UNIVERSIDADE FEDERAL DE SANTA CATARINA

LINSE - LABORATÓRIO DE CIRCUITOS E PROCESSAMENTO DE SINAIS

Sistema de aquisição de áudio e classificação via Machine Learning para identificação de problemas em motores

João Paulo Vieira

I. Introdução

O presente sistema visa, utilizando microfones digitais e uma placa ESP32, realizar uma aquisição periódica de trechos de áudio de um motor em funcionamento, que serão armazenados num cartão de memória (SD) e posteriormente enviados a um servidor. Após o envio, esses arquivos poderão ser acessados por um PC, com um programa que realiza o download dos arquivos e passa-os por um modelo de Machine Learning (SVM), que classifica os áudios em 'normal' ou 'anormal', identificando uma possível falha no motor através do som produzido. A Figura 1 exibe o diagrama de blocos desse sistema.

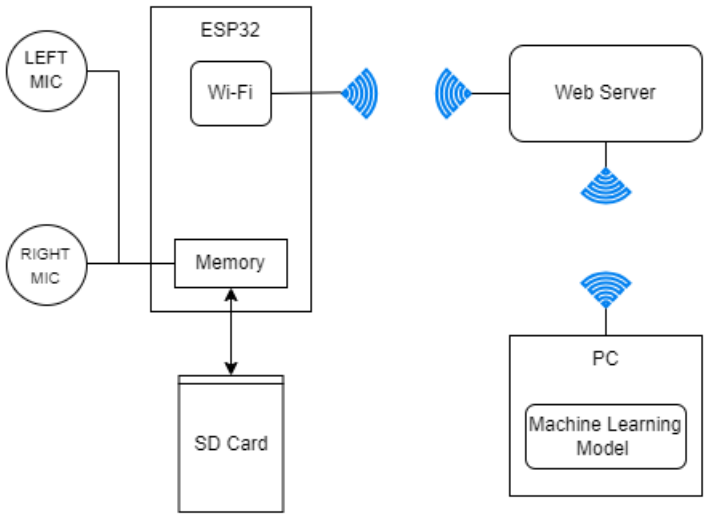

Figura 1: Diagrama de blocos do sistema.

II. Problema

Uma série de motores, no final da linha de montagem, passam por avaliações que desejam verificar a qualidade do produto. Um dos critérios utilizados é o ruído gerado pelo motor, sendo este alto ou baixo, e se soa de uma forma específica que indique algum tipo de mau funcionamento. Até então, existe uma implementação de solução que busca identificar tais problemas com base num algoritmo que extrai a FFT de um sinal de áudio da máquina operando e busca medir a potência de algumas componentes de frequência, dentre outras características. No entanto, observa-se que tal método, considerando que está restringido a poucos parâmetros, está sujeito a apresentar falhas na classificação.

III. Solução Proposta

Deseja-se, com esse sistema, utilizar um modelo de machine learning (SVM) treinado com diversas características sonoras extraídas das faixas de áudio, do tipo espectral, temporal, harmônico e rítmico, para ser capaz de classificar se o motor está em correto funcionamento. Para captar o áudio, utiliza-se um microfone digital (do tipo MEMS), com um protocolo de comunicação I2S. A ESP32 oferece suporte a esse protocolo, e permite um fluxo de dados adequado via DMA (em que as amostras são enviadas diretamente à memória). No entanto, considerando as limitações da memória interna do ESP32 utilizado (cerca de 3MB), optou-se por acoplar um cartão SD ao sistema, de modo que seja escrito no cartão conforme se adquire as amostras de áudio. Para transmitir o áudio para um servidor, utiliza-se a biblioteca HTTP, realizando-se HTTP Posts para a um servidor no Google Drive (através da API da Google). A partir disso, utiliza-se um programa no PC para baixar os arquivos do Drive, e classificá-los em normal ou anormal com o modelo.

IV. Descrição dos blocos

A seguir, será feita uma descrição detalhada de cada bloco que compõe o sistema, sendo esses divididos em três grupos: aquisição, envio do áudio ao servidor e classificação.

A. Aquisição

O processo de aquisição do sinal depende de realizar várias configurações, das quais serão descritas a seguir. Estão envolvidos, nesse bloco, os microfones, o cartão SD, e a ESP32 (incluindo seus módulos WiFi e I2S).

1) Data: Para realizar uma temporização adequada, e marcar o horário de cada gravação, é feito uso da biblioteca NTPClient, do qual requer conexão Wi-Fi. A linha de código da Figura 2 exemplifica como é feita a aquisição da data.

```
String date = timeClient.getFormattedDate();
```

Figura 2: Função para adquirir a data.

É necessário também configurar a biblioteca para a timezone brasileira.

2) Montagem do SD Card: Para montar o SD Card, usam-se as bibliotecas “FS”, “SPI” e “SD” disponíveis para a ESP32. As funções mountSDCARD e unmountSDCARD são utilizadas ao longo do código para montar e desmontar o cartão, quando necessário. A Figura 3 exibe as funções citadas.

```
/// mountSDCARD: monta o SD card no ESP32. -> Após isso, está pronto para uso.
bool mountSDCARD(){

  if(!SD.begin()){
    Serial.println("Card Mount Failed, restarting ESP.");
    return false;
  }

  uint8_t cardType = SD.cardType();

  if(cardType == CARD_NONE){
    Serial.println("No SD card attached");
    return false;
  }

  Serial.print("SD Card Type: ");

  if(cardType == CARD_MMC){
        Serial.println("MMC");
  } else if(cardType == CARD_SD){
        Serial.println("SDSC");
  } else if(cardType == CARD_SDHC){
        Serial.println("SDHC");
  } else {
        Serial.println("UNKNOWN");
  }
```

Figura 3: Funções mountSDCARD e unmountSDCARD.

3) Interface I2S: Inicialmente, configura-se a interface I2S através da função I2SMEMSAMPLER, ilustrada na Figura 4.

```
I2SSampler *input = new I2SMEMSSampler(I2S_NUM_0, i2s_mic_pins, i2s_mic_Config); // configura a interface I2S
```

Figura 4: Função I2SMEMSAMPLER, para realizar a configuração da I2S.

Nessa função, são passados parâmetros como os pinos em que estão conectados os microfones e outras configurações (presentes na variável i2s_mic_config). A seguir, na Figura 5, são dados os valores configurados nessa variável.

```
// i2s config for reading from I2S
i2s_config_t i2s_mic_Config = {
    .mode = (i2s_mode_t)(I2S_MODE_MASTER | I2S_MODE_RX),
    .sample_rate = SAMPLE_RATE,
    .bits_per_sample = I2S_BITS_PER_SAMPLE_32BIT,
    .channel_format = I2S_CHANNEL_FMT_RIGHT_LEFT,
    .communication_format = I2S_COMM_FORMAT_I2S_MSB,
    .intr_alloc_flags = ESP_INTR_FLAG_LEVEL1,
    .dma_buf_count = 32,
    .dma_buf_len = 1024,
    .use_apll = false,
    .tx_desc_auto_clear = false};
```

Figura 5: Struct i2s$_m ic_C onfig, contendo os parâmetros de configuração da I2S.$

Para o parâmetro '.mode', configura-se a ESP32 em modo receiver (para recepção de dados). É também definida a taxa de amostragem (16 kHz), o número de bits por amostra (32) e o formato do canal (I2S_CHANNEL_FMT_LEFT_RIGHT), configurado para operar em stereo. Através de diversos testes, constatou-se que é necessário utilizar um número grande de buffers (.dma_buf_count = 32) e um comprimento também longo para cada buffer (.dma_buf_len = 1024), pois caso não o seja feito, são geradas descontinuidades nas gravações (a ESP não consegue processar as amostras na velocidade desejada). Otimizaram-se tais valores de modo a garantir uma amostragem segura em 16 kHz, valor máximo que o sistema conseguiu atingir até então. Observou-se que aumentar o número de buffers levava a um overflow na memória de dados, então foi necessário controlar esse valor para que não ocorresse nenhuma falha.

4) Gravação: Para gravar o áudio, aloca-se dinamicamente na memória um buffer de 1024 posições de 64 bits (totalizando 8kiB), permitindo que a cada leitura sejam lidas 2048 amostras (1024 para cada canal). A Figura 6 exibe o trecho de código em que é declarado o buffer e inicia-se a comunicação I2S.

```
int buffer_size =  1024;
int64_t* samples = (int64_t *)malloc(sizeof(int64_t) * buffer_size); // buffer utilizado para gravação
input->start(); // Inicia o I2S
```

Figura 6: Declaração do buffer utilizado para armazenar as amostras adquiridas.

Após a alocação, cria-se o arquivo e é escrita uma header do formato WAV, de modo que o arquivo possa ser lido posteriormente. Para escrever a header, é utilizada a função writeWavHeader (Figura 7), que escreve no arquivo as variáveis da struct _wav_header, dada na Figura 8.

```
File fp = SD.open(fname, "wb");
int count_r = 0;  int mult = ceil(TEMPO*SAMPLE_RATE/buffer_size); // tempo de gravação

int fsize = mult * 2 * buffer_size * 2;
writeWavHeader(fp, fsize);
```

Figura 7: Declaração do arquivo de áudio e escrita da header WAVE.

```
typedef struct _wav_header
{
  // RIFF Header
  uint8_t riff_header[4]; // Contains "RIFF"
  int wav_size = 0;    // Size of the wav portion of the file, which follows the first 8 bytes. File size - 8
  uint8_t wave_header[4]; // Contains "WAVE"

  // Format Header
  uint8_t fmt_header[4];      // Contains "fmt " (includes trailing space)
  int fmt_chunk_size = 16; // Should be 16 for PCM
  short audio_format = 1;  // Should be 1 for PCM. 3 for IEEE Float
  short num_channels = 2;
  int sample_rate = 16000;
  int byte_rate = sample_rate* num_channels * 2;      // Number of bytes per second. sample_rate * num_channels * Bytes Per Sample
  short sample_alignment = 4; // num_channels * Bytes Per Sample
  short bit_depth = 16;       // Number of bits per sample

  // Data
  uint8_t data_header[4]; // Contains "data"
  int data_bytes = 0;  // Number of bytes in data. Number of samples * num_channels * sample byte size
  // uint8_t bytes[]; // Remainder of wave file is bytes
  _wav_header()
  {
    riff_header[0] = 'R';
    riff_header[1] = 'I';
    riff_header[2] = 'F';
    riff_header[3] = 'F';
    wave_header[0] = 'W';
    wave_header[1] = 'A';
    wave_header[2] = 'V';
    wave_header[3] = 'E';
    fmt_header[0] = 'f';
    fmt_header[1] = 'm';
    fmt_header[2] = 't';
    fmt_header[3] = ' ';
    data_header[0] = 'd';
    data_header[1] = 'a';
    data_header[2] = 't';
    data_header[3] = 'a';
  }
} wav_header_t;
```

Figura 8: Dados da header WAVE.

Nessa struct (Figura 8), são dadas informações sobre as faixas de áudio, como a taxa de amostragem, número de canais, taxa de bytes, alinhamento dos samples, e etc. Estão presentes, também, os caracteres necessários para a header (”RIFF”, ”WAVE”, ”fmt ”e ”data”).

Para realizar a leitura da porta I2S, utiliza-se a função I2SMEMSAMPLER::read, dada na Figura 9.

```
int I2SMEMSSampler::read(int64_t *samples, int count)
{
    size_t bytes_read = 0;
    i2s_read(m_i2sPort, samples, sizeof(int64_t) * count, &bytes_read, portMAX_DELAY);
    int samples_read = bytes_read / sizeof(int64_t);
    return samples_read;
}
```

Figura 9: Função para adquirir a data.

Nessa função. todos os dados lidos são salvos no buffer samples, definido na Figura 6. Para gravar as amostras lidas no cartão SD, executa-se a função ’read’ e são realizados deslocamentos nos bits das amostras originais, pois o microfone envia apenas 16 bits para cada amostra. Assim, os dados são escritos no cartão, como ilustra o código da Figura 10.

```
while (turn == 1)
{
  int samples_read = input->read(samples, buffer_size);
  uint8_t acquired_data[4];

  for(int i=0; i<samples_read;i++){
    acquired_data[0] = samples[i]>>48;
    acquired_data[1] = samples[i]>>56;
    acquired_data[2] = samples[i]>>16;
    acquired_data[3] = samples[i]>>24;
    fp.write(acquired_data,4);
  }

  count_r += samples_read;
  if (count_r >= mult*buffer_size){turn = 0; count_r = 0;}

}
```

Figura 10: Função de leitura das amostras.

Após a gravação, os ponteiros alocados são liberados da memória (de modo garantir que não haja overflow). Observou-se que durante a gravação, é necessário manter o WiFi desligado, pois há conflito entre as interrupções (I2S x WiFi).

B. Envio dos áudios ao servidor

Para realizar o envio das gravações, utiliza-se a ESP32 e o Google AppScript, plataforma que permite utilizar a API do Google Drive. Através da AppScript, é possível produzir um endereço para o qual a ESP32 pode enviar HTTP Gets e Posts, sendo assim possível realizar a transmissão dos dados. A seguir, serão descritos os códigos para as duas plataformas (ESP32 e AppScript).

1) Código ESP32: Nessa seção, será dada a descrição das funções utilizadas para realizar o envio dos áudios pela ESP32.

- **sendBinaryDataToAppScript:**
  Para realizar o envio do arquivo, utiliza-se a função ‘sendBinaryDataToAppScript’. Nela, inicialmente é feito um HTTP Get, enviando ao servidor o nome do arquivo e seu tamanho, para que posteriormente seja checado se o arquivo recebido equivale (em tamanho) ao que foi enviado. Para garantir um resultado positivo no Get, são feitas até 5 tentativas, e caso não seja possível enviar o filename, é encerrada a operação. Esse mesmo procedimento é também realizado para outras funções que dependem do WiFi, como os HTTP Posts, por onde são enviadas as amostras. A Figura 11 exibe as configurações inciais feitas para realizar o Get, em que a variável ’url’ representa o endereço fornecido para acessar a API.

```
HTTPClient http;
http.begin(url + "?filename=" + result + "&fileSize=" + String(fileSize) + "&check_filesize=false"  + "&delete_file=false");
int GETCODE = http.GET();  i = 0;
Serial.println("GET result code: " + String(GETCODE));
while((GETCODE != 302)) {
  GETCODE = http.GET(); i++;
  Serial.println("GET result code: " + String(GETCODE));
  if (i>= n_try) {Serial.println("Failed to send filename!"); return; }
}
http.end();
```

A seguir, na Figura 12, é definido um buffer (utilizado para armazenar os dados lidos do SDCARD), e define-se o tipo de dado a ser enviado (binário).

Figura 11: HTTP Get realizado para enviar o nome e tamanho do arquivo.

```
uint8_t * postbuffer = (uint8_t *)malloc(sizeof(uint8_t) * BUFFER_SIZE);
int bytesRead; int totalSent = 0;

http.setTimeout(20000); // Timeout elevado pois os buffers possuem 32kB.
http.begin(url);
http.addHeader("Content-Type", "application/octet-stream"); // Headerless binary files
```

Figura 12: Configurações para envio das amostras.

Após isso, são enviados os dados em pacotes de tamanho 32kiB, através de HTTP Posts, conforme o seguinte trecho de código da Figura 13.

```
// Leitura do arquivo:
while (file.available()) {

  bytesRead = file.read(postbuffer, BUFFER_SIZE);
  Serial.println(bytesRead);

  if (bytesRead > 0) {
    Serial.printf("Sending chunk %d - %d bytes...\n", totalSent, totalSent + bytesRead);
    bool success;
    for (int retry = 0; retry < NUM_RETRIES; retry++) {

      int httpResponseCode = http.POST((postbuffer), bytesRead);

      if ((httpResponseCode > 0)){
        Serial.printf("HTTP Response code: %d\n", httpResponseCode);
        String payload = http.getString();
        Serial.println(payload);
        //Serial.println("Sent chunk successfully.");
        success = true; break;
      } else {
        Serial.printf("Error sending data (Retry %d): %s\n", retry + 1, http.errorToString(httpResponseCode).c_str());
        delay(RETRY_DELAY);
      }

    }

    if (!success) {
      Serial.println("Failed to send buffer. Aborting transmission.");
      break;
    }

    totalSent += bytesRead;

  } else {

    Serial.println("Error reading file");
    break;
  }
}
```

Figura 13: Trecho que realiza consecutivamente os HTTP Posts.

Nesse código, é feita a leitura do arquivo, em que armazena-se o 32kiB de dados no buffer, e então é feito o envio. Caso o servidor retorne um erro, são feitas até 5 tentativas.
Por fim, libera-se o buffer da memória, fecha-se o arquivo, e desliga-se a conexão HTTP.

- **Checagem dos dados**
Com a função 'checkFileSize', envia-se um Get para a API de modo a identificar se o tamanho do arquivo salvo é igual ao tamanho do arquivo salvo no cartão SD, como ilusta a Figura 14.

```
String checkFileSizeWithAppsScript(const String& fname) {

  char result[100]; // You can adjust the size as per your requirement
  getSubstringFromSecond(fname, result);
  String urlGET = url + "?filename=" + result + "&fileSize=" + String(fileSize) + "&check_filesize=true"  + "&delete_file=false"; // Include the parameters in the URL
  //String urlGET = url + "?check_filesize=true"; // Include the parameters in the URL
  Serial.println(urlGET);
  HTTPClient http;
  http.begin(urlGET);
  http.setFollowRedirects(HTTPC_STRICT_FOLLOW_REDIRECTS);
  int httpCode = http.GET();
  Serial.println(httpCode);
  if (httpCode == 200 ) {
    String response = http.getString();
    http.end(); // Release resources and close the HTTP connection
    return response;
  } else {
    Serial.println("Error in HTTP GET request to Google Apps Script");
    http.end(); // Release resources and close the HTTP connection
    return ""; // Return an empty string to indicate an error
  }
}
```

Figura 14: Função para checar o tamanho do arquivo no Drive.

Nessa função, o parâmetro ’check_filesize’ presente na URL é definido como ’true’, o que fará com o que o servidor execute uma função para comparar os tamanhos dos arquivos. Caso a função retorne ’false’, é chamada a função deleteOldFile, que realiza outro Get, agora com um comando para deletar o arquivo (parâmetro delete_file = true). Desse modo, o programa tenta novamente enviar o arquivo, até um limite máximo de tentativas falhas. A FIgura 15 exibe a função deleteOldFile.

```
String deleteOldFile(const String& fname)
{
  char result[100]; // You can adjust the size as per your requirement
  getSubstringFromSecond(fname, result);
  String urlGET = url + "?filename=" + result + "&fileSize=" + String(fileSize) + "&check_filesize=false" + "&delete_file=true"; // Include the parameters in the URL

  HTTPClient http;
  http.begin(urlGET);
  http.setFollowRedirects(HTTPC_STRICT_FOLLOW_REDIRECTS);
  int httpCode = http.GET();
  Serial.println(httpCode);
  if (httpCode == 200 ) {
    String response = http.getString();
    http.end(); // Release resources and close the HTTP connection
    return response;
  } else {
    Serial.println("Error in HTTP GET request to Google Apps Script");
    http.end(); // Release resources and close the HTTP connection
    return ""; // Return an empty string to indicate an error
  }
}
```

Figura 15: Função para deletar o arquivo no Drive.

2) Código AppScript: O código em javascript possui duas funções: doGet e doPost, associadas as operações de GET e POST realizadas pela ESP32.

- **doGet:**
  Na função doGet, são enviados parâmetros a API, como o nome do arquivo a ser salvo, seu tamanho, e variáveis condicionais, para realizar a checagem do tamanho ou deletar o arquivo (na situação em que os tamanhos diferem). A seguir, na Figura 16, é dado o trecho inicial da função doGet, em que são salvos globalmente alguns parâmetros a serem utilizados no POST.

```
const DESTINATION_FOLDER_ID = 'XXXXX';

function doGet(e) {
  var filename = e.parameter.filename;
  var fileSize = e.parameter.fileSize;
  var checkFileSize = e.parameter.check_filesize;
  var deleteFile = e.parameter.delete_file;

  // Store the filename in Script Properties
  PropertiesService.getScriptProperties().setProperty("filename", filename);
  PropertiesService.getScriptProperties().setProperty("fileSize", fileSize);
```

Figura 16: Trecho da função doGet em que são salvos globalmente os valores de filename e fileSize.

Após isso, são checadas as condicionais mencionadas anteriormente, ou seja, se o código deve ou não comparar os tamanhos ou deletar o arquivo. A Figura 17 exibe o trecho de código para essas condicionais.

```
// Check if the checkFileSize parameter is set to true
if (checkFileSize && checkFileSize.toLowerCase() === "true") {

  var fileId = getFileIdByName(filename); // Get the file ID of the file with the provided filename
  if (fileId) {

    PropertiesService.getScriptProperties().setProperty("fileId", fileId);
    var fileSizeOnDrive = getFileSize(fileId);  // Get the file size of the file in Google Drive
    var isFileSizeMatching = (fileSizeOnDrive == fileSize); // Compare the file size from the ESP32 with the file size on Google Drive

    return ContentService.createTextOutput(isFileSizeMatching.toString());   // Return the result as a response
  } else {
    return ContentService.createTextOutput("false"); // File not found in Google Drive, return false

  }
}

if (deleteFile && deleteFile.toLowerCase() == "true") {

  var fileId = getFileIdByName(filename);
  var result = deleteFileById(fileId);

  var i = 0;

  while (result == "false") {
    i = i+1;
    result = deleteFileById(fileId);
    if (i>= 3) {
      break;
    }
  }

  return ContentService.createTextOutput(result);
}
else {
  return ContentService.createTextOutput("Request received successfully");
}
```

Figura 17: Trecho da função doGet para comparar o tamanho do arquivo com o valor da variável fileSize, ou deletar o arquivo (caso a comparação seja falsa).

- **doPost:**
  Na função doPost, são salvos os dados binários (chunks) enviados pela ESP. Na primeira execução, cria-se o arquivo com o filename enviado, e após isso os próximos chunks vão sendo adicionados ao arquivo. Utiliza-se o formato 'octet-stream', que representam dados binários sem header. A Figura 18 exibe a função doPost completa, contendo comentários para cada seção.

```
function doPost(e) {

  var filename = PropertiesService.getScriptProperties().getProperty("filename");
  var fileSize = PropertiesService.getScriptProperties().getProperty("fileSize");
  var contentBlob = e.postData;
  var bytes = contentBlob.getBytes();


  var folder = DriveApp.getFolderById(DESTINATION_FOLDER_ID); // Save the binary data to Google Drive
  var files = folder.getFilesByName(filename);

  // Check if the file already exists
  if (files.hasNext()) {
    var file = files.next();
    var fileId = file.getId();

    // Fetch the existing file content as a Blob
    var existingData = file.getBlob();

    // Append the new data to the existing file content
    var updatedData = concatUint8Arrays(existingData.getBytes(),bytes);
    Drive.Files.update({}, fileId, Utilities.newBlob(updatedData,"application/octet-stream"));  // Use the Advanced Drive Service to update the file content
  } else {
    // If the file doesn't exist, create a new one
    var blob = Utilities.newBlob(bytes, "application/octet-stream", filename);
    var file = folder.createFile(blob);
  }

  // Return a response to the ESP32
  var response = ContentService.createTextOutput("File saved successfully");
  response.setMimeType(ContentService.MimeType.TEXT);
  return response;
}
```

Figura 18: Função doPost.

São utilizadas também, funções extra, como concatUint8Arrays (Figura 19), que serve para concatenar os dados de forma adequada, e as funções getFileIdByName, getFileSize e deleteFileById (Figura 20), que obtêm o ID do arquivo, o tamanho, e deleta o arquivo.

```
function concatUint8Arrays(array1, array2) {
  var newArray = new Uint8Array(array1.length + array2.length);
  newArray.set(array1, 0);
  newArray.set(array2, array1.length);
  return newArray;
}
```

Figura 19: Função concatUint8Arrays.

```
function getFileIdByName(filename) {
  try {
    var files = DriveApp.getFilesByName(filename);
    if (files.hasNext()) {
      return files.next().getId();
    }
    return null; // Return null if the file is not found
  } catch (error) {
    return null; // Return null if an error occurs
  }
}

function getFileSize(fileId) {
  try {
    var file = DriveApp.getFileById(fileId);
    return file.getSize();
  } catch (error) {
    return -1; // Return -1 to indicate an error
  }
}

function deleteFileById(fileId) {
  try {
    Drive.Files.remove(fileId);
    Logger.log("File with ID '" + fileId + "' was successfully deleted.");
    return "true";
  } catch (error) {
    Logger.log("Error deleting file with ID '" + fileId + "': " + error);
    return "false";
  }
}
```

Figura 20: Funções getFileIdByName, getFileSize e deleteFileById.

C. *Classificação*

imgs/fig1a.png

((a)) Circuito 1) a) i).

imgs/fig1b.png

((b)) Circuito 1) a) ii).

Figura 21: Circuito para extração da potência de ruído disponível de um resistor.

imgs/fig1d.png

Figura 22