1. ***FORMULAS***

Amplitude modulation (AM):

$s(t) = [1 + m * \cos(2*\pi*f_m*t)] * \cos(2*\pi*f_c*t)$

f_c = carrier frequency (Hz)
f_m = message / modulating frequency (Hz)
m = modulation index (0 to 1)
t = time (seconds)

The term $[1 + m*\cos(2*\pi*f_m*t)]$ is the "envelope": a gain that rises and falls over time and is applied to the carrier wave.

Sideband expansion (product-to-sum identity):

$\cos(A)*\cos(B) = 1/2 * [\cos(A-B) + \cos(A+B)]$

Applying this to the AM formula above gives:

$s(t) = \cos(2*\pi*f_c*t)$
$+ (m/2) * \cos(2*\pi*(f_c - f_m)*t)$ <- lower sideband
$+ (m/2) * \cos(2*\pi*(f_c + f_m)*t)$ <- upper sideband

This shows that an AM signal always contains exactly three frequencies: the carrier, and two sidebands offset from it by exactly f_m. Nothing else is present in the signal.

Fourier transform (discrete, real-valued signal):

$X[k] = \text{sum over n of } x[n] * \exp(-2j*\pi*k*n/N)$

Implemented with numpy's rfft (real FFT), which returns the spectrum for non-negative frequencies only. Converting to decibels:

$\text{magnitude_dB} = 20 * \log10(|X[k]| / \max(|X|))$

This expresses each frequency bin's magnitude relative to the strongest bin (0 dB = peak, negative values = weaker).

Frequency resolution of an FFT:

resolution (Hz per bin) = sample_rate / number_of_samples

A longer recording (more samples) gives finer frequency resolution, i.e. a narrower spectral peak per tone.

2. *VARIABLES*

f1_khz, f2_khz - the two frequencies typed into the GUI, in kHz.

f_carrier - the higher of the two input frequencies, in Hz (f1_khz or f2_khz * 1000).

f_message - the lower of the two input frequencies, in Hz.

mod_index - modulation index m, entered in the GUI (0 to 1).

duration - length of the signal to generate/export, in seconds.

fs - sample rate in Hz, picked automatically so it safely covers the highest frequency present (f_carrier + f_message).

t - array of time values (seconds), one per audio sample.

message - array holding cos(2*pi*f_message*t), the modulating tone.

envelope - array holding 1 + mod_index * message.

am_signal - the final AM waveform, s(t), normalized so its peak amplitude is 1 (so it can be safely saved to a WAV file without clipping).

lower_sb - the lower sideband frequency, f_carrier - f_message.

upper_sb - the upper sideband frequency, f_carrier + f_message.

freqs - array of frequency values (Hz) returned by the FFT.

mag_db - array of magnitudes (dB) at each frequency in freqs.

resolution - FFT frequency resolution, fs / number_of_samples.

half_width - how far (in Hz) each spectrum plot zooms out from its peak, so the peak's shape is visible instead of a single-pixel spike.

3. ***FUNCTIONS***

choose_sample_rate(f_carrier, f_message)

Picks a standard audio sample rate (from a list of common rates: 44100, 48000, 88200, 96000, 176400, 192000 Hz) that is comfortably more than twice the highest frequency present in the signal (the upper sideband). This satisfies the Nyquist sampling theorem, so the signal can be digitized and later exported without aliasing (frequency distortion from under-sampling).

generate_am(f_carrier, f_message, fs, duration, mod_index)

Builds the actual AM signal at the audio sample rate fs. Returns the time array, the message signal, and the normalized AM waveform. This is the array used for the Fourier transform and for the WAV export.

generate_am_curve(f_carrier, f_message, duration, mod_index, points_per_cycle, max_points)

Builds the same AM signal, but on a much denser, evenly spaced time grid (400 points per carrier cycle by default, capped at max_points total so very long windows don't create huge arrays) instead of at the audio sample rate. This is used only for the on-screen waveform plot, so the curve looks smooth no matter what frequencies are entered, rather than being limited by how few audio samples land within one cycle. The waveform plot calls this with duration = 5 / f_message, so the window always spans 5 full envelope cycles, regardless of how the carrier and message frequencies relate to each other.

compute_spectrum(signal, fs)

Runs the Fourier transform. Applies a Hann window to the signal first (tapers the edges of the recording to zero, which reduces artifacts called "spectral leakage"), then computes the real FFT (numpy.fft.rfft), then converts magnitude to decibels relative to the strongest frequency present.

export_wav(path, signal, fs)

Converts the normalized signal (range -1 to 1) into 16-bit PCM audio samples and writes it to a .wav file at the given sample rate, using scipy.io.wavfile. The resulting file can be opened in any standard audio player or editor.

DirectionalSpeakerApp.generate_and_plot(self)

Reads the values from the GUI's input fields, validates them, calls generate_am and generate_am_curve to build the signal, calls compute_spectrum to analyze it, and redraws the waveform plot and the three spectrum panels (carrier, lower sideband, upper sideband). Also updates the results text box with the computed frequencies.

DirectionalSpeakerApp.export(self)

Opens a "Save As" file dialog and, once a location is chosen, calls export_wav to write the most recently generated AM signal to disk.

4. ***HOW IT ALL FITS TOGETHER***

We type two frequencies (kHz) into the GUI.

generate_am_curve() builds a smooth version of the signal for the waveform plot; generate_am() builds the actual sampled signal used for analysis and export.

compute_spectrum() runs the Fourier transform on that sampled signal,

confirming the three predicted frequencies (carrier, lower sideband, upper sideband) are exactly where the AM formula says they should be.

export_wav() lets you save the sampled signal as a standard .wav file, which you can then open in any audio application to listen to it.