Universidade Federal de Santa Catarina
Centro Tecnológico
Programa de Pós-Graduação em Engenharia Elétrica
Discentes: Carlos André Antunes, João Lucas Pereira dos Santos de Paula e Shirley Karolina da Silva Ferreira

Processamento de Sinais Biomédicos (T2)

Considerações Iniciais

Para o desenvolvimento deste trabalho, foi utilizada a linguagem de programação Python e a ferramenta de desenvolvimento Colab.

Considerando os sinais dos canais X, Y e Z e o sinal de referência, mostrados na Figura 1.

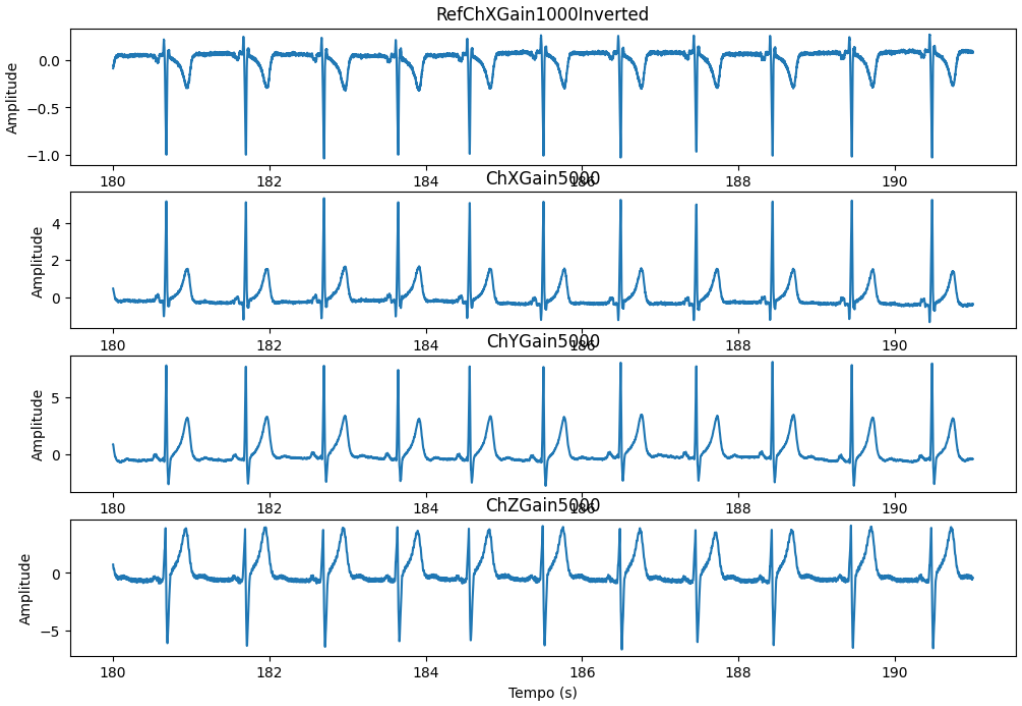

Figura 1 – Sinais brutos de ECG.

Problema 1: Interpolação dos Sinais

Como primeira etapa foi realizada a interpolação do sinal para uma $F_s = 4000$ amostras por segundo. Como visto na Figura 1, o sinal de referência está invertido, então foi necessário multiplicá-lo por -1. Foram criadas as operações de interpolação para cada canal mais o canal de referência, conforme mostrado a seguir.

```
interp_funcao_x = interp1d(tempo, amplitude_x, kind='linear')
interp_funcao_y = interp1d(tempo, amplitude_y, kind='linear')
interp_funcao_z = interp1d(tempo, amplitude_z, kind='linear')
interp_funcao_ref = interp1d(tempo, sinal_referencia, kind='linear')
```

Dada a interpolação, teremos novas variáveis como a frequência de amostragem e o tempo, agora reamostrado.

```
nova_frequencia_amostragem_4000 = 4000 # 1200 Hz
# Gerar novos pontos de tempo para a nova frequência de amostragem
tempo_reamostrado_4000Hz = np.arange(tempo[0], tempo[-1],
↪  1/nova_frequencia_amostragem_4000)
```

Uma vez com as novas variáveis criadas, as operações de interpolação foram aplicadas e posteriormente um *dataframe* foi criada com os sinais interpolados:

```
# Interpolar para obter os novos valores de amplitude
amplitude_reamostrada_4000Hz_x = interp_funcao_x(tempo_reamostrado_4000Hz)
amplitude_reamostrada_4000Hz_y = interp_funcao_y(tempo_reamostrado_4000Hz)
amplitude_reamostrada_4000Hz_z = interp_funcao_z(tempo_reamostrado_4000Hz)
amplitude_reamostrada_4000Hz_ref = interp_funcao_ref(tempo_reamostrado_4000Hz)

# criando um dataframe para os canais reamostrados
ecg_interpol = pd.DataFrame({
    "Tempo (s)": tempo_reamostrado_4000Hz,
    "Amplitude X": amplitude_reamostrada_4000Hz_x,
    "Amplitude Y": amplitude_reamostrada_4000Hz_y,
    "Amplitude Z": amplitude_reamostrada_4000Hz_z,
    "Referencia": amplitude_reamostrada_4000Hz_ref
}).set_index('Tempo (s)')
```

A Figura 2 mostra os sinais resultantes após a operação de interpolação.

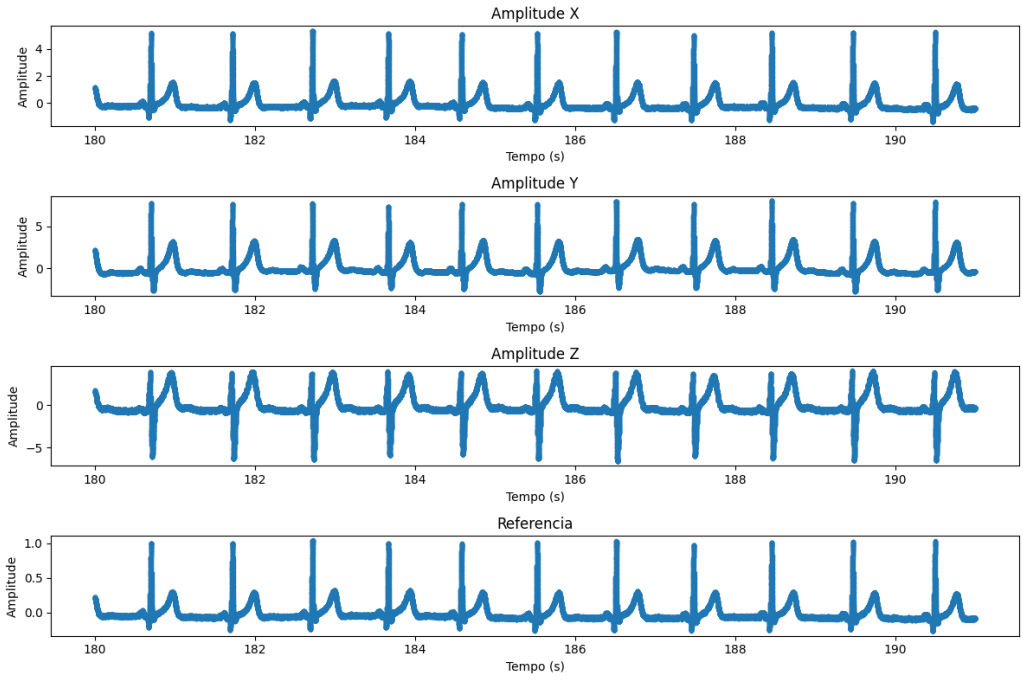

Figura 2 – Canais reamostrados à 4000Hz.

Problema 2: Detecção do Complexo PQRST e Promediação

Nesta etapa, primeiramente foi realizada a filtragem dos sinais de cada canal, através de um filtro passa-banda, de $[0.05 - 50]Hz$. O algoritmo desenvolvido é descrito abaixo:

```
# Filtros
cutoff_high = 0.5  # Frequência de corte do filtro passa-alta
cutoff_low = 50  # Frequência de corte do filtro passa-baixa

order_high = 3  # Ordem do filtro passa-alta
order_low = 4  # Ordem do filtro passa-baixa

# Filtro passa-alta
b_high, a_high = butter(order_high, cutoff_high / (4000 / 2), btype='high')
amplitude_reamostrada_4000Hz_x_high = filtfilt(b_high, a_high,
↪  amplitude_reamostrada_4000Hz_x)
amplitude_reamostrada_4000Hz_y_high = filtfilt(b_high, a_high,
↪  amplitude_reamostrada_4000Hz_y)
amplitude_reamostrada_4000Hz_z_high = filtfilt(b_high, a_high,
↪  amplitude_reamostrada_4000Hz_z)
amplitude_reamostrada_4000Hz_ref_high = filtfilt(b_high, a_high,
↪  amplitude_reamostrada_4000Hz_ref)

# Filtro passa-baixa
b_low, a_low = butter(order_low, cutoff_low / (4000 / 2), btype='low')
amplitude_reamostrada_4000Hz_x_filtered = filtfilt(b_low, a_low,
↪  amplitude_reamostrada_4000Hz_x_high)
amplitude_reamostrada_4000Hz_y_filtered = filtfilt(b_low, a_low,
↪  amplitude_reamostrada_4000Hz_y_high)
amplitude_reamostrada_4000Hz_z_filtered = filtfilt(b_low, a_low,
↪  amplitude_reamostrada_4000Hz_z_high)
amplitude_reamostrada_4000Hz_ref_filtered = filtfilt(b_low, a_low,
↪  amplitude_reamostrada_4000Hz_ref_high)
```

A Figura 3 mostra o resultado da filtragem aplicada. Veja que há uma melhora do sinal.

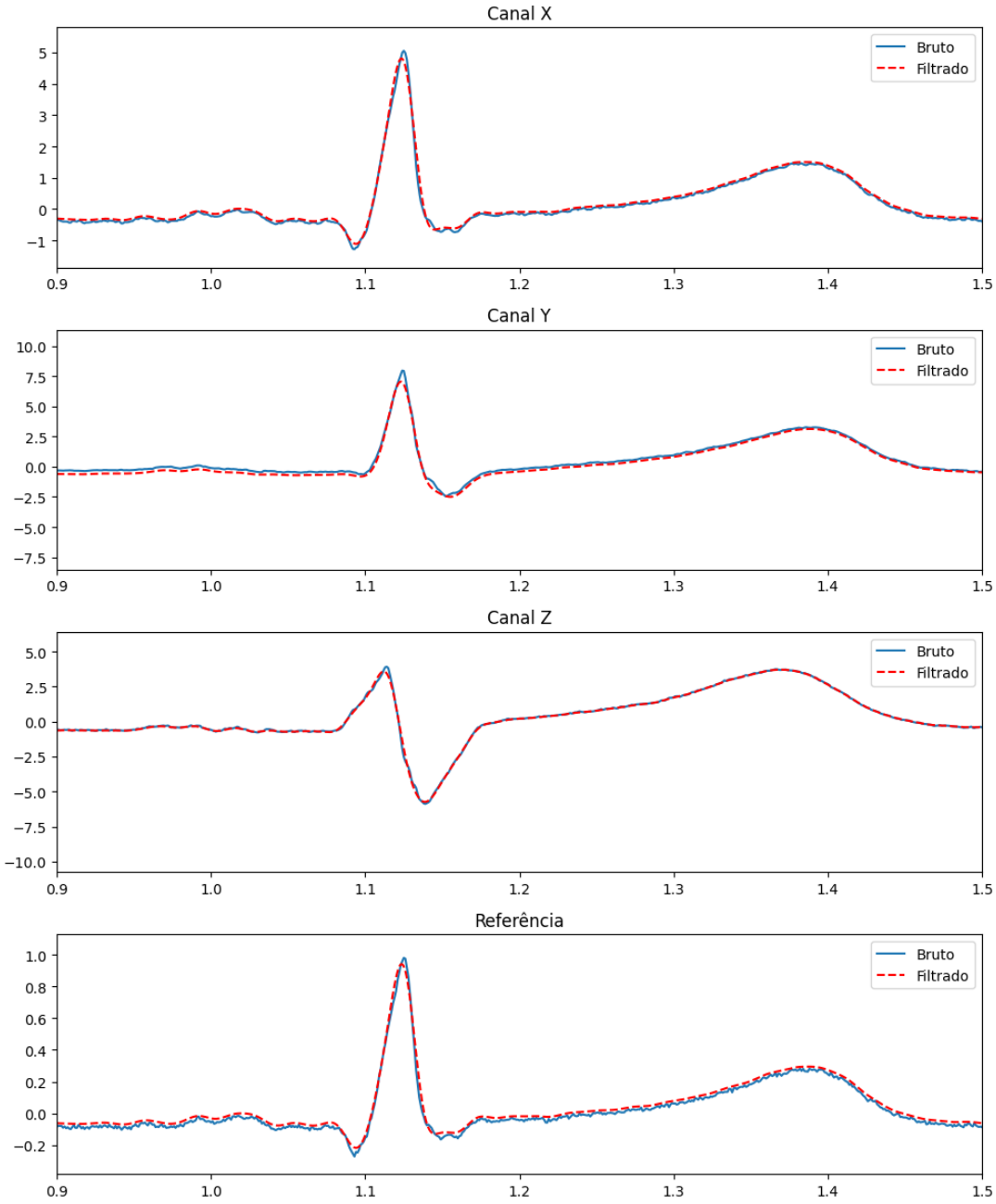

Figura 3 – Canais filtrados em para uma banda de 0.5 a 50Hz.

Para a detecção dos picos R, foi desenvolvida uma função chamada *detecta_R*(), conforme detalhada abaixo:

```
# Função criada para detectar as ondas R
def detecta_R(tempo, sinal, limiar_taxa):
  # diminuindo o limiar_taxa, a onda T é detectada
  # aumentando o limiar_taxa, somente o pico R é detectado

   intervalo = np.max(sinal) - np.min(sinal) # delta_y (escalar)
   limiar = limiar_taxa*intervalo + np.min(sinal) #0.x% do intervalo menos a
   ↪  amplitude negativa (escalar)
```

```
    amplitudes_maximas = []
    amplitudes_maximas_indices = []
    picoR_indices = []
    primeira_condicao = False # "chave" pra analisar as amplitudes que ultrapassam
    ↪  o limiar

    for n in range(0,len(sinal)):

     if sinal[n] >=limiar: # se a amplitude ultrapassa o limiar,
       primeira_condicao = True
       amplitudes_maximas.append(sinal[n]) # guardamos os valores dessa amplitude
       ↪  que ultrapassa o limiar
       amplitudes_maximas_indices.append(n) # guardamos os índices dessa amplitude
       ↪  que ultrapassa o limiar

     # analisando agora se esta amplitude se trata de um pico R:
     elif primeira_condicao == True and sinal[n] < limiar:
       amplitude_maxima_global =
       ↪  amplitudes_maximas.index(np.max(amplitudes_maximas))

       picoR_indices.append(amplitudes_maximas_indices[amplitude_maxima_global]) #
       ↪  armazena os picos R

       amplitudes_maximas = [] # "zera"
       amplitudes_maximas_indices = [] # "zera"

       primeira_condicao = False # volta à condição inicial
    return picoR_indices
```

Cujos argumentos são:

- tempo: vetor de tempo do sinal.
- Sinal: sinal ECG.
- limiar_taxa: compreendendo valores entre 0 e 1, consiste no percentual para criar um limiar de comparação entre o máximo do sinal e o mínimo e então, detecção do pico R.

Assim, para o canal X, aplica-se:

```
sinal_x = ecg_interpol_filtered['Amplitude X'].to_numpy()
picos_R_x = detecta_R(tempo_reamostrado_4000Hz, sinal_x, 0.7)
```

Cujo resultado é mostrado na Figura 4. Veja a eficiência do algoritmo desenvolvido ao se detectar todos os picos R.

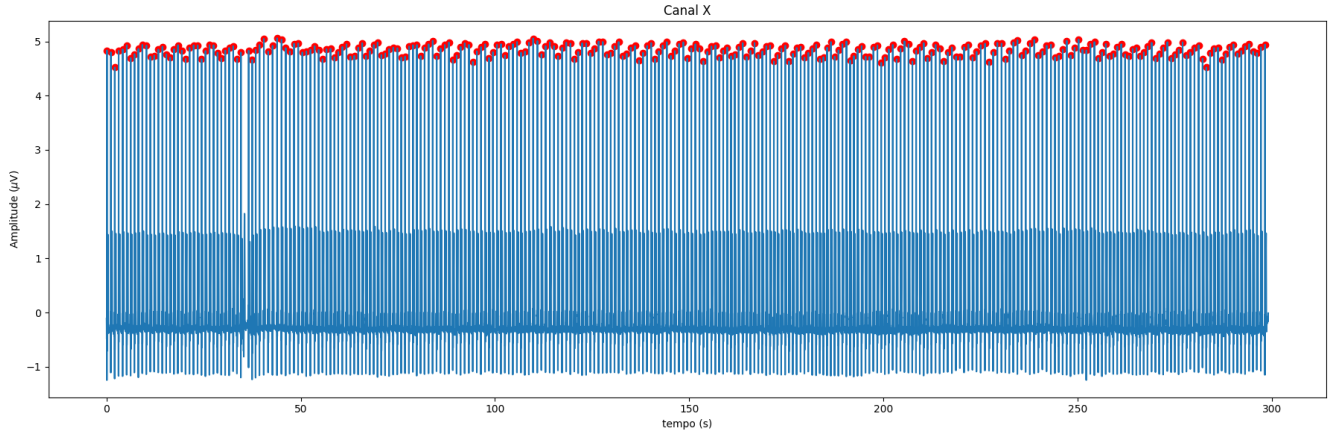

Figura 4 – Detecção dos picos R no canal X.

Para o canal Y, aplica-se:

```
sinal_y = ecg_interpol_filtered['Amplitude Y'].to_numpy()
picos_R_y = detecta_R(tempo_reamostrado_4000Hz, sinal_y, 0.7)
```

Resultando nos picos detectados na Figura 5.

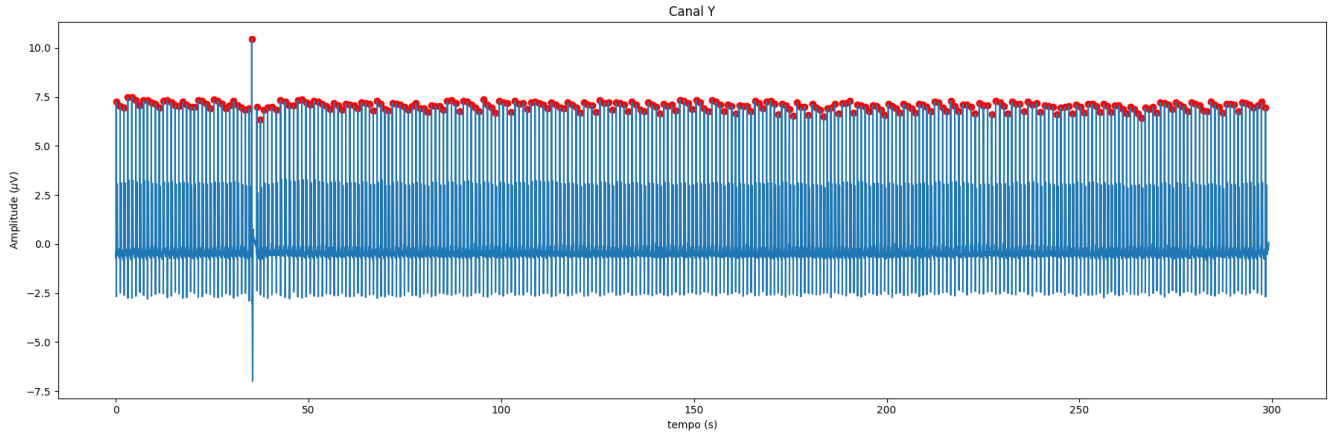

Figura 5 – Detecção dos picos R no canal Y.

E por fim, para o canal Z, aplica-se:

```
sinal_z = ecg_interpol_filtered['Amplitude Z'].to_numpy()
picos_R_z = detecta_R(tempo_reamostrado_4000Hz, sinal_z,0.85)
```

Resultando nos picos detectados na Figura 6.

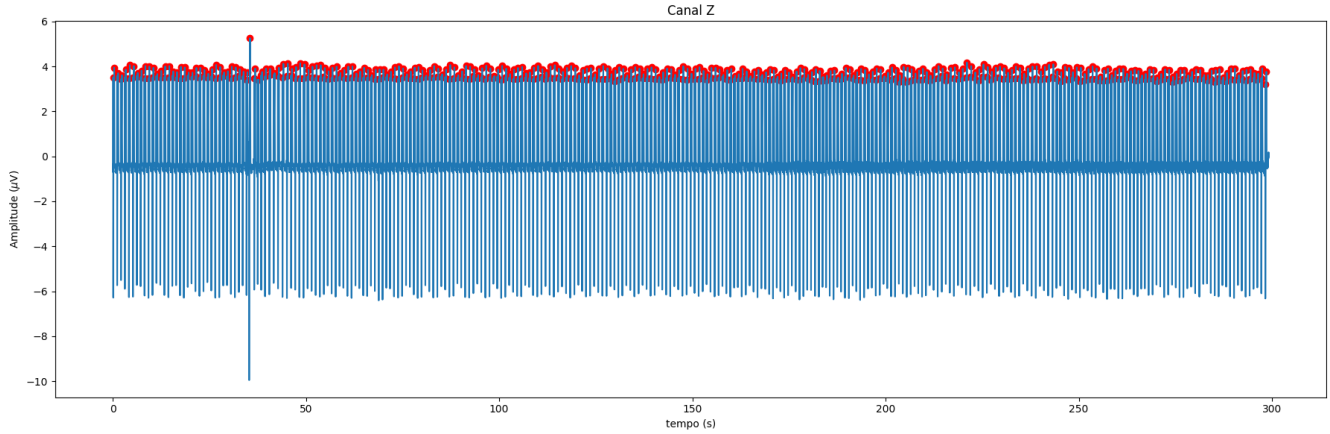

Figura 6 – Detecção dos picos R no canal Z.

Porém, como a onda T possui, em muitos trechos, a mesma amplitude que a onda R, o algoritmo erra ao detectar ambos os picos. Assim, em específico para o canal Z, a função desenvolvida apresentou falhas.

Para o cálculo da promediação, inicialmente foi escolhido um sinal *template* como meio de comparação. Uma vez detectado o batimento modelo, com seu respectivo pico R, definiu-se uma janela em *ms* em torno do pico R que servirá para correlacionar com os sinais de cada canal, identificando no tempo as ocorrências dos picos R e permitindo o alinhamento temporal de cada batimento. A Figura 7 detalha o sinal *template* escolhido e a janela de para correlação.

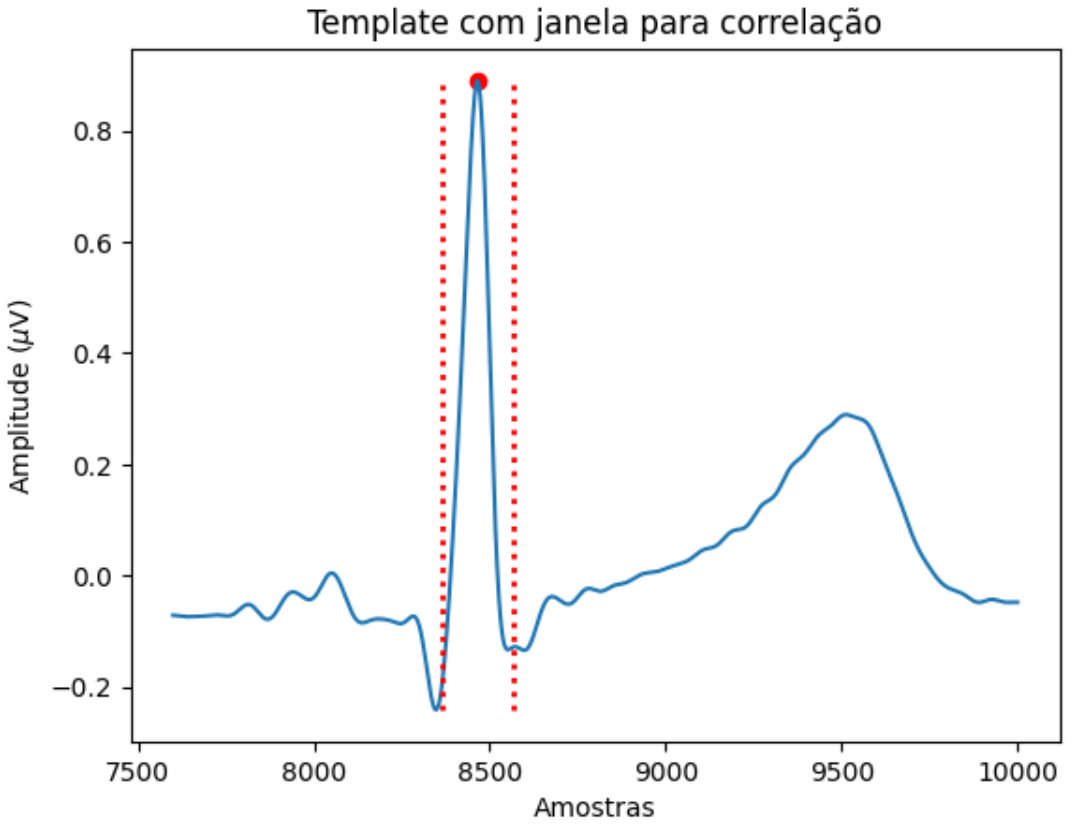

Figura 7 – *Template* com a janela para correlação.

A geração do *template* foi com base no seguinte código:

```
tempo_janela = 25  # milissegundos
janela_amostras = nova_frequencia_amostragem_4000*tempo_janela/1000 # tamanho da janela em amostras
t1 = int(1.9*nova_frequencia_amostragem_4000)
t2 = int(2.5*nova_frequencia_amostragem_4000)

condition1 = int(t1)< np.array(picos_R_template)
condition2 = np.array(picos_R_template) < t2
condition = condition1 & condition2
pico_R_template = np.array(picos_R_template)[condition]

limite_inferior = pico_R_template - janela_amostras
limite_superior = pico_R_template + janela_amostras

template_samples = range(int(limite_inferior), int(limite_superior))
template = sinal_referencia[template_samples]
```

Foi desenvolvida a função *promediacao*(), detalhada abaixo.

```
def promediacao(sinal, template, corr_threshold, janela_correlacao, fs, janela_batimento):
  # ref_menor = np.max(sinal) + np.min(sinal)
  intervalo = np.max(sinal) - np.min(sinal) # delta_y (escalar)
  ref_menor = 0.7*intervalo + np.min(sinal)
  samples_window_step = (janela_correlacao/2)*fs/1000 # a cada este passo, a correlação é calculada
  samples_window_size = len(template) # janela de dados do sinal de entrada para calculo da correlação
  numWin = math.floor((len(sinal) - samples_window_size)/samples_window_step +1) # numero de vezes que se janela o sinal de entrada para calculo da correlação

  count_alignments = 0
  count_rejected = 0
  r_alinhado = []
  indexes = []

  for winIdx in range(numWin):

    p1 = int(samples_window_step*winIdx) # posicao p1 da janela deslizante
    p2 = int(samples_window_size + samples_window_step*winIdx) # posicao p2 da janela deslizante
    windowed_signal = sinal[p1:p2] # janelando o sinal de entrada
    id = (p1,p2) # captura as posicoes

```

```
    correlacao = np.corrcoef(template, windowed_signal)

    if round(correlacao[0,1],2) > corr_threshold:
      r_possivel = np.max(windowed_signal) # possivel pico r
      if ref_menor < r_possivel:  # confirmando se realmente trata de um pico r
         r_alinhado.append(r_possivel)
         count_alignments = count_alignments +1 # contagem de complexos alinhados
         indexes.append(id)
      else:
        count_rejected = count_rejected +1

    # janela_batimento =0.45
  janela_batimento_samples =
  ↪  janela_batimento*nova_frequencia_amostragem_4000/1000
  x = len(template_samples) + 2*janela_batimento_samples
  sum = np.zeros((int(x)))
  for i in range(0,len(indexes)):
     i1 =indexes[i][0]-janela_batimento_samples
     i2 = indexes[i][1] + janela_batimento_samples
     sum = sum + sinal[int(i1):int(i2)] # soma ponto a ponto de cada batimento
     ↪  alinhado
  sinal_promediado = sum/count_alignments # divisao entre os n batimentos
  ↪  alinhados

  return sinal_promediado, count_alignments, count_rejected # a função retorna o
  ↪  canal promediado, o numero de complexos alinhados e o numero de complexos
  ↪  rejeitados
```

Em que os argumentos da função criada são:

- **sinal** corresponde ao sinal a ser promediado.
- **template** ao sinal tido como modelo.
- **corr_threshold** ao limiar da correlação entre o sinal e a janela definida do *template*.
- **janela_correlacao** à janela em *ms* em torno do pico R do *template*.
- **fs** à frequência de amostragem.
- **janela_batimento** ao tempo, em *ms*, de um batimento cardíaco (em média conforme reportado pela literatura).

A função retorna o sinal promediado, o número de complexos alinhados e o número de complexos rejeitados. Como variáveis envolvidas foi definido um *threshold* da correlação igual a 0,8, uma janela de $50ms$, centralizada no pico R e um tempo médio de batimento igual a $450ms$.

Para o canal X, a função de promediação é chamada da forma:

```
canal_x_promediado, complexos_alinhados, complexos_rejeitados =
↪  promediacao(sinal_x, template, 0.8, 50, nova_frequencia_amostragem_4000, 450)
```

```
print(f'Número de complexos alinhados:{complexos_alinhados}')
print(f'Número de complexos rejeitados:{complexos_rejeitados}')
```

Para se estimar o ruído, foi utilizada uma janela de $20ms$ em um trecho do sinal promediado em que se tem atividade cardíaca basal. Para o canal X o seguinte código estimar o ruído:

```
janela = 20 # ms
janela_samples = janela*nova_frequencia_amostragem_4000/1000
noise = np.std(canal_x_promediado[0:int(janela_samples)])
noise # em milivolts
```

Resultando em um ruído de aproximadamente $\pm 0,00235 \mu V$, ou seja, consideravelmente pequeno. O sinal promediado no canal X é mostrado na Figura 8. Veja a definição que o sinal adquire.

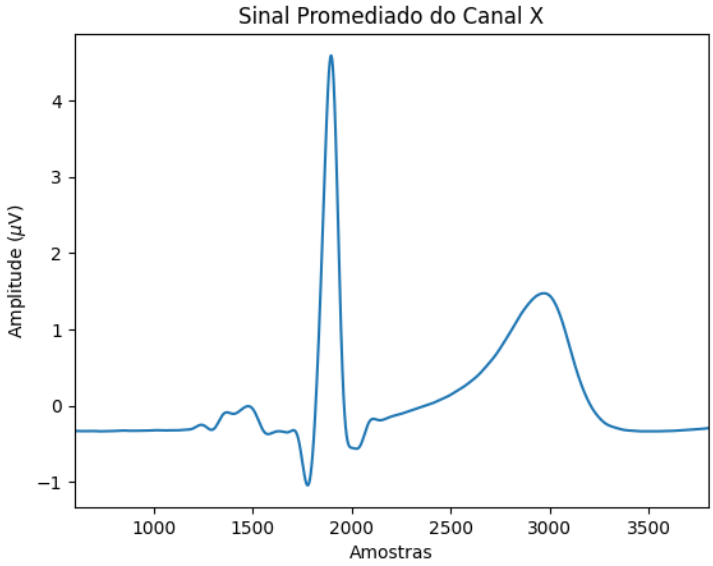

Figura 8 – Sinal do canal X promediado. O ruído resultante é da ordem de $\pm 0,00235 \mu V$.

Ainda em relação ao canal X, a Tabela 1 detalha o número de complexos alinhados e rejeitados durante a operação de promediação.

Tabela 1 – Complexos incluídos e rejeitados na promediação do canal X.

Complexos	**Total**
Incluídos	114
Rejeitados	159

Para o canal Y, a chamada da função de promediação é dada por:

```
canal_y_promediado, complexos_alinhados, complexos_rejeitados =
↪  promediacao(sinal_y, template, 0.8, 50, nova_frequencia_amostragem_4000, 450)
print(f'Número de complexos alinhados:{complexos_alinhados}')
print(f'Número de complexos rejeitados:{complexos_rejeitados}')
```

A estimativa do ruído, também considerando uma janela de $20ms$, na atividade cardíaca basal, resultou em um ruído de aproximadamente $\pm 0.00795 \mu V$.

Para o canal Y, o sinal promediado é mostrado na Figura 9.

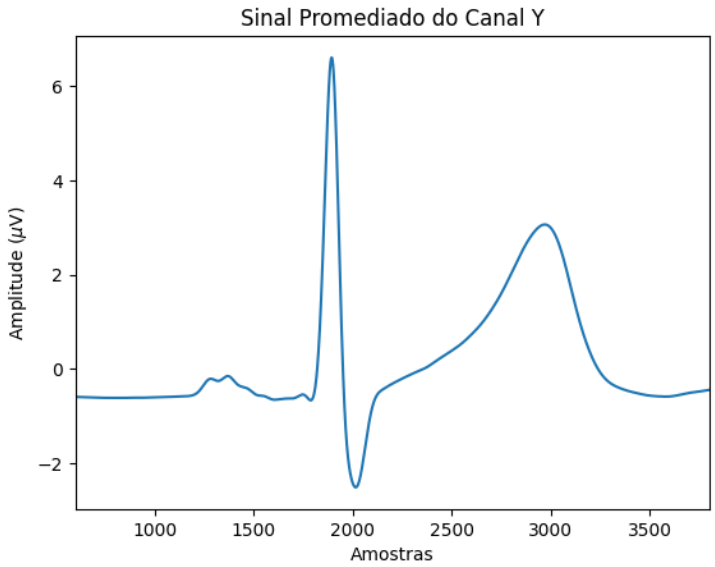

Figura 9 – Sinal do canal Y promediado. O ruído estimado é de $\pm 0.00795 \mu V$.

A Tabela 2 detalha o número de complexos alinhados e rejeitados durante a promediação do sinal no canal Y.

Tabela 2 – Complexos incluídos e rejeitados na promediação do canal Y.

Complexos	**Total**
Incluídos	113
Rejeitados	176

E por fim, para o canal Z, a chamada da função para promediação fica da forma:

```
canal_z_promediado, complexos_alinhados, complexos_rejeitados =
↪  promediacao(sinal_z, template, 0.8, 50, nova_frequencia_amostragem_4000, 450)
print(f'Número de complexos alinhados:{complexos_alinhados}')
print(f'Número de complexos rejeitados:{complexos_rejeitados}')
```

A estimativa do ruído no sinal promediado, considerando uma janela de $20ms$ na atividade cardíaca basal, foi de aproximadamente $\pm 0.00575 \mu V$.

Resultando no sinal mostrado na Figura 10.

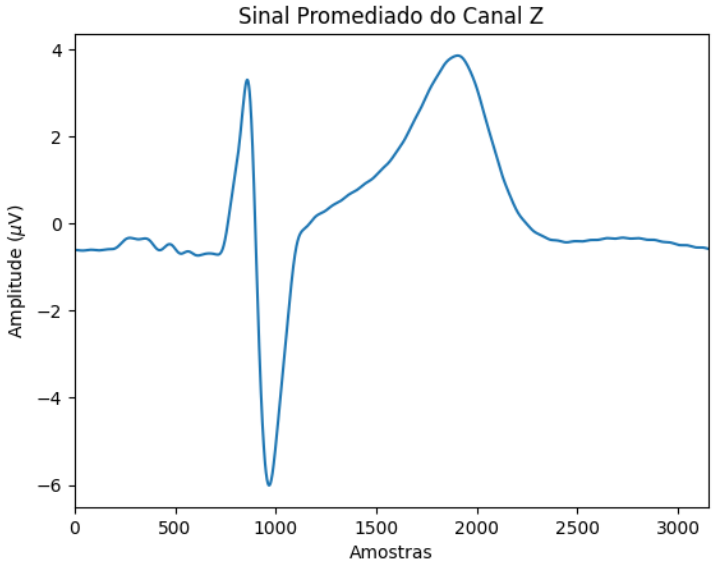

Figura 10 – Sinal do canal Z promediado. O ruído estimado foi se aproximadamente $\pm 0.00575 \mu V$.

O número de complexos incluídos e rejeitados é detalhado na Tabela 3.

Tabela 3 – Complexos incluídos e rejeitados na promediação do canal Z.

Complexos	**Total**
Incluídos	50
Rejeitados	173

Problema 3: Intervalos RR

Nesta etapa o objetivo é determinar, em ms, os intervalos RR. Para isto, foi calculada a diferença entre os picos R:

```
# Vamos assumir que "picos_R" é um array de índices dos picos R
tempos_picos_R = tempo_reamostrado_4000Hz[picos_R_x]
rr_intervals = np.diff(tempos_picos_R)
```

Que ao se plotar com o vetor no domínio do tempo, em ms, fica da forma mostrada na Figura 11.

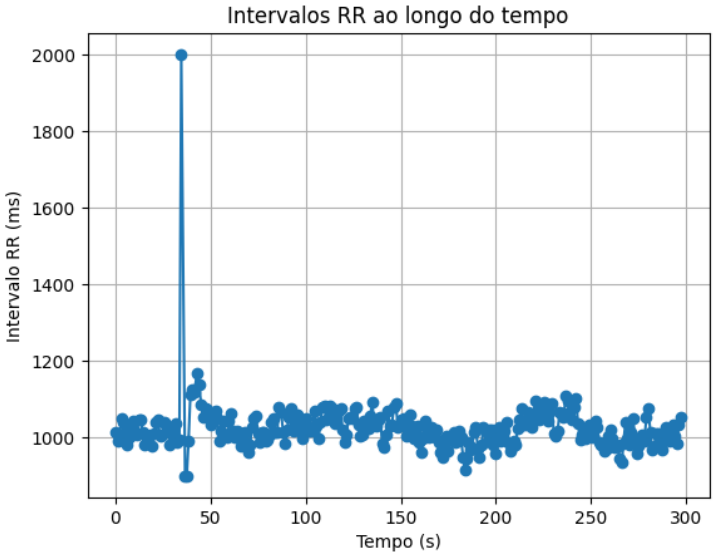

Figura 11 – Intervalos RR ao longo tempo.

O histograma destes intervalos RR é mostrado na Figura 12.

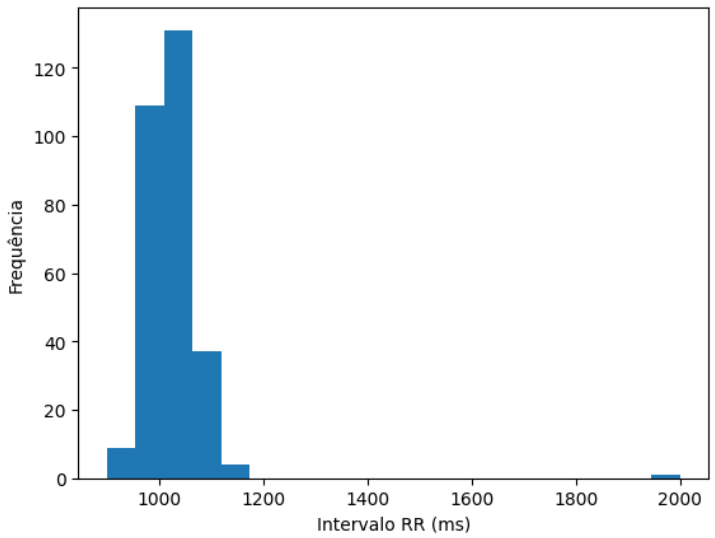

Figura 12 – Histograma dos intervalos RR. É possível verificar a presença de *outlier*.

E o boxplot desta distribuição na Figura 13.

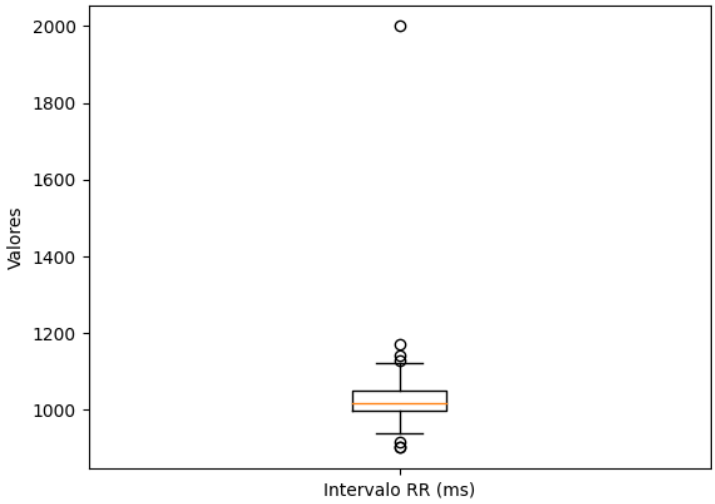

Figura 13 – Boxplot dos intervalos RR.

Problema 4: Detecção de Complexos - Q e T

Nesta etapa desenvolvemos um algoritmo para detecção dos complexos do ECG, em especial do Q e T. Primeiramente, desenvolveu-se o código para detecção do Q e S:

```
# Calculando os índices mínimos
mins_try = []

for R_peak_i in picos_R_x:
    left_interval = sinal_x[R_peak_i-700:R_peak_i]
    right_interval = sinal_x[R_peak_i:R_peak_i+700]

    if left_interval.size > 0:  # se não é vazio
        mins_try.append(R_peak_i - 700 +
        ↪  (list(left_interval).index(min(left_interval))))

    if right_interval.size > 0:  # se não é vazio
        mins_try.append(R_peak_i +
        ↪  (list(right_interval).index(min(right_interval))))
```

A Figura 14 ilustra a detecção resultante do código utilizando como exemplo um batimento do canal X.

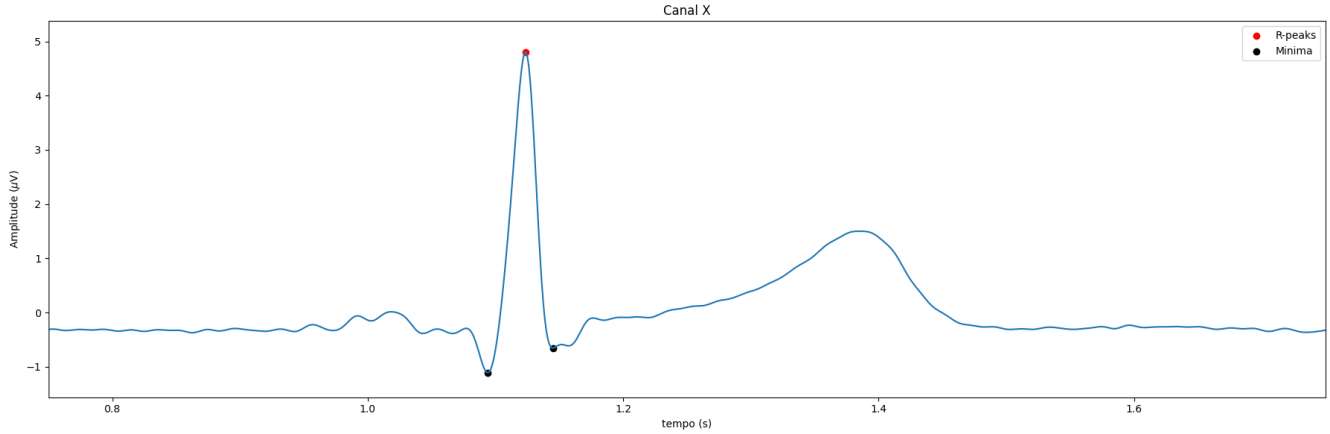

Figura 14 – Detecção dos complexos Q e S.

Estes complexos, então, serviram como pontos de partida para se detectar os complexos P e T, através da implementação do seguinte código:

```
# Mínimos índices
Q_try = []
S_try = []

for R_peak_i in picos_R_x:
    left_interval = sinal_x[R_peak_i-700:R_peak_i]
    right_interval = sinal_x[R_peak_i:R_peak_i+700]

    if left_interval.size > 0:
        Q_try.append(R_peak_i - 700 +
        ↪  (list(left_interval).index(min(left_interval))))

    if right_interval.size > 0:
        S_try.append(R_peak_i +
        ↪  (list(right_interval).index(min(right_interval))))

# adaptando para calcular os maximos em torno de Q e S
P_try = []
T_try = []

q_and_s_waves = Q_try + S_try  # combinando os indices de Q e S

for wave_peak in Q_try:
    left_interval = sinal_x[max(0, wave_peak - 700):wave_peak]

    if left_interval.size > 0:
        P_try.append(wave_peak - 700 + np.argmax(left_interval))

for wave_peak in S_try:
```

```
    right_interval = sinal_x[wave_peak:min(len(sinal_x), wave_peak + 1000)]
    if right_interval.size > 0:
        T_try.append(wave_peak + np.argmax(right_interval))
```

O resultado, é mostrado na Figura 15. Veja que todos os complexos são certamente detectados.

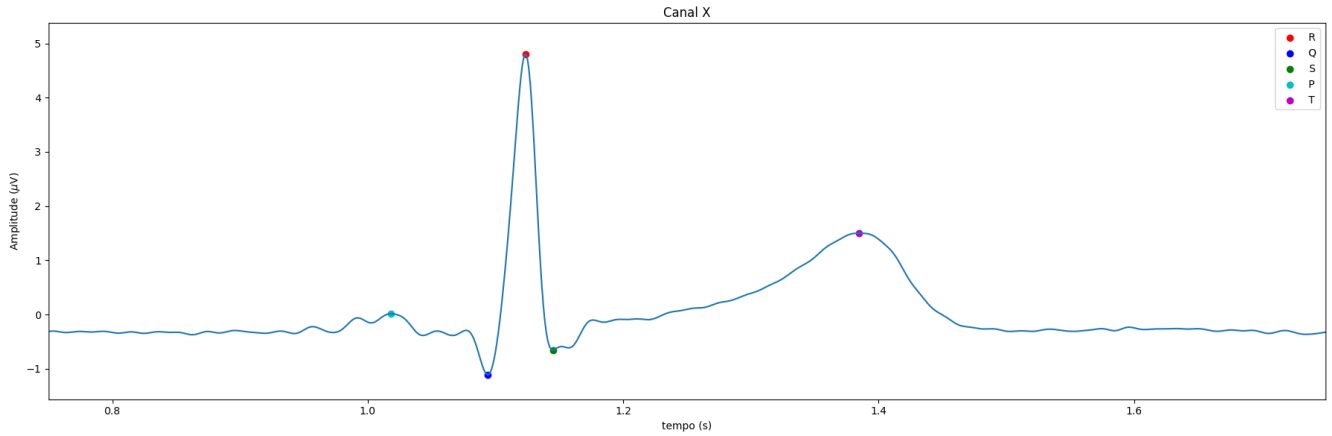

Figura 15 – Detecção dos Complexos PQRST.

Problema 5: Filtro Média Móvel

Nesta etapa, foram calculadas as estatísticas dos intervalos RR através do seguinte código:

```
# Calcular estatísticas descritivas
mean_s = np.mean(rr_intervals)
sdn_s = np.std(rr_intervals)
variancia_s = np.var(rr_intervals)
rms_s = np.sqrt(np.mean(np.square(rr_intervals)))

# Imprimir os resultados
print(f'Média: {mean_s} ms')
print(f'Desvio Padrão: {sdn_s} ms')
print(f'Variância: {variancia_s} (ms^2)')
print(f'RMS: {rms_s} ms')
```

A Tabela 4 detalha as estatísticas dos intervalos RR.

Tabela 4 – Estatísticas dos intervalos RR.

Média	1024,66 ms
Desvio Padrão	69,26 ms
Variância	4796,82 ms
RMS	1027 ms

Para aplicação do filtro média móvel de três termos, aplicamos:

```python
# Aplicar um filtro de média móvel de 3 termos
filtro_MA = np.convolve(rr_intervals, np.ones(3)/3, mode='valid')

# Ajustar o tempo para ter o mesmo comprimento que o resultado do filtro
tempo_filtrado = tempos_picos_R[:len(filtro_MA)]
```

Aplicou-se então um filtro média móvel de três termos de 5 minutos, a partir do seguinte algoritmo.


```python
# Aplicar um filtro de média móvel de 3 termos
filtro_MA = np.convolve(rr_intervals, np.ones(3)/3, mode='valid')

# Ajustar o tempo para ter o mesmo comprimento que o resultado do filtro
tempo_filtrado = tempos_picos_R[:len(filtro_MA)]
```


O resultado é mostrado na Figura 16.

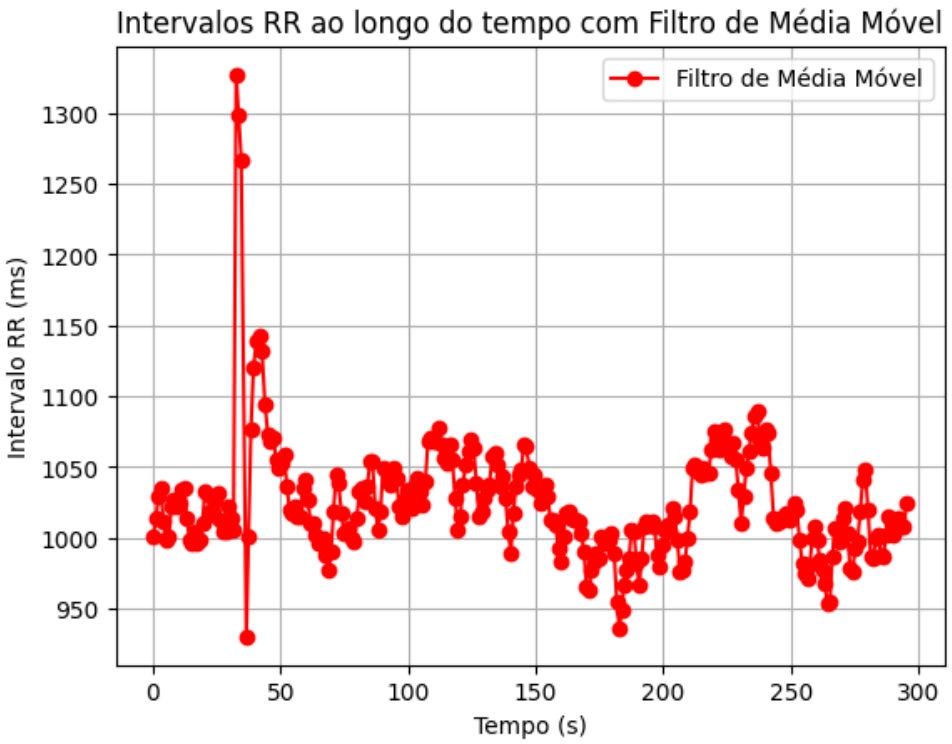

Figura 16 – Intervalo RR com média móvel de 3 termos.

Após a filtragem, foram calculadas as estatísticas do sinal RR, agora filtrado, resultando na Tabela 5.

Tabela 5 – Estatísticas dos intervalos RR após a filtragem utilizando filtro média móvel.

Média	1024,64 ms
Desvio Padrão	42,59 ms
Variância	1814,10 ms
RMS	1025,52 ms

Problema 6: Sinais RR para Faixa $M_{RR} - 2\sigma < RR < M_{RR} + 2\sigma$

Nesta etapa, o primeiro passo foi a remoção de *outliers* nos intervalos RR.

```
intervalos_RR_out = remove_outliers(rr_intervals)
```

O histograma resultante é mostrado na Figura 17.

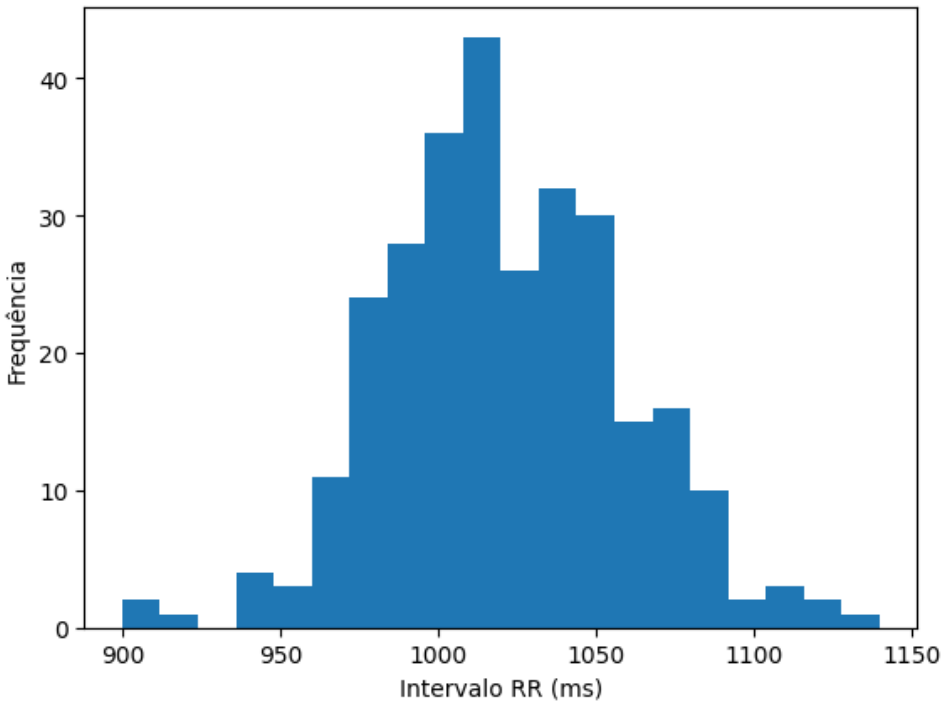

Figura 17 – Histograma dos intervalos RR sem *outliers*.

Os intervalos RR ao longo do tempo, após a remoção de outliers, são mostrados na Figura 18.

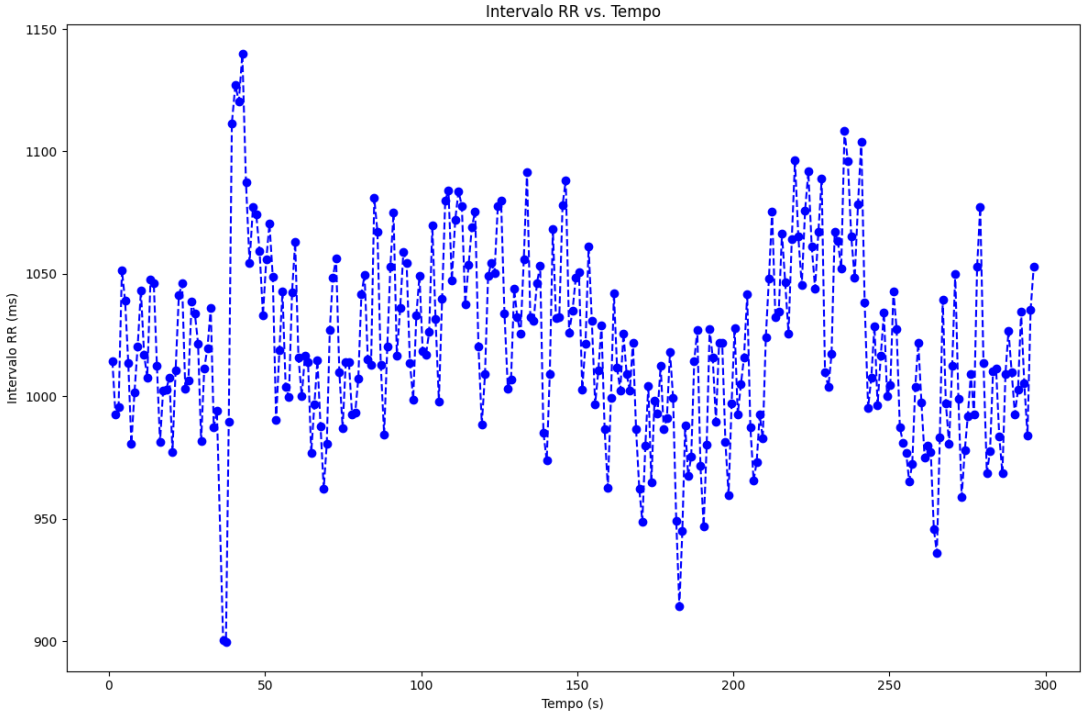

Figura 18 – Intervalos RR ao longo tempo sem *outliers*.

Problema 7: Espectro de Frequência

Nesta etapa foram calculados os espectros de frequência do sinal RR bruto (Figura 19) e sem *outliers* (Figura 20).

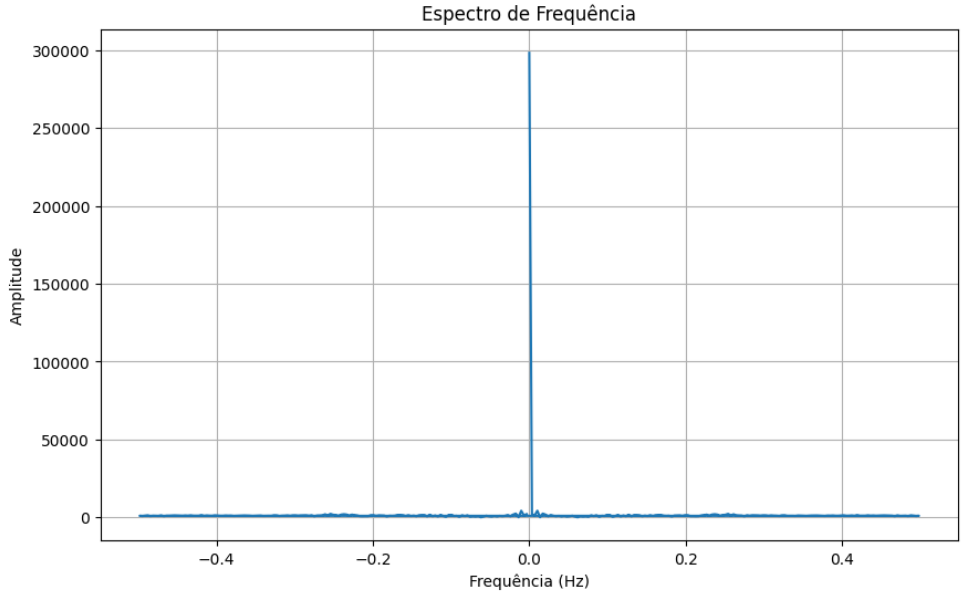

Figura 19 – Espectro de frequência RR.

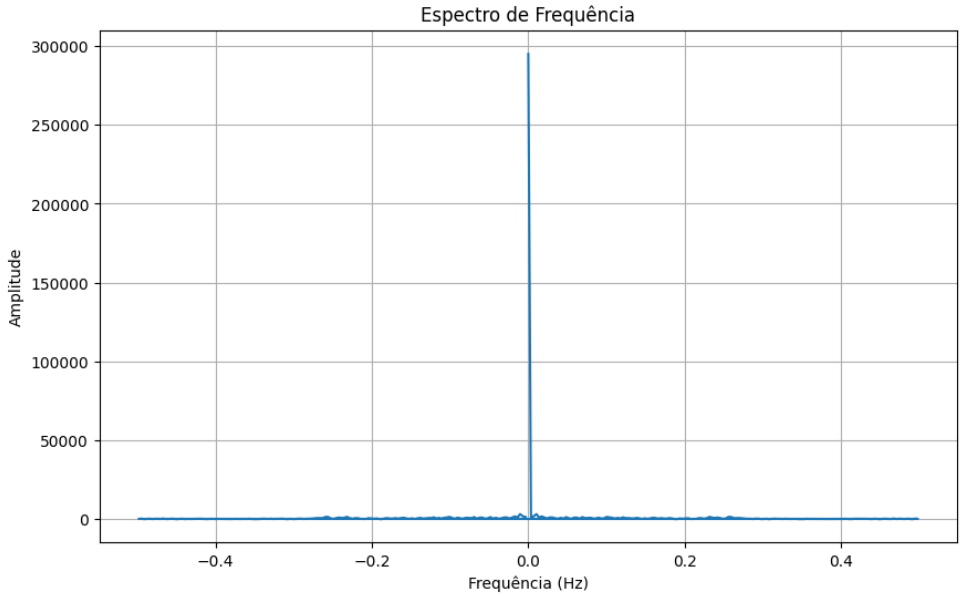

Figura 20 – Espectro de frequência RR sem *outliers*.