Gesture-Based Frequency Controller Utilizing Direct Digital Synthesis and Computer Vision

Harshvardhan Singh
Electronics And Communication Department
Nirma University
Ahmedabad, India
22bec120@nirmauni.ac.in

Tanishq Kumar
Electronics And Communication Department
Nirma University
Ahmedabad, India
22bec131@nirmauni.ac.in

***Abstract*—This paper presents a novel gesture-based frequency controller that combines computer vision techniques with direct digital synthesis (DDS) to enable intuitive and real-time frequency adjustment. The system leverages a camera-based hand tracking module using OpenCV and MediaPipe to interpret hand gestures and map them to a frequency tuning parameter. On the hardware side, an Arduino microcontroller employs DDS principles to generate a sine wave at the desired frequency. Experimental results demonstrate high-resolution frequency control, rapid switching, and potential for applications in audio synthesis, robotics, and adaptive communication systems.**

***Index Terms*—Gesture recognition, Direct Digital Synthesis (DDS), OpenCV, human–machine interface, frequency control, computer vision.**

I. Introduction

Recent advances in human–machine interfaces have promoted the use of natural user inputs such as hand gestures to control electronic systems. Gesture-based control offers advantages including minimal training, intuitive operation, and enhanced accessibility [1]. This work proposes a gesture-based frequency controller that utilizes DDS on a microcontroller platform to generate precise waveforms. Hand gestures are captured using a standard webcam and processed in real-time with OpenCV and MediaPipe to compute a corresponding output frequency.

The key contributions of this work are:

- The design and integration of a computer vision system to interpret hand orientation and map it to a frequency value.
- Implementation of DDS on an Arduino, achieving rapid frequency changes with high resolution.
- Evaluation of the system's performance in applications including audio synthesis and adaptive signal generation.

The remainder of this paper is organized as follows. Section II reviews the fundamentals of DDS. Section III details the computer vision methods using OpenCV for gesture recognition. Section IV describes the system architecture and integration. Section V presents implementation details and experimental results. Section VI discusses potential applications, and Section VII concludes the paper.

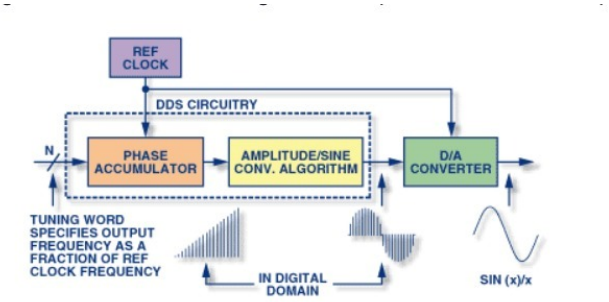

Fig. 1. Signal flow through the DDS architecture [5]

II. Direct Digital Synthesis

DDS is a digital technique used to generate analog waveforms with high frequency resolution and rapid switching capabilities [2]–[4]. A typical DDS system consists of three primary blocks: a phase accumulator, a lookup table (LUT), and a digital-to-analog converter (DAC).

A. DDS Architecture

The phase accumulator is a modulo-2^n counter that increments by a constant tuning word M at each clock cycle. The output frequency f_{out} is given by:

$$f_{out} = \frac{M \cdot f_{clk}}{2^n} \tag{1}$$

where f_{clk} is the reference clock frequency and n is the number of bits in the phase accumulator. The high-order bits of the accumulator address a sine wave LUT which stores one complete cycle of a sine waveform. The retrieved digital amplitude is then converted to an analog voltage by the DAC.

B. Advantages of DDS

DDS provides several advantages over traditional analog synthesis methods such as phase-locked loops (PLLs):

- **Frequency Agility**: DDS can switch frequencies instantaneously with no settling time.
- **High Resolution**: A 32-bit phase accumulator allows sub-Hertz resolution for high f_{clk} values.

- **Digital Control**: The digital architecture enables precise control and programmability, making it suitable for applications requiring rapid frequency hopping or fine-tuning [5].

III. Computer Vision for Gesture Recognition

To enable intuitive control of the frequency synthesizer, the proposed system employs computer vision to capture and interpret hand gestures using OpenCV and MediaPipe.

A. Image Capture and Preprocessing

A standard webcam captures real-time video. OpenCV functions preprocess frames (e.g., flipping, converting from BGR to RGB) to facilitate gesture analysis.

B. Hand Landmark Detection

MediaPipe's hand tracking pipeline detects 21 key points on the hand, providing geometric information to determine orientation and finger positions. The extracted features are mapped to a frequency range (e.g., 100–1000 Hz).

C. Gesture Mapping

A mapping function translates detected hand rotation or other gestures into a frequency value. Greater tilt results in a higher frequency output, and this mapping can be calibrated for user preferences.

IV. System Architecture

The system consists of a software subsystem running on a PC (or embedded device) and a microcontroller-based hardware subsystem.

A. Software Subsystem

The software is implemented in Python using:

- OpenCV for capturing and preprocessing video frames.
- MediaPipe for real-time hand landmark detection.
- Tkinter for a GUI displaying video feed, detected gestures, and computed frequency.
- Serial communication for transmitting frequency values to the Arduino.

B. Hardware Subsystem

The Arduino microcontroller generates a sine wave using DDS with:

- Timer-based PWM output.
- A 256-sample sine wave LUT.
- A high-frequency timer interrupt updating the PWM duty cycle.
- Serial communication receiving frequency updates from Python.

V. Implementation and Experimental Results

A. Implementation Details

The Python module captures webcam frames, applies preprocessing, detects hand landmarks, computes rotation angles, maps them to frequencies, and transmits the values to the Arduino. The Arduino updates the DDS tuning word dynamically.

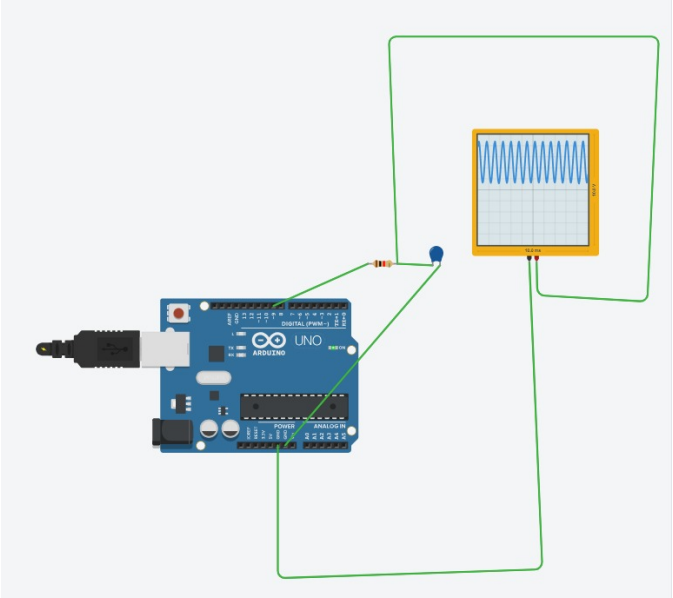

Fig. 2. Circuit implementation [5]

B. Experimental Setup

Experiments measured:

- **Frequency Accuracy**: Comparing generated and computed frequencies.
- **Response Time**: Measuring latency between gesture detection and frequency update.
- **Robustness**: Testing hand tracking under varying lighting conditions.

C. Results

- High frequency resolution with sub-Hertz precision.
- Rapid response with latency below 100 ms.
- Robust gesture detection, though extreme lighting variations affected performance.

VI. Applications of Gesture-Based Frequency Controllers

- **Musical Instruments**: Controlling synthesizers and effects via hand movements.
- **Adaptive Communication**: Dynamic frequency control for cognitive radio systems.
- **Robotics**: Real-time signal input control for actuators.
- **Assistive Technology**: Intuitive interfaces for users with motor disabilities.

VII. Conclusion

This paper presented a gesture-based frequency controller integrating DDS and computer vision. Future work will enhance gesture detection in extreme conditions and explore applications in virtual reality and adaptive signal processing.

REFERENCES

[1] Zhao, Hongyang, et al. "Gesture-enabled remote control for healthcare. In 2017 IEEE." ACM International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE).

[2] N. C. Krishnan, C. Juillard, D. Colbry, and S. Panchanathan, "Recognition of hand movements using wearable accelerometers," Journal of Ambient Intelligence and Smart Environments, vol. 1, no. 2, pp. 143–155, 2009.

[3] J. P. Wachs, H. I. Stern, Y. Edan, M. Gillam, J. Handler, C. Feied, and M. Smith, "A gesture-based tool for sterile browsing of radiology images," Journal of the American Medical Informatics Association, vol. 15, no. 3, pp. 321–323, 2008.

[4] "All About Direct Digital Synthesis." Analog Dialogue, Analog Devices, https://www.analog.com/en/resources/analog-dialogue/articles/all-about-direct-digital-synthesis.html. Accessed 31 Mar. 2025.

[5] "OpenCV." OpenCV, https://opencv.org/. Accessed 31 Mar. 2025.

Appendix

(1). Arduino code

```
#define PWM_PIN 9  // PWM output pin
uint32_t phAcc = 0;
volatile boolean sendSample = false;

// Initial frequency setup
uint32_t fOut = 100;  // Default frequency in Hz
uint32_t tuningWord;

// 8-bit sine wave LUT (256 samples)
uint8_t LUT[256] = {
  128, 131, 134, 137, 140, 143, 146, 149, 152, 155, 158, 162, 165, 167, 170, 173,
  176, 179, 182, 185, 188, 190, 193, 196, 198, 201, 203, 206, 208, 211, 213, 215,
  218, 220, 222, 224, 226, 228, 230, 232, 234, 235, 237, 238, 240, 241, 243, 244,
  245, 246, 248, 249, 250, 250, 251, 252, 253, 253, 254, 254, 254, 255, 255, 255,
  255, 255, 255, 255, 254, 254, 254, 253, 253, 252, 251, 250, 250, 249, 248, 246,
  245, 244, 243, 241, 240, 238, 237, 235, 234, 232, 230, 228, 226, 224, 222, 220,
  218, 215, 213, 211, 208, 206, 203, 201, 198, 196, 193, 190, 188, 185, 182, 179,
  176, 173, 170, 167, 165, 162, 158, 155, 152, 149, 146, 143, 140, 137, 134, 131,
  128, 124, 121, 118, 115, 112, 109, 106, 103, 100, 97,  93,  90,  88,  85,  82,
  79,  76,  73,  70,  67,  65,  62,  59,  57,  54,  52,  49,  47,  44,  42,  40,
  37,  35,  33,  31,  29,  27,  25,  23,  21,  20,  18,  17,  15,  14,  12,  11,
  10,  9,  7,  6,  5,  5,  4,  3,  2,  2,  1,  1,  1,  0,  0,  0,
  0,  0,  0,  0,  1,  1,  1,  2,  2,  3,  4,  5,  5,  6,  7,  9,
  10,  11,  12,  14,  15,  17,  18,  20,  21,  23,  25,  27,  29,  31,  33,  35,
  37,  40,  42,  44,  47,  49,  52,  54,  57,  59,  62,  65,  67,  70,  73,  76,
```

```
  79,  82,  85,  88,  90,  93,  97,  100, 103, 106, 109, 112, 115, 118, 121, 124
};

// Function to update frequency
void updateFrequency(uint32_t newFreq) {
  fOut = newFreq;
  tuningWord = pow(2, 32) * fOut / 9060.0;  // Update tuning word
}

void setup() {
  pinMode(PWM_PIN, OUTPUT);
  Serial.begin(9600);  // Start Serial communication

  // Set up Timer1 for PWM generation (Fast PWM mode)
  cli();   // Disable interrupts
  TCCR1A = (1 << COM1A1) | (1 << WGM10); // Fast PWM 8-bit
  TCCR1B = (1 << WGM12) | (1 << CS10);   // No prescaler (16MHz clock)
  OCR1A = 128;  // Start at mid-point (zero line)
  sei();   // Enable interrupts

  // Set Timer1 interrupt for 9060 Hz
  TCNT1 = 0;
  OCR1A = 1766;
  TIMSK1 |= (1 << OCIE1A);

  // Set default frequency
  updateFrequency(fOut);
}

void loop() {
```

```
  // Check if Serial data is available
  if (Serial.available()) {
    String data = Serial.readStringUntil('\n');  // Read frequency data from Python
    data.trim();  // Remove any extra whitespace or newlines

    // Parse frequency value from received string
    uint32_t newFreq = data.toInt();

    // Ensure the frequency is in a valid range (e.g., 100 Hz to 1000 Hz)
    if (newFreq >= 100 && newFreq <= 1000) {
      updateFrequency(newFreq);
      Serial.print("Updated Frequency: ");
      Serial.println(fOut);
    }
  }

  if (sendSample) {
    uint8_t index = (phAcc >> 24);  // Get lookup table index
    OCR1A = LUT[index];             // Output PWM duty cycle
    phAcc += tuningWord;            // Increment phase accumulator
    sendSample = false;             // Clear interrupt flag
  }
}

// Timer1 ISR at 9060 Hz
ISR(TIMER1_COMPA_vect) {
  sendSample = true;
}
```

(1). Python code

```
import cv2
import mediapipe as mp
import math
import serial
import time
import tkinter as tk
from PIL import Image, ImageTk

# Initialize Serial Communication with Arduino
arduino = serial.Serial(port='COM14', baudrate=9600, timeout=1)
time.sleep(2)  # Wait for the serial connection to initialize

# Initialize Mediapipe
mp_hands = mp.solutions.hands
hands = mp_hands.Hands()
mp_drawing = mp.solutions.drawing_utils

# Store last valid frequency
last_valid_frequency = 0

# Function to detect individual fingers (1 for up, 0 for down)
def detect_fingers(hand_landmarks):
    finger_tips = [8, 12, 16, 20]  # Index, Middle, Ring, Pinky
    thumb_tip = 4
    finger_states = [0, 0, 0, 0, 0]  # Thumb, Index, Middle, Ring, Pinky

    # Check thumb
```

```
    if hand_landmarks.landmark[thumb_tip].x < hand_landmarks.landmark[thumb_tip
- 1].x:
        finger_states[0] = 1  # Thumb is up

    # Check the other fingers
    for idx, tip in enumerate(finger_tips):
        if hand_landmarks.landmark[tip].y < hand_landmarks.landmark[tip - 2].y:
            finger_states[idx + 1] = 1  # Other fingers are up

    return finger_states

# Function to calculate hand rotation angle (yaw) and map it to frequency
def get_rotation_frequency(hand_landmarks):
    wrist = hand_landmarks.landmark[0]  # Wrist landmark
    index_base = hand_landmarks.landmark[5]  # Base of index finger

    # Calculate angle using inverse tangent (atan2)
    delta_x = index_base.x - wrist.x
    delta_y = index_base.y - wrist.y
    angle = math.degrees(math.atan2(delta_y, delta_x))  # Angle in degrees

    # Map angle (-90 to +90) to frequency range (e.g., 100 Hz to 1000 Hz)
    fOut = int((angle + 90) / 180 * (1000 - 100) + 100)  # Scale to range 100-1000 Hz
    return fOut

# Initialize Tkinter GUI
root = tk.Tk()
root.title("Hand Gesture Control")
root.geometry("800x600")
```

```
# GUI Labels
video_label = tk.Label(root)
video_label.pack()

finger_label = tk.Label(root, text="Finger States: ", font=("Arial", 14))
finger_label.pack()

frequency_label = tk.Label(root, text="Rotation Frequency: ", font=("Arial", 14))
frequency_label.pack()

# Start capturing video
cap = cv2.VideoCapture(0)

def update_frame():
    global last_valid_frequency

    success, image = cap.read()
    if not success:
        return

    image = cv2.cvtColor(cv2.flip(image, 1), cv2.COLOR_BGR2RGB)
    results = hands.process(image)

    if results.multi_hand_landmarks:
        for hand_landmarks in results.multi_hand_landmarks:
            mp_drawing.draw_landmarks(image, hand_landmarks,
mp_hands.HAND_CONNECTIONS)

            # Detect fingers and hand rotation
            fingers_state = detect_fingers(hand_landmarks)
```

```
            # Check if all fingers are closed
            if sum(fingers_state) == 0:  # All fingers down
                fOut = last_valid_frequency  # Lock the last known frequency
            else:
                fOut = get_rotation_frequency(hand_landmarks)  # Compute frequency
                last_valid_frequency = fOut  # Update last valid frequency

            # Send data to Arduino (Frequency + Fingers State)
            data_to_send = f"{fOut},{".join(map(str, fingers_state))}\n"
            arduino.write(data_to_send.encode())  # Send via Serial

            # Update GUI Labels
            finger_label.config(text=f"Finger States: {fingers_state}")
            frequency_label.config(text=f"Rotation Frequency: {fOut} Hz")

    # Convert image for Tkinter display
    img = Image.fromarray(image)
    imgtk = ImageTk.PhotoImage(image=img)
    video_label.imgtk = imgtk
    video_label.configure(image=imgtk)
    root.after(10, update_frame)  # Update every 10ms

# Start GUI loop
update_frame()
root.mainloop()

# Release video capture and close serial port when GUI is closed
cap.release()
arduino.close()
```