Code Breakdown

Server Boot & File Loading

```
s.boot;
~buffer = Buffer.read(s, "path");
```

- **`s.boot`**: Starts the SuperCollider audio server
- **`~buffer`**: Global environment variable storing the loaded audio
- **`Buffer.read()`**: Loads file into server memory for playback
- Only need to change the path for this, we can also change to live input

SynthDef Structure

```
SynthDef(\drumDetectorFile, {
    arg bufnum, threshold = 0.2, rate = 1.0, startPos = 0;
    // Processing code here
}).add;
```

- **`SynthDef()`**: Creates a reusable audio processing template
- **`\drumDetectorFile`**: Symbol name for this synth definition
- **`arg`**: Parameters for the synth def
  - `bufnum`: Which buffer to analyze
  - `threshold`: Sensitivity level (default 0.2)
  - `rate`: Playback speed (1.0 = normal)
  - `startPos`: Where to start in the file (sample number)
- **`.add`**: Sends definition to server

Audio Input & Timing

```
input = PlayBuf.ar(1, bufnum, rate, startPos: startPos, loop: 0, doneAction: 2);
currentTime = Phasor.ar(0, rate / SampleRate.ir, startPos, BufFrames.kr(bufnum));
```

- **`PlayBuf.ar()`**: Plays audio from buffer
  - `1`: Mono (1 channel) // 2 channels breaks some stuff, we only need 1 channel for fft anyways
  - `bufnum`: Which buffer to play
  - `rate`: Playback speed
  - `loop: 0`: Don't loop
  - `doneAction: 2`: Free the synth when playback ends
- **`Phasor.ar()`**: Creates a ramp for tracking playback position
  - Used to generate timestamps for detected events

Juicy stuff

Frequency Band Separation

```
kickBand = LPF.ar(input, 80);           // Bass: 20-80Hz
snareBand = BPF.ar(input, 250, 2.0);    // Snare: ~150-400Hz
hihatBand = HPF.ar(input, 8000);        // Hi-hat: 8kHz+
```

- **`LPF.ar()`**: Low-pass filter (keeps frequencies below 80Hz for kick detection)
- **`BPF.ar()`**: Band-pass filter (keeps frequencies around 250Hz for snare)
  - `2.0`: Bandwidth parameter (higher = wider frequency range)
- **`HPF.ar()`**: High-pass filter (keeps frequencies above 8kHz for hi-hat)

Amplitude Analysis

```
kickEnv = Amplitude.ar(kickBand, 0.01, 0.1);
snareEnv = Amplitude.ar(snareBand, 0.01, 0.1);
hihatEnv = Amplitude.ar(hihatBand, 0.001, 0.05);
totalEnergy = Amplitude.ar(input, 0.01, 0.1);
```

- **`Amplitude.ar()`**: Tracks volume/energy of a signal over time
  - First number: Attack
  - Second number: Release
- **Different timing for hi-hat**: Faster response (0.001, 0.05) because hi-hats are quicker

Feature Extraction

```
zeroCrossings = ZeroCrossing.ar(input);
fft = FFT(LocalBuf(1024), input);
spectralCentroid = SpecCentroid.kr(fft).clip(20, 20000);
```

- **`ZeroCrossing.ar()`**: Counts how often signal crosses zero (indicates "noisiness")
- **`FFT()`**: Fast Fourier Transform for frequency analysis
  - `LocalBuf(1024)`: Creates 1024-sample buffer for FFT analysis
- **`SpecCentroid.kr()`**: Calculates "brightness" of sound (where most energy is)
- **`.clip(20, 20000)`**: Limits values to reasonable frequency range

Trigger Logic (Kick)

```
kickTrig = Trig1.ar(
    (kickEnv > (threshold * 2)) *
    (kickEnv > (snareEnv + 0.01)) *     // Kick should dominate over snare
    (kickEnv > (hihatEnv * 2)),         // Kick should dominate over hihat
    0.15  // Longer holdoff to prevent multiple triggers
);
```

- `Trig1.ar()`: Creates a trigger that fires once when condition is true
  - Condition: All three comparisons must be true (multiplication in supercollider is the same as using AND)
  - `0.15`: Holdoff time (won't trigger again for 150ms)
- **Logic**: Kick energy must be:
  1. Above threshold × 2
  2. Greater than snare energy + small offset
  3. Greater than hi-hat energy × 2

Trigger Logic (Snare)

```
snareTrig = Trig1.ar(
    (snareEnv > threshold) *
    (totalEnergy > (threshold * 0.5)) * // General activity threshold
    (snareEnv > (kickEnv * 0.4)),       // Some independence from kick
    0.1
);
```

- **Snare conditions**:
  1. Snare energy above threshold
  2. Total energy above half threshold (something is happening)
  3. Snare energy greater than 40% of kick energy

Trigger Logic (Hi-hat)

```
hihatTrig = Trig1.ar(
    (hihatEnv > (threshold * 0.3)) *
    (hihatEnv > (kickEnv * 1.5)) *     // Hi-hat should be louder than kick
    (zeroCrossings > 0.1),             // High-frequency content indicator
    0.06  // Shorter holdoff for rapid hi-hat patterns
);
```

- **Hi-hat conditions**:
  1. Hi-hat energy above 30% of threshold (more sensitive)
  2. Hi-hat louder than kick × 1.5
  3. High zero-crossing rate (indicates high-frequency content)
- **Shorter holdoff**: 60ms allows for rapid hi-hat patterns

Note: We can tweak these values to make the detection more accurate.

Data Capture

```
kickLevel = Latch.ar(kickEnv, kickTrig);
snareLevel = Latch.ar(snareEnv, snareTrig);
hihatLevel = Latch.ar(hihatEnv, hihatTrig);
```

- `Latch.ar()`: "Freezes" a value when trigger fires
- Captures the exact energy level at the moment of detection

OSC Message Sending

```
SendReply.ar(kickTrig, '/kick', [
    currentTime / SampleRate.ir,        // Timestamp in seconds
    kickLevel,
    Latch.ar(spectralCentroid, kickTrig),
    Latch.ar(zeroCrossings, kickTrig),
    kickEnv / (snareEnv + 0.001),       // Kick dominance ratio
    kickEnv / (hihatEnv + 0.001)        // Kick vs hihat ratio
]);
```

- **`SendReply.ar()`**: Sends OSC message when trigger fires
  - First arg: When to send (the trigger)
  - Second arg: OSC address ('/kick')
  - Third arg: Array of data to send
- **Data sent**: Timestamp, energy level, spectral info, dominance ratios

OSC Responders

```
OSCdef(\kickResponder, {
    arg msg, time;
    var timestamp = msg[3];
    var level = msg[4];
    // ... extract other values
    "KICK @ %s - Level: %, Centroid: %Hz, Dominance: %".postf(/*values*/);
}, '/kick');
```

- **`OSCdef()`**: Sets up listener for OSC messages
- **`msg`**: Array containing the sent data
  - `msg[3]` = first data value (timestamp)
  - `msg[4]` = second data value (level), etc.
- **`.postf()`**: Formatted printing to console

Control & Cleanup

```
~detector = Synth(\drumDetectorFile, [\bufnum, ~buffer]);
~detector.set(\threshold, 0.4);
~detector.free;
s.freeAll;
```

- **`Synth()`**: Creates instance of the SynthDef
- **`.set()`**: Changes parameters while running
- **`.free`**: Stops and removes synth
- **`s.freeAll`**: Stops all synths on server
- **`OSCdef().free`**: Removes OSC listeners
- **`~buffer.free`**: Frees buffer memory

What This Code Does (or atleast is supposed to do):

1. **Loads audio file** into memory
2. **Splits into 3 frequency bands** (kick, snare, hi-hat ranges)
3. **Tracks energy** in each band continuously
4. **Detects sudden increases** (onsets) with smart logic to avoid false positives
5. **Sends detailed analysis data** via OSC when drums are detected
6. **Prints results** to console with timestamps and features
7. **Plays original audio** for monitoring