VILNIAUS GEDIMINO TECHNIKOS UNIVERSITETAS

ELEKTRONIKOS FAKULTETAS

ELEKTRONIKOS INŽINERIJOS KATEDRA

REALAUS LAIKO OPERACINĖS SISTEMOS

Laboratorinis darbas 3

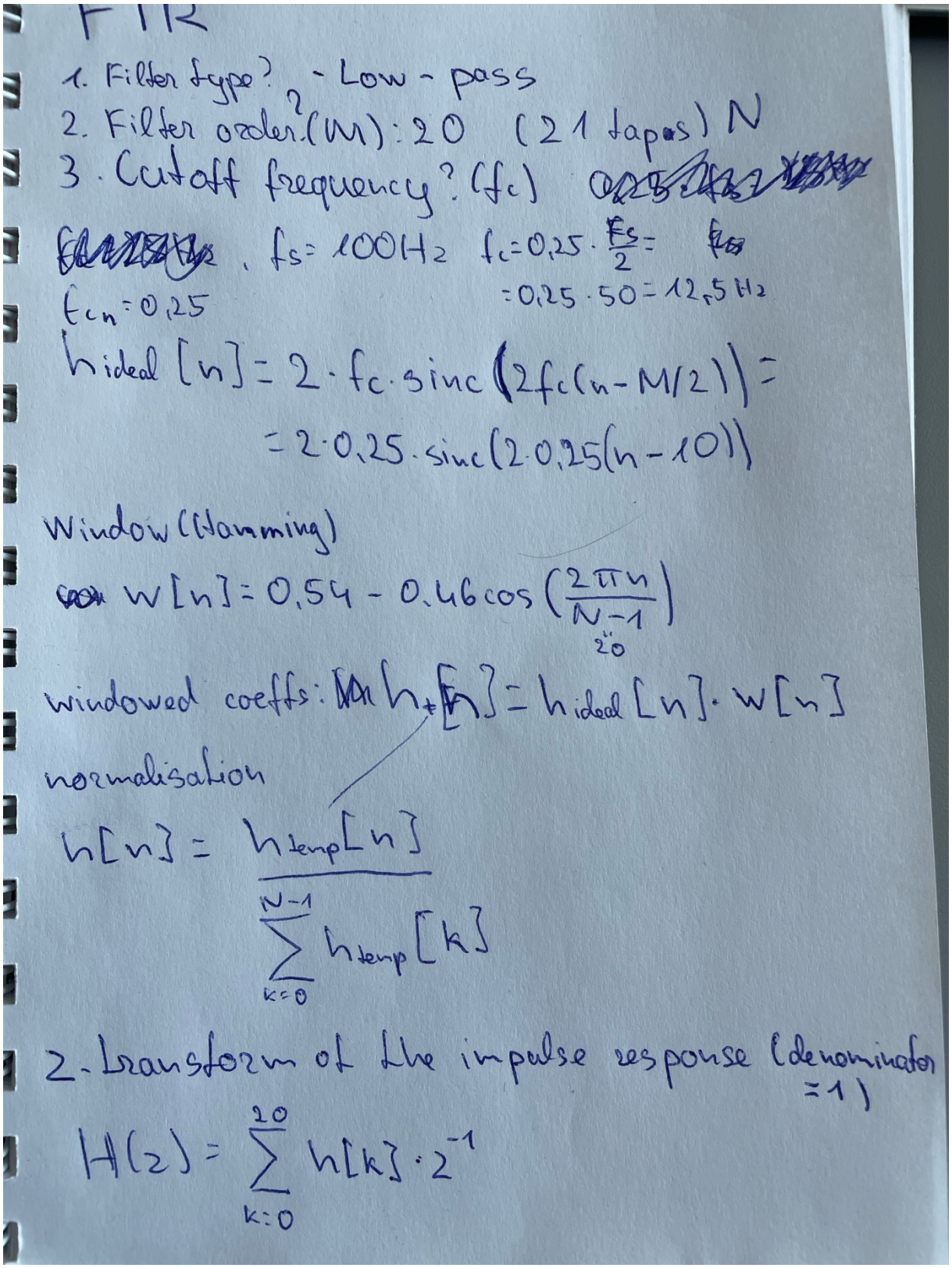
Atliko: EKSFM-24 gr. studentas Ignas Malinauskas

Tikrino: dr. Saulius Sakavičius

VILNIUS 2025

FIR filtras

1. FIR filtro koeficientų apskaičiavimo žingsniai ir perdavimo funkcija Z srityje.

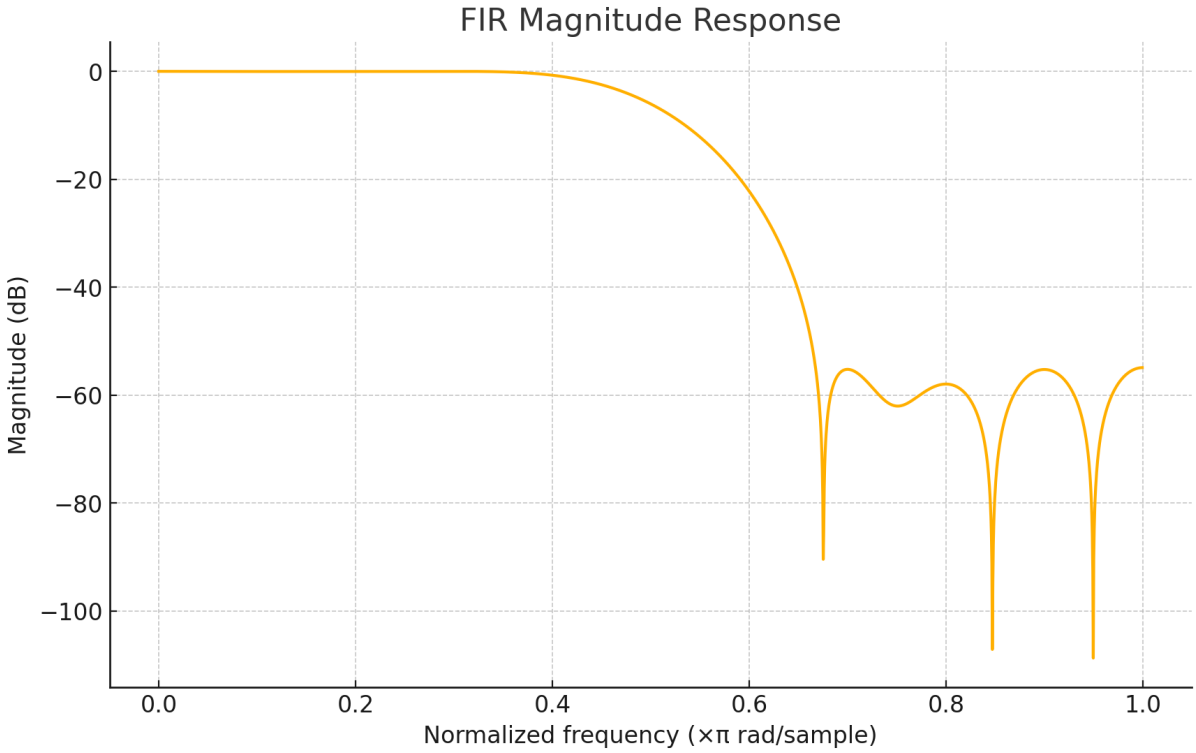


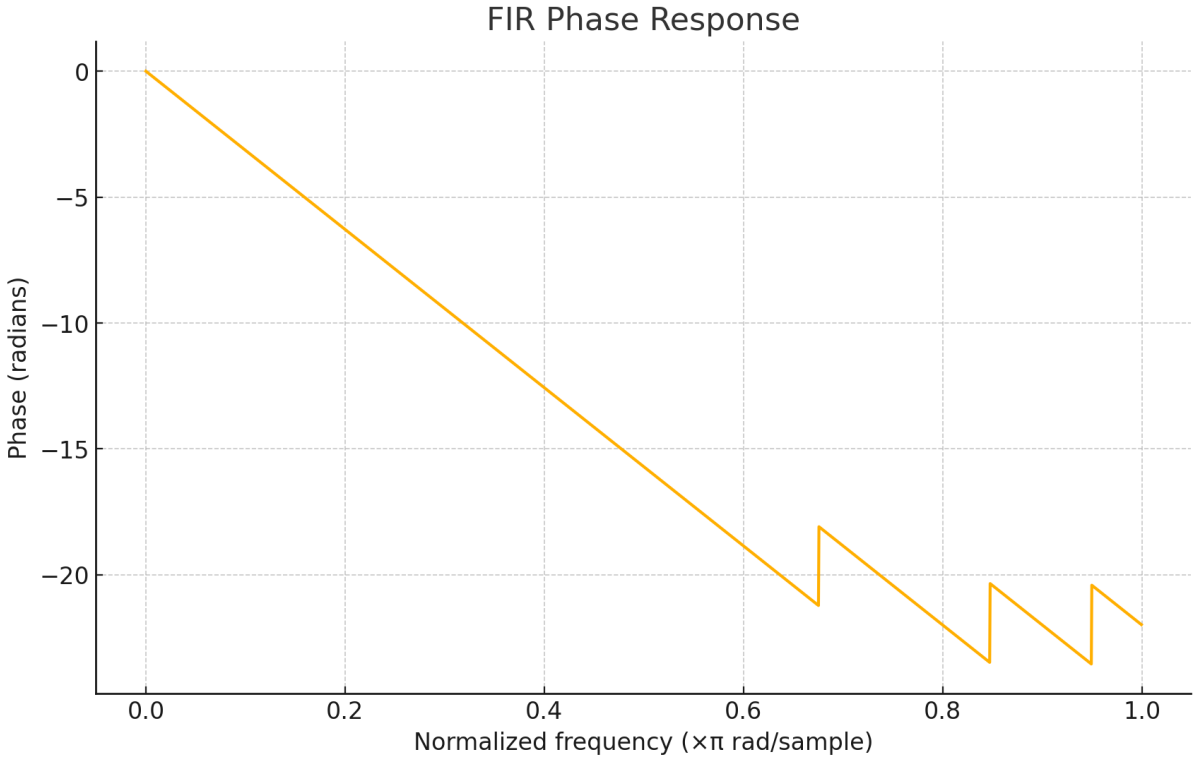
Koeficientai:

h = { 0.0000000000, 0.0036191610, -0.0000000000, 0.0122382502, 0.0000000000, 0.0343155511, 0.0000000000, -0.0858292531, 0.0000000000, 0.3105830617, 0.4990994590,

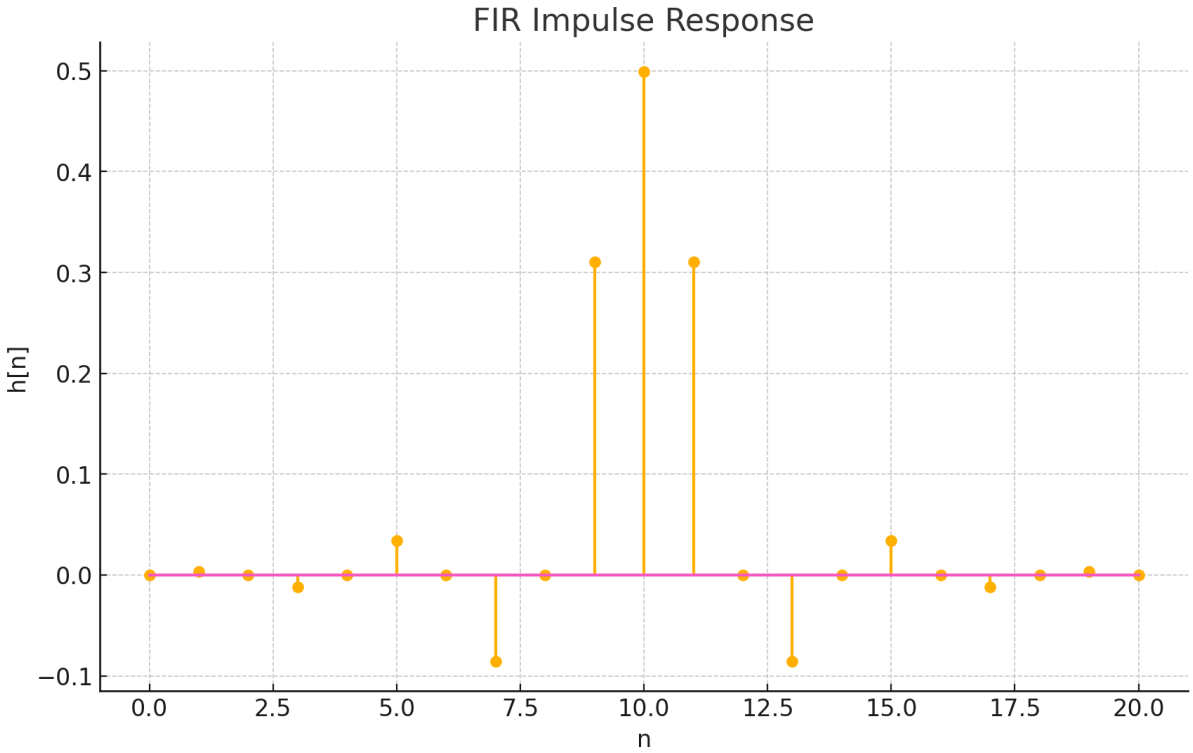
0.3105830617, 0.0000000000, -0.0858292531, 0.0000000000, 0.0343155511, 0.0000000000, -0.0122382502, 0.0000000000, 0.0036191610, 0.0000000000 };

2. Dažnio ir fazės atsako grafai

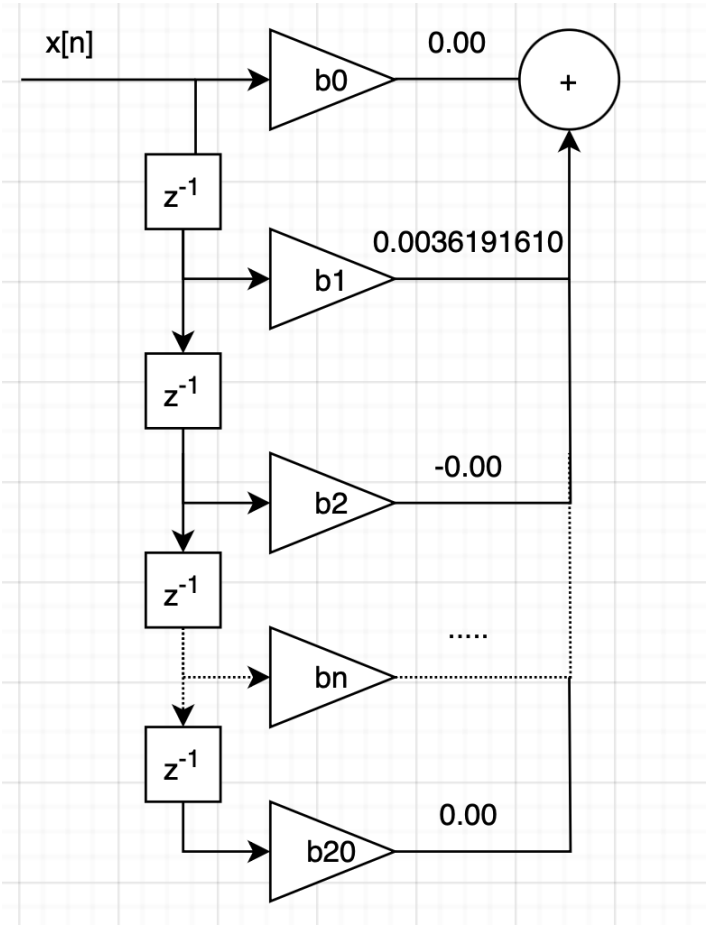




3. Impulso atsako grafas



4. Direct Form I diagrama su koeficientais



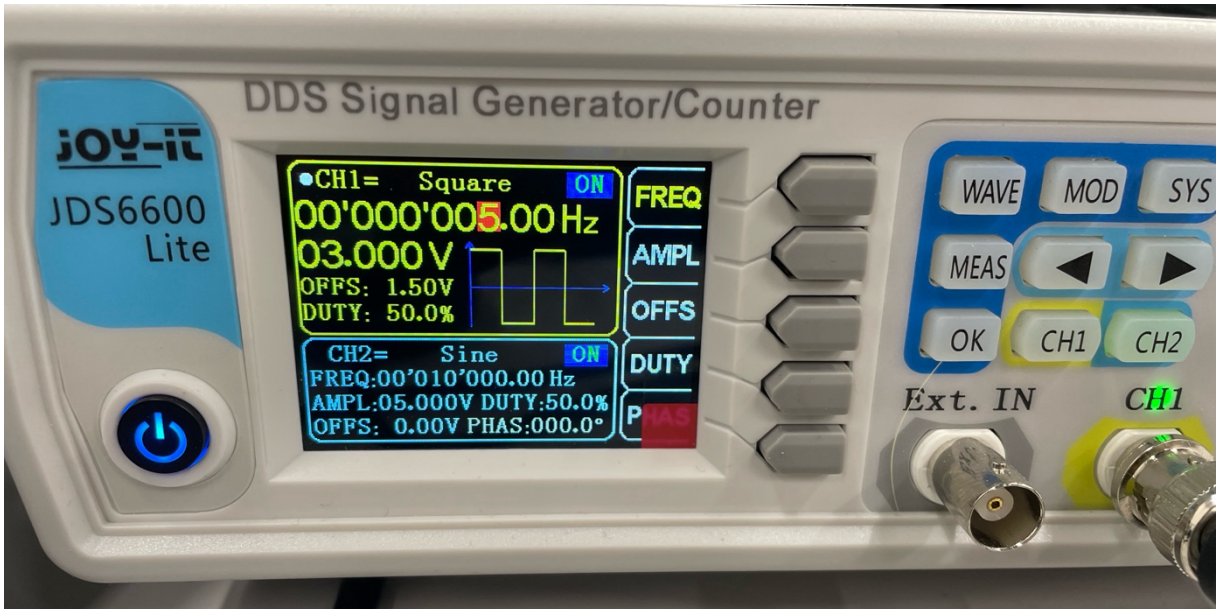
5. Nulių ir polių diagrama

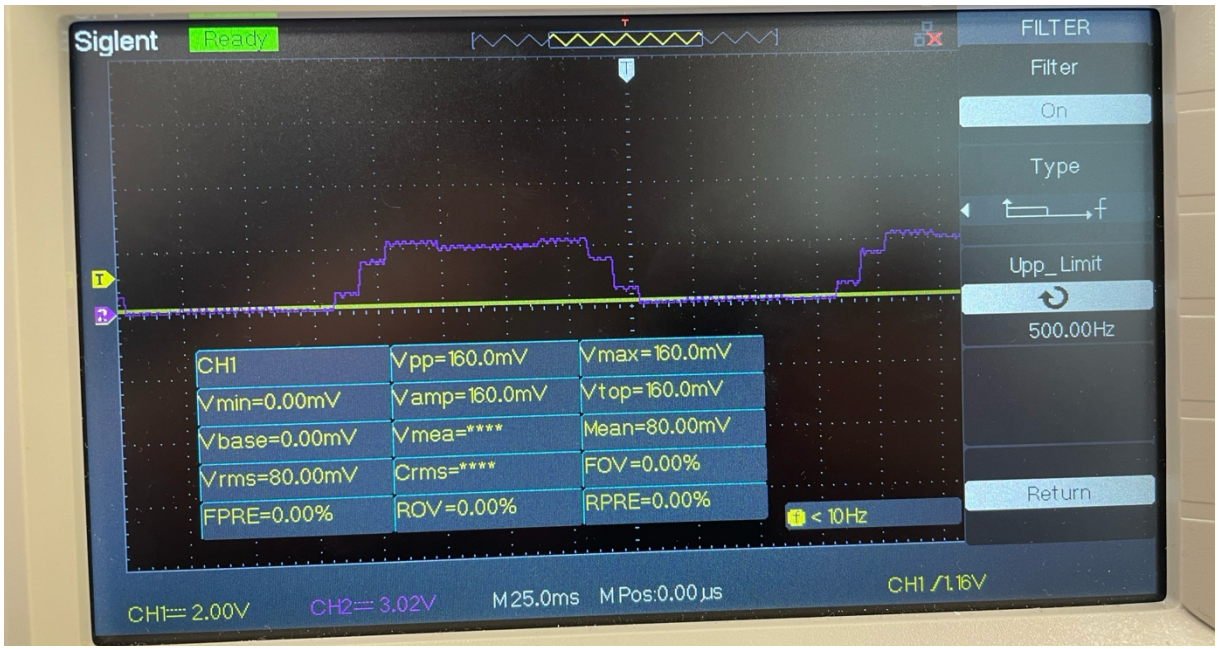
6. FIR filtro implementacijos kodas

```c
static void fir_filter_task(void *arg)
{
    (void)arg;
    float ring[FIR_TAPS] = {0};
    size_t w = 0;

    uint16_t raw;
    while (xQueueReceive(filt_queue, &raw, portMAX_DELAY) == pdTRUE) {
        ring[w] = (float)raw;

        /* Convolution (direct-form): h · x  */
        float acc = 0.0f;
        size_t r = w;
        for (size_t k = 0; k < FIR_TAPS; k++) {
            acc += fir_coeff[k] * ring[r];
            r = (r == 0) ? (FIR_TAPS - 1) : (r - 1);   // circular buffer walk
        }
        w = (w + 1) % FIR_TAPS;

        /* Clip to ADC range */
        float clamped = fminf(fmaxf(acc, 0.0f), 4095.0f);
        uint16_t filt = (uint16_t)clamped;

        xQueueSend(pwm_queue, &filt, 0);                // to PWM

        /* Send to logger (non-blocking) */
        adc_sample_t sample = {
            .raw  = raw,
```

```
            .filt = filt,
            .mv   = esp_adc_cal_raw_to_voltage(raw, &adc_chars),
            .duty = (filt * ((1 << LEDC_DUTY_RES) - 1)) / 4095
        };
        xQueueSend(log_queue, &sample, 0);
    }
}
```

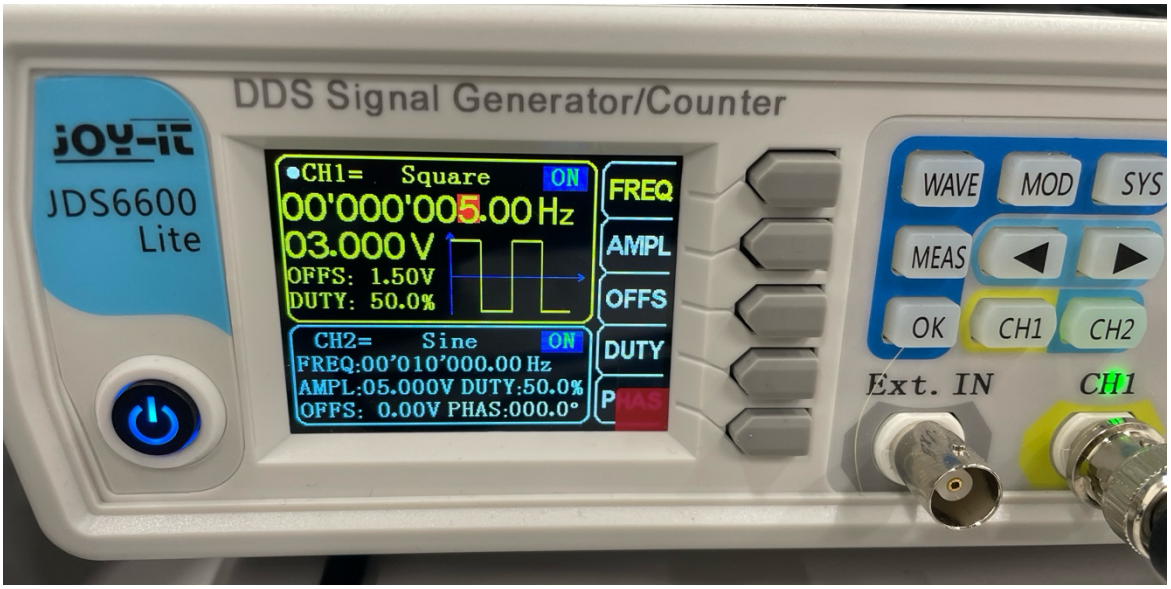
7. Testinio signalo (kvadratinės bangos) charakteristikos ir filtruotas signalas

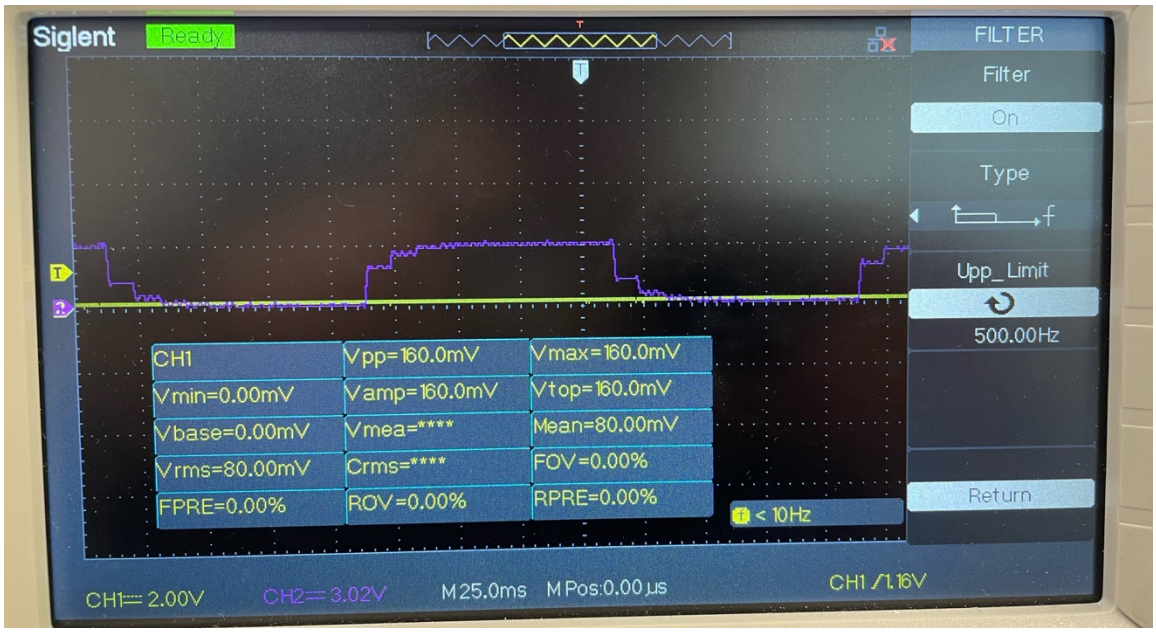




IIR filtras

Pirmo laipsnio exponential smoothing filtras



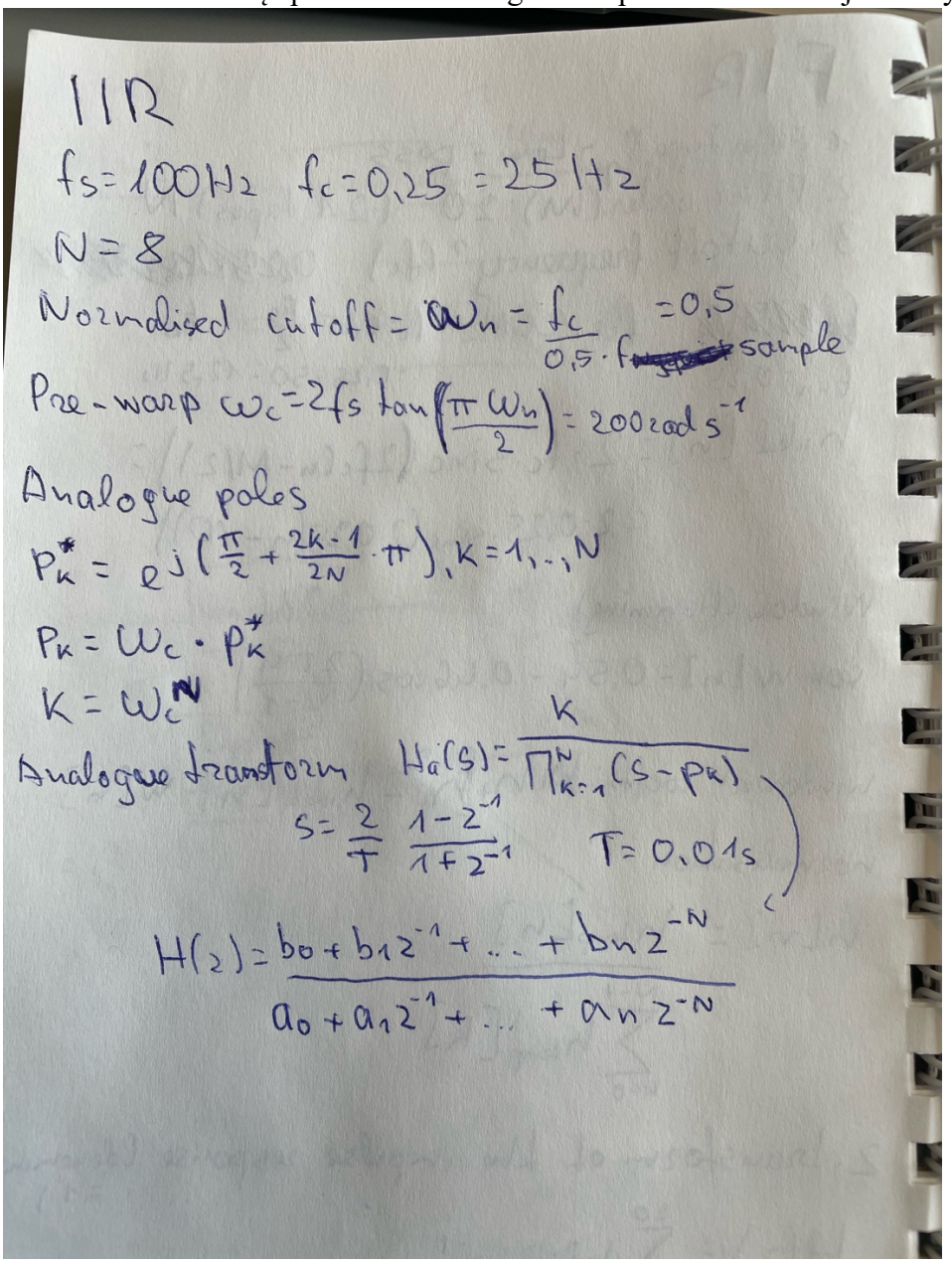



```c
#define IIR_ALPHA 0.611f
static void iir_filter_task(void *arg)
{
    (void)arg;
    float y = 0.0f;          // previous output (starts at 0)

    uint16_t raw;
    while (xQueueReceive(filt_queue, &raw, portMAX_DELAY) == pdTRUE) {

        /* Exponential smoothing: y ← y + α · (x − y) */
        y += IIR_ALPHA * ((float)raw - y);

        /* Clip to ADC range */
        float clamped = fminf(fmaxf(y, 0.0f), 4095.0f);
        uint16_t filt = (uint16_t)clamped;

        xQueueSend(pwm_queue, &filt, 0);                // to PWM
```

```
        /* Send to logger (non-blocking) */
        adc_sample_t sample = {
            .raw  = raw,
            .filt = filt,
            .mv   = esp_adc_cal_raw_to_voltage(raw, &adc_chars),
            .duty = (filt * ((1 << LEDC_DUTY_RES) - 1)) / 4095
        };
        xQueueSend(log_queue, &sample, 0);
    }
}
```

8 laipsnio butterworth filtras:

1. IIR filtro koeficientų apskaičiavimo žingsniai ir perdavimo funkcija Z srityje



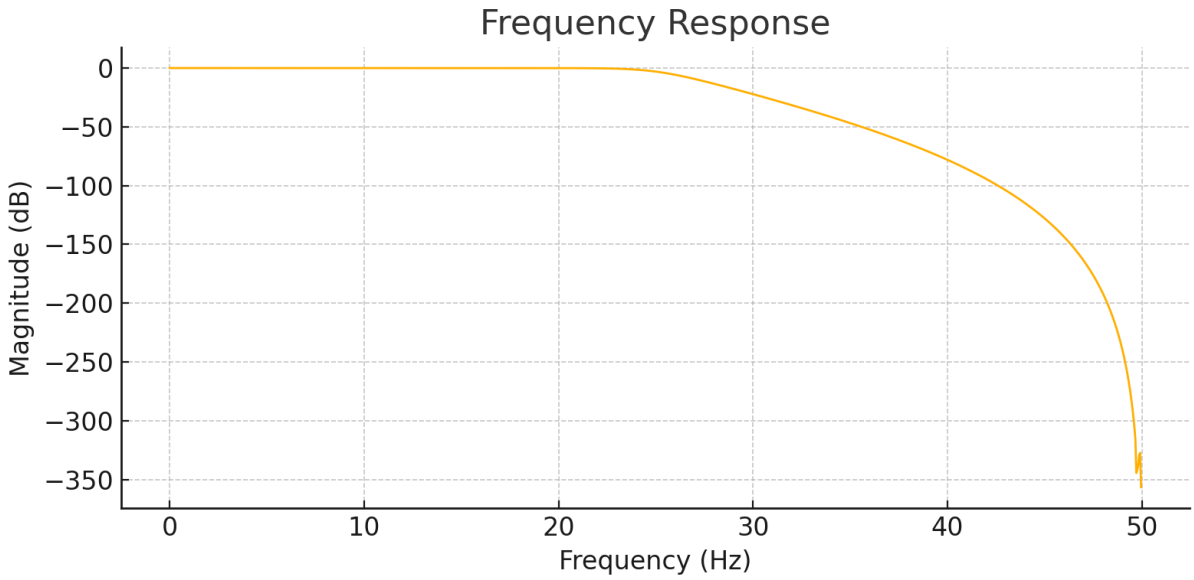
Koeficientai:

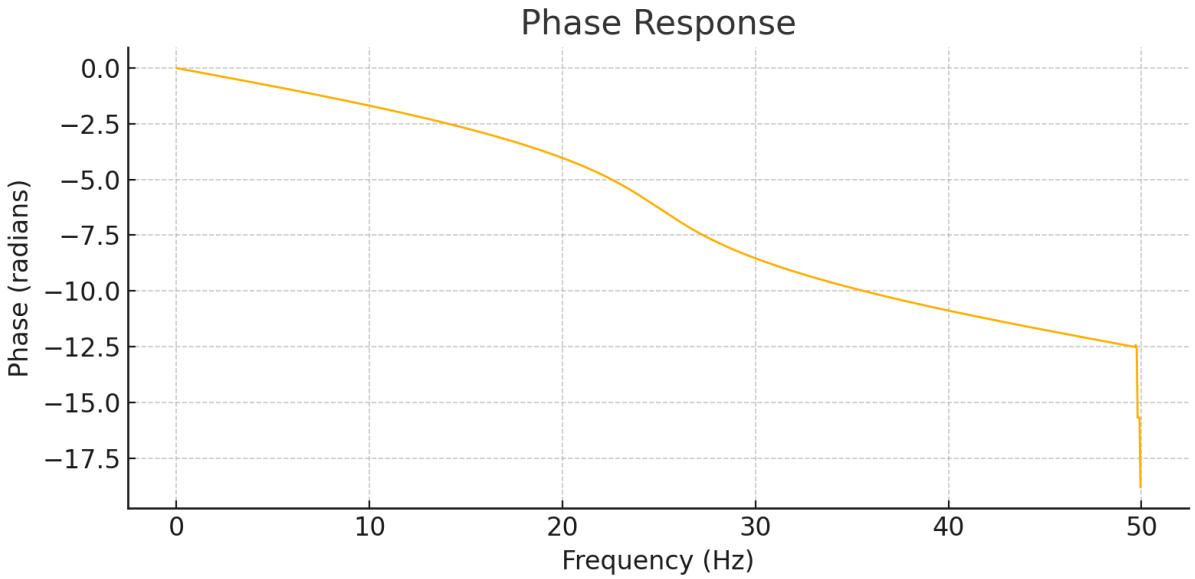
b = [0.0092672856, 0.074138285, 0.25948400, 0.51896799

0.64870999, 0.51896799, 0.25948400, 0.074138285, 0.0092672856]

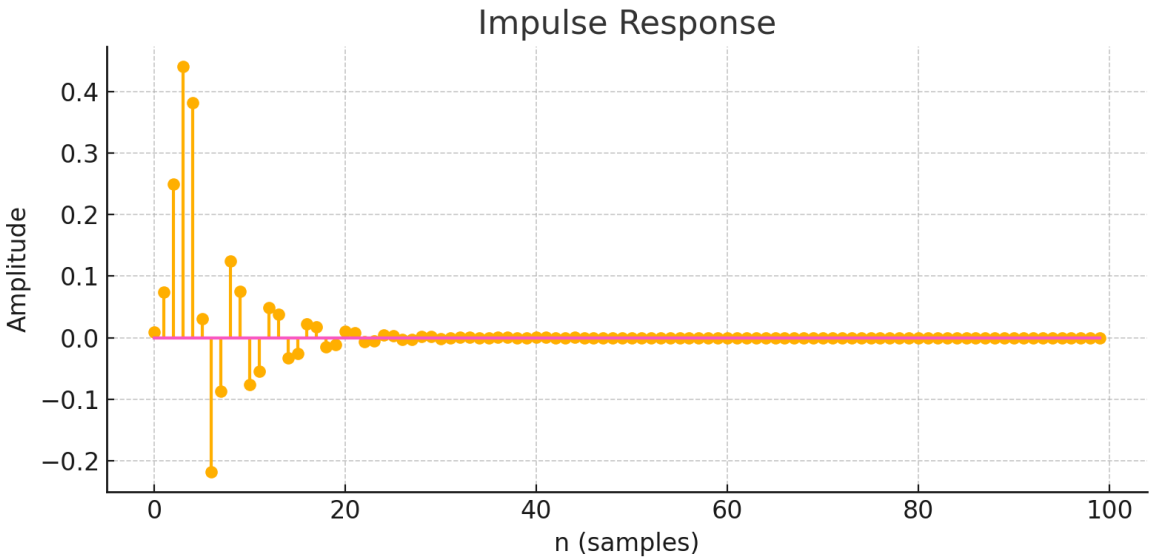
a = [1.00000000, 0, 1.06093560, 0, 0.290888157, 0, 0.0204295879, 0, 1.71765164e-4]

2. Dažnio ir fazės atsako grafai

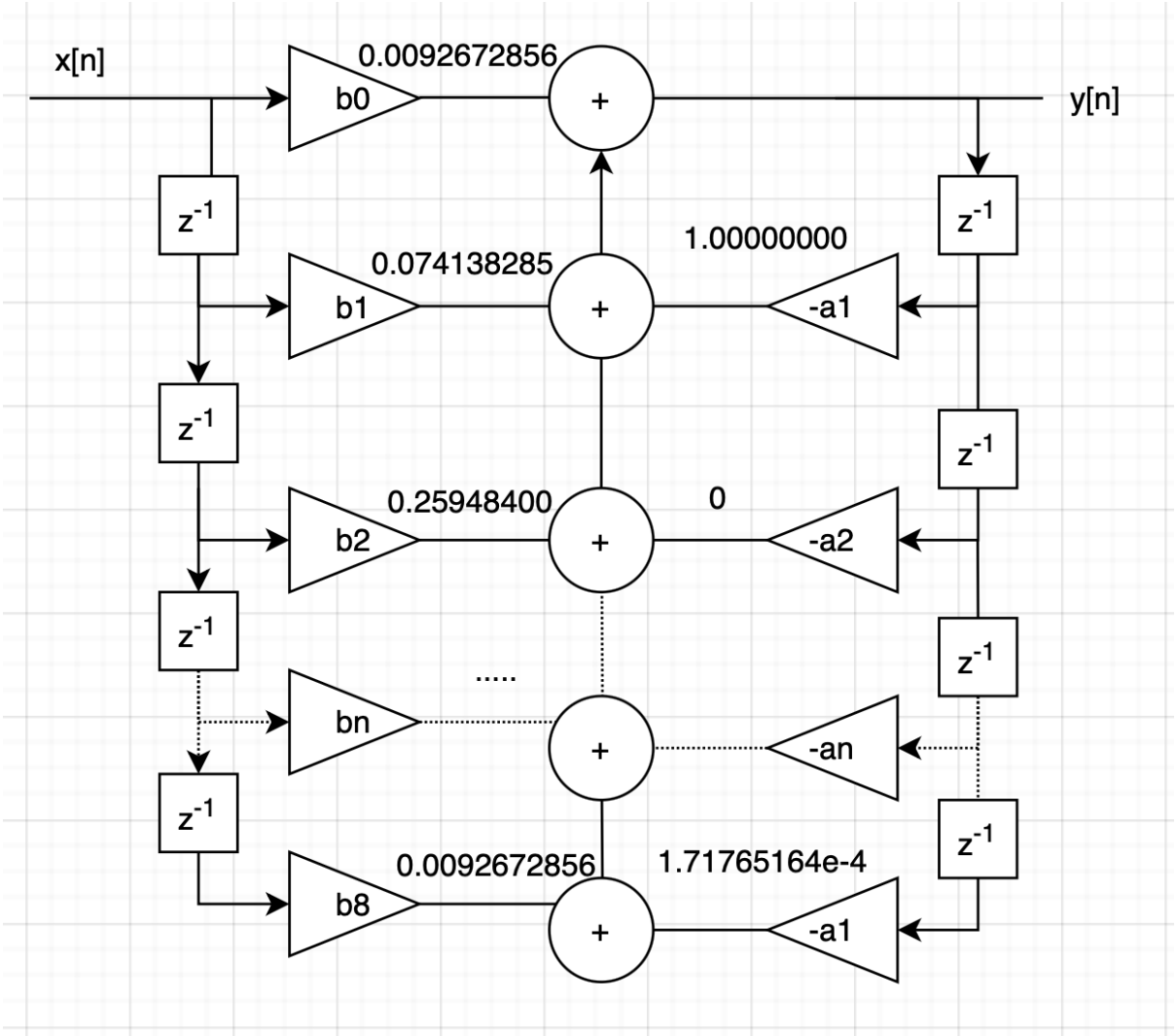




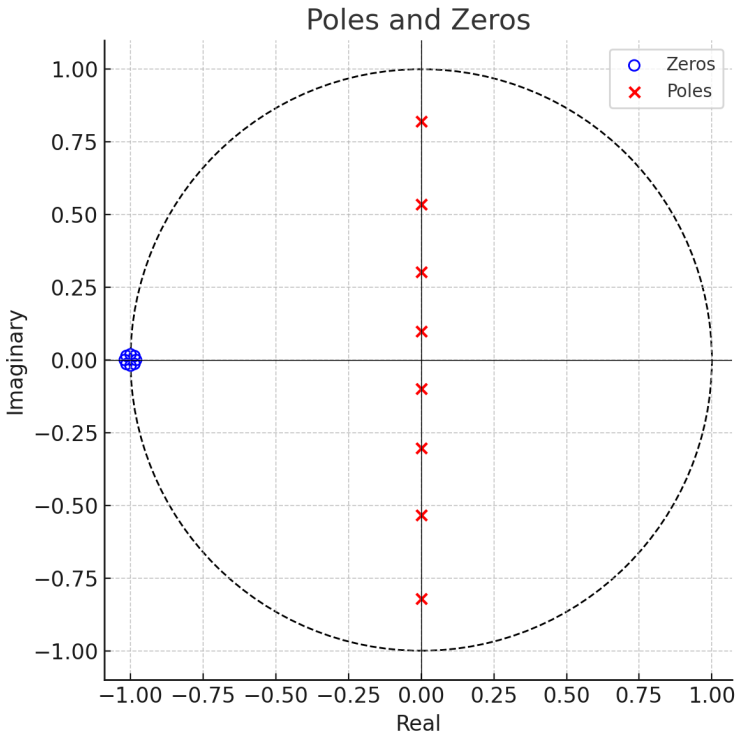
3. Impulso atsako grafas



4. Direct Form I diagrama su koeficientais



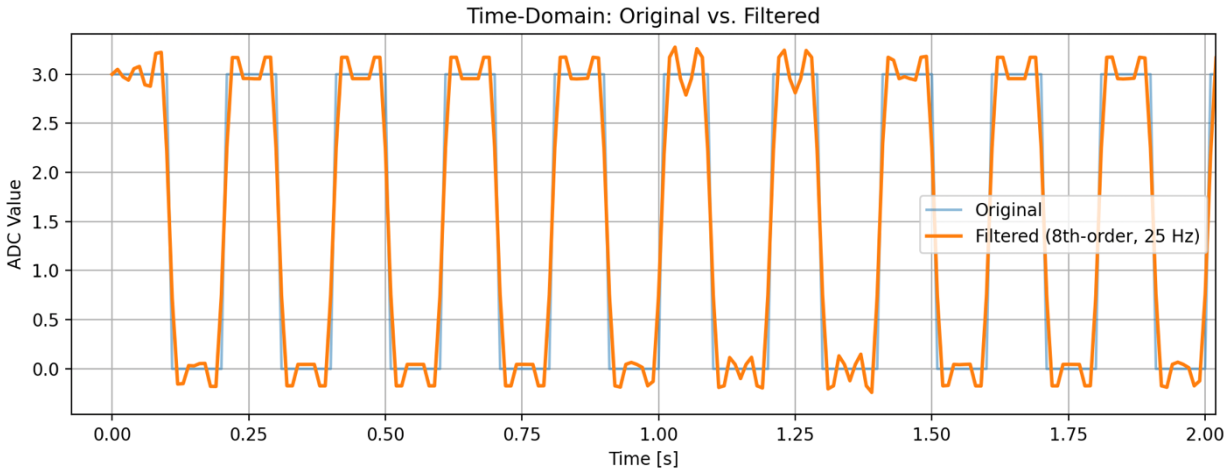
5. Nulių ir polių diagrama

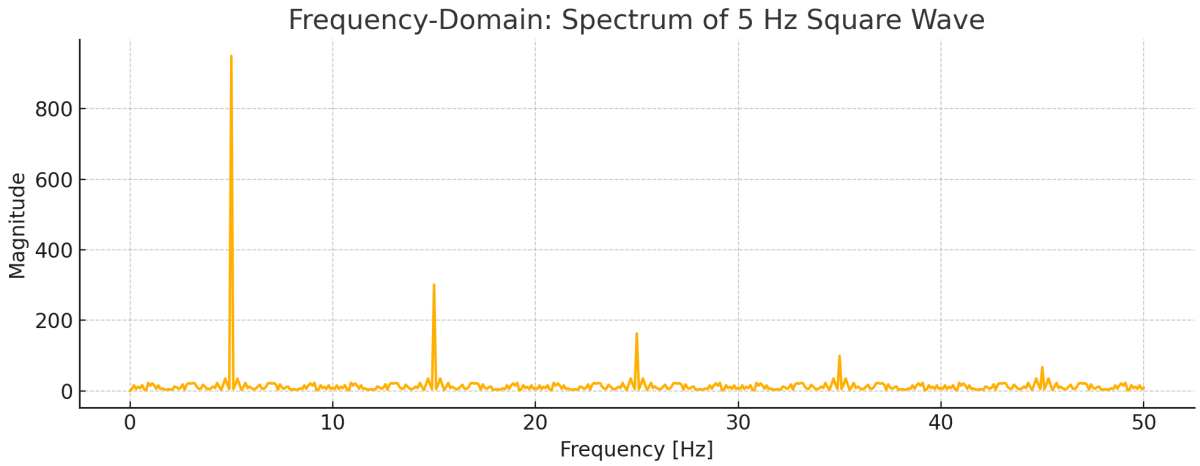


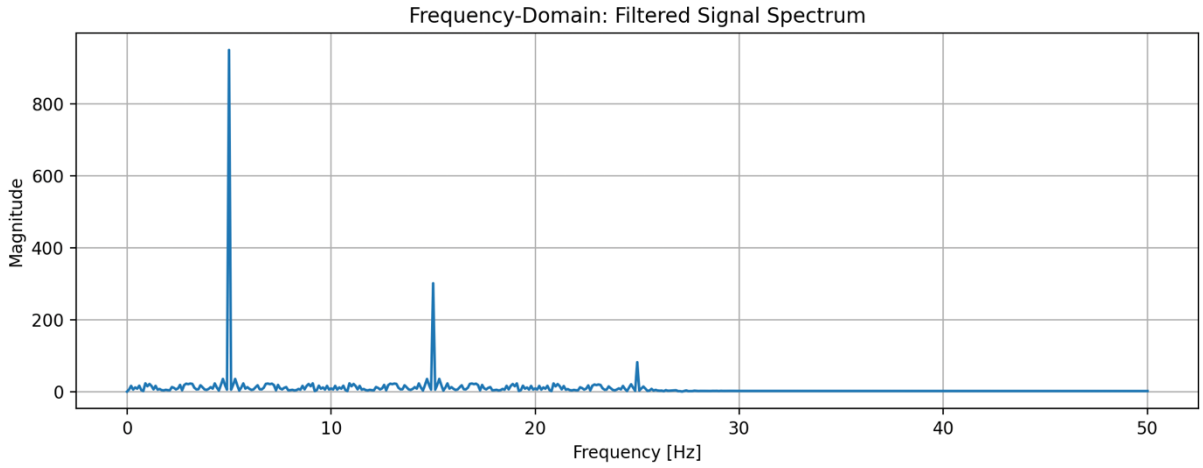
6. FIR filtro implementacijos kodas

```
# 3. Design 8th-order Butterworth IIR low-pass
order = 8
cutoff = 25  # Cutoff frequency in Hz
nyquist = fs / 2
normal_cutoff = cutoff / nyquist  # Normalized cutoff (0 to 1)

b, a = butter(order, normal_cutoff, btype='low', analog=False)

# 4. Apply filter (zero-phase)
filtered = filtfilt(b, a, adc_values)
```

17. Testinio signalo (kvadratinės bangos) charakteristikos ir filtruotas signalas







Išvados

Testuojant filtruotą signalą naudojant FIR ir IIR per osciloskopą buvo naudojamas osciloskopo skaitmeninis filtras, kuris išlygindavo kvadratinę bangą ir buvo galima matyti, kad abu filtrai veikia. Kadangi testavimo metu buvo išbandytas tik pirmo laipsnio IIR filtras, gauti ADC rodmenys buvo išsiųsti per UART į terminalą, išsaugoti ir vėliau apdoroti su Python programavimo kalba, kuria buvo sukurtas 8 laipsnio Butterworth filtras ir išanalizuti duomenys.

Visos sistemos kodas su FIR filtru (kodas toks pat ir su pirmo laipsnio filtru išskyrus filtravimo užduotį)

```
/*
 * ADC-to-PWM Mirror + UART Logger + FIR Filter (ESP32-S3)
 * ------------------------------------------------------
 * • Samples ADC1-CH5 (GPIO 5) every 10 ms (100 Hz).
 * • **FIR task**: low-pass filters the 12-bit sample using a 20th-order
 *   (21-tap) Hamming-window FIR (fc = 0.25·fs).
 * • **PWM task**: converts the filtered 12-bit sample to a 13-bit LEDC duty
 *   at 8 kHz.
 * • **Logger task**: prints raw, filtered and calibrated millivolts.
 */

#include <stdio.h>
#include <inttypes.h>
```

```c
#include <math.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/queue.h"
#include "driver/adc.h"
#include "driver/ledc.h"
#include "esp_adc_cal.h"
#include "esp_log.h"
#include "esp_system.h"

#define TAG                 "ADC_PWM_FIR"

/* ---------------------------  GPIOs  ---------------------------- */
#define PWM_GPIO            7                       // LEDC output pin

/* ---------------------------  LEDC  ----------------------------- */
#define LEDC_TIMER          LEDC_TIMER_0
#define LEDC_MODE           LEDC_LOW_SPEED_MODE
#define LEDC_CHANNEL        LEDC_CHANNEL_0
#define LEDC_FREQ_HZ        8000                    // 8 kHz carrier
#define LEDC_DUTY_RES       LEDC_TIMER_13_BIT       // 0–8191

/* ---------------------------- ADC ------------------------------ */
#define ADC_UNIT_USED       ADC_UNIT_1
#define ADC_CHANNEL_USED    ADC1_CHANNEL_5          // GPIO6
#define ADC_ATTEN_USED      ADC_ATTEN_DB_11
#define ADC_SAMPLES_MS      10                      // 100 Hz sampling

/* -------------------------  FIR filter  ------------------------- */
#define FIR_TAPS 21
static const float fir_coeff[FIR_TAPS] = {
    0.0000000000f,  0.0036191610f,  0.0000000000f, -0.0122382502f,  0.0000000000f,
    0.0343155511f,  0.0000000000f, -0.0858292531f,  0.0000000000f,  0.3105830617f,
    0.4990994590f,  0.3105830617f,  0.0000000000f, -0.0858292531f,  0.0000000000f,
    0.0343155511f,  0.0000000000f, -0.0122382502f,  0.0000000000f,  0.0036191610f,
    0.0000000000f
};

/* ---------------------------- Types ---------------------------- */
typedef struct {
    uint16_t raw;       // 0–4095 (ADC)
    uint16_t filt;      // 0–4095 (FIR)
    uint32_t mv;        // millivolts
    uint16_t duty;      // 0–8191 (LEDC duty)
} adc_sample_t;

/* ----------------------  Queues / RTOS  ------------------------- */
static QueueHandle_t filt_queue;   // raw → FIR
static QueueHandle_t pwm_queue;    // filtered → PWM
static QueueHandle_t log_queue;    // struct → Logger

/* ---------------------------  ADC  ------------------------------ */
static esp_adc_cal_characteristics_t adc_chars;
```

```c
static void adc_sample_task(void *arg)
{
    (void)arg;
    while (1) {
        uint16_t raw = adc1_get_raw(ADC_CHANNEL_USED);     // 0-4095
        xQueueSend(filt_queue, &raw, 0);                   // to FIR
        vTaskDelay(pdMS_TO_TICKS(ADC_SAMPLES_MS));
    }
}

/* ----------------------  FIR filter task  ---------------------- */
static void fir_filter_task(void *arg)
{
    (void)arg;
    float ring[FIR_TAPS] = {0};
    size_t w = 0;

    uint16_t raw;
    while (xQueueReceive(filt_queue, &raw, portMAX_DELAY) == pdTRUE) {
        ring[w] = (float)raw;

        /* Convolution (direct-form): h · x  */
        float acc = 0.0f;
        size_t r = w;
        for (size_t k = 0; k < FIR_TAPS; k++) {
            acc += fir_coeff[k] * ring[r];
            r = (r == 0) ? (FIR_TAPS - 1) : (r - 1);   // circular buffer walk
        }
        w = (w + 1) % FIR_TAPS;

        /* Clip to ADC range */
        float clamped = fminf(fmaxf(acc, 0.0f), 4095.0f);
        uint16_t filt = (uint16_t)clamped;

        xQueueSend(pwm_queue, &filt, 0);                // to PWM

        /* Send to logger (non-blocking) */
        adc_sample_t sample = {
            .raw  = raw,
            .filt = filt,
            .mv   = esp_adc_cal_raw_to_voltage(raw, &adc_chars),
            .duty = (filt * ((1 << LEDC_DUTY_RES) - 1)) / 4095
        };
        xQueueSend(log_queue, &sample, 0);
    }
}

/* ----------------------  PWM output task  ---------------------- */
static void pwm_output_task(void *arg)
{
    (void)arg;
    uint16_t filt;
```

```c
    while (xQueueReceive(pwm_queue, &filt, portMAX_DELAY) == pdTRUE) {
        uint32_t duty = (filt * ((1 << LEDC_DUTY_RES) - 1)) / 4095;
        ledc_set_duty(LEDC_MODE, LEDC_CHANNEL, duty);
        ledc_update_duty(LEDC_MODE, LEDC_CHANNEL);
    }
}

/* --------------------------  UART logger  ------------------------ */
static void uart_logger_task(void *arg)
{
    (void)arg;
    adc_sample_t s;
    while (xQueueReceive(log_queue, &s, portMAX_DELAY) == pdTRUE) {
        ESP_LOGI(TAG, "raw:%4" PRIu16 " → filt:%4" PRIu16 " | %4" PRIu32 " mV |
duty:%4" PRIu16,
                 s.raw, s.filt, s.mv, s.duty);
    }
}

/* ----------------------  Peripheral setup  ---------------------- */
static void init_ledc(void)
{
    ledc_timer_config_t timer_conf = {
        .speed_mode       = LEDC_MODE,
        .duty_resolution  = LEDC_DUTY_RES,
        .timer_num        = LEDC_TIMER,
        .freq_hz          = LEDC_FREQ_HZ,
        .clk_cfg          = LEDC_AUTO_CLK
    };
    ledc_timer_config(&timer_conf);

    ledc_channel_config_t ch_conf = {
        .gpio_num       = PWM_GPIO,
        .speed_mode     = LEDC_MODE,
        .channel        = LEDC_CHANNEL,
        .timer_sel      = LEDC_TIMER,
        .duty           = 0,
        .hpoint         = 0
    };
    ledc_channel_config(&ch_conf);
}

static void init_adc(void)
{
    adc1_config_width(ADC_WIDTH_BIT_12);
    adc1_config_channel_atten(ADC_CHANNEL_USED, ADC_ATTEN_USED);
    esp_adc_cal_characterize(ADC_UNIT_USED, ADC_ATTEN_USED, ADC_WIDTH_BIT_12, 0,
&adc_chars);
}

/* --------------------------  app_main  ------------------------ */
void app_main(void)
{
```

```
    ESP_LOGI(TAG, "Starting ADC → FIR → PWM");

    init_ledc();
    init_adc();

    filt_queue = xQueueCreate(8, sizeof(uint16_t));
    pwm_queue  = xQueueCreate(8, sizeof(uint16_t));
    log_queue  = xQueueCreate(8, sizeof(adc_sample_t));

    /* Pinning strategy
     *  – ADC & FIR tasks on core 1 (low latency)
     *  – PWM task also on core 1 (shares queue, avoids cross-core hop)
     *  – Logger on core 0 (non-critical)
     */
    xTaskCreatePinnedToCore(adc_sample_task,  "adc_task",  2048, NULL, 6, NULL, 1);
    xTaskCreatePinnedToCore(fir_filter_task,  "fir_task",  3072, NULL, 5, NULL, 1);
    xTaskCreatePinnedToCore(pwm_output_task,  "pwm_task",  2048, NULL, 4, NULL, 1);
    xTaskCreatePinnedToCore(uart_logger_task, "uart_task", 2048, NULL, 3, NULL, 0);

    ESP_LOGI(TAG, "Setup complete (LEDC %u Hz, FIR %u taps)", LEDC_FREQ_HZ,
FIR_TAPS);
}
```