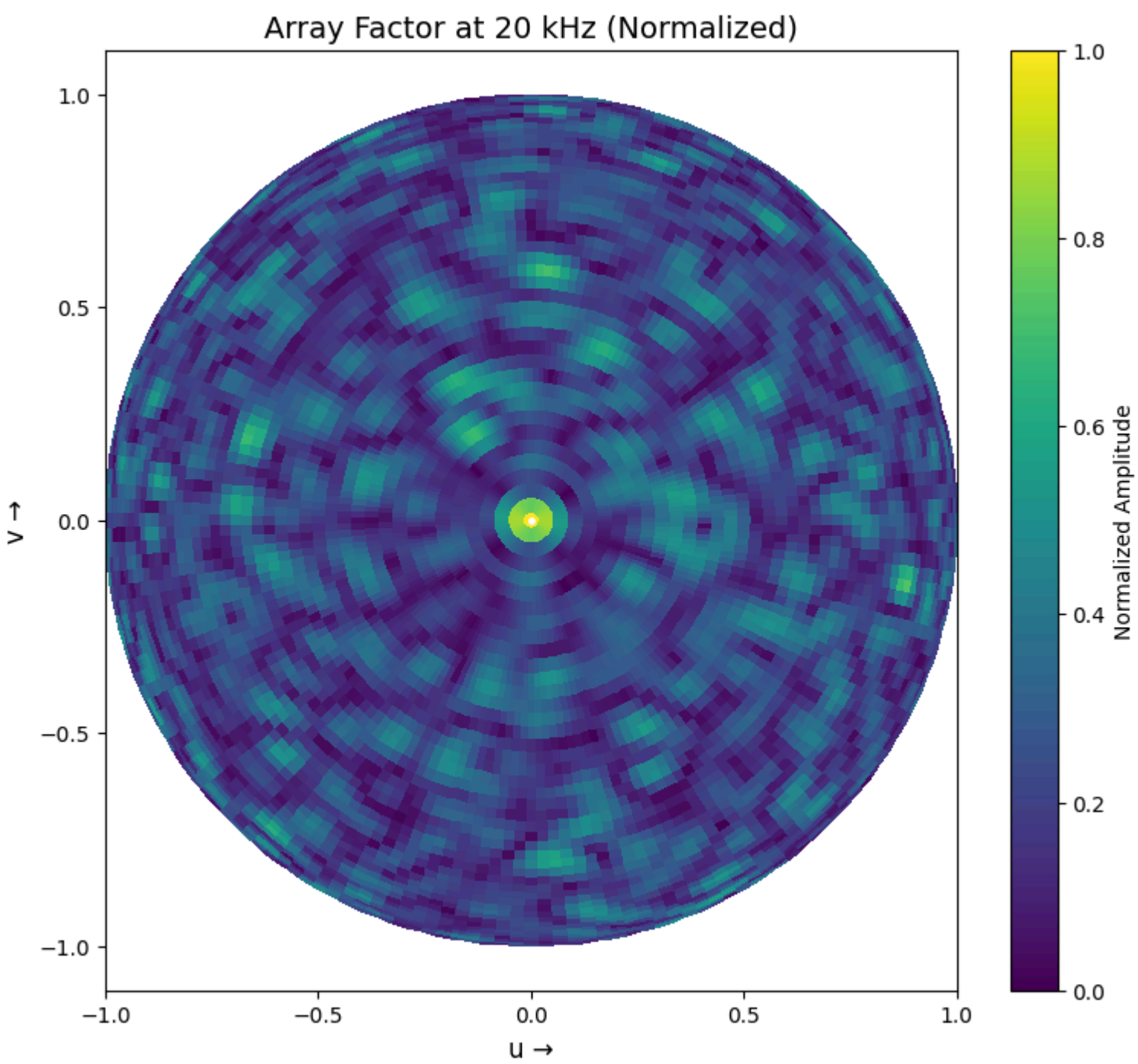
In [12]:
```python
def convert_to_uv_coordinates(azimuth_angles: np.ndarray,
                              elevation_angles: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
    """
    Convert spherical angles to u,v coordinates.

    Args:
        azimuth_angles: The azimuth angles.
        elevation_angles: The elevation angles.

    Returns:
        U: The u coordinates.
    """
    Az, El = np.meshgrid(azimuth_angles, elevation_angles)
    U = np.sin(Az) * np.sin(El)
    V = np.cos(Az) * np.sin(El)
    return U, V
```

In [13]:
```python
def plot_uv_map(grid_data: np.ndarray, azimuth_angles: np.ndarray,
                elevation_angles: np.ndarray, title: str):
    """
    Generic plotting function for data on a u,v grid.

    Args:
        grid_data: The data to plot.
        azimuth_angles: The azimuth angles.
        elevation_angles: The elevation angles.
        title: The title of the plot.

    Returns:
        None
    """
    U, V = convert_to_uv_coordinates(azimuth_angles, elevation_angles)

    plt.figure(figsize=(9, 8))
    plt.pcolormesh(U, V, grid_data, shading='auto',
                   cmap='viridis', vmin=0, vmax=1)

    cbar = plt.colorbar()
    cbar.set_label('Normalized Amplitude')
    plt.plot(0, 0, 'w.', markersize=5)  # Center dot

    plt.title(title, fontsize=14)
    plt.xlabel('u →', fontsize=12)
    plt.xticks(np.arange(-1, 1.5, 0.5)) # 0.5 steps
    plt.ylabel('v →', fontsize=12)
```

```
    plt.yticks(np.arange(-1, 1.5, 0.5)) # 0.5 steps
    plt.axis('equal')
    plt.xlim(-1, 1)
    plt.ylim(-1, 1)
    plt.show()
```

In [14]:
```
# Number of elements
M = element_coords.shape[0]

# Define angular grid for scanning in radians.
azimuths_rad = np.deg2rad(np.arange(0, 360, ANGULAR_STEP_DEG))
elevations_rad_polar = np.deg2rad(np.arange(0, 91, ANGULAR_STEP_DEG))
```

In [15]:
```
# The look direction (u0) is set to boresight (straight up, z-axis).
# In uv-coordinates, this corresponds to (u,v) = (0,0).
look_direction_vector = np.array([0, 0, 1])

af_grid = np.zeros((len(azimuths_rad), len(
    elevations_rad_polar)), dtype=complex)

for i, az in enumerate(azimuths_rad):
    for j, el_polar in enumerate(elevations_rad_polar):
        # Convert spherical angles (az, el) to a Cartesian unit vector e(u).
        # e = [sin(α)sin(ε), cos(α)sin(ε), cos(ε)]
        e_direction = compute_direction_vector(az, el_polar)
        # Compute the Array Factor for this direction.
        af_grid[i, j] = compute_array_factor(
            e_direction,
            look_direction_vector,
            frequency=20000,
        )

af_magnitude_normalized = np.abs(af_grid) / M  # Maximum value of the AF is M

plot_uv_map(
    af_magnitude_normalized.T, azimuths_rad, elevations_rad_polar,
    f"Array Factor at 20 kHz (Normalized)"
)
```



In [16]:
```
distances = pdist(element_coords)
d_min = np.min(distances)
wavelength_20k = SOUND_VELOCITY / 20000
```

```
print(f"Frequency (f): {20000 / 1e3:.1f} kHz")
print(f"Wavelength (λ): {wavelength_20k:.4f} m")
print(f"Smallest element spacing (d_min): {d_min:.4f} m")

is_unambiguous = d_min < (wavelength_20k / 2)
print(f"Is unambiguous: {is_unambiguous}")
```

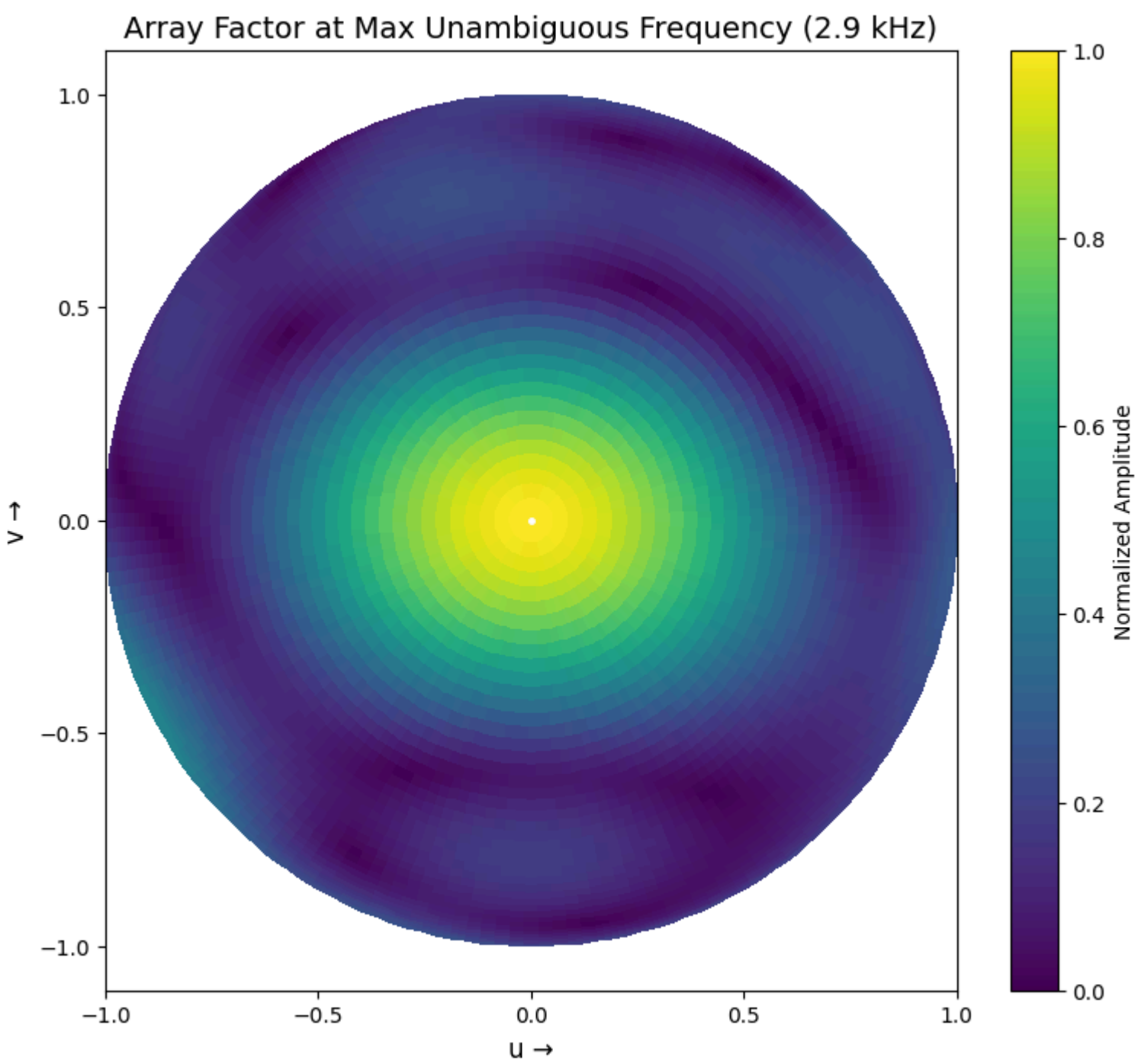
```
Frequency (f): 20.0 kHz
Wavelength (λ): 0.0171 m
Smallest element spacing (d_min): 0.0601 m
Is unambiguous: False
```

As the condition of smallest element spacing $d_{min} < \lambda/2$ is not met, it will likely produce grating lobes and ambiguous DF results.

In [17]:
```
f_max_unambiguous = SOUND_VELOCITY / (2 * d_min)
print(f"Maximum unambiguous frequency: {f_max_unambiguous / 1e3:.1f} kHz")
```

```
Maximum unambiguous frequency: 2.9 kHz
```

In [18]:
```
# Recompute the array factor at the maximum unambiguous frequency
af_grid_max_f = np.zeros((len(azimuths_rad), len(elevations_rad_polar)), dtype=complex)
for i, az in enumerate(azimuths_rad):
    for j, el_polar in enumerate(elevations_rad_polar):
        e_direction = compute_direction_vector(az, el_polar)
        af_grid_max_f[i, j] = compute_array_factor(
            e_direction,
            look_direction_vector,
            frequency=f_max_unambiguous,
        )

af_mag_norm_max_f = np.abs(af_grid_max_f) / M

plot_uv_map(
    af_mag_norm_max_f.T, azimuths_rad, elevations_rad_polar,
    f"Array Factor at Max Unambiguous Frequency ({f_max_unambiguous / 1e3:.1f} kHz)"
)
```



Task 4: (20 points)

Now use the pre-processed data again and consider the first second of the measurement data:

- Compute the spectrum of the signal via Fast Fourier Transformation with $N = 400$. For reasons of symmetry, use only the first half of 200 values for the following steps.
- Implement the incoherent broadband beamforming function based on the array transfer vector with $K = 200$ frequency bins:

$$BF = \sum_{k=1}^{K} \left| \mathbf{a}^H(\mathbf{u}; \omega_k) \mathbf{Z}(\omega_k) \right|^2$$

- Calculate the function values of the beamforming function for a grid ($\alpha \in [0, 2\pi]$, $\epsilon \in [0, \pi]$) with a step size of 2°.

In [19]:
```python
def broadband_beamformer_power(direction_vector: np.ndarray, Z_freq: np.ndarray,
                               freqs: np.ndarray) -> float:
    """
    Calculates the incoherent broadband beamformer power for a single direction.

    Args:
        direction_vector: The unit vector e(u) = [u, v, w] to scan.
        Z_freq: An (M x K) matrix of FFT coefficients for M channels and K frequency bins.
        freqs: A (K x 1) array of the frequencies for each bin.

    Returns:
        The total power.
    """
    total_power = 0.0
    # Sum over all K frequency bins
    for k in range(1, len(freqs)):
        fk = freqs[k]

        # Get steering vector for this direction and frequency
        a_k = compute_steering_vector(direction_vector, fk)

        # Get the FFT data for this frequency bin
        Z_k = Z_freq[:, k]

        # Calculate power for this bin and add to the sum: |a^H * Z|^2
        power_k = np.abs(np.vdot(a_k, Z_k))**2
        total_power += power_k

    return total_power
```

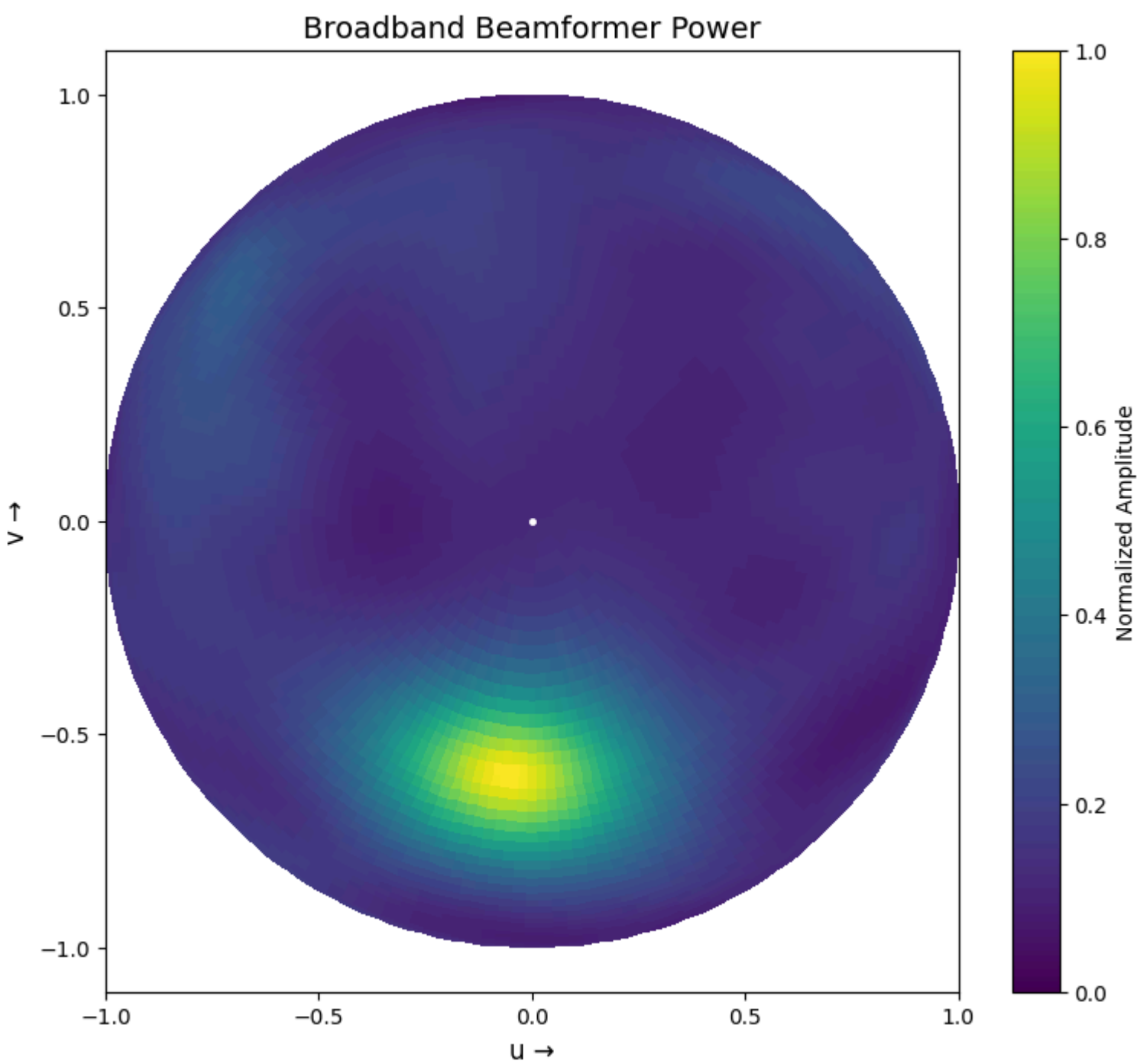
In [20]:
```python
# We take the first SAMPLE_RATE samples of each channel.
audio_segment = np.zeros((M, SAMPLE_RATE))
for i, data in enumerate(channel_data):
    audio_segment[i, :] = data[:SAMPLE_RATE]
```

```
# Compute the FFT of the audio segment. N=400, K=200
Z_freq = np.fft.fft(audio_segment, n=N_FFT_BEAMFORMING, axis=1)
Z_freq_half = Z_freq[:, :N_FREQ_BINS]

# Define the frequency axis.
freq_axis = np.fft.fftfreq(N_FFT_BEAMFORMING, 1/SAMPLE_RATE)[:N_FREQ_BINS]

bf_power_grid = np.zeros((len(azimuths_rad), len(elevations_rad_polar)))
for i, az in enumerate(azimuths_rad):
    for j, el_polar in enumerate(elevations_rad_polar):
        # Convert spherical angles (az, el) to a Cartesian unit vector e(u).
        # e = [sin(α)sin(ε), cos(α)sin(ε), cos(ε)]
        e_direction = compute_direction_vector(az, el_polar)
        # Compute the broadband beamformer power for this direction.
        bf_power_grid[i, j] = broadband_beamformer_power(
            e_direction, Z_freq_half,
            freqs=freq_axis,
        )
```

Task 5: (12 points)

- Plot the beamforming function in uv-coordinates.
- Determine the maximum of the Beamforming Function and the corresponding azimuth and elevation angles in degrees: $\hat{\mathbf{u}} = \arg\max_{\mathbf{u}} \sum_{k=1}^{K} \left|\mathbf{a}^H(\mathbf{u};\omega_k)\mathbf{Z}(\omega_k)\right|^2$. An accuracy of 2° is sufficient.

In [21]:
```
# Normalize the broadband beamformer power
bf_power_normalized = bf_power_grid / np.max(bf_power_grid)
plot_uv_map(
    bf_power_normalized.T,
    azimuth_angles=azimuths_rad,
    elevation_angles=elevations_rad_polar,
    title="Broadband Beamformer Power"
)
```



In [22]:
```
max_idx = np.unravel_index(np.argmax(bf_power_grid), bf_power_grid.shape)
az_est_deg = np.rad2deg(azimuths_rad[max_idx[0]])
el_est_deg = np.rad2deg(90 - elevations_rad_polar[max_idx[1]])
```

```
print(f"Estimated Azimuth: {az_est_deg:.2f} degrees")
print(f"Estimated Elevation: {el_est_deg:.2f} degrees")
```

```
Estimated Azimuth: 186.00 degrees
Estimated Elevation: 5120.62 degrees
```

Task 6: (12 points)

- Now divide the audio data in 99 windows of 1 second length. For each window perform the steps described in Task 4 and 5 in order to estimate the DOA of the impinging signal.
- Plot the maximum of the beamforming function as a track in uv-coordinates for the complete measurement with one estimate per second
- Transform the uv-results into azimuth and elevation angles and plot both as a track over time.

In [23]:
```
def estimate_doa_for_segment(audio_segment: np.ndarray) -> Tuple[float, float, float, float]:
    """
    Estimate the DOA for a given audio segment.

    Args:
        audio_segment: The audio segment.

    Returns:
        The estimated azimuth, elevation, u, and v.
    """
    # Compute the FFT of the audio segment. N=400, K=200
    Z_freq = np.fft.fft(
        audio_segment[:, :N_FFT_BEAMFORMING], n=N_FFT_BEAMFORMING, axis=1)
    Z_freq_half = Z_freq[:, :N_FREQ_BINS]
    freq_axis = np.fft.fftfreq(N_FFT_BEAMFORMING, 1/SAMPLE_RATE)[:N_FREQ_BINS]

    bf_power_grid = np.zeros((len(azimuths_rad), len(elevations_rad_polar)))
    for i, az in enumerate(azimuths_rad):
        for j, el_polar in enumerate(elevations_rad_polar):
            direction_vector = compute_direction_vector(az, el_polar)
            bf_power_grid[i, j] = broadband_beamformer_power(
                direction_vector,
                Z_freq_half,
                freqs=freq_axis,
            )

    # Find the maximum power and its index
    max_idx = np.unravel_index(np.argmax(bf_power_grid), bf_power_grid.shape)
    az_est_rad = azimuths_rad[max_idx[0]]
    el_est_rad_polar = elevations_rad_polar[max_idx[1]]

    # Convert the estimated azimuth and elevation to u and v coordinates
```

```
    u_est = np.sin(az_est_rad) * np.sin(el_est_rad_polar)
    v_est = np.cos(az_est_rad) * np.sin(el_est_rad_polar)

    # Convert the estimated azimuth and elevation to degrees
    az_est_deg = np.rad2deg(az_est_rad)
    el_est_deg = np.rad2deg(el_est_rad_polar)

    return az_est_deg, el_est_deg, u_est, v_est
```

In [24]:

```
def plot_tracking_results(u_track: np.ndarray, v_track: np.ndarray, azimuth_track: np.ndarray,
                          elevation_track: np.ndarray, time_axis: np.ndarray) -> None:
    """
    Plot the tracking results.

    Args:
        u_track: The u-track.
        v_track: The v-track.
        azimuth_track: The azimuth track.
        elevation_track: The elevation track.
        time_axis: The time axis.
    """
    fig = plt.figure(figsize=(18, 6))
    grid = plt.GridSpec(2, 3, wspace=0.4, hspace=0.3)

    ax1 = fig.add_subplot(grid[:, 0])
    ax1.scatter(u_track, v_track, c=time_axis, cmap='viridis', s=20)

    circle1 = plt.Circle((0, 0), np.cos(np.deg2rad(30)),
                         color='k', fill=False, lw=1.5)
    circle2 = plt.Circle((0, 0), np.cos(np.deg2rad(60)),
                         color='k', fill=False, lw=1.5)
    circle3 = plt.Circle((0, 0), np.cos(np.deg2rad(90)),
                         color='k', fill=False, lw=1.5)
    ax1.add_artist(circle1)
    ax1.add_artist(circle2)
    ax1.add_artist(circle3)

    ax1.set_title('DOA Track in uv-coordinates')
    ax1.set_xlabel('u →')
    ax1.set_ylabel('v →')
    ax1.set_xlim(-1, 1)
    ax1.set_ylim(-1, 1)
    ax1.set_xticks([-1, -0.5, 0, 0.5, 1])
    ax1.set_yticks([-1, -0.5, 0, 0.5, 1])
    ax1.set_aspect('equal', adjustable='box')
    ax1.grid(True, linestyle='--', alpha=0.6)
```

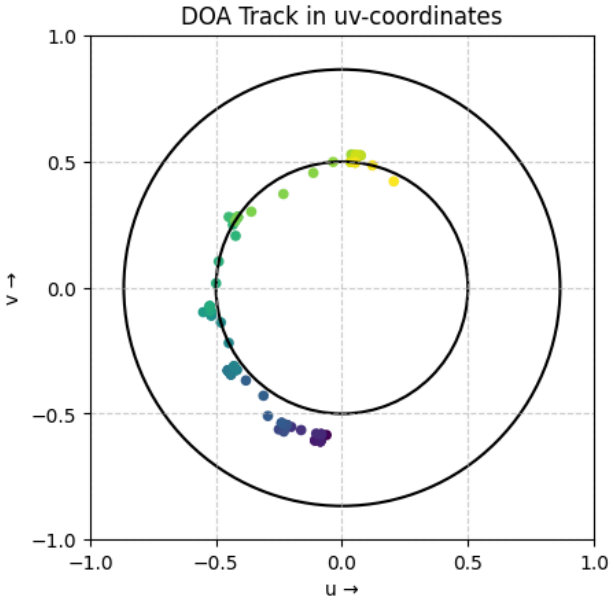
```
    ax2 = fig.add_subplot(grid[0, 1:])
    ax2.plot(time_axis, azimuth_track, '+', markersize=5)
    ax2.set_title('Azimuth Angle Track')
    ax2.set_ylabel('Azimuth angle (degree)')
    ax2.set_ylim(0, 360)
    ax2.grid(True, linestyle='--', alpha=0.6)

    ax3 = fig.add_subplot(grid[1, 1:])
    ax3.plot(time_axis, elevation_track, '+', markersize=5)
    ax3.set_title('Elevation Angle Track')
    ax3.set_xlabel('Time (s)')
    ax3.set_ylabel('Elevation angle (degree)')
    ax3.set_ylim(0, 90)
    ax3.grid(True, linestyle='--', alpha=0.6)

    plt.show()
```

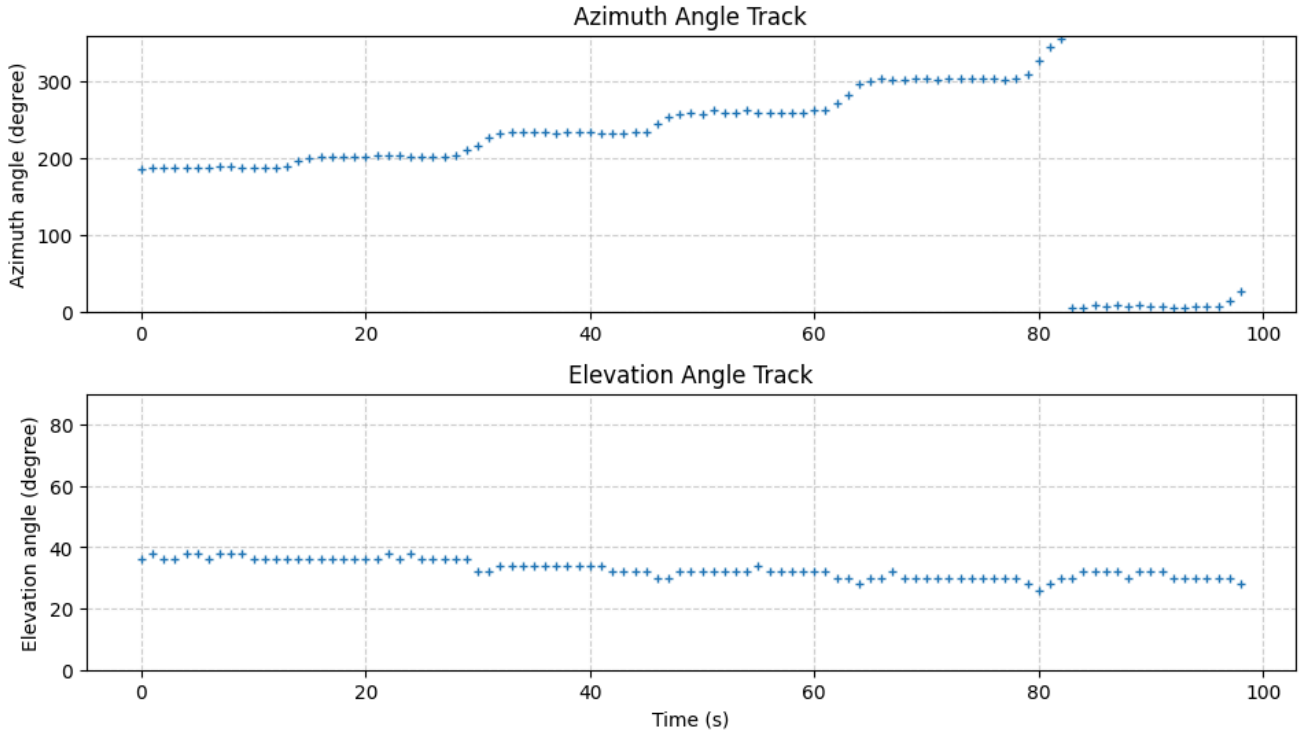
In [25]:

```
NUM_WINDOWS = 99

# Initialize lists to store the tracking results
azimuth_track = []
elevation_track = []
u_track = []
v_track = []

print(f"Starting DOA tracking for {NUM_WINDOWS} windows...")
for i in range(NUM_WINDOWS):
    if i % 10 == 0:
        print(f"Window {i+1}/{NUM_WINDOWS} complete.")

    start_sample = i * SAMPLE_RATE
    end_sample = start_sample + SAMPLE_RATE

    # Build the audio_segment for the current window from the list of channels.
    current_segment_list = [channel[start_sample:end_sample]
                            for channel in channel_data]
    audio_segment = np.vstack(current_segment_list)

    # Estimate the DOA for the current window.
    az_deg, el_deg, u_est, v_est = estimate_doa_for_segment(
        audio_segment=audio_segment
    )

    # Store the results for this window.
    azimuth_track.append(az_deg)
```

```
    elevation_track.append(el_deg)
    u_track.append(u_est)
    v_track.append(v_est)

time_axis = np.arange(NUM_WINDOWS)
plot_tracking_results(
    u_track=u_track,
    v_track=v_track,
    azimuth_track=azimuth_track,
    elevation_track=elevation_track,
    time_axis=time_axis
)
```

```
Starting DOA tracking for 99 windows...
Window 1/99 complete.
Window 11/99 complete.
Window 21/99 complete.
Window 31/99 complete.
Window 41/99 complete.
Window 51/99 complete.
Window 61/99 complete.
Window 71/99 complete.
Window 81/99 complete.
Window 91/99 complete.
```





Task 7: (12 points)

- In two points in time the UAV signal is overlaid by speech and the DOA of the speech is determined. Think about how the UAV can still be located here.
- Think about how to reduce noise and amplify the UAV signal with array signal processing techniques.

Solution

UAV Location with Speech Interference

We can locate the UAV, even when its signal is overlaid by speech, by exploiting differences in both its spatial origin and frequency content.

- **Spatial Filtering**: Since the UAV and the speech source have different directions of arrivals, we can use advanced beamforming. By this, we can use an adaptive beamformer like the **Capon (MVDR)** beamformer to suppress the speech interference, or we can also use **deterministic nulling** to place a blind spot in our beam pattern directly at the speech source's DOA.

- **Frequency Filtering**: The UAV's sound is dominated by high-frequency harmonics, while speech is broadband and at lower frequencies. We can use a targeted **band-pass filter** focused only on the UAV's harmonic frequencies to remove most of the speech signal before beamforming.

Noise Reduction and Signal Amplification

We can reduce noise and amplify the UAV signal by using **beamforming**.

This process allows us to achieve **beamforming gain**, which improves the Signal-to-Noise Ratio (SNR). We make it work by coherently adding the phase-aligned signal from the UAV's direction across all our microphone elements, causing it to be amplified. Simultaneously, the random, uncorrelated background noise from each microphone adds incoherently, effectively reducing its overall level relative to the signal.