EE217 Lab 2 Report
PRBS-Based Capacitive Touch Sensing

1 Part I: PRBS Drive and ADC Sampling

Sampling Rate of the ADC

The ADS1256 was configured with gain = 2× and operated in continuous conversion mode. The SPI interface was clocked at 3 MHz.

By timing a large batch of ADC reads and dividing the number of samples by the elapsed time, the effective sampling throughput was measured to be approximately

$$f_{\text{ADC}} \approx 15{,}000 \text{ samples/sec.}$$

Sampling Rate Optimization

Several implementation choices were made to increase achievable sampling throughput:

- Increasing the SPI clock frequency to 3 MHz
- Enabling the ADS1256 input buffer
- Avoiding unnecessary register reconfiguration between conversions
- Minimizing settle-discard samples after MUX switching
- Using tight timing loops for PRBS clock generation

These optimizations increased the achievable read rate while maintaining stable correlation measurements.

2 Part II: Touch Sensing (Correlator + Centroid)

Frame Rate

A PRBS order of $n = 7$ was used:

$$L = 2^7 - 1 = 127.$$

With five drive electrodes and seven sense channels, the resulting measurement frame rate was approximately

$$\boxed{4.0 \text{ frames/sec}}.$$

Effect of PRBS Length

The PRBS length determines the duration of the correlation sequence used to estimate coupling between each drive and sense electrode pair. Increasing the sequence length improves signal-to-noise ratio because the correlation integrates over a longer excitation pattern. However, the time required to acquire one frame increases proportionally with PRBS length, which reduces the frame rate.

To evaluate this trade-off, multiple PRBS lengths were tested and the resulting frame rate was measured.

PRBS Order n	Sequence Length $(2^n - 1)$	Frame Rate (fps)
5	31	15.8
6	63	8.1
7	127	4.0

Table 1: Measured frame rate as a function of PRBS sequence length.

As expected, longer sequences reduce frame rate while improving correlation stability. In practice, $n = 6$ was the shortest sequence that still produced reliable centroid localization. Shorter sequences such as $n = 5$ resulted in noticeably noisier centroid estimates. Order $n = 7$ was therefore selected as a compromise between spatial stability and interactive frame rate.

Baseline Capacitance Map

A baseline capacitance map was obtained by averaging 30 frames while the sensor was untouched. The baseline represents the nominal correlation value for each drive–sense pair and serves as the reference used to detect touch-induced capacitance changes.

The measured baseline noise level was approximately

$$\overline{\sigma}_{\text{baseline}} \approx 1.18 \times 10^{-4}.$$

Touch detection was implemented using a threshold condition

$$|\Delta C_{d,s}| > 6\sigma_{d,s}$$

which robustly suppresses false detections due to measurement noise.

Thumb and Pinky Heatmaps

Representative touch heatmaps are shown below. These figures display the baseline-subtracted correlation magnitude, the detected touch region, the centroid location, and an ellipse approximation of the contact area.

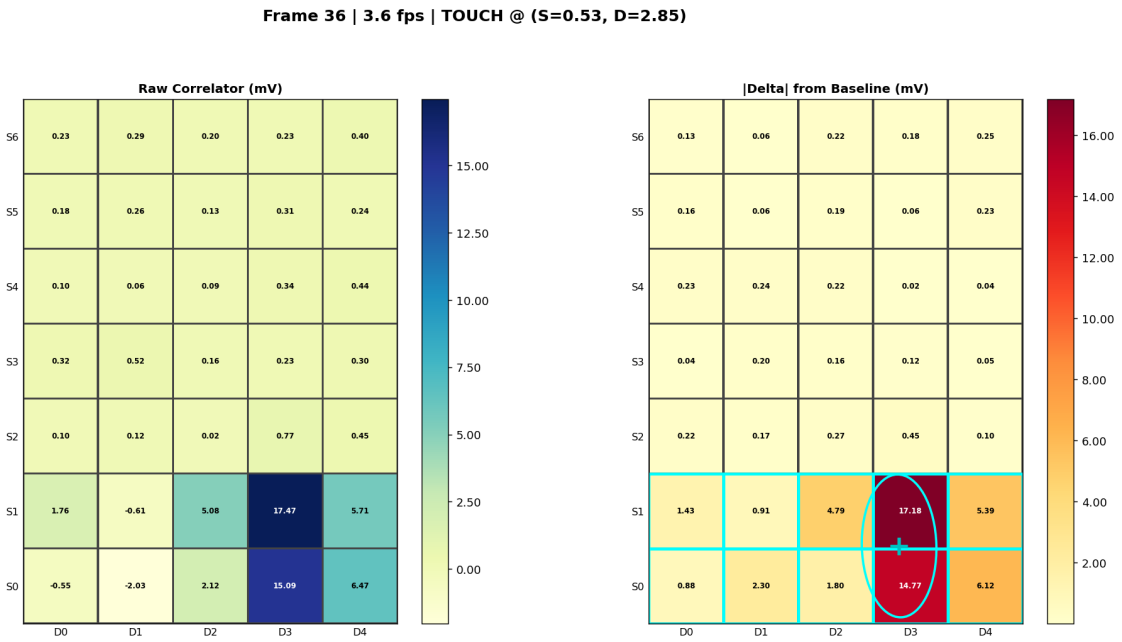

Figure 1: Thumb contact heatmap with centroid and ellipse estimate.

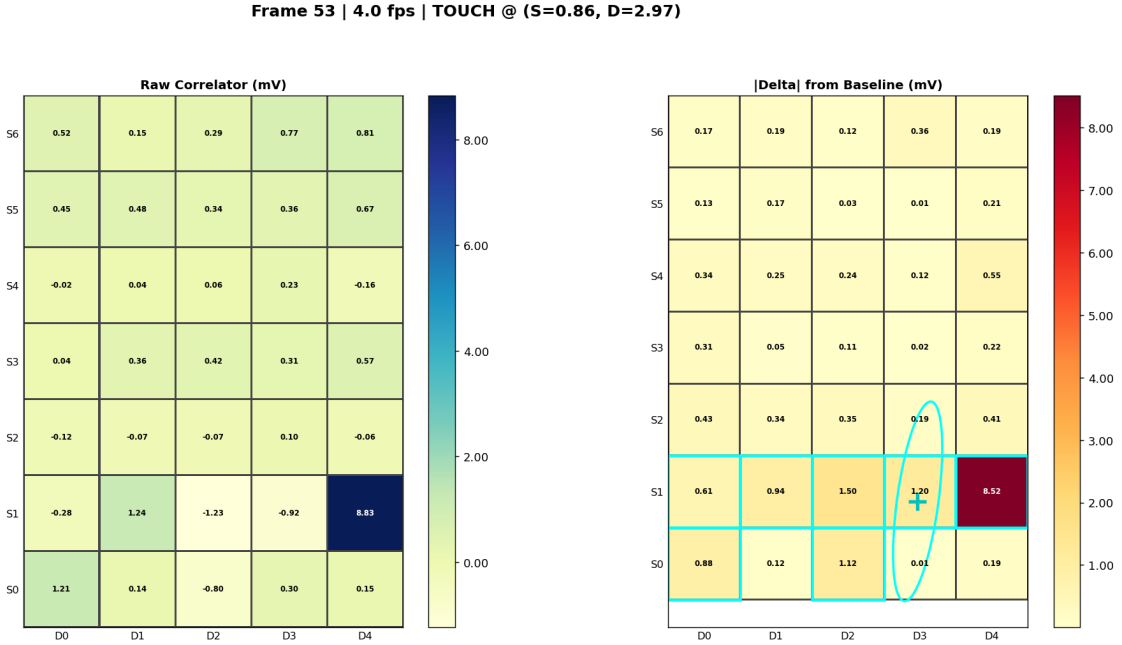

Figure 2: Pinky tip contact heatmap with centroid and ellipse estimate.

Centroid Estimation

Touch position was estimated using a weighted centroid over the positive correlation region:

$$x_c = \frac{\sum w_{d,s} s}{\sum w_{d,s}}, \qquad y_c = \frac{\sum w_{d,s} d}{\sum w_{d,s}}$$

where

$$w_{d,s} = \max(\Delta C_{d,s}, 0).$$

This provides a continuous position estimate within the discrete drive–sense electrode grid.

Jitter Measurement (1000 Samples)

To quantify localization noise, a conductive penny was placed on the sensor surface and held stationary while recording 1000 centroid samples.

The measured jitter statistics were:

- $\sigma_x = 0.0264$ nodes
- $\sigma_y = 0.0591$ nodes
- RMS jitter $= 0.0647$ nodes

This corresponds to an estimated spatial resolution of

$$\boxed{0.065 \text{ nodes (RMS)}}.$$

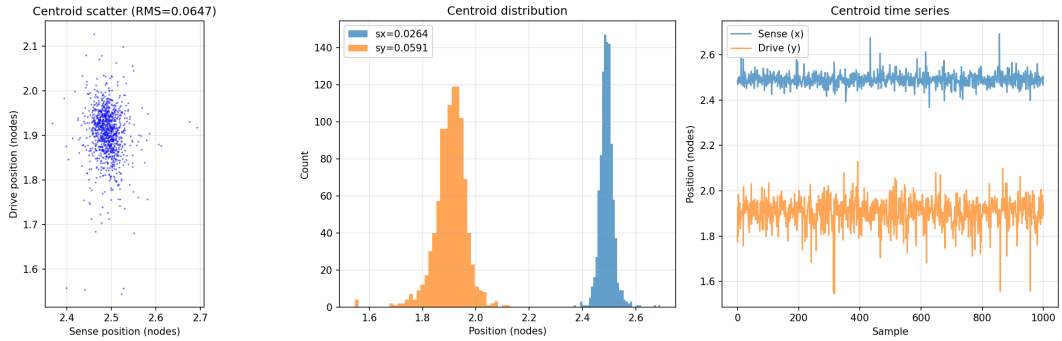

Figure 3: Centroid scatter and distribution for 1000 stationary samples.

3 Part III: Kalman-Filtered Touch Tracking

To reduce centroid jitter and estimate finger motion, a Kalman filter was applied to the raw centroid measurements.

State Representation

The filter state contains both position and velocity components:

$$\mathbf{x}_k = \begin{bmatrix} p_x \\ p_y \\ v_x \\ v_y \end{bmatrix}.$$

The measurement vector consists of the observed centroid position:

$$\mathbf{z}_k = \begin{bmatrix} p_x \\ p_y \end{bmatrix}.$$

System Model

A constant-velocity motion model was assumed:

$$\mathbf{x}_{k+1} = F\mathbf{x}_k + \mathbf{w}_k$$

$$F = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The measurement equation is

$$\mathbf{z}_k = H\mathbf{x}_k + \mathbf{v}_k$$

with

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}.$$

Noise Models

Measurement noise was estimated from the stationary 1000-sample test in Part II:

$$R = \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix} = \begin{bmatrix} 0.17^2 & 0 \\ 0 & 0.13^2 \end{bmatrix}.$$

Process noise was modeled as random acceleration with standard deviation

$$\sigma_a = 2.0 \text{ nodes/s}^2.$$

This produces the covariance matrix

$$Q = \sigma_a^2 \begin{bmatrix} \frac{\Delta t^3}{3} & 0 & \frac{\Delta t^2}{2} & 0 \\ 0 & \frac{\Delta t^3}{3} & 0 & \frac{\Delta t^2}{2} \\ \frac{\Delta t^2}{2} & 0 & \Delta t & 0 \\ 0 & \frac{\Delta t^2}{2} & 0 & \Delta t \end{bmatrix}.$$

Tracking Results

The Kalman filter significantly reduces noise in the centroid estimate and produces a smooth trajectory of finger motion.

During a slow slide across the sensor surface the system achieved:

- Mean speed: 6.23 nodes/s
- Maximum speed: 15.14 nodes/s

- Mean v_x: -1.52 nodes/s
- Mean v_y: 0.05 nodes/s

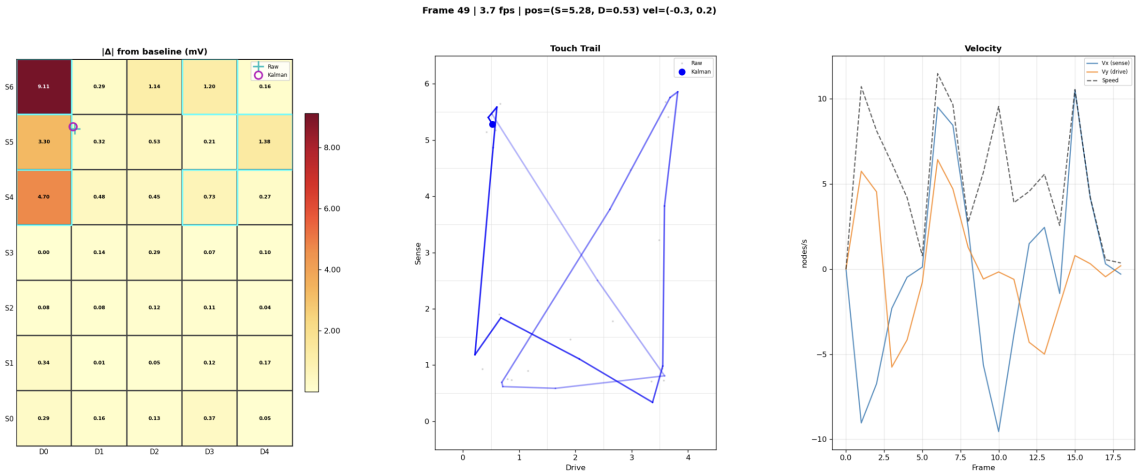

Figure 4: Live tracking frame showing heatmap, raw centroid points, Kalman filtered trajectory, and velocity estimates.

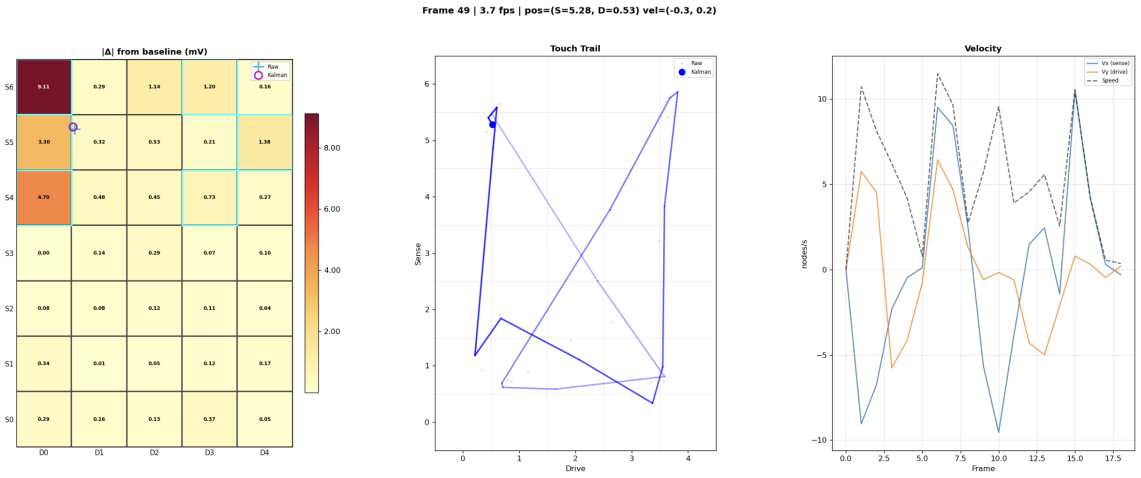

Figure 5: Example frame from the live tracking interface showing the touch heatmap, raw centroid estimate, Kalman filtered trajectory, and velocity estimates.

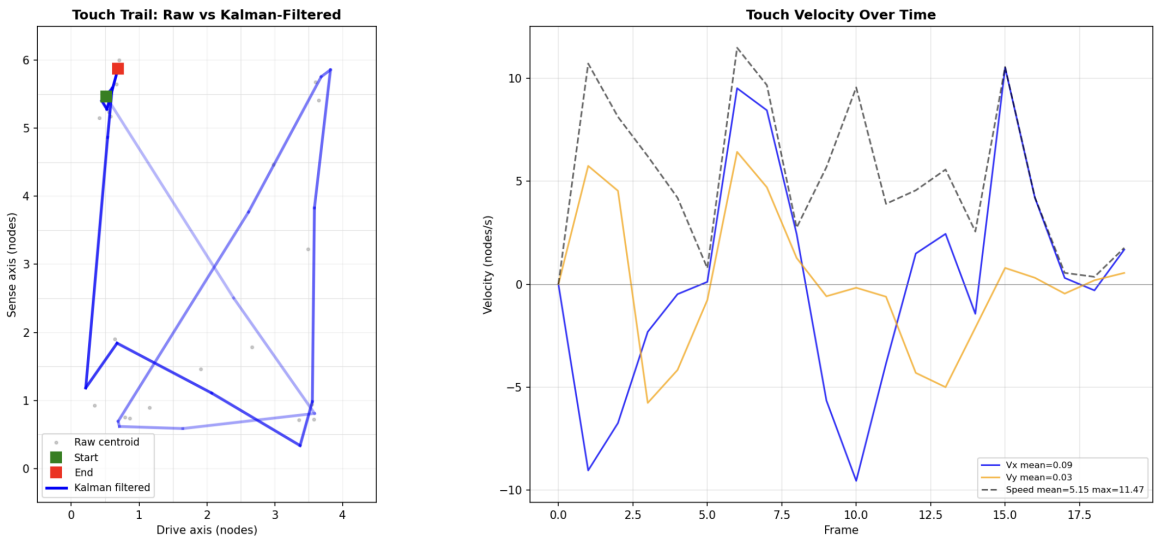

Figure 6: Summary of Kalman-filtered touch tracking showing the smoothed trajectory and estimated velocity over time.

Kalman vs Raw Comparison Video

A video demonstrating simultaneous raw centroid tracking and Kalman-filtered tracking is available at:

Google Drive

4 Part IV: Surface Dielectric Experiment (Packing Tape)

To evaluate how an insulating surface layer affects capacitive touch sensing, the entire electrode grid was covered with a single layer of clear packing tape and the experiments from Part II were repeated.

The tape acts as a dielectric layer between the finger and the sensor electrodes. This increases the distance between the conductive object and the electrodes, reducing capacitive coupling strength.

Experimental Setup

The same system parameters used in Part II were maintained:

- PRBS order: $n = 7$ (length 127)
- ADC gain: 2×
- 5 drive electrodes and 7 sense electrodes
- Baseline computed from 30 frames with no touch

The measured frame rate remained approximately

$$\boxed{4.0 \text{ frames/sec}}$$

since the PRBS length and ADC configuration were unchanged.
The measured baseline noise level was

$$\overline{\sigma}_{\text{baseline}} \approx 1.14 \times 10^{-4}$$

which is comparable to the baseline noise measured without the tape layer.

Effect of Tape on Touch Signal

With the insulating tape layer present, the magnitude of the correlation signal produced by finger contact was reduced compared to the direct-touch case. This occurs because the tape increases the separation between the conductive object and the sensing electrodes, decreasing the effective capacitance change.

Despite the reduced coupling, touch detection and centroid localization remained stable using the same PRBS order and threshold parameters. The centroid estimation algorithm continued to produce consistent position estimates, though the detected touch region was slightly more diffuse.

Example Heatmaps with Tape Layer

Representative correlation heatmaps for touches with the tape layer are shown below.

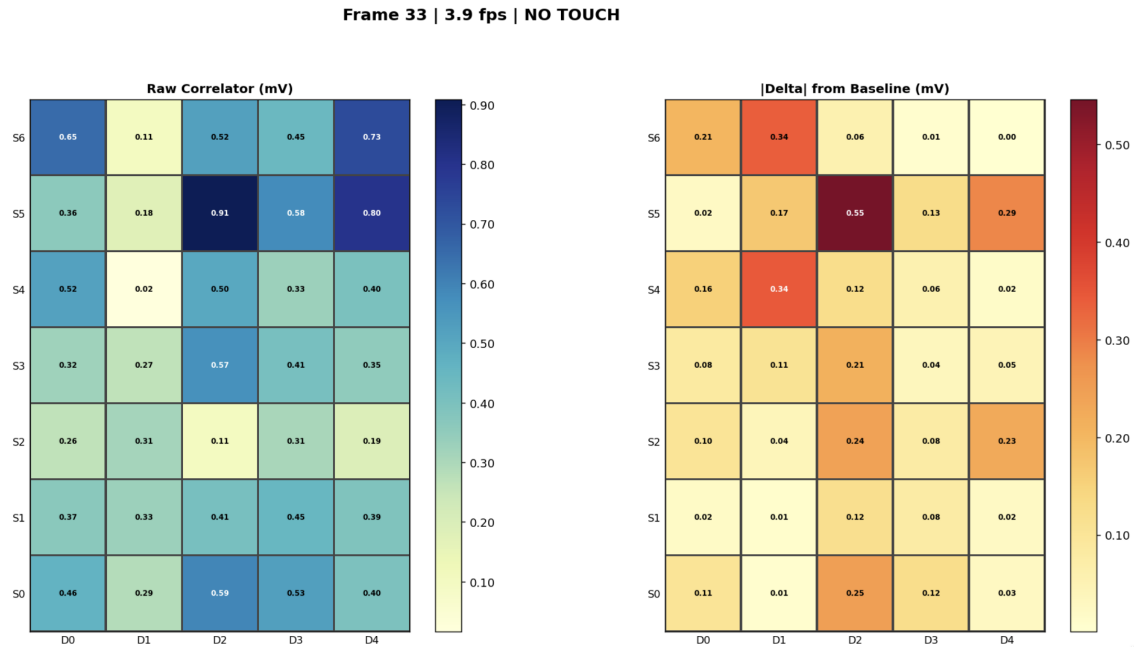

Figure 7: Thumb contact heatmap with packing tape covering the sensor.

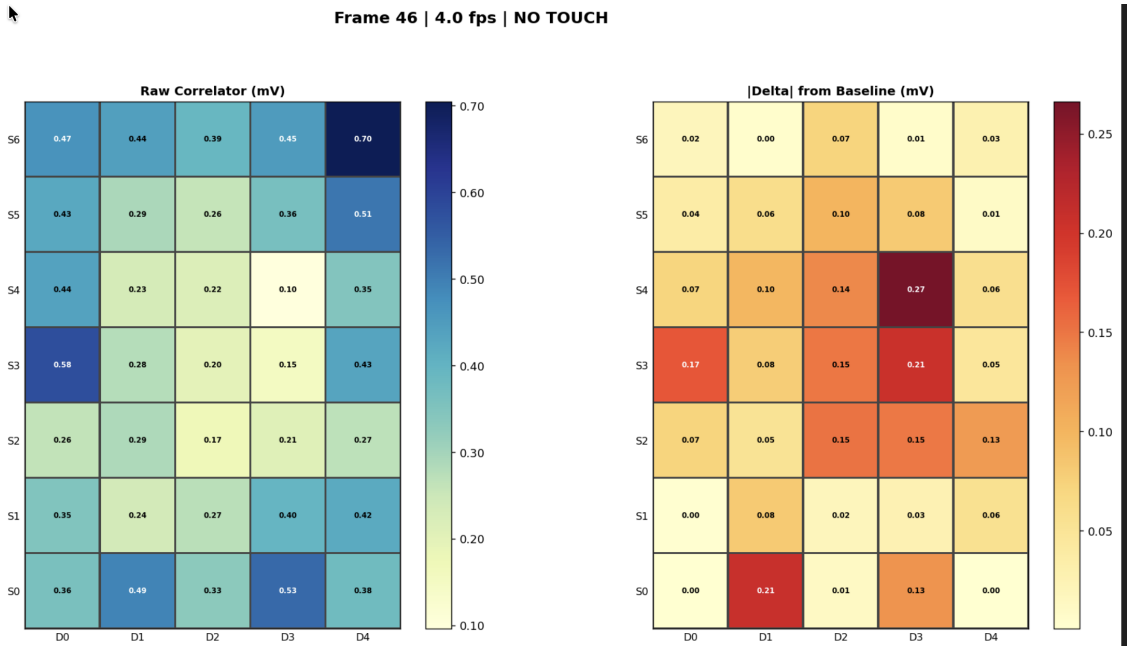

Figure 8: Pinky contact heatmap with packing tape covering the sensor.

Conductive Object Response

A conductive penny was also placed on the taped sensor surface to test stationary-contact behavior. Unlike the direct-contact case in Part II, the penny produced little to no measurable correlation response.

This occurs because the insulating tape prevents strong capacitive coupling between the metal coin and the electrodes. Without direct capacitive interaction through the dielectric layer, the signal change remains below the detection threshold.

Discussion

Adding the tape layer effectively reduces the sensor's sensitivity by increasing the dielectric thickness between the electrode plane and the touching object. This lowers the magnitude of the capacitance change induced by contact.

In practical capacitive touch systems, similar protective layers (such as glass or plastic screens) are present. These systems compensate for the reduced coupling by increasing electrode sensitivity, adjusting gain, or using longer correlation sequences.

In this experiment, the PRBS order $n = 7$ remained sufficient to detect finger touches even through the insulating layer, demonstrating that the correlation-based sensing approach is robust to moderate dielectric separation.

5 Appendix: Code

Part I Code

```
import RPi.GPIO as GPIO
import time
```

```
import numpy as np

# =========================
# Configuration Parameters
# =========================

LFSR_ORDER = 7 # PRBS order
BIT_RATE_HZ = 15000 # PRBS clock frequency
GPIO_DRIVE_LINES = [7, 12, 16, 20, 21]

# Valid LFSR tap configurations (maximal-length)
LFSR_TAPS = {
    2: [2, 1], 3: [3, 2], 4: [4, 3], 5: [5, 3],
    6: [6, 5], 7: [7, 6], 8: [8, 6, 5, 4], 9: [9, 5],
    10: [10, 7], 11: [11, 9], 12: [12, 11, 10, 4],
    13: [13, 12, 11, 8], 14: [14, 13, 12, 2],
    15: [15, 14], 16: [16, 15, 13, 4],
}


# =========================
# PRBS Generation
# =========================

def create_prbs_sequence(order, taps):
    """Generate␣a␣maximal-length␣PRBS␣using␣an␣LFSR."""
    sequence_length = (2 ** order) - 1
    shift_register = [1] * order
    output = []

    for _ in range(sequence_length):
        bit_out = shift_register[-1]
        output.append(bit_out)

        feedback = 0
        for tap in taps:
            feedback ^= shift_register[tap - 1]

        shift_register = [feedback] + shift_register[:-1]

    return np.array(output)


def build_drive_matrix(prbs_sequence, num_lines):
    """
␣␣␣␣Create␣phase-shifted␣PRBS␣pattern␣for␣each␣drive␣line.
␣␣␣␣Each␣drive␣uses␣a␣different␣phase␣for␣orthogonality.
␣␣␣␣"""
    seq_len = len(prbs_sequence)
    phase_step = seq_len // num_lines

    phase_offsets = [i * phase_step for i in range(num_lines)]

    drive_pattern = []
```

```
    for t in range(seq_len):
        drive_row = [
            int(prbs_sequence[(t + offset) % seq_len])
            for offset in phase_offsets
        ]
        drive_pattern.append(drive_row)

    return drive_pattern, phase_offsets


# =========================
# GPIO Setup
# =========================

def initialize_gpio(pins):
    """Configure␣GPIO␣pins␣for␣PRBS␣drive."""
    GPIO.setmode(GPIO.BCM)
    GPIO.setwarnings(False)
    for pin in pins:
        GPIO.setup(pin, GPIO.OUT, initial=GPIO.LOW)


# =========================
# Main Execution
# =========================

def main():

    # Generate PRBS
    prbs_seq = create_prbs_sequence(LFSR_ORDER, LFSR_TAPS[LFSR_ORDER])
    sequence_length = len(prbs_seq)

    drive_pattern, offsets = build_drive_matrix(
        prbs_seq,
        len(GPIO_DRIVE_LINES)
    )

    # Print configuration summary
    print("\n===␣PRBS␣Drive␣Configuration␣===")
    print(f"LFSR␣Order␣␣␣␣␣␣␣:␣{LFSR_ORDER}")
    print(f"Sequence␣Length␣␣␣:␣{sequence_length}")
    print(f"Target␣Bit␣Rate␣␣␣:␣{BIT_RATE_HZ}␣Hz")
    print("Drive␣Line␣Phases:")
    for i, pin in enumerate(GPIO_DRIVE_LINES):
        print(f"␣␣Line␣{i}␣->␣GPIO␣{pin}␣(phase␣offset␣{offsets[i]})")

    # Initialize hardware
    initialize_gpio(GPIO_DRIVE_LINES)

    bit_period = 1.0 / BIT_RATE_HZ
    pattern_index = 0
    bit_counter = 0

    print("\nStarting␣PRBS␣output...␣(Ctrl-C␣to␣stop)\n")
```

```
    start_time = time.perf_counter()
    next_toggle = start_time + bit_period

    try:
        while True:
            # Busy-wait for precise timing
            while time.perf_counter() < next_toggle:
                pass

            next_toggle += bit_period

            GPIO.output(GPIO_DRIVE_LINES, drive_pattern[pattern_index])

            pattern_index = (pattern_index + 1) % sequence_length
            bit_counter += 1

    except KeyboardInterrupt:
        pass

    # Compute actual achieved frequency
    elapsed_time = time.perf_counter() - start_time
    actual_rate = bit_counter / elapsed_time if elapsed_time > 0 else 0

    print("\n=== Run Summary ===")
    print(f"Total Bits Output : {bit_counter}")
    print(f"Elapsed Time       : {elapsed_time:.2f} s")
    print(f"Measured Bit Rate : {actual_rate:.0f} Hz")

    GPIO.cleanup()
    print("GPIO cleaned up. Done.\n")


if __name__ == "__main__":
    main()
```

Part II Code

```
import time
import numpy as np
import RPi.GPIO as GPIO

import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse, Rectangle

import ADS1256
import config

# ============================================================
# Configuration
# ============================================================
```

```
LFSR_ORDER = 7

ADC_GAIN = 2 # PGA gain
ADC_VREF = 3.3 # ADC reference voltage (V)
N_AVG = 4 # average N sweeps per channel per frame
MUX_SETTLE_READS = 4 # toss initial reads after mux switch

GPIO_DRIVE_LINES = [7, 12, 16, 20, 21]
ADC_SENSE_CHANNELS = [1, 2, 3, 4, 5, 6, 7]

N_DRIVE = len(GPIO_DRIVE_LINES)
N_SENSE = len(ADC_SENSE_CHANNELS)

BASELINE_FRAMES = 30
THRESHOLD_SIGMA = 6
BASELINE_UPDATE_ALPHA = 0.02

# Penny / conductive circle noise test (1000-sample centroid jitter)
RUN_PENNY_TEST = True
PENNY_SAMPLES = 1000

# LFSR taps (maximal-length)
LFSR_TAPS = {
    2: [2, 1], 3: [3, 2], 4: [4, 3], 5: [5, 3],
    6: [6, 5], 7: [7, 6], 8: [8, 6, 5, 4], 9: [9, 5],
    10: [10, 7], 11: [11, 9], 12: [12, 11, 10, 4],
    13: [13, 12, 11, 8], 14: [14, 13, 12, 2],
    15: [15, 14], 16: [16, 15, 13, 4],
}

GAIN_CODE = {1: 0, 2: 1, 4: 2, 8: 3, 16: 4, 32: 5, 64: 6}


# ============================================================
# PRBS / Reference Generation
# ============================================================

def create_prbs(order, taps):
    """Generate a maximal-length PRBS from an LFSR (bits in {0,1})."""
    seq_len = (2 ** order) - 1
    reg = [1] * order
    out = []

    for _ in range(seq_len):
        out.append(reg[-1])

        fb = 0
        for tap in taps:
            fb ^= reg[tap - 1]
        reg = [fb] + reg[:-1]

    return np.array(out, dtype=np.int8)
```

```
def make_drive_pattern(prbs_bits, n_lines):
    """Phase-shift␣PRBS␣bits␣across␣drive␣lines␣to␣keep␣codes␣orthogonal."""
    seq_len = len(prbs_bits)
    phase_step = seq_len // n_lines
    phase_offsets = [i * phase_step for i in range(n_lines)]

    pattern = []
    for t in range(seq_len):
        pattern.append([int(prbs_bits[(t + off) % seq_len]) for off in phase_offsets])

    return pattern, phase_offsets


def make_bipolar_refs(prbs_bits, n_lines):
    """Create␣bipolar␣PRBS␣references␣in␣{-1,+1}␣for␣correlation."""
    seq_len = len(prbs_bits)
    phase_step = seq_len // n_lines

    refs = np.zeros((n_lines, seq_len), dtype=np.float64)
    for d in range(n_lines):
        off = d * phase_step
        for t in range(seq_len):
            refs[d, t] = 2.0 * float(prbs_bits[(t + off) % seq_len]) - 1.0
    return refs


# ============================================================
# Hardware / Sampling
# ============================================================

def init_gpio_drive(pins):
    """Initialize␣drive␣GPIO␣lines."""
    GPIO.setmode(GPIO.BCM)
    GPIO.setwarnings(False)
    for pin in pins:
        GPIO.setup(pin, GPIO.OUT, initial=GPIO.LOW)


def init_adc():
    """Initialize␣ADS1256␣and␣set␣gain/buffer␣parameters."""
    adc = ADS1256.ADS1256()
    adc.ADS1256_init()

    config.SPI.max_speed_hz = 3_000_000

    adc.ADS1256_ConfigADC(GAIN_CODE[ADC_GAIN], 0xF0) # data rate param from demo code
    adc.ADS1256_WriteReg(0x00, 0x02) # enable input buffer
    adc.ADS1256_WriteCmd(0xF0) # sync/standby command used in your original
    time.sleep(0.05)

    return adc
```

```
def read_one_sense_sweep(adc, channel, drive_pins, drive_pattern, seq_len):
    """
␣␣␣␣Read␣one␣sense␣channel␣across␣the␣full␣PRBS␣length.
␣␣␣␣We␣set␣the␣NEXT␣drive␣pattern␣before␣reading␣so␣the␣ADC␣pipeline␣aligns.
␣␣␣␣"""
    adc.ADS1256_SetChannal(channel)
    adc.ADS1256_WriteCmd(0xFC) # sync
    adc.ADS1256_WriteCmd(0x00) # wakeup

    lsb_volts = ADC_VREF / (ADC_GAIN * 0x7FFFFF)

    # Prime the ADC + drive with the first pattern, discard settling reads
    GPIO.output(drive_pins, drive_pattern[0])
    for _ in range(MUX_SETTLE_READS):
        adc.ADS1256_Read_ADC_Data()

    accum = np.zeros(seq_len, dtype=np.float64)

    for _ in range(N_AVG):
        for i in range(seq_len):
            GPIO.output(drive_pins, drive_pattern[(i + 1) % seq_len])
            accum[i] += adc.ADS1256_Read_ADC_Data() * lsb_volts

    return accum / float(N_AVG)


def correlate_frame(adc, drive_pins, drive_pattern, refs, seq_len):
    """Scan␣all␣sense␣channels␣and␣compute␣the␣drive/sense␣correlation␣map."""
    corr_map = np.zeros((N_DRIVE, N_SENSE), dtype=np.float64)

    for s_idx, adc_ch in enumerate(ADC_SENSE_CHANNELS):
        samples = read_one_sense_sweep(adc, adc_ch, drive_pins, drive_pattern, seq_len)
        samples = samples - np.mean(samples) # remove DC offset

        # Correlate vs each drive's reference
        for d in range(N_DRIVE):
            corr = 0.0
            for t in range(seq_len):
                corr += refs[d, t] * samples[t]
            corr_map[d, s_idx] = corr / float(seq_len)

    return corr_map


# ============================================================
# Touch Estimation
# ============================================================

def centroid_from_map(weight_map):
    """Compute␣weighted␣centroid␣in␣(sense,␣drive)␣node␣coordinates."""
    w = np.maximum(weight_map, 0.0)
    total = float(w.sum())
    if total < 1e-12:
        return np.nan, np.nan
```

```
    cx = 0.0 # sense axis (S index)
    cy = 0.0 # drive axis (D index)
    for d in range(w.shape[0]):
        for s in range(w.shape[1]):
            cy += w[d, s] * d
            cx += w[d, s] * s

    return cx / total, cy / total


def ellipse_from_second_moments(weight_map, cx, cy):
    """Estimate ellipse major/minor axes + angle via weighted covariance."""
    w = np.maximum(weight_map, 0.0)
    total = float(w.sum())
    if total < 1e-12:
        return 0.0, 0.0, 0.0

    mxx = myy = mxy = 0.0
    for d in range(w.shape[0]):
        for s in range(w.shape[1]):
            dx = (s - cx)
            dy = (d - cy)
            ww = w[d, s]
            mxx += ww * dx * dx
            myy += ww * dy * dy
            mxy += ww * dx * dy

    mxx /= total
    myy /= total
    mxy /= total

    disc = max((mxx - myy) ** 2 / 4.0 + mxy ** 2, 0.0)
    half = (mxx + myy) / 2.0

    major = 2.0 * np.sqrt(max(half + np.sqrt(disc), 0.0))
    minor = 2.0 * np.sqrt(max(half - np.sqrt(disc), 0.0))
    angle = np.degrees(0.5 * np.arctan2(2.0 * mxy, mxx - myy))

    return major, minor, angle


# ============================================================
# Visualization
# ============================================================

def _draw_grid_heatmap(ax, values, title, cmap_name):
    cmap = plt.get_cmap(cmap_name)

    im = ax.pcolormesh(
        np.arange(N_DRIVE + 1),
        np.arange(N_SENSE + 1),
        values.T,
        cmap=cmap,
```

```
        edgecolors="#444444",
        linewidth=1.2,
    )

    ax.set_xticks(np.arange(N_DRIVE) + 0.5)
    ax.set_xticklabels([f"D{d}" for d in range(N_DRIVE)], fontsize=9)

    ax.set_yticks(np.arange(N_SENSE) + 0.5)
    ax.set_yticklabels([f"S{s}" for s in range(N_SENSE)], fontsize=9)

    ax.set_title(title, fontsize=11, fontweight="bold")
    ax.set_aspect("equal")
    ax.tick_params(length=0)

    vmin, vmax = float(values.min()), float(values.max())
    norm = plt.Normalize(vmin, vmax)

    # annotate cell values
    for d in range(N_DRIVE):
        for s in range(N_SENSE):
            val = float(values[d, s])
            rgba = cmap(norm(val))
            brightness = 0.299 * rgba[0] + 0.587 * rgba[1] + 0.114 * rgba[2]
            txt_color = "white" if brightness < 0.5 else "black"
            ax.text(d + 0.5, s + 0.5, f"{val:.2f}", ha="center", va="center",
                    fontsize=6.5, color=txt_color, fontweight="bold")

    return im


def save_frame_plot(raw_corr, delta_corr, frame_idx, fps, cx, cy,
                    major, minor, angle, touch_mask, out_path="touchmap.png"):
    """Save␣a␣side-by-side␣plot:␣raw␣correlation␣and␣|delta|␣heatmap."""
    fig, (ax_raw, ax_delta) = plt.subplots(1, 2, figsize=(16, N_SENSE + 1))

    im1 = _draw_grid_heatmap(ax_raw, raw_corr * 1000.0, "Raw␣Correlator␣(mV)", "YlGnBu")
    fig.colorbar(im1, ax=ax_raw, fraction=0.046, pad=0.04, format="%.2f")

    im2 = _draw_grid_heatmap(ax_delta, np.abs(delta_corr) * 1000.0, "|Delta|␣from␣
        Baseline␣(mV)", "YlOrRd")
    fig.colorbar(im2, ax=ax_delta, fraction=0.046, pad=0.04, format="%.2f")

    if touch_mask is not None:
        for d in range(N_DRIVE):
            for s in range(N_SENSE):
                if touch_mask[d, s]:
                    ax_delta.add_patch(Rectangle((d, s), 1, 1, fill=False,
                                                 edgecolor="cyan", linewidth=2.5))

    if not np.isnan(cx):
        ax_delta.plot(cy + 0.5, cx + 0.5, "c+", ms=16, mew=3)
        if major > 1e-2:
            ax_delta.add_patch(Ellipse((cy + 0.5, cx + 0.5),
                                       width=major, height=minor, angle=angle,
```

```
                                    fill=False, edgecolor="cyan", linewidth=2))

    status = "NO␣TOUCH" if np.isnan(cx) else f"TOUCH␣@␣(S={cx:.2f},␣D={cy:.2f})"
    fig.suptitle(f"Frame␣{frame_idx}␣|␣{fps:.1f}␣fps␣|␣{status}", fontsize=14, fontweight
        ="bold")

    fig.tight_layout(rect=[0, 0, 1, 0.93])
    fig.savefig(out_path, dpi=130, bbox_inches="tight", facecolor="white")
    plt.close(fig)


def save_centroid_noise_plots(samples, out_path="centroid_scatter.png"):
    """Save␣scatter␣+␣hist␣+␣time-series␣plots␣for␣centroid␣noise."""
    arr = np.array(samples, dtype=np.float64)
    cx_all = arr[:, 0]
    cy_all = arr[:, 1]
    maj_all = arr[:, 2]
    min_all = arr[:, 3]

    sx = float(np.std(cx_all))
    sy = float(np.std(cy_all))
    rms = float(np.sqrt(sx * sx + sy * sy))

    fig, axes = plt.subplots(1, 3, figsize=(16, 5))

    axes[0].plot(cx_all, cy_all, "b.", ms=2, alpha=0.4)
    axes[0].set_xlabel("Sense␣position␣(nodes)")
    axes[0].set_ylabel("Drive␣position␣(nodes)")
    axes[0].set_title(f"Centroid␣scatter␣(RMS={rms:.4f})")
    axes[0].set_aspect("equal")
    axes[0].grid(True, alpha=0.3)

    axes[1].hist(cx_all, bins=40, alpha=0.7, label=f"sx={sx:.4f}")
    axes[1].hist(cy_all, bins=40, alpha=0.7, label=f"sy={sy:.4f}")
    axes[1].set_xlabel("Position␣(nodes)")
    axes[1].set_ylabel("Count")
    axes[1].set_title("Centroid␣distribution")
    axes[1].legend()
    axes[1].grid(True, alpha=0.3)

    axes[2].plot(cx_all, label="Sense␣(x)", alpha=0.7)
    axes[2].plot(cy_all, label="Drive␣(y)", alpha=0.7)
    axes[2].set_xlabel("Sample")
    axes[2].set_ylabel("Position␣(nodes)")
    axes[2].set_title("Centroid␣time␣series")
    axes[2].legend()
    axes[2].grid(True, alpha=0.3)

    fig.tight_layout()
    fig.savefig(out_path, dpi=150, facecolor="white")
    plt.close(fig)

    return sx, sy, rms, float(np.mean(maj_all)), float(np.std(maj_all)), float(np.mean(
        min_all)), float(np.std(min_all))
```

```
# ============================================================
# Main Loop
# ============================================================

def main():
    # Build PRBS and drive references
    prbs_bits = create_prbs(LFSR_ORDER, LFSR_TAPS[LFSR_ORDER])
    seq_len = len(prbs_bits)

    drive_pattern, phase_offsets = make_drive_pattern(prbs_bits, N_DRIVE)
    refs = make_bipolar_refs(prbs_bits, N_DRIVE)

    # Estimate work per frame (rough)
    est_reads_per_frame = N_SENSE * (seq_len * N_AVG + MUX_SETTLE_READS)

    print("\n=== Part II: Touch Sensing (Correlator + Centroid) ===")
    print(f"PRBS              : order={LFSR_ORDER}, length={seq_len}")
    print(f"ADC               : gain={ADC_GAIN}x, vref={ADC_VREF} V, avg={N_AVG}")
    print(f"Sense channels    : {ADC_SENSE_CHANNELS}")
    print(f"Drive GPIO        : {GPIO_DRIVE_LINES}")
    print(f"Est ADC reads/frame: {est_reads_per_frame}")
    print("Drive phase offsets:")
    for i, pin in enumerate(GPIO_DRIVE_LINES):
        print(f"  D{i}: GPIO {pin} (offset {phase_offsets[i]})")

    if RUN_PENNY_TEST:
        print("\n--- Penny Noise Test Enabled ---")
        print(f"Will record {PENNY_SAMPLES} centroid samples after baseline capture.")
        print("After baseline locks: place conductive object and keep it still.\n")

    # Hardware init
    init_gpio_drive(GPIO_DRIVE_LINES)
    adc = init_adc()

    baseline = None
    baseline_stack = []
    noise_sigma = None
    threshold = None

    frame_idx = 0
    centroid_log = []

    print(f"Baseline capture: {BASELINE_FRAMES} frames (keep fingers off)...")

    try:
        while True:
            t0 = time.perf_counter()

            corr_map = correlate_frame(adc, GPIO_DRIVE_LINES, drive_pattern, refs, seq_len
                )

            t1 = time.perf_counter()
```

```
dt = t1 - t0
fps = (1.0 / dt) if dt > 0 else 0.0

# ----- Baseline phase -----
if baseline is None:
    baseline_stack.append(corr_map.copy())
    frame_idx += 1
    print(f"\r  baseline {frame_idx}/{BASELINE_FRAMES}  ({fps:.1f} fps)",
          end="", flush=True)

    if frame_idx >= BASELINE_FRAMES:
        stack = np.array(baseline_stack)
        baseline = np.mean(stack, axis=0)
        noise_sigma = np.std(stack, axis=0)

        threshold = THRESHOLD_SIGMA * noise_sigma
        threshold = np.maximum(threshold, 1e-6)

        baseline_stack = None
        print("\nBaseline locked.")
        print(f"  mean noise sigma: {noise_sigma.mean():.6f}")
        print("Running live touch tracking... (Ctrl-C to stop)\n")

    continue

# ----- Touch detect -----
delta = corr_map - baseline
abs_delta = np.abs(delta)

touch_mask = abs_delta > threshold
has_touch = bool(touch_mask.any())

cx = cy = np.nan
major = minor = angle = 0.0

if has_touch:
    gated = abs_delta.copy()
    gated[~touch_mask] = 0.0

    cx, cy = centroid_from_map(gated)
    if not np.isnan(cx):
        major, minor, angle = ellipse_from_second_moments(gated, cx, cy)
        centroid_log.append((cx, cy, major, minor, angle))
else:
    # slowly adapt baseline when no touch
    baseline = (1.0 - BASELINE_UPDATE_ALPHA) * baseline + BASELINE_UPDATE_ALPHA
         * corr_map

# ----- Status output -----
if RUN_PENNY_TEST:
    if has_touch:
        print(f"\r  penny samples: {len(centroid_log)}/{PENNY_SAMPLES} "
              f"| pos=(S={cx:.2f},D={cy:.2f}) | nodes={int(touch_mask.sum())}",
              end="     ", flush=True)
```

```
            else:
                print(f"\r␣␣penny␣samples:␣{len(centroid_log)}/{PENNY_SAMPLES}␣"
                      f"|␣waiting␣for␣touch...",
                      end="␣␣␣␣␣", flush=True)
        else:
            if has_touch:
                print(f"\rFrame␣{frame_idx:4d}␣|␣{fps:4.1f}␣fps␣|␣TOUCH␣"
                      f"(S={cx:.2f},D={cy:.2f})␣|␣maj={major:.2f}␣min={minor:.2f}␣|␣
                          nodes={int(touch_mask.sum())}",
                      end="␣␣␣␣␣", flush=True)
            else:
                print(f"\rFrame␣{frame_idx:4d}␣|␣{fps:4.1f}␣fps␣|␣no␣touch",
                      end="␣␣␣␣␣", flush=True)

        # Save heatmap each frame (skip in penny test to maximize sample rate)
        if not RUN_PENNY_TEST:
            save_frame_plot(corr_map, delta, frame_idx, fps, cx, cy,
                            major, minor, angle, touch_mask,
                            out_path="touchmap.png")

        frame_idx += 1

        # Stop after collecting enough penny samples
        if RUN_PENNY_TEST and len(centroid_log) >= PENNY_SAMPLES:
            print(f"\n\nCollected␣{PENNY_SAMPLES}␣samples.")
            break

except KeyboardInterrupt:
    print("\n\nStopped␣by␣user.")

# ----- Cleanup -----
for pin in GPIO_DRIVE_LINES:
    GPIO.output(pin, GPIO.LOW)
GPIO.cleanup()

# ----- Noise summary (if enough samples) -----
if len(centroid_log) >= 20:
    arr = np.array(centroid_log, dtype=np.float64)
    cx_all = arr[:, 0]
    cy_all = arr[:, 1]

    sx = float(np.std(cx_all))
    sy = float(np.std(cy_all))
    rms = float(np.sqrt(sx * sx + sy * sy))

    print("\n===␣Centroid␣Noise␣Summary␣===")
    print(f"Samples␣␣␣␣␣␣␣␣␣␣␣:␣{len(arr)}")
    print(f"Mean␣position␣␣␣␣␣:␣(S={np.mean(cx_all):.4f},␣D={np.mean(cy_all):.4f})")
    print(f"Sigma␣(sense)␣␣␣␣␣:␣{sx:.6f}␣nodes")
    print(f"Sigma␣(drive)␣␣␣␣␣:␣{sy:.6f}␣nodes")
    print(f"RMS␣jitter␣␣␣␣␣␣␣␣:␣{rms:.6f}␣nodes")
    print(f"Peak-to-peak␣(S)␣␣:␣{np.ptp(cx_all):.4f}␣nodes")
    print(f"Peak-to-peak␣(D)␣␣:␣{np.ptp(cy_all):.4f}␣nodes")
```

```
        maj_mu = float(np.mean(arr[:, 2]))
        maj_sd = float(np.std(arr[:, 2]))
        min_mu = float(np.mean(arr[:, 3]))
        min_sd = float(np.std(arr[:, 3]))
        print(f"Major␣axis␣␣␣␣␣␣␣␣:␣{maj_mu:.4f}␣+/-␣{maj_sd:.4f}")
        print(f"Minor␣axis␣␣␣␣␣␣␣␣:␣{min_mu:.4f}␣+/-␣{min_sd:.4f}")

        if RUN_PENNY_TEST:
            print(f"Estimated␣resolution:␣~{rms:.4f}␣nodes")

        save_centroid_noise_plots(centroid_log, out_path="centroid_scatter.png")
        print("Saved␣centroid_scatter.png")

    print("\nDone.\n")


if __name__ == "__main__":
    main()
```

Part III Code

```
import time
import numpy as np

import RPi.GPIO as GPIO

import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

import ADS1256
import config

# ------------------------------
# Configuration (same as Part 2)
# ------------------------------
PRBS_ORDER = 7

ADC_GAIN = 2
ADC_VREF = 3.3
NUM_AVG = 4
SETTLE_THROWAWAYS = 4

DRIVE_GPIO = [7, 12, 16, 20, 21]
SENSE_ADC_CH = [1, 2, 3, 4, 5, 6, 7]

N_DRIVE = len(DRIVE_GPIO)
N_SENSE = len(SENSE_ADC_CH)

BASELINE_FRAMES = 30
THRESH_SIGMA = 6
BASELINE_LP_ALPHA = 0.02
```

```
TRAIL_LEN = 80

# LFSR taps
LFSR_TAPS = {
    2: [2, 1], 3: [3, 2], 4: [4, 3], 5: [5, 3],
    6: [6, 5], 7: [7, 6], 8: [8, 6, 5, 4], 9: [9, 5],
    10: [10, 7], 11: [11, 9], 12: [12, 11, 10, 4],
    13: [13, 12, 11, 8], 14: [14, 13, 12, 2],
    15: [15, 14], 16: [16, 15, 13, 4],
}

GAIN_CODE = {1: 0, 2: 1, 4: 2, 8: 3, 16: 4, 32: 5, 64: 6}

# Penny-derived measurement noise (you can update if you want)
KF_R = np.diag([0.17**2, 0.13**2]) # [sense(x), drive(y)] std dev in nodes
KF_ACCEL_STD = 2.0 # tuning knob (nodes/s^2)


# ------------------------------
# PRBS / reference building
# ------------------------------
def prbs_lfsr(order_n: int, taps_1idx: list[int]) -> np.ndarray:
    """Generate␣one␣full␣PRBS␣period␣(0/1)␣using␣an␣LFSR␣with␣given␣taps␣(1-indexed)."""
    period = (2**order_n) - 1
    reg = [1] * order_n
    seq = np.empty(period, dtype=np.int8)

    for i in range(period):
        seq[i] = reg[-1]
        fb = 0
        for t in taps_1idx:
            fb ^= reg[t - 1]
        reg = [fb] + reg[:-1]
    return seq


def build_drive_pattern(bits01: np.ndarray, n_drive: int):
    """Return␣GPIO␣output␣pattern␣rows␣(period␣x␣n_drive)␣and␣per-drive␣phase␣offsets."""
    period = len(bits01)
    step = period // n_drive
    offsets = [d * step for d in range(n_drive)]

    pattern = []
    for k in range(period):
        pattern.append([int(bits01[(k + off) % period]) for off in offsets])
    return pattern, offsets


def build_corr_refs(bits01: np.ndarray, n_drive: int) -> np.ndarray:
    """Return␣+/-1␣references␣for␣correlation,␣shape:␣(n_drive,␣period)."""
    period = len(bits01)
    step = period // n_drive
    refs = np.zeros((n_drive, period), dtype=np.float64)
```

```
    for d in range(n_drive):
        off = d * step
        # map 0->-1, 1->+1
        refs[d, :] = 2.0 * bits01[(np.arange(period) + off) % period].astype(np.float64) -
             1.0
    return refs


# ----------------------------
# Hardware / ADC acquisition
# ----------------------------
def init_hw():
    GPIO.setmode(GPIO.BCM)
    GPIO.setwarnings(False)
    for p in DRIVE_GPIO:
        GPIO.setup(p, GPIO.OUT, initial=GPIO.LOW)

    adc = ADS1256.ADS1256()
    adc.ADS1256_init()
    config.SPI.max_speed_hz = 3_000_000

    adc.ADS1256_ConfigADC(GAIN_CODE[ADC_GAIN], 0xF0)
    adc.ADS1256_WriteReg(0x00, 0x02) # enable input buffer
    adc.ADS1256_WriteCmd(0xF0) # sync
    time.sleep(0.05)
    return adc


def sample_prbs_channel(adc, adc_channel: int, pattern_rows, period: int) -> np.ndarray:
    """
␣␣␣␣Read␣one␣full␣PRBS␣period␣worth␣of␣samples␣for␣one␣sense␣channel.
␣␣␣␣We␣set␣NEXT␣pattern␣then␣read␣(pipeline␣alignment).
␣␣␣␣"""
    adc.ADS1256_SetChannal(adc_channel)
    adc.ADS1256_WriteCmd(0xFC) # sync-ish
    adc.ADS1256_WriteCmd(0x00)

    scale = ADC_VREF / (ADC_GAIN * 0x7FFFFF)

    GPIO.output(DRIVE_GPIO, pattern_rows[0])
    for _ in range(SETTLE_THROWAWAYS):
        adc.ADS1256_Read_ADC_Data()

    accum = np.zeros(period, dtype=np.float64)
    for _ in range(NUM_AVG):
        for i in range(period):
            GPIO.output(DRIVE_GPIO, pattern_rows[(i + 1) % period])
            accum[i] += adc.ADS1256_Read_ADC_Data() * scale
    return accum / float(NUM_AVG)


def correlate_frame(adc, pattern_rows, refs_pm1: np.ndarray, period: int) -> np.ndarray:
    """Compute␣the␣(drive␣x␣sense)␣correlator␣matrix␣for␣this␣frame."""
```

```
    cap = np.zeros((N_DRIVE, N_SENSE), dtype=np.float64)

    for s_idx, ch in enumerate(SENSE_ADC_CH):
        wave = sample_prbs_channel(adc, ch, pattern_rows, period)
        wave = wave - np.mean(wave) # remove DC bias

        # correlation per drive
        # (explicit loop to keep same behavior as your code)
        for d in range(N_DRIVE):
            cap[d, s_idx] = float(np.dot(refs_pm1[d, :], wave)) / float(period)

    return cap


# ----------------------------
# Touch processing (centroid)
# ----------------------------
def centroid_from_weights(weight_mat: np.ndarray):
    """Return␣(cx_sense,␣cy_drive)␣in␣node␣coordinates␣or␣(nan,␣nan)."""
    w = np.maximum(weight_mat, 0.0)
    total = float(np.sum(w))
    if total < 1e-12:
        return np.nan, np.nan

    # cx over sense index, cy over drive index
    cx = 0.0
    cy = 0.0
    for d in range(w.shape[0]):
        for s in range(w.shape[1]):
            cy += w[d, s] * d
            cx += w[d, s] * s
    return cx / total, cy / total


# ----------------------------
# Kalman filter (2D constant velocity)
# State: [x_sense, y_drive, vx, vy]
# ----------------------------
class KalmanCV2D:
    def __init__(self, r_meas: np.ndarray, accel_std: float):
        self.x = np.zeros(4, dtype=np.float64)
        self.P = np.eye(4, dtype=np.float64) * 10.0
        self.r = r_meas.astype(np.float64).copy()
        self.accel_std = float(accel_std)
        self.ready = False

    def reset(self):
        self.x[:] = 0.0
        self.P[:, :] = np.eye(4) * 10.0
        self.ready = False

    def start(self, mx: float, my: float):
        self.x[:] = np.array([mx, my, 0.0, 0.0], dtype=np.float64)
        self.P[:, :] = np.diag([0.04, 0.04, 1.0, 1.0])
```

```
        self.ready = True

    def predict(self, dt: float):
        F = np.array([
            [1, 0, dt, 0],
            [0, 1, 0, dt],
            [0, 0, 1, 0],
            [0, 0, 0, 1],
        ], dtype=np.float64)

        q = (self.accel_std ** 2)
        Q = q * np.array([
            [dt**3/3, 0, dt**2/2, 0 ],
            [0, dt**3/3, 0, dt**2/2],
            [dt**2/2, 0, dt, 0 ],
            [0, dt**2/2, 0, dt ],
        ], dtype=np.float64)

        self.x = F @ self.x
        self.P = F @ self.P @ F.T + Q

    def update(self, mx: float, my: float):
        H = np.array([
            [1, 0, 0, 0],
            [0, 1, 0, 0],
        ], dtype=np.float64)

        z = np.array([mx, my], dtype=np.float64)
        innov = z - H @ self.x
        S = H @ self.P @ H.T + self.r
        K = self.P @ H.T @ np.linalg.inv(S)

        self.x = self.x + K @ innov

        I = np.eye(4, dtype=np.float64)
        I_KH = I - K @ H
        # Joseph form for numerical stability
        self.P = I_KH @ self.P @ I_KH.T + K @ self.r @ K.T


# ------------------------------
# Plotting
# ------------------------------
def draw_heatmap(ax, mat, nd, ns, cmap_name, title):
    cmap = plt.get_cmap(cmap_name)
    im = ax.pcolormesh(
        np.arange(nd + 1),
        np.arange(ns + 1),
        mat.T,
        cmap=cmap,
        edgecolors="#444444",
        linewidth=1.2,
    )
    ax.set_xticks(np.arange(nd) + 0.5)
```

```
    ax.set_xticklabels([f"D{d}" for d in range(nd)], fontsize=9)
    ax.set_yticks(np.arange(ns) + 0.5)
    ax.set_yticklabels([f"S{s}" for s in range(ns)], fontsize=9)
    ax.set_title(title, fontsize=11, fontweight="bold")
    ax.set_aspect("equal")
    ax.tick_params(length=0)

    vmin, vmax = float(mat.min()), float(mat.max())
    norm = plt.Normalize(vmin, vmax)
    for d in range(nd):
        for s in range(ns):
            val = float(mat[d, s])
            rgba = cmap(norm(val))
            brightness = 0.299*rgba[0] + 0.587*rgba[1] + 0.114*rgba[2]
            tc = "white" if brightness < 0.5 else "black"
            ax.text(d + 0.5, s + 0.5, f"{val:.2f}",
                    ha="center", va="center",
                    fontsize=6.5, color=tc, fontweight="bold")
    return im


def save_live_frame(cap, delta, frame_idx, fps,
                    raw_xy, touch_mask,
                    trail_raw, trail_kf, vel_hist,
                    kf: KalmanCV2D):
    nd, ns = cap.shape[0], cap.shape[1]
    fig, (ax_hm, ax_tr, ax_v) = plt.subplots(1, 3, figsize=(22, ns + 2))

    # (1) heatmap of |delta|
    im = draw_heatmap(ax_hm, np.abs(delta) * 1000.0, nd, ns, "YlOrRd", "|| from baseline (mV)")
    fig.colorbar(im, ax=ax_hm, fraction=0.046, pad=0.04, format="%.2f")

    if touch_mask is not None:
        for d in range(nd):
            for s in range(ns):
                if bool(touch_mask[d, s]):
                    ax_hm.add_patch(Rectangle((d, s), 1, 1, fill=False,
                                              edgecolor="cyan", linewidth=2))

    cx, cy = raw_xy
    if not np.isnan(cx):
        ax_hm.plot(cy + 0.5, cx + 0.5, "c+", ms=14, mew=2, label="Raw")

    if kf.ready:
        fx, fy = float(kf.x[0]), float(kf.x[1])
        ax_hm.plot(fy + 0.5, fx + 0.5, "mo", ms=10, mew=2, fillstyle="none", label="Kalman")

    if ax_hm.get_legend_handles_labels()[1]:
        ax_hm.legend(fontsize=7, loc="upper right")

    # (2) trail view
    ax_tr.set_xlim(-0.5, nd - 0.5)
```

```
ax_tr.set_ylim(-0.5, ns - 0.5)
ax_tr.set_xlabel("Drive")
ax_tr.set_ylabel("Sense")
ax_tr.set_title("Touch␣Trail", fontsize=11, fontweight="bold")
ax_tr.set_aspect("equal")

for d in range(nd + 1):
    ax_tr.axvline(d - 0.5, color="#ddd", linewidth=0.5)
for s in range(ns + 1):
    ax_tr.axhline(s - 0.5, color="#ddd", linewidth=0.5)

if trail_raw:
    tr = np.array(trail_raw)
    ax_tr.plot(tr[:, 1], tr[:, 0], ".", color="#bbb", ms=3, alpha=0.5, label="Raw")

if trail_kf:
    tf = np.array(trail_kf)
    for i in range(1, len(tf)):
        a = 0.25 + 0.75 * (i / len(tf))
        ax_tr.plot(tf[i-1:i+1, 1], tf[i-1:i+1, 0], "-", color="blue", linewidth=2,
            alpha=a)
    ax_tr.plot(tf[-1, 1], tf[-1, 0], "bo", ms=8, label="Kalman")

if ax_tr.get_legend_handles_labels()[1]:
    ax_tr.legend(fontsize=7, loc="upper␣right")

# (3) velocity history
ax_v.set_title("Velocity", fontsize=11, fontweight="bold")
ax_v.set_xlabel("Frame")
ax_v.set_ylabel("nodes/s")
ax_v.grid(True, alpha=0.3)

if vel_hist:
    vh = np.array(vel_hist)
    t = np.arange(len(vh))
    ax_v.plot(t, vh[:, 0], label="Vx␣(sense)", alpha=0.8)
    ax_v.plot(t, vh[:, 1], label="Vy␣(drive)", alpha=0.8)
    speed = np.sqrt(vh[:, 0]**2 + vh[:, 1]**2)
    ax_v.plot(t, speed, "k--", label="Speed", alpha=0.6)
    ax_v.legend(fontsize=7)

# title
if kf.ready:
    title = (f"Frame␣{frame_idx}␣|␣{fps:.1f}␣fps␣|␣"
             f"pos=(S={kf.x[0]:.2f},␣D={kf.x[1]:.2f})␣"
             f"vel=({kf.x[2]:.1f},␣{kf.x[3]:.1f})")
else:
    title = f"Frame␣{frame_idx}␣|␣{fps:.1f}␣fps␣|␣No␣touch"

fig.suptitle(title, fontsize=12, fontweight="bold")
fig.subplots_adjust(left=0.04, right=0.97, bottom=0.08, top=0.88, wspace=0.35)
fig.savefig("tracking.png", dpi=120, facecolor="white")
plt.close(fig)
```

```
def save_summary_report(trail_raw, trail_kf, vel_hist, fps_last):
    vh = np.array(vel_hist, dtype=np.float64)
    speeds = np.sqrt(vh[:, 0]**2 + vh[:, 1]**2)

    fig = plt.figure(figsize=(16, 14))

    ax1 = fig.add_subplot(2, 2, 1)
    ax1.set_xlim(-0.5, N_DRIVE - 0.5)
    ax1.set_ylim(-0.5, N_SENSE - 0.5)
    ax1.set_xlabel("Drive␣axis␣(nodes)")
    ax1.set_ylabel("Sense␣axis␣(nodes)")
    ax1.set_title("Touch␣Trail:␣Raw␣vs␣Kalman-Filtered", fontweight="bold")
    ax1.set_aspect("equal")

    for d in range(N_DRIVE + 1):
        ax1.axvline(d - 0.5, color="#ddd", linewidth=0.5)
    for s in range(N_SENSE + 1):
        ax1.axhline(s - 0.5, color="#ddd", linewidth=0.5)

    if trail_raw:
        tr = np.array(trail_raw)
        ax1.plot(tr[:, 1], tr[:, 0], ".", color="#aaa", ms=5, alpha=0.6, label="Raw␣
            centroid")

    if trail_kf:
        tf = np.array(trail_kf)
        for i in range(1, len(tf)):
            a = 0.25 + 0.75 * (i / len(tf))
            ax1.plot(tf[i-1:i+1, 1], tf[i-1:i+1, 0], "-", color="blue", linewidth=2.5,
                alpha=a)
        ax1.plot(tf[0, 1], tf[0, 0], "gs", ms=10, label="Start")
        ax1.plot(tf[-1, 1], tf[-1, 0], "rs", ms=10, label="End")
        ax1.plot([], [], "-", color="blue", linewidth=2.5, label="Kalman␣filtered")

    ax1.legend(fontsize=9)
    ax1.grid(True, alpha=0.15)

    ax2 = fig.add_subplot(2, 2, 2)
    ax2.set_title("Touch␣Velocity␣Over␣Time", fontweight="bold")
    ax2.set_xlabel("Frame")
    ax2.set_ylabel("Velocity␣(nodes/s)")
    ax2.grid(True, alpha=0.3)

    t = np.arange(len(vh))
    ax2.plot(t, vh[:, 0], "b-", linewidth=1.5, alpha=0.8,
            label=f"Vx␣mean={np.mean(vh[:,0]):.2f}")
    ax2.plot(t, vh[:, 1], color="orange", linewidth=1.5, alpha=0.8,
            label=f"Vy␣mean={np.mean(vh[:,1]):.2f}")
    ax2.plot(t, speeds, "k--", linewidth=1.5, alpha=0.6,
            label=f"Speed␣mean={np.mean(speeds):.2f}␣max={np.max(speeds):.2f}")
    ax2.axhline(0, color="gray", linewidth=0.5)
    ax2.legend(fontsize=8)
```

```
    ax3 = fig.add_subplot(2, 1, 2)
    ax3.axis("off")

    n_frames = len(vh)
    text = (
        "Kalman␣Filter␣Design\n\n"
        "We␣use␣a␣2D␣constant-velocity␣Kalman␣filter␣to␣smooth␣the␣raw␣touch␣centroid\n"
        "and␣estimate␣velocity.\n\n"
        "State:␣␣␣␣␣␣␣x␣=␣[px,␣py,␣vx,␣vy]\n"
        "Measurement:␣z␣=␣[px,␣py]\n\n"
        "Measurement␣noise␣R␣(from␣conductive␣test):\n"
        "␣␣sx␣=␣0.17␣nodes␣(sense),␣sy␣=␣0.13␣nodes␣(drive)\n\n"
        f"Process␣noise␣(accel␣std):␣{KF_ACCEL_STD:.2f}␣nodes/s^2\n"
        "␣␣Higher␣->␣less␣lag,␣noisier\n"
        "␣␣Lower␣␣->␣smoother,␣more␣lag\n\n"
        "Touch␣handling:\n"
        "␣␣touch-down:␣initialize␣at␣raw␣centroid\n"
        "␣␣touch:␣␣␣␣␣␣predict␣+␣update␣each␣frame\n"
        "␣␣touch-up:␣␣␣coast␣for␣5␣frames␣then␣reset\n\n"
        f"Results␣({n_frames}␣frames,␣~{fps_last:.1f}␣fps)\n"
        f"␣␣mean␣speed:␣{np.mean(speeds):.2f}␣nodes/s\n"
        f"␣␣max␣speed␣:␣{np.max(speeds):.2f}␣nodes/s\n"
        f"␣␣mean␣vx␣␣␣:␣{np.mean(vh[:,0]):.2f}␣nodes/s\n"
        f"␣␣mean␣vy␣␣␣:␣{np.mean(vh[:,1]):.2f}␣nodes/s\n"
    )

    ax3.text(0.02, 0.98, text, transform=ax3.transAxes, fontsize=10.5,
            fontfamily="monospace", va="top",
            bbox=dict(boxstyle="round,pad=0.5", facecolor="#f8f8f8", edgecolor="#cccccc")
                )

    fig.suptitle("Part␣III␣-␣Kalman-Filtered␣Touch␣Tracking", fontsize=15, fontweight="
        bold")
    fig.tight_layout(rect=[0, 0, 1, 0.95])
    fig.savefig("kalman_report.png", dpi=150, facecolor="white")
    plt.close(fig)

    print("\n[Saved]␣kalman_report.png")


# ------------------------------
# Main
# ------------------------------
def main():
    bits = prbs_lfsr(PRBS_ORDER, LFSR_TAPS[PRBS_ORDER])
    period = len(bits)
    gpio_pattern, phase_offsets = build_drive_pattern(bits, N_DRIVE)
    corr_refs = build_corr_refs(bits, N_DRIVE)

    # header
    print("\n===␣Part␣III:␣Kalman-Filtered␣Touch␣Tracking␣===")
    print(f"PRBS:␣order={PRBS_ORDER},␣length={period}")
    print(f"ADC␣:␣gain={ADC_GAIN}x,␣vref={ADC_VREF:.1f}␣V,␣avg={NUM_AVG}")
    print(f"GPIO␣drives:␣{DRIVE_GPIO}")
```

```
print(f"Sense␣ADC␣ch:␣{SENSE_ADC_CH}")
print("Drive␣phase␣offsets:")
for d in range(N_DRIVE):
    print(f"␣␣D{d}:␣GPIO␣{DRIVE_GPIO[d]}␣(offset␣{phase_offsets[d]})")
print(f"Kalman:␣R=diag([{np.sqrt(KF_R[0,0]):.3f}^2,␣{np.sqrt(KF_R[1,1]):.3f}^2]),␣
    accel_std={KF_ACCEL_STD:.2f}")
print(f"Baseline:␣{BASELINE_FRAMES}␣frames,␣threshold={THRESH_SIGMA}\n")

adc = init_hw()
kf = KalmanCV2D(KF_R, KF_ACCEL_STD)

baseline = None
baseline_stack = []
frame_idx = 0
no_touch_frames = 0

trail_raw = []
trail_kf = []
vel_hist = []

print("Capturing␣baseline...␣keep␣hands␣off␣the␣sensor.")

try:
    while True:
        t0 = time.perf_counter()
        cap = correlate_frame(adc, gpio_pattern, corr_refs, period)
        t1 = time.perf_counter()

        dt = max(t1 - t0, 1e-9)
        fps = 1.0 / dt

        # baseline capture
        if baseline is None:
            baseline_stack.append(cap.copy())
            frame_idx += 1
            print(f"\r␣␣baseline␣frame␣{frame_idx}/{BASELINE_FRAMES}", end="", flush=
                True)
            if frame_idx >= BASELINE_FRAMES:
                stacked = np.array(baseline_stack)
                baseline = np.mean(stacked, axis=0)
                noise = np.std(stacked, axis=0)
                thresh = THRESH_SIGMA * noise
                thresh = np.maximum(thresh, 1e-6)
                baseline_stack = None
                print(f"\nBaseline␣locked.␣~{fps:.1f}␣fps")
                print("Now␣slide␣your␣finger␣slowly.␣Ctrl-C␣to␣stop.\n")
            continue

        delta = cap - baseline
        abs_delta = np.abs(delta)

        touch_mask = abs_delta > thresh
        has_touch = bool(np.any(touch_mask))
```

```
raw_cx = np.nan
raw_cy = np.nan

if has_touch:
    masked = abs_delta.copy()
    masked[~touch_mask] = 0.0
    raw_cx, raw_cy = centroid_from_weights(masked)
    no_touch_frames = 0
else:
    no_touch_frames += 1
    # slow baseline tracking when idle
    baseline = (1 - BASELINE_LP_ALPHA) * baseline + BASELINE_LP_ALPHA * cap

# Kalman lifecycle
if has_touch and not np.isnan(raw_cx):
    if not kf.ready:
        kf.start(raw_cx, raw_cy)
    else:
        kf.predict(dt)
        kf.update(raw_cx, raw_cy)

    trail_raw.append((raw_cx, raw_cy))
    trail_kf.append((kf.x[0], kf.x[1]))
    vel_hist.append((kf.x[2], kf.x[3]))

    if len(trail_raw) > TRAIL_LEN:
        trail_raw = trail_raw[-TRAIL_LEN:]
    if len(trail_kf) > TRAIL_LEN:
        trail_kf = trail_kf[-TRAIL_LEN:]

elif kf.ready:
    # coast briefly after touch-up
    if no_touch_frames < 5:
        kf.predict(dt)
        trail_kf.append((kf.x[0], kf.x[1]))
        vel_hist.append((kf.x[2], kf.x[3]))
        if len(trail_kf) > TRAIL_LEN:
            trail_kf = trail_kf[-TRAIL_LEN:]
    else:
        kf.reset()
        trail_raw = []
        trail_kf = []

# status line
if kf.ready:
    print(f"\rFrame␣{frame_idx:4d}␣|␣{fps:4.1f}␣fps␣|␣"
          f"KF␣pos=(S={kf.x[0]:.2f},␣D={kf.x[1]:.2f})␣"
          f"vel=({kf.x[2]:+.2f},␣{kf.x[3]:+.2f})",
          end="␣␣␣", flush=True)
else:
    print(f"\rFrame␣{frame_idx:4d}␣|␣{fps:4.1f}␣fps␣|␣no␣touch",
          end="␣␣␣", flush=True)

save_live_frame(
```

```
                cap=cap,
                delta=delta,
                frame_idx=frame_idx,
                fps=fps,
                raw_xy=(raw_cx, raw_cy),
                touch_mask=touch_mask,
                trail_raw=trail_raw,
                trail_kf=trail_kf,
                vel_hist=vel_hist,
                kf=kf,
            )

            frame_idx += 1

    except KeyboardInterrupt:
        print("\n\nStopped by user.")

    finally:
        for p in DRIVE_GPIO:
            GPIO.output(p, GPIO.LOW)
        GPIO.cleanup()

    if vel_hist:
        vh = np.array(vel_hist, dtype=np.float64)
        speed = np.sqrt(vh[:, 0]**2 + vh[:, 1]**2)

        print("\n=== Velocity Summary ===")
        print(f"Frames: {len(vh)}")
        print(f"Mean speed : {np.mean(speed):.2f} nodes/s")
        print(f"Max  speed : {np.max(speed):.2f} nodes/s")
        print(f"Mean Vx    : {np.mean(vh[:,0]):.2f} nodes/s")
        print(f"Mean Vy    : {np.mean(vh[:,1]):.2f} nodes/s")

        save_summary_report(trail_raw, trail_kf, vel_hist, fps_last=fps)

    print("\nDone.")


if __name__ == "__main__":
    main()
```