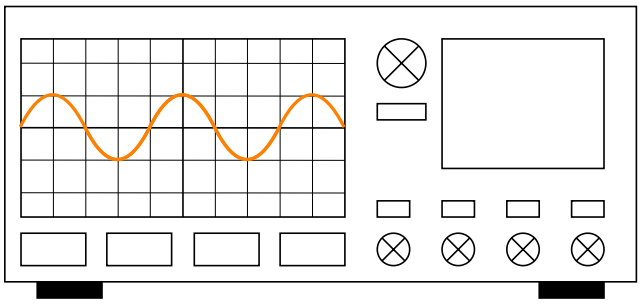
Engenharia Eletrotécnica e de Computadores

Sistemas Eletrónicos

µOscilloscope

3.º Trabalho de Laboratório



Autores:
João Gonçalves, ist199995
- jrazevedogoncalves@tecnico.ulisboa.pt
Teresa Nogueira, ist1100029
- maria.teresa.ramos.nogueira@tecnico.ulisboa.pt
Tatiana Delgado, ist1100089
- tatianadelgado@tecnico.ulisboa.pt

Docente:
Diogo Miguel Caetano

Grupo 64. Turno: terça-feira 14:30-16:00. Laboratório LPT.

Março 2024

I. Introdução

A) Enquadramento, Objetivos e Organização do Trabalho

Um sistema embebido (*embedded system*) é um conjunto de *hardware* e *software* concebido para executar uma tarefa específica, i.e., não tem por finalidade a computação genérica.

Em termos de componentes, um sistema embebido poderá consistir em:

1. *Sensores*: captam informações externas, convertendo-as em dados digitais posteriormente processados.
2. *Processadores*: responsáveis por processar os dados coletados pelos sensores e executar algoritmos que geram informação útil através destes dados.
3. *Portas I/O digitais*: permitem a comunicação do sistema com o mundo exterior, controlando dispositivos e recebendo informações de botões, interruptores e/ou outros sensores digitais.
4. *Portas série*: facilitam a comunicação com outros dispositivos e computadores, através de protocolos específicos para a troca de dados.
5. *Conversores analógico-digital* (ADCs): transformam sinais analógicos, como tensões e correntes, em valores digitais que podem ser processados pelo sistema.
6. *Conversores digital-analógico* (DACs): realizam a conversão inversa, i.e., transformam valores digitais em sinais analógicos que podem ser utilizados para controlar dispositivos externos.

A *Internet* das Coisas (*Internet of Things*, IoT) surge como um paradigma que conecta dispositivos físicos à *internet*, permitindo a troca de dados e a comunicação em tempo real.

Neste trabalho, apresentamos um sistema embebido que implementa um osciloscópio. O projeto µOscilloscope consiste no desenvolvimento de uma aplicação de *software* para implementar um pequeno osciloscópio através de *hardware* para desenvolvimento de soluções IoT. Os objetivos do trabalho podem ser resumidos em:

1. Domínio do circuito de *hardware* base do projeto.
2. Implementação do *software* do µOscilloscope.
3. Depuração e otimização do *software*.
4. Calibração o µOscilloscope.

O módulo IoT apresenta uma implementação de *software* compatível com Python 3, MicroPython®, otimizada para microcontroladores. O código apresentado é desenvolvido com este *subset* da linguagem em mente.

A linguagem de programação Python® é referenciada pela documentação oficial [1, 2], enquanto os métodos de processamento de sinais digitais discretos são baseados em Oppenheim e Schafer (2010) [3] e Beard (2004) [4].

É esperado que este trabalho prático complemente o estudo da:

- *Eletrónica dos sistemas embebidos*, i.e., a interface entre os microcontroladores e os componentes externos.

O que se segue neste documento é organizado da seguinte forma: na Secção II abordamos em detalhe o desenvolvimento do *software*. Segue-se, na Secção III, os resultados dos testes realizados no simulador. Na Secção IV discutimos os testes efetuados no laboratório. Por fim, apresentamos as conclusões na Secção V.

B) Descrição do Hardware Base

O µOscilloscope é um dispositivo IoT essencialmente composto por:

1. *Módulo IoT*: integra multiplos componentes, entre os quais se destaca a placa microcontroladora baseada num processador da família `ESP32`. Este processador opera a 240 MHz, possui 32 bits de largura e integra memória flash de 4 MB e SRAM de 560 kB. A placa microcontroladora também inclui três botões (um de *reset* e dois programáveis), diversas interfaces modulares, conectividade Wi-Fi e Bluetooth, bateria de iões de lítio (Li-ion) recarregável, porta USB Tipo-C para carregamento da bateria e transferência de *software*, ecrã TFT e uma placa de interface para entrada de sinais.
2. *Placa de interface de sinais e ADC*: o ADC de 12 bits permite a conversão de tensões de 0 V a 2 V numa palavra digital. Para aumentar a gama de medição, a placa de interface do módulo IoT realiza um *offset* de 1 V no terminal negativo do sinal de entrada e possui divisores resistivos selecionáveis por transístores. A escala utilizada (segunda), tem um fator de divisão de $^1/_{29.3}$, e permite leituras entre −30 V e 30 V.
3. *Ecrã TFT* (*thin-film transistor liquid-crystal display* de 1.14"): controlável através de uma biblioteca fornecida (`T_Display.py`), suporta cores e tem uma resolução de 240 × 135 pixels (`width` × `height`).

II. Desenvolvimento do Programa

Nesta secção, será abordado o desenvolvimento do programa[1], detalhando a sua estrutura (*top-down*), as funcionalidades implementadas e os métodos de desenvolvimento utilizados.

A) Estrutura do Programa

O cabeçalho do *script* tem como objetivo principal importar as bibliotecas necessárias para o funcionamento do programa[2] e inicializar um conjunto de variáveis globais que definem a representação gráfica dos dados no ecrã TFT. Esta escolha proporciona flexibilidade e acessibilidade global, i.e., as variáveis podem ser acedidas e modificadas por qualquer função, facilitando a manipulação das configurações gráficas em diferentes partes do código, sem a necessidade de passar parâmetros entre funções e evitando a utilização de "números mágicos".

```
#!/usr/bin/env python3
import sys, gc, cmath, math, T_Display

# Global variables
tft = T_Display.TFT()           # TFT display interface
width = 240                     # Max width (px)
height = 135                    # Max height (px)
x_div = 10                      # Number of horizontal divisions
x_range = [5, 10, 20, 50]       # Time scale (ms)
x_range_index = 0               # Time scale starting index
y_div = 6                       # Number of vertical divisions
y_range = [1, 2, 5, 10]         # Amplitude scale (V)
y_range_index = 1               # Amplitude scale starting index
```

O ciclo principal (*main loop*) do *script* define o comportamento do sistema em resposta aos cliques nos botões programáveis do módulo IoT, tal como descrito no guia laboratorial. Cada ação é mapeada para uma função específica, garantindo a modularidade e clareza do código.

```
if __name__ == "__main__":
    # Create an instance of the uOscilloscope object
    osc = uOscilloscope()

    # Main loop
    while tft.working():
        button = tft.readButton()
        if button != tft.NOTHING:
            print("Button pressed:", button)
            if button == 11:    # Fast click button 1
                osc.time_display()
            elif button == 12:  # Long click button 1
                osc.send_email()
            elif button == 13:  # Double click button 1
                osc.write_to_display()
            elif button == 21:  # Fast click button 2
                osc.change_x_scale()
            elif button == 22:  # Long click button 2
                osc.change_y_scale()
            elif button == 23:  # Double click button 2
                osc.freq_display()
            else: print("Invalid button key")
```

Cada botão mapeia as seguintes funcionalidades:

- ☞ Botão `(11)`: Reinicia o *display* e realiza uma nova leitura, representando a forma de onda no tempo.
- ☞ Botão `(12)`: Envia um email com os valores de tensão e uma mensagem para os endereços *hardcoded*.
- ☞ Botão `(13)`: Apaga o *display* e apresenta os valores de `avg`, `rms`, `min` e `max` do sinal de entrada atual.
- ☞ Botão `(21)`: Altera a escala vertical, passando para a escala imediatamente acima e de forma circular.
- ☞ Botão `(22)`: Altera a escala horizontal, passando para a escala imediatamente acima e de forma circular.
- ☞ Botão `(23)`: Calcula a transformada de Fourier e apresenta o espectro do sinal no domínio da frequência.

Todas as funções foram implementadas como métodos da classe `uOscilloscope` de forma a otimizar a organização e a legibilidade do código, bem como promover o encapsulamento (consequentemente, evitar conflitos) e melhorar a capacidade de teste e manutenção do *software*, como será discutido na secção que se segue.

[1] O ficheiro `main.py` pode ser encontrado na integra em anexo na pasta comprimida.
[2] Compatíveis com MicroPython®, como descrito na documentação oficial [aqui]. A biblioteca `T_Display` encontra-se por padrão em memória nos dispositivos e realiza a interface programável do módulo IoT (API para a aquisição, atualização do ecrã, etc).

B) Funcionalidades Principais do Programa

A classe `uOscilloscope` condensa um conjunto de funções do sistema embebido que implementa um osciloscópio, com exceção de rotinas *low level* que fazem a interface com o *hardware* do módulo IoT, como é o caso da biblioteca disponibilizada, `T_display`. Este trabalho, portanto, consistiu na implementação de diversos algoritmos que realizam as funcionalidades requeridas e na realização de uma interface de utilizador para o pequeno osciloscópio, como se apresenta em seguida.

Em primeira instância, utilizou-se o construtor da classe, `__init__()`, de forma a garantir a inicialização da tela do dispositivo IoT, a configuração das definições iniciais de exibição, e a representação do sinal analógico, à entrada do programa, com a criação do objeto criado a partir da classe.

```
16 class uOscilloscope:
17     """
18     This class provides essential functions for interacting with the uOscilloscope.
19     - Data Acquisition: Samples and processes analog signals from an ADC.
20     - Display and Visualization: Renders signal waveforms on a TFT display.
21     - Signal Analysis: Calculates essential signal statistics (maximum, minimum, average, and RMS values).
22     - Data Export: Sends measurement summaries and signal data via email.
23     - Scale Adjustments: Provides controls to modify the X-axis (time/freq.) and Y-axis (amplitude) scales.
24     - Frequency Analysis: Performs a Fourier transform to display the spectrum of the input signal.
25     """
26     def __init__(self):
27         # Init IoT display for the first time
28         self.time_display()
29 
30     ...
```

A função `time_display` é responsável pela representação do sinal no domínio do tempo. A flag `function_flag`, que indica o modo de exibição atual (tempo ou frequência), garante a correta disposição dos parâmetros no *display*. Consequentemente, a função `clear_display`, responsável por limpar o ecrã e imprimir o painel do osciloscópio, adapta-se ao modo selecionado, ajustando as escalas de acordo. Os cálculos de escala garantem que o sinal seja ajustado aos limites do *display*, considerando a amplitude e as divisões selecionadas.

```
127     def time_display(self):
128         x, y = [], []
129         self.function_flag = "time" # Sets display mode to "time"
130 
131         # Sampling and converting to the right range
132         self.read_samples()
133 
134         # Plot sampled values in the current X and Y scale
135         y_div_factor = y_range[y_range_index] * y_div / (height - 16)
136         x = list(range(len(self.amplitudes)))
137         y = [
138             round(max(0, min((height - 16) / 2 + value / y_div_factor, height - 16)))
139             for value in self.amplitudes
140         ]
141 
142         # Clear the display and print the plot
143         self.clear_display()
144         tft.display_nline(tft.YELLOW, x, y) # Display the plot
```

A função de aquisição de amostras foi desenvolvida de forma modular, o que permite que as restantes funções de representação operem com os dados mais recentes (`time_display`, `write_to_display`, `freq_display`). A atribuição da reta de conversão das amostras para amplitudes é condicionada ao ambiente de aquisição, i.e., módulo IoT ou simulação. No caso do módulo IoT, a reta de conversão depende da calibração prévia. Acrescenta-se que a conversão do nível digital para amplitude foi implementada com base numa função `lambda` de forma a manter o código conciso e permitir melhor interface com a lógica condicional.

```
36     def read_samples(self):
37         # Read raw ADC samples:
38         self.samples = tft.read_adc(240, x_range[x_range_index] * x_div)
39 
40         # Conditional conversion based on environment:
41         if sys.implementation.name == "micropython":  # Check if running on a MicroPython device
42             convert_sample = lambda sample: 0.0120 * sample - 24.06 # IoT 03.0003
43         else:
44             convert_sample = lambda sample: 0.0129 * sample - 26.62 # Simulator
45 
46         # Convert raw samples to amplitudes:
47         self.amplitudes = [convert_sample(sample) for sample in self.samples]
```

Conforme já mencionado, a função `clear_display` é responsável pela exibição das escalas e ícones no *display*. Ressalta-se a representação adicional da frequência do sinal (obtida pela função `estimated_frequency` através do método de *zero crossings* como veremos na secção II-C) e de um indicador de *out of scale*, ativo quando o sinal excede os limites do *display* para a escala atual.

```
88  def clear_display(self):
89      tft.display_set(tft.BLACK, 0, 0, width, height)                              # Erase display
90      tft.set_wifi_icon(width - 16, height - 16)                                   # Set wifi icon
91
92      # Conditional Behavior Based on Display Mode:
93      if self.function_flag == "time":                                             # Time Domain Display
94          tft.display_write_grid(0, 0, width, height - 16, x_div, y_div, True)     # Set grid
95          estimated_freq = round(self.estimate_frequency())
96
97          formated_strings = [    # Labels for the X and Y scales, and freq.
98              (f"{y_range[y_range_index]:02d}V/", 0),
99              (f"{x_range[x_range_index]:02d}ms/", 45),
100             (f"f={estimated_freq:03d} Hz", 100)
101         ]
102
103         # Print scales and signal frequency
104         for text, x_pos in formated_strings:
105             tft.display_write_str(tft.Arial16, text, x_pos, height - 16)
106         # Print signal out of range error
107         if (max(abs(value) for value in self.amplitudes) > y_range[y_range_index] * y_div / 2):
108             tft.display_write_str(tft.Arial16, "scale", 175, height - 16, tft.RED)
109
110     elif self.function_flag == "freq":                                           # Freq. Domain Display
111         tft.display_write_grid(0, 0, width, height - 16, x_div, y_div, False)    # Set grid
112
113         formated_strings = [    # Labels for the X and Y scales
114             (f"{y_range[y_range_index]/2:.1f}V/", 0),
115             (f"{round(1200/(x_range[x_range_index])):03d}Hz/", 50)
116         ]
117
118         # Print scales
119         for text, x_pos in formated_strings:
120             tft.display_write_str(tft.Arial16, text, x_pos, height - 16)
121         # Print signal out of range error
122         if (max(value for value in self.magnitudes) > y_range[y_range_index] * y_div / 2):
123             tft.display_write_str(tft.Arial16, "scale", 175, height - 16, tft.RED)
```

O envio dos emails com os pontos adquiridos e as métricas do sinal é desempenhado pela função `send_email`. A mensagem é enviada para os três endereços de email *hardcoded* de forma simultânea, conforme se segue.

```
145 def send_email(self):
146     # Sampling and converting to the right range
147     self.read_samples()
148
149     # Sample period and calculations on the sampled values
150     fs = 240 * 1000 / (x_range[x_range_index] * x_div)      # Sampling frequency (Hz)
151     estimated_freq = self.estimate_frequency()              # Estimate signal frequency
152     self.signal_metrics()
153
154     # Message formatting
155     stats = [self.max, self.min, self.avg, self.rms, estimated_freq]
156     labels = ["Vmax", "Vmin", "Vavg", "rms", "\nEstimated Freq"]
157     message = ", ".join(f"{label}: {value:.2f}" for label, value in zip(labels, stats))
158
159     # Prepare recipient list:
160     names = ["jrazevedogoncalves", "maria.teresa.ramos.nogueira", "tatianadelgado"]
161     emails = ",".join(f"{name}@tecnico.ulisboa.pt" for name in names)
162
163     tft.send_mail(1 / fs, self.amplitudes, message, emails)  # Send email
```

Por sua vez, a função `signal_metrics` é responsável por obter os valores de média (`avg`), valor eficaz (`rms`), mínimo (`min`) e máximo (`max`). O cálculo do valor eficaz segue a fórmula: $x_{\text{rms}} = \sqrt{(x_1^2 + x_2^2 + \cdots + x_n^2)/N}$.

```
48  def signal_metrics(self):
49      """
50      Calculates basic signal metrics including max, min, average and RMS from a list of signal samples.
51      """
52      self.max, self.min = max(self.amplitudes), min(self.amplitudes)
53      self.avg = sum(self.amplitudes) / len(self.amplitudes)
54      self.rms = (sum(value ** 2 for value in self.amplitudes) / len(self.amplitudes)) ** 0.5
```

A função `write_to_display` é responsável por apresentar os valores das métricas do sinal no display. Salienta-se a inclusão da frequência do sinal entre as informações exibidas.

```
164    def write_to_display(self):
165        tft.display_set(tft.BLACK, 0, 0, width, height) # Erase the display
166        tft.set_wifi_icon(width - 16, height - 16)      # Set wifi icon
167
168        # Sampling and converting to the right range
169        self.read_samples()
170
171        # Calculations on the sampled values
172        self.signal_metrics()
173        estimated_freq = self.estimate_frequency()
174
175        formated_strings = [ # Define data for display
176            (f"Vmax = {self.max:.2f}", 105),
177            (f"Vmin = {self.min:.2f}", 75),
178            (f"Vavg = {self.avg:.2f}", 45),
179            (f"Vrms = {self.rms:.2f}", 15)
180        ]
181
182        for text, y_pos in formated_strings: # Iterate through data and display it
183            tft.display_write_str(tft.Arial16, text, 20, y_pos)
184
185        tft.display_write_str(tft.Arial16, f"f = {estimated_freq:.2f} Hz", 130, 15) # Displays estimated freq.
```

O ajuste das escalas é realizada pelas funções `change_x_scale` e `change_y_scale`, de forma cíclica.

```
190    def change_x_scale(self):               # Cycles the horizontal scale to the next available option.
191        global x_range_index
192        x_range_index = x_range_index + 1
193        if x_range_index == len(x_range):   # Check if we reached the end of the list
194            x_range_index = 0               # Reset the index to wrap around
195        self.current_function()             # Update the display with the new scale
196
197    def change_y_scale(self):               # Cycles the vertical scale to the next available option.
198        global y_range_index
199        y_range_index = y_range_index + 1
200        if y_range_index == len(y_range):   # Check if we reached the end of the list
201            y_range_index = 0               # Reset the index to wrap around
202        self.current_function()             # Update the display with the new scale
```

A subsequente atualização do display depende do método `current_function`. Esta função depende da `function_flag` que garante que a representação atual (frequência ou tempo) utiliza a escala mais recente.

```
30     def current_function(self):             # Redirection function
31         if self.function_flag == "time":
32             self.time_display()
33         elif self.function_flag == "freq":
34             self.freq_display()
```

Por fim, o cálculo da transformada de Fourier para posterior representação do espectro do sinal, conforme o especificado no enunciado do trabalho é desempenhado pela função `dft` (*Discrete Fourier Transform*, DFT):

```
204 def dft(self):
205   """
206   Calculates the DFT of a signal. This implementation directly computes the DFT formula:
207   X[k] = sum_{n=0}^{N-1} x[n] * exp(-j * 2 * pi * k * n / N)
208   """
209   N = len(self.amplitudes), mag = [0.0] * N
210
211   for k in range(N//2 - 1): # Iterate over frequencies up to half the sample rate (Nyquist limit)
212     real, imag = 0, 0
213     for n in range(N):                                  # Iterate over signal samples
214       theta = -k * (2 * math.pi) * (float(n) / N)       # Calculate the phase angle
215       real += self.amplitudes[n] * math.cos(theta)      # Accumulate real component
216       imag += self.amplitudes[n] * math.sin(theta)      # Accumulate imaginary component
217
218     magnitude = abs(complex(real, imag)) / N            # Calculate magnitude from complex number
219     magnitude *= 2 if k != 0 else 1                     # Double for non-zero frequencies (symmetry)
220
221     mag[2 * k] = magnitude                              # Store magnitudes in pairs for efficient plotting
222     mag[2 * k + 1] = magnitude
223
224   return mag
```

A função `freq_display` é responsável pela representação do espectro do sinal amostrado. Em termos de estrutura, assemelha-se à função responsável pela representação do andamento temporal do sinal, efetuando os cálculos de escala (agora no domínio da frequência e com escala de amplitude dupla da escala temporal) e *setup* do ecrã. Salienta-se apenas que o cálculo da transformada de Fourier do sinal depende do método implementado selecionado (`dft` ou `czt`), conforme adiantamos na secção seguinte.

```
331     def freq_display(self):
332         x, y = [], []
333         self.function_flag = "freq"
334 
335         # Sampling and converting to the right range
336         self.read_samples()
337 
338         # Method to calculate the frequency domain representation of the signal
339         method = "czt"
340         if method == "dft":
341             self.magnitudes = self.dft()
342         elif method == "czt":
343             self.magnitudes = self.czt()
344 
345         # Display the frequency representation in the current scales
346         y_div_factor = (y_range[y_range_index] * y_div) / (2 * (height - 16))
347         x = list(range(len(self.magnitudes)))
348         y = [
349             round(max(0, min(value / y_div_factor, height - 16)))
350             for value in self.magnitudes
351         ]
352 
353         # Clear the display and print the plot
354         self.clear_display()
355         tft.display_nline(tft.MAGENTA, x, y)
```

C) Funcionalidades Adicionais

A DFT é o método padrão para calcular a representação de um sinal no domínio da frequência, com uma complexidade de $\mathcal{O}(N^2)$. A *Fast Fourier Transform* (FFT) surge como um algoritmo otimizado para a DFT, oferecendo uma vantagem significativa em termos de tempo de processamento, especialmente para conjuntos de dados extensos, já que a sua complexidade computacional é de $\mathcal{O}(N \log N)$.

A principal restrição da FFT reside no tamanho da entrada, que deve ser uma potência de dois para garantir máxima eficiência. A *Chirp Z-Transform* (CZT) [5] é uma generalização da DFT. Enquanto a DFT amostra o plano Z em pontos uniformemente espaçados na circunferência unitária, a CZT amostra ao longo de arcos espirais no plano Z, correspondentes a linhas retas no plano S. Consequentemente:

- A CZT oferece melhor resolução de frequência em relação à FFT.
- O número de amostras, N, não tem que ser igual ao número de amostras da transformada, M, embora o caso especifico da DFT admita $M = N$.
- M e N não necessitam de ser inteiros compostos, nem potências de dois.

Para motivar a CZT, consideremos a expressão da DFT:

$$X_k = \sum_{n=0}^{N-1} x_n \left(e^{-j2\pi/N}\right)^{kn}, \quad k = 0, \ldots, N-1.$$

Considerando a identidade algébrica de Bluestein,

$$kn = \frac{1}{2}\left(k^2 + n^2 - (k-n)^2\right),$$

é possível chegar à expressão da CZT para o caso da transformada de Fourier:

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n \left(e^{-j2\pi/N}\right)^{\frac{1}{2}(k^2+n^2-(k-n)^2)} \\ &= \left(e^{-j2\pi/N}\right)^{\frac{1}{2}k^2} \cdot \sum_{n=0}^{N-1} \left[x_n \left(e^{-j2\pi/N}\right)^{\frac{1}{2}n^2} \left(e^{-j2\pi/N}\right)^{-\frac{1}{2}(k-n)^2} \right]. \end{aligned}$$

Onde o *chirp* é a componente $\left(e^{-j2\pi/N}\right)^{\frac{1}{2}k^2}$. A implementação deste algoritmo pode ser encontrada em anexo.

Na prática, este algoritmo é implementado com a FFT e a sua inversa (IFFT) de forma a otimizar a operação de convolução. Para valores de M e N suficientemente grandes, o tempo de computação torna-se aproximadamente proporcional a $(N+M)\log(N+M)$ (consequentemente $N \log N$ visto que $N = M$). Esta relação contrasta com a complexidade quadrática da avaliação direta da DFT, evidenciando uma otimização temporal significativa. As Figuras 1 e 2 demonstram a evolução dos dois algoritmos para diferentes tamanhos de N.

```
f =
     Linear model Poly2:
     f(x) = p1*x^2 + p2*x + p3
     Coefficients (with 95% confidence bounds):
       p1 =   2.297e-06  (1.693e-06, 2.9e-06)
       p2 =  -7.825e-05  (-0.00132, 0.001164)
       p3 =    -0.03699  (-0.5764, 0.5024)
```

```
f =
     General model:
     f(x) = a*x*log(x) + b
     Coefficients (with 95% confidence bounds):
       a =   5.884e-06  (4.718e-06, 7.049e-06)
       b =    0.008971  (-0.0006641, 0.01861)
.
```

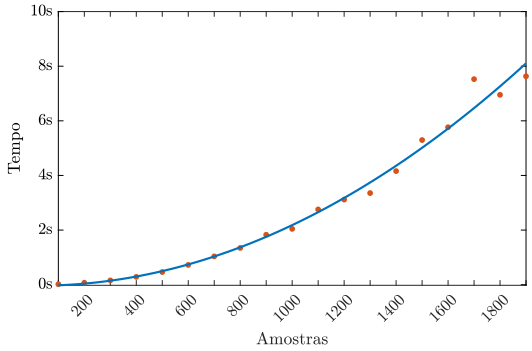


Fig. 1: Evolução temporal da DFT.
$\mathcal{O}(N^2)$

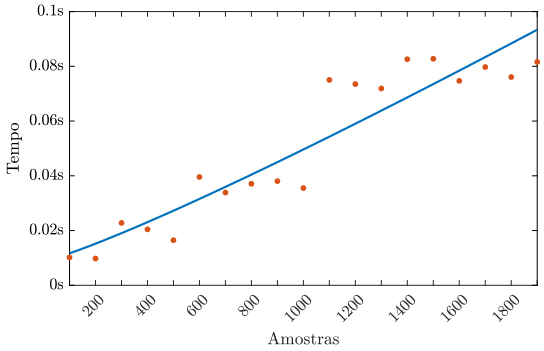


Fig. 2: Evolução temporal da CZT.
$\mathcal{O}(N \log N)$

Para obter as curvas de complexidade utilizámos a ferramenta `fit` do MATLAB® para os modelos explicitados. Note-se que para o tamanho do sinal amostrado em prática neste trabalho ($N = 240$), existe uma relação de cerca de duas ordens de grandeza na resposta temporal entre os dois algoritmos para o processador utilizado[3].

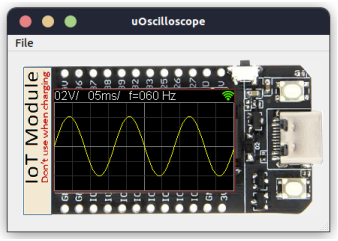
Outra funcionalidade suplementar adicionada ao `uOscilloscope` foi a capacidade de estimar a frequência do sinal amostrado, através de um simples algoritmo (com pequena *footprint*) que calcula as *zero crossings*.

```
    def estimate_frequency(self):
        """
        Estimate frequency by counting zero crossings.
        Works well for long low-noise sines, square, triangle, etc
        """
        fs = 240 * 1000/(x_range[x_range_index] * x_div) # Sampling rate [S/s]

        # Remove the DC offset from the signal
        self.avg = sum(self.amplitudes) / len(self.amplitudes)
        x = [value - self.avg for value in self.amplitudes]

        # Detect both rising and falling edges
        indices = []
        for i in range(1, len(x)):
            if (x[i] >= 0 and x[i-1] < 0) or (x[i] <= 0 and x[i-1] > 0):
                indices.append(i)

        if len(indices) < 2:
            return 0  # Not enough edges to determine frequency

        # Linear interpolation to find intersample
        crossings = [i - x[i] / (x[i] - x[i-1]) for i in indices[:-1]]
        diff = []
        previous = None
        for value in crossings:
            if previous is not None:
                diff.append(value - previous)
            previous = value

        # Check for empty diff array
        if not diff:
            return 0 # No valid frequency calculation possible

        return fs / (2 * sum(diff) / len(diff))
```

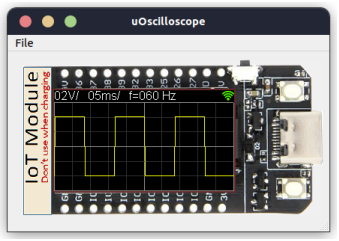
[3] De facto, a arquitetura e *clock speed* do processador desempenham um fator preponderante na resposta temporal de qualquer computação. No entanto, estes resultados servem apenas para visualizar as proposições teóricas da complexidade dos algoritmos.

III. Testes no Simulador

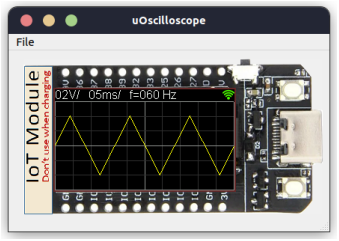
Esta secção condensa a discussão essencial do funcionamento do µOscilloscope em regime de simulação. Salienta-se que a função de conversão do nível digital das amostras para amplitudes foi obtida diretamente por inspeção do ficheiro `T_Simulator.py` disponibilizado (que é chamado na classe `TFT` presente na biblioteca `T_Display.py` quando o ambiente MicroPython® não é detetado). As seguintes imagens visam demonstrar a correta operação das funcionalidades do programa:



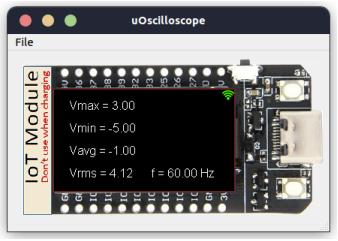
(a) Onda sinusoidal 60Hz, 4V.



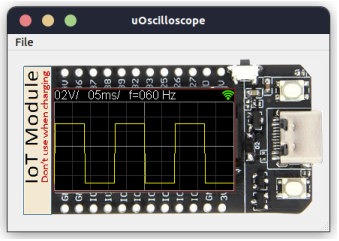
(b) Onda quadrada 60Hz, 4V.



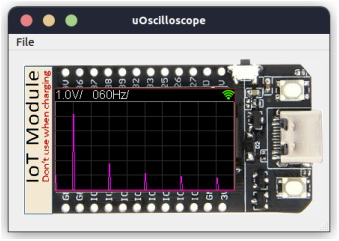
(c) Onda triangular 60Hz, 4V.



(d) Onda quadrada 60Hz, 4V, -1Vdc: `avg`, `rms`, `min`, `max` e freq.



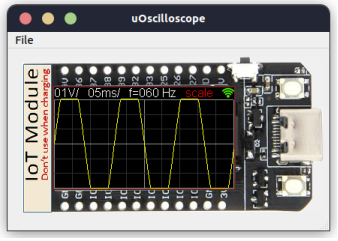
(e) Onda quadrada 60Hz, 4V, -1Vdc: Vizualização no tempo.



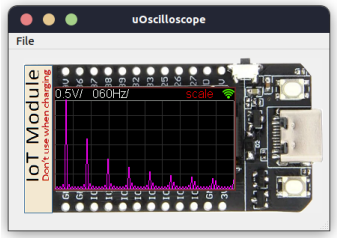
(f) Onda quadrada 60Hz, 4V, -1Vdc: Vizualização na frequência.

Fig. 3: Visualização do *output* no *display* simulado do osciloscópio. Para garantir consistência visual, utilizou-se uma escala vertical de 2V/div e uma escala horizontal de 5ms/div para as diversas formas de onda. A transformada é exibida com escala vertical de 1V/div e escala horizontal de 60Hz/div, e obtida com o algoritmo CZT.

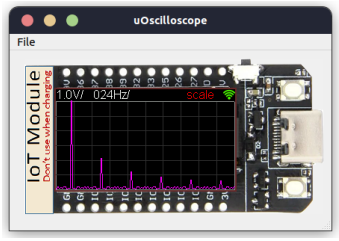
Destaca-se o indicador *out of scale* presente nas três figuras seguintes, onde o sinal excede os limites da escala vertical na visualização temporal e espectral.



(a) Onda sinusoidal 60Hz, 5V: *Out of scale.*



(b) Onda quadrada 35Hz, 4V: Espalhamento (*leakage*), $f_S/f \in \mathbb{Q}$.

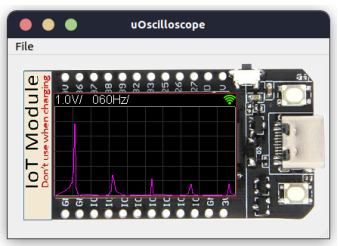


(c) Onda quadrada 500Hz, 4V: Espelhamento (*aliasing*), $f > f_S/2$.

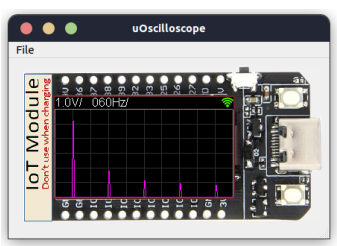
Fig. 4: Visualização do funcionamento do indicador *out of scale*.

Ressalta-se que as duas últimas visualizações do espectro escolhidas permitem visualizar: (4b) o fenómeno de espalhamento espectral (*spectral leakage*), uma vez que a frequência do sinal de entrada (35 Hz) não é múltiplo inteiro da frequência de amostragem (1200 Hz); (4c) o espelhamento espectral (*folding* ou *aliasing*), uma vez que o Teorema da Amostragem não é cumprido (neste cenário $f = 500\,\text{Hz}$ e $f_S = 480\,\text{Hz}$), verificando-se, portanto, uma frequência aparente de $500\,\text{Hz} - 480\,\text{Hz} = 20\,\text{Hz}$.

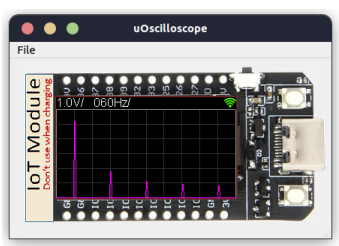
É ainda crucial analisar a visualização do resultado dos três algoritmos de cálculo da transformada:



(a) Onda quadrada 60Hz, 4V: FFT.



(b) Onda quadrada 60Hz, 4V: DFT.



(c) Onda quadrada 60Hz, 4V: CZT.

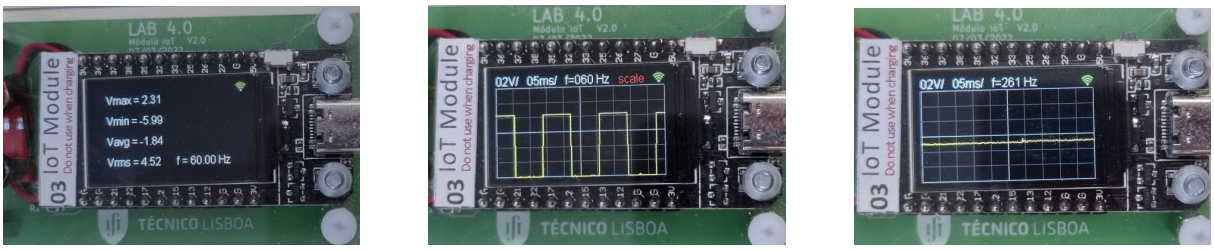
Fig. 5: Visualização do sinal no domínio da frequência para três algoritmos diferentes.

Para realizar a FFT do sinal de entrada, com tamanho de 240 amostras, foi efetuado um *zero padding* até à próxima potência de dois, 256. O *zero padding* quebra artificialmente a periodicidade do sinal, introduzindo transições bruscas nas suas fronteiras. Isto causa dispersão de energia para frequências adjacentes, expandindo e distorcendo as harmónicas do espectro. Em contraste, a CZT obtém um espectro com resolução idêntica à da DFT, preservando a mesma complexidade temporal da FFT [5].

IV. Testes no Laboratório

Após a exploração do funcionamento do µOscilloscope em regime de simulação, esta secção dedica-se à descrição e análise dos testes realizados em ambiente de laboratório. A transição do ambiente simulado para o real implica a necessidade de calibração dos instrumentos e o contacto com variáveis aleatórias, como veremos em seguida. Os ensaios em ambiente laboratorial não se destinam unicamente a verificar a funcionalidade do dispositivo em condições práticas, mas também à identificação de possíveis discrepâncias entre os resultados simulados e os obtidos em condições reais.

Começámos por analisar a resposta do osciloscópio a uma onda quadrada de frequência 60 Hz com 4 V de amplitude ($V_{\text{pico-pico}} = 8\,\text{V}$) e tensão DC de $-1\,\text{V}$, sem realizar qualquer alteração ao código utilizado em regime de simulação, conforme apresentado na Figura 6.



(a) Métricas do sinal: `avg`, `rms`, `min`, `max` e freq.

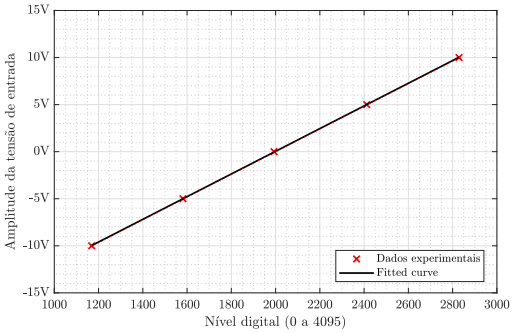
(b) Vizualização no tempo: *Out of scale*.

(c) Entrada 0Vdc: *Offset* de conversão.

Fig. 6: Visualização da onda quadrada antes da calibração e *offset* DC extra para uma entrada aproximadamente nula (0 V).

A ausência de calibração traduz-se num *offset* que descentra os sinais e em distorção (pouco visível[4]). No caso da onda quadrada, o valor médio, que deveria ser $-1\,\text{V}$, aproxima-se de $-2\,\text{V}$; por outro lado, a entrada nula, que deveria estar centrada na origem da *grid* do osciloscópio, apresenta um deslocamento em relação ao centro.

O processo de calibração do µOscilloscope consistiu na aquisição de valores de tensão DC conhecidos ($-10\,\text{V}$, $-5\,\text{V}$, $0\,\text{V}$, $5\,\text{V}$ e $10\,\text{V}$) e na conversão em níveis digitais, através do *script* auxiliar fornecido, `main_exemplo2.py`, que, após carregado no dispositivo IoT, realizou a média de 100 amostras para um intervalo de 50 ms. Em seguida, com os dados experimentais, foi realizada uma análise em MATLAB® com base no modelo de regressão linear `fitlm`, cujo resultado[5] pode ser visualizado na Figura 7.




```
f1 =

    Linear regression model:
        y ~ 1 + x1

    Estimated Coefficients:
                           Estimate        SE          pValue
                           ________    __________    __________

        (Intercept)         -24.059      0.053085    2.3687e-08
        x1                 0.012049    2.5506e-05    2.0919e-08

    Number of observations: 5, Error degrees of freedom: 3
    Root Mean Squared Error: 0.0335
    R-squared: 1,  Adjusted R-Squared: 1
    F-statistic vs. constant model: 2.23e+05, p-value = 2.09e-08
```

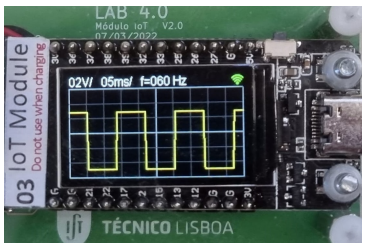
Fig. 7: Reta de regressão linear que relaciona os valores das amplitudes medidas com os níveis digitais do conversor.

Os resultados após a calibração podem ser visualizados na Figura 8. Note-se que, nesta configuração, a tensão DC de *offset* extra previamente mencionada (consequência do uso da mesma função de conversão das amostras utilizada em regime de simulação) e as pequenas distorções observadas, são bastante menos pronunciadas.

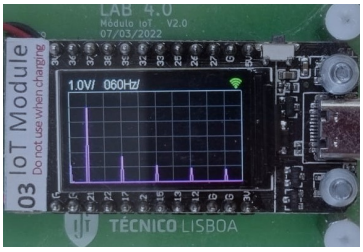
[4] Talvez devido à semelhança entre a função de conversão do nível digital para amplitude antes e após a calibração (os declives apresentam um erro simulação-experimental de cerca de 5%), onde a maior discrepância se verifica na constante de interceção com o eixo vertical das duas reta (erro de cerca de 10%).

[5] O módulo utilizado em laboratório foi o `IoT 03.0003`.

(a) Métricas do sinal.



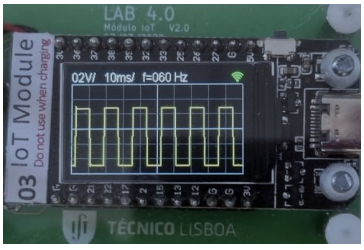
(b) Vizualização no tempo.



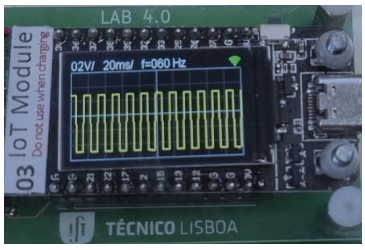
(c) Visualização do espectro.

Fig. 8: Visualização da onda quadrada com 4 V de amplitude ($V_{\text{pico-pico}} = 8\,\text{V}$) e tensão DC de $-1\,\text{V}$, depois da calibração.

Na Figura 9 é possível observar a mesma onda quadrada para diferentes escalas horizontais.



(a) Escala horizontal a 10 ms/div.



(b) Escala horizontal a 20 ms/div.

Fig. 9: Continuação da visualização temporal da onda quadrada com 4 V de amplitude e tensão DC de $-1\,\text{V}$, depois da calibração.

Por fim, acrescenta-se que a funcionalidade de enviar as métricas do sinal amostrado por email se verificou regular em regime laboratorial, enquanto em regime de simulação ocasionalmente se encontrava fora de serviço.

V. Conclusões

Os resultados obtidos demonstram que o osciloscópio desenvolvido é capaz de capturar e visualizar formas de onda bem como as respetivas representações espectrais com boa precisão. A calibração do sistema foi fundamental para aumentar a exatidão das medidas, eliminando possíveis distorções que comprometem a fidelidade e a confiabilidade das representações em causa.

Os objetivos do trabalho foram plenamente atingidos, tendo sido possível:

1. *Dominar o circuito de hardware base do projeto*: Através do estudo do guia e do subsequente contacto em meio laboratorial, foi adquirido um conhecimento profundo do *hardware* utilizado, nomeadamente da interface entre o microcontrolador e os componentes externos.
2. *Implementar o software do µOscilloscope*: O *software* foi desenvolvido em Python® (restrito ao ambiente MicroPython®). A depuração e otimização do *software* foram realizadas através de testes em simulador e em ambiente laboratorial.
3. *Aplicar conhecimento sobre processamento de sinais digitais*: Através de algoritmos para a captura e análise de sinais, onde se destacam a DFT, a FFT, a CZT, o método de estimação de frequência por *zero crossings*, bem como conceitos como o espelhamento, espalhamento e o Teorema da Amostragem.
4. *Calibrar o Osciloscópio*: A calibração do osciloscópio foi realizada em laboratório, ajustando o *software* ao *hardware* específico utilizado, com base em ferramentas computacionais.

Conclui-se que a execução e resultados deste projeto consolidam o entendimento e aplicação dos conceitos de sistemas embebidos e *internet* das coisas, através da conceção e ajuste do µOscilloscope. Demonstrou-se, assim, a correlação entre o desempenho teórico esperado e os resultados experimentais obtidos, enfatizando a eficácia das técnicas de processamento de sinais digitais.

Referências

[1] Python Software Foundation, "Python Documentation v3.9," last accessed in 2024-03-28.

[2] D. George, P. Sokolovsky, et al., "MicroPython Documentation (latest)," last accessed in 2024-03-28.

[3] A. Oppenheim and R. Schafer, *Discrete-Time Signal Processing*, 3rd ed. Prentice Hall, 2010, ch. 8–10.

[4] J. Beard, *The FFT in the 21st Century: Eigenspace Processing*, 1st ed. Springer, 2004, pp. 33–81, ch. 2–3.

[5] L. Rabiner, R. Schafer, and C. Rader, "The Chirp Z-Transform Algorithm," *IEEE Transactions on Audio and Electroacoustics*, vol. 17, pp. 86–92, 1969.

Anexo

i) Implementação do Algoritmo CZT

A CZT pode ser entendida como um processo de três etapas [5]:

1. Criar uma nova sequência y_n a partir de x_n, de acordo com a equação:

$$y_n = x_n A^{-n} W^{n^2/2}, \quad n = 0, \ldots, N-1.$$

2. Realizar a convolução da sequência y_n com a sequência v_n definida como $v_n = W^{-n^2/2}$, de forma a obter uma sequência g_k, onde

$$g_k = \sum_{n=0}^{N-1} y_n v_{k-n}, \quad k = 0, \ldots, M-1.$$

3. Multiplicar g_k por $W^{k^2/2}$ de forma a obter X_k:

$$X_k = g_k W^{k^2/2}, \quad k = 0, \ldots, M-1.$$

Onde, para o caso especifico do cálculo da DFT, admitimos:

$$A = 1, \quad W = e^{-j2\pi/N}, \quad M = N.$$

```python
    def czt(self):
        """
        Calculates the Chirp Z-Transform.
        """
        n = len(self.amplitudes)
        m = n                               # Number of output points
        w = cmath.exp(-2j * cmath.pi / m) # Ratio between successive points on the spiral contour
        a = 1                               # Starting point on the Z-plane

        # Precompute chirp values
        chirp = [w**(i**2 / 2.0) for i in range(1 - n, max(n, m))]

        # Zero padding and preparation for FFTs
        N2 = int(2 ** math.ceil(math.log2(m + n - 1))) # Next power of 2

        dummy = chirp[n - 1 : n + n - 1]
        xp = [self.amplitudes[i] * a ** -i * dummy[i] for i in range(n)]    # yn
        xp = xp + [0.0] * (N2 - n)
        del dummy

        ichirpp = [1/i for i in chirp[: m + n - 1]]
        ichirpp = ichirpp + [0.0] * (N2 - (m + n - 1))

        # Calculate FFTs
        fft_xp, fft_ichirpp = self.fft(xp), self.fft(ichirpp)
        del xp, ichirpp
        gc.collect()

        # Convolution in frequency domain becomes a product
        k = [fft_xp[i] * fft_ichirpp[i] for i in range(N2)]
        r = self.ifft(k)                                                    # gk
        r = [v / len(r) for v in r]
        r = r[n - 1 : m + n - 1]
        del k, fft_xp, fft_ichirpp
        gc.collect()

        # Scale and adjust phase based on chirp
        chirp = chirp[n - 1 : m + n - 1]
        result = [r[i] * chirp[i] for i in range(len(r) // 2)]              # Xk
        del chirp
        gc.collect()

        # Grouping values two by two and magnitude scaling with normalization
        mag = [0.0] * 2 * len(result)
        for i in range(m // 2):
            mag[2 * i] = result[i]
            mag[2 * i + 1] = mag[2 * i]

        mag = [abs(value) / len(mag) for value in mag]
        for i in range(len(mag)):
            mag[i] *= 2 if i>1 else 1

        return mag
```

A FFT e a sua inversa tomam uma implementação clássica como se apresenta em seguida:

```
233     def fft(self, x):
234         """
235         A recursive implementation of the 1D Cooley-Tukey FFT, the input should have a length of power of 2.
236         """
237         N = len(x)
238 
239         if N == 1:
240             return x
241         else:
242             X_even = self.fft(x[0:N:2])
243             X_odd = self.fft(x[1:N:2])
244 
245             out = [0.0] * N
246             for k in range(N//2):
247                 out[k] = X_even[k] + self.exp(N, -k) * X_odd[k]
248                 out[k + N // 2] = X_even[k] - self.exp(N, -k) * X_odd[k]
249 
250         return out
```

```
252     def ifft(self, x):
253         """
254         Implements the 1D Inverse Fast Fourier Transform (IFFT). The input should have a length of power of 2.
255         """
256         N = len(x)
257 
258         if N == 1:
259             return x
260         else:
261             X_even = self.ifft(x[0:N:2])
262             X_odd = self.ifft(x[1:N:2])
263 
264             out = [0.0] * N
265             for k in range(N//2):
266                 out[k] = X_even[k] + self.exp(N, k) * X_odd[k]  # Conjugate twiddle factor
267                 out[k + N // 2] = X_even[k] - self.exp(N, k) * X_odd[k]
268 
269             return out
```