Workflow :

Phase 1: Initialization

1. **Script Start:** You run demo_feature_extraction.py.
2. **Entry Point:** The script begins execution in the if __name__ == "__main__": block, which immediately calls the main() function.
3. **Directory Setup:** The main() function's first action is os.makedirs('test_results', exist_ok=True). This creates a new folder named test_results in the same directory as the script. exist_ok=True cleverly prevents the script from crashing if the folder already exists from a previous run.
4. **Signal Generation:** The main() function prints "Generating test signals..." and then calls generate_test_signals().

Phase 2: Synthetic Signal Generation

This phase happens entirely within the generate_test_signals() function.

1. **Setup:** It defines a sampling frequency (fs=1e6 Hz) and duration (duration=0.001 s), which means it will create **1000 data samples** for each signal. A time vector t is created to represent these 1000 points.
2. **Test Signal Creation:** It builds four different complex I/Q signals from scratch.
    - **QPSK:** A digital signal is created by:
        1. Choosing random phase "jumps" (0, 90, 180, 270 degrees).
        2. Creating complex symbols from these phases (e.g., $1+0j$, $0+1j$, $-1+0j$).
        3. Repeating each symbol to simulate a symbol rate (making the signal "blocky").
        4. Multiplying the result by a complex carrier wave to shift its frequency to **100 kHz**.
        5. Adding a small amount of random complex noise.
    - **FM:** An analog signal is created by:
        1. Creating a low-frequency (5 kHz) sine wave (the "modulating" signal).
        2. Creating a 150 kHz carrier.

3. The *phase* of the carrier is modified directly by the modulating sine wave.
4. The final signal is generated as $e^{j \cdot \text{phase}(t)}$, resulting in a signal with constant amplitude but varying frequency.
5. Adding noise.

- **BURST:** An on-off signal is created by:
  1. Creating a simple 200 kHz carrier wave.
  2. Creating an "envelope" array of all zeros.
  3. Looping and setting "chunks" of the envelope to 1.0 (e.g., samples 0-200 are 1, 201-499 are 0, 500-700 are 1, etc.).
  4. Multiplying the carrier by this 0/1 envelope, effectively turning it on and off.
  5. Adding noise.
- **AM:** An analog signal is created by:
  1. Creating a 120 kHz carrier wave.
  2. Creating a 3 kHz modulating sine wave.
  3. Creating an amplitude envelope based on the formula $(1 + 0.5 \times \text{modulating\_wave})$.
  4. Multiplying the carrier by this *analog* envelope, resulting in a signal with varying amplitude but constant frequency.
  5. Adding noise.

3. **Return:** The function returns a dictionary signals (containing the four numpy arrays), the time vector t, and the sample rate fs back to the main() function.

Phase 3: Main Processing Loop

Back in main(), the script now loops through the signals dictionary, processing one signal at a time. We will follow the **'qpsk'** signal through the entire test_all_features function.

Loop 1: signal_name = 'qpsk'

The main function calls test_all_features('qpsk', iq_signal, t, fs).

Inside test_all_features('qpsk', ...):

1. **Spectrogram:**
    - Calls compute_spectrogram(iq_signal, fs, nperseg=256).
    - **Inside compute_spectrogram:** This function from signal_features.py uses scipy.signal.spectrogram to perform a **Short-Time Fourier Transform (STFT)**. It breaks the 1000-sample signal into 256-sample segments (with 128 samples of overlap), calculates the Power Spectral Density (PSD) for each, and converts the power to **decibels (dB)** using $10 \times \log_{10}(\text{power})$.
    - plot_spectrogram_result is called, which uses matplotlib to create a heatmap of the result (time vs. frequency vs. power) and saves it as test_results/qpsk_spectrogram.png.
2. **Center Frequency & PSD:**
    - Calls estimate_center_frequency(iq_signal, fs, method='peak').
    - **Inside estimate_center_frequency:**
        1. Calculates the **Fast Fourier Transform (FFT)** of the *entire* signal using fft().
        2. Calculates the PSD (power) by taking the squared absolute value of the FFT result.
        3. It shifts the FFT output (fftshift) so that 0 Hz is in the center.
        4. Because method='peak', it finds the *single frequency bin with the most power* (np.argmax(psd)) and returns that frequency as the center_freq.
    - plot_psd is called, which plots the PSD (power vs. frequency) and saves it as test_results/qpsk_psd.png.
3. **Bandwidth:**
    - Calls estimate_bandwidth(iq_signal, fs, threshold_db=-10).
    - **Inside estimate_bandwidth:**
        1. It *first* calls estimate_center_frequency to get the full PSD.
        2. It converts the PSD to dB.
        3. It finds the maximum power (peak_power_db).
        4. It calculates a threshold (e.g., peak_power_db - 10).
        5. It finds all frequencies where the power is *above* this threshold.

6. The bandwidth is calculated as the difference between the *highest* and *lowest* frequencies that met this threshold.

4. **Constellation Diagram:**
    - Calls plot_constellation(iq_signal). This is a simple plot. It just creates a scatter plot of the **real part (I)** vs. the **imaginary part (Q)** of the complex data.
    - Saves the plot as test_results/qpsk_constellation.png.
5. **Burst Detection:**
    - Calls detect_bursts(iq_signal, fs, threshold_db=-10).
    - **Inside detect_bursts:**
        1. Calculates the signal's instantaneous amplitude, or **envelope**, using np.abs(iq_data).
        2. Converts the envelope to dB.
        3. Finds the **median** power level of the entire signal.
        4. Sets a detection threshold 10 dB *above* this median.
        5. Finds all contiguous blocks of samples that are *above* this threshold.
        6. It returns a list of these blocks (bursts) and the envelope itself.
6. **Duty Cycle:**
    - Calls compute_duty_cycle(bursts, len(iq_signal), fs).
    - **Inside compute_duty_cycle:** It simply sums the total duration (in seconds) of all bursts found in the previous step and divides by the total signal duration (0.001 s).
7. **Time Domain Plot:**
    - Calls plot_time_domain(t, iq_signal, envelope=envelope).
    - This function creates two subplots: one for the **I (real)** component and one for the **Q (imaginary)** component over time.
    - It overlays the envelope (from step 5) as a red dashed line.
    - Saves the plot as test_results/qpsk_timedomain.png.
8. **Modulation Classification:**
    - Calls classify_modulation_simple(iq_signal, fs).

- **Inside classify_modulation_simple:** This is a **heuristic** (rule-based) function.
    1. It calculates amplitude statistics (how much the amplitude varies).
    2. It calculates phase statistics (how much the *unwrapped* phase varies).
    3. It uses a simple if/else tree to make a guess. (e.g., "IF amplitude variation is high AND phase variation is low, THEN classify as 'AM'").

9. **Round-Trip Test (IQ -> Waveform):**
    - Calls iq_to_waveform(iq_signal[:1000], fs).
    - **Inside iq_to_waveform:** This converts the complex signal back to a real one by interpolating (upsampling) it and shifting its frequency.
10. **Round-Trip Test (Waveform -> IQ):**
    - Calls waveform_to_iq(waveform, fs_real).
    - **Inside waveform_to_iq:** This uses the **Hilbert Transform** (scipy.signal.hilbert) to reconstruct the complex analytic signal from the real waveform.

Loop 2, 3, 4:

The script repeats this entire 10-step test_all_features process for the 'fm', 'burst', and 'am' signals, saving their respective plots.

Phase 4: Summary & Cleanup

1. **Report Generation:** After the main loop is finished, the main() function now has the all_results dictionary, which contains the numerical results for all four signals.
2. **Formatting:** It loops through all_results and builds a large string (summary_text) that neatly formats these results into a table.
3. **Save & Print:** This summary string is:
    - Written to a new file: test_results/summary.txt.
    - Printed to the console.
4. **Finish:** The main() function ends, and the script execution is complete.

Unexpected or Incorrect Results :

The discrepancies stem from specific logic flaws in the signal_features.py module and the parameters used in the demo_feature_extraction.py script.

---

1. AM Signal Bandwidth is 0.0 kHz

- **Unexpected Result:** The AM signal shows a bandwidth of **0.0 kHz**.
- **Expected Result:** The AM signal is generated with a 3 kHz modulating tone (fm = 3e3). This should create two sidebands, one at +3 kHz and one at -3 kHz from the carrier. The total bandwidth should be **6.0 kHz**.
- **Reason:** This error is caused by the combination of the estimate_bandwidth function's threshold and the AM signal's modulation index.
    1. The estimate_bandwidth function is called with threshold_db=-10. This means it looks for all frequency components that are within -10 dB of the peak power.
    2. The AM signal is generated with m = 0.5. The power of the sidebands relative to the carrier (the peak) is calculated as $10 \times \log_{10}((m/2)^2)$.
    3. $10 \times \log_{10}((0.5/2)^2) = 10 \times \log_{10}(0.0625) \approx \textbf{-12.04 dB}$.
    4. Because the sidebands (-12.04 dB) are weaker than the -10 dB threshold, the *only* frequency component strong enough to be detected is the carrier itself. This results in the function measuring a bandwidth of 0.

---

2. QPSK and FM Center Frequencies are Incorrect

- **Unexpected Result:**
    - QPSK is centered at **88.0 kHz** instead of its generated 100 kHz.
    - FM is centered at **130.0 kHz** instead of its generated 150 kHz.
- **Reason:** The demo script uses estimate_center_frequency(..., method='peak').
    - This method finds the **single frequency bin with the most power**.

- For modulated signals like QPSK (which has a suppressed carrier) and wideband FM, the *carrier frequency* is often **not** the strongest point in the spectrum. The power is distributed among the sidebands.
- The function is correctly finding the *peak sideband* (e.g., at 88 kHz) but incorrectly reporting this as the *center frequency*.
- **Note:** The bandwidth calculation *is* centered correctly (e.g., the FM range is 120-180 kHz, centered at 150 kHz), but the separate "Center Freq" metric reported in the table is wrong because it only used the peak method. Using method='weighted' would likely fix this.

---

3. BURST Signal Detection is Incorrect

- **Unexpected Result:** The BURST signal shows **5 bursts** and a **47.90%** duty cycle.
- **Expected Result:** The signal is generated with a 200 µs pulse every 500 µs. In a 1000 µs (1 ms) duration, this should create **2 bursts** and a **40.0%** duty cycle ($2 \times 200 \mu s / 1000 \mu s$).
- **Reason:** This error is caused by an incorrect parameter in the detect_bursts function call.
    1. The function is called with threshold_db=-10.
    2. Inside detect_bursts, the threshold is set as threshold = median_power + threshold_db.
    3. For the BURST signal, the signal is "off" (just noise) 60% of the time. Therefore, the median_power will be the median power of the **noise floor**.
    4. The code sets the detection threshold to median(noise_floor) - 10. This is **10 dB *below* the noise floor**.
    5. As a result, the function isn't detecting the signal bursts. It's detecting random, brief **dips *in the noise*** (fades) that momentarily drop below this very low threshold. This results in a random number of "bursts" (in this case, 5).
    6. This also explains why QPSK, FM, and AM all show "1 burst" and "100% duty cycle"—their signal power never drops below the noise floor, so they are detected as one continuous burst.

---

4. FM Modulation Classification is Wrong

- **Unexpected Result:** The FM signal is classified as **"PSK-like"** instead of "FM/PM."
- **Reason:** The heuristic logic in classify_modulation_simple is flawed.
    1. The function distinguishes between FM and PSK based on phase_std, which is the standard deviation of np.diff(phase_unwrapped). This measures the standard deviation of the **instantaneous frequency**.
    2. It expects FM to have a *high* standard deviation (> 1.0) and PSK to have a *low* one.
    3. However, the generated FM signal's frequency varies *smoothly* (following a sine wave). The PSK signal, in contrast, has an instantaneous frequency of zero *except* for large, sharp spikes at symbol changes.
    4. The FM signal's smooth frequency variation results in a *lower* standard deviation than the classifier's 1.0 threshold, causing it to be misclassified as "PSK-like."