



Universidade de Brasília
Departamento da Ciência da Computação
Teleinformática e Redes 1

Simre;Camfe

Simulador de Redes: Camadas Física e Enlace

Grupo: 03

Henrique França	Mat: 242039130
José Antônio de Andrade	Mat: 232013031
Rafael Sapienza	Mat: 232006215

Professor:
Marcelo Antônio Marotta

25 de novembro de 2025

1 Introdução

A comunicação de dados em redes de computadores depende de uma série de procedimentos organizados em camadas, cada uma responsável por uma parte específica do processo de transmissão. Entre essas etapas, destacam-se a camada física e a camada de enlace, fundamentais para garantir que a informação seja corretamente convertida, modulada, transmitida, enquadrada, analisada e reconstruída no destino. O objetivo deste trabalho é desenvolver um simulador capaz de representar, de maneira modular, o funcionamento dessas duas camadas utilizando técnicas clássicas de modulação e protocolos de enquadramento, detecção e correção de erros.

Na camada física, o simulador implementa modulações digitais do tipo NRZ-Polar, Manchester e Bipolar, além de modulações por portadora como ASK, FSK, QPSK e 16-QAM, reproduzindo a forma como sinais elétricos ou ondas são utilizados para transportar bits no meio físico.

Já na camada de enlace, o simulador incorpora diferentes protocolos de enquadramento — incluindo contagem de caracteres, FLAGS com inserção de bytes e FLAGS com inserção de bits — que delimitam os quadros e evitam ambiguidades durante a recepção. Em seguida, são aplicados mecanismos de detecção de erros, como bit de paridade, Checksum e CRC-32, além de um método de correção de erros baseado no código de Hamming, permitindo identificar ou corrigir falhas introduzidas no fluxo de bits durante a transmissão.

O sistema integra essas funcionalidades em uma interface gráfica, permitindo visualizar passo a passo o processo de envio e recepção de mensagens. Assim, o simulador oferece uma compreensão prática e unificada dos principais mecanismos que tornam a comunicação de dados confiável.

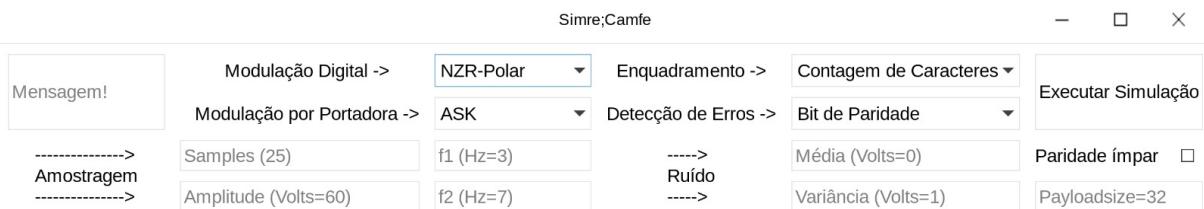


Figura 1: GUI do simulador

1.1 Instalação

Para executar o projeto, deve-se:

1. Instalar Python 3.x no sistema.
2. Instalar GTK no sistema.
3. Criar um Virtual Environment para o python.
4. Usar o pip para instalar: PyGObject (gtk), matplotlib, numpy e socket.
5. Executar o arquivo InterfaceGrafica.py.

2 Implementação

2.1 Camada Física

O funcionamento da Camada Física é bem simples do ponto de vista completo do protocolo. Todas os métodos estabelecidos no trabalho (no arquivo CamadaFisica.py) recebem uma lista de booleanos e parâmetros relacionados ao método de codificação, e retornam uma lista de valores de tensão que simulam uma onda eletromagnética. Similarmente, na decodificação, recebem uma lista de amostras de tensão e convertem para o valor aproximado mais provável (seja este verdadeiro ou falso). Segue exemplo do método na Listagem 1 com Amplitude Shift Keying.

Listagem 1: (De)Modulação ASK

```

1 import numpy as np
2 import numpy.typing as npt
3
4 def ASK_modulation(digitals: list[bool], *,
5                     smplcnt: int = 220,
6                     f: float = 100,
7                     amp: float = 500,
8                     lowwave: bool = False) -> npt.NDArray:
9
10    # Variaveis basicas
11    result: npt.NDArray = np.empty(0)
12    period: np.float64 = np.pi * 2 * f
13
14    # Pre-calculos para eficiencia
15    ## Cria uma lista com smplcnt valores de 0 ate 2*pi*f e usa-os para
16    ## construir uma onda
17    rOne: npt.NDArray = np.sin(np.arange(0, period, period / smplcnt)) *
18    amp
19    ## Apenas zeros
20    rZero: npt.NDArray = np.zeros(smplcnt)
21
22    # Criacao de sinal
23    for b in digitals:
24        # Checa se b eh 0/1, e entao inverte a codificacao se tal foi
25        ## requisitado
26        result = np.append(result, rOne if (b ^ lowwave) else rZero)
27
28    return result
29
30 def ASK_demodulation(signal: npt.NDArray, *,
31                      smplcnt: int = 220,
32                      f: float = 100,
33                      amp: float = 500,
34                      lowwave: bool = False) -> list[bool]:
35
36    # Pre-calculos para eficiencia
37    period: np.float64 = np.pi * 2 * f
38    ## Valor esperado para onda "1"
39    eOne: npt.NDArray = np.sin(np.arange(0, period, period / smplcnt)) *
40    amp
41    ## Valor esperado para onda "0"
42    eZero: npt.NDArray = np.zeros(smplcnt)
43
44    # Itera pelos chunks de samples de cada codigo (smplcnt)

```

```

41     result: list[bool] = []
42     for i in range(0, len(signal), smplcnt):
43         # Checa a diferença absoluta acumulada entre o sinal recebido
44         # e as ondas esperadas.
45         absdiffOne: np.float64 = np.sum(np.abs(signal[i:i+smplcnt] - eOne))
46         absdiffZero: np.float64 = np.sum(np.abs(signal[i:i+smplcnt] - eZero)
47     )
48
49     # O menor valor eh escolhido, seguindo escolha da inversao de
50     # codificacao
51     result.append(bool(absdiffOne < absdiffZero) ^ lowwave)
52
53     return result

```

No caso de modulações digitais, os métodos recebem uma lista de booleanos e retornam uma lista de inteiros, representando a devida modulação. Em geral, tais valores são 1, 0 e -1. Para cada modulação, um método construtor distinto deve ser aplicado mas, em geral, sua formação depende de valores anteriores, diferente dos métodos de portadora que todos apenas se interessam no valor do bit atual. Segue exemplo da (de)modulação bipolar, na Listagem 2.

Para a geração de valores de tensão nas modulações de portadora, usa-se uma função seno com os devidos parâmetros fornecidos (amplitude, frequência e fase) e divide-se um período completo de acordo com os parâmetros de amostragem. Especificamente, um período completo é dividido de tal forma que, para N amostras, o intervalo entre elas seja $2\pi/N$. Por exemplo, para uma onda de 1Hz, com 4 amostras teremos a lista de tensão $[0, A, 0, -A]$ onde A é o valor máximo de tensão. Cada símbolo é gerado individualmente de acordo com o protocolo escolhido, e então utilizado na formação da lista final de valores de tensão.

Para a demodulação, compara blocos de amostras do sinal recebido e um conjunto de ondas de referência pré-computadas para determinar qual é o símbolo recebido. Utiliza-se como critério a diferença absoluta acumulada e, se necessário, a energia do sinal.

Listagem 2: (De)Modulação Bipolar

```

1 def bipolar(bit_string: list[bool]) -> list[int]:
2     # Lista de resultados
3     sinal_codificado = []
4     # Valor inicial da codificacao
5     aux = 1
6
7     for bit in bit_string:
8         if not bit:
9             # Zero
10            sinal_codificado.append(0)
11        else:
12            # Um (+1)
13            sinal_codificado.append(aux)
14            aux = -aux # Trocar polaridade apos todo "1"
15
16    return sinal_codificado
17
18 def bipolar_decoder(bit_string: list[int]) -> list[bool]:
19     # Lista de resultados
20     sinal_decodificado = []
21     # Valor inicial da codificacao
22     aux = 1
23
24     for bit in bit_string:
25         if bit > -0.5 and bit <= 0.5:

```

```

26         # Zero
27         sinal_decodificado.append(False)
28     elif bit > 0.5:
29         # Um (+1)
30         if aux == -1:
31             # Erro na codificacao
32             return None
33         sinal_decodificado.append(True)
34         aux = -aux
35     else:
36         # Um (-1)
37         if aux == 1:
38             # Erro na codificacao
39             return None
40         sinal_decodificado.append(True)
41         aux = -aux
42
43     return sinal_decodificado

```

No sistema, a Camada Física entra logo antes da mensagem ser enviada para o socket — momento em que todo o protocolo da Camada de Enlace já foi aplicado — para simular o trajeto de transmissão. Ruído é adicionado logo em seguida e, após o socket receber a mensagem, tal é decodificada pelo mesmo método escolhido pelo usuário. Por prática, mesmo que o sinal esteja irreconhecível devido ao erro, todos métodos de portadora retornarão algum resultado que pode ser, então, analizado pela Camada de Enlace. A aplicação dos métodos para simulação está disposta na Listagem 3.

Listagem 3: Aplicação da Camada Física

```

1 import numpy as np
2 import numpy.typing as npt
3 import threading
4
5 # ...
6
7 class Programa(Gtk.Window):
8     # ...
9     def run(self,
10             message,
11             digital_mod,
12             analog_mod,
13             framing_method,
14             error_method,
15             errorcurve_mean,
16             errorcurve_variance,
17             parity_status) -> bool:
18         # ...
19
20         # Codificar por portadora
21         amsg: npt.NDArray = np.empty(0)
22         if analog_mod == GUI_ASK:
23             amsg = cf.ASK_modulation(dmsg, smplcnt=GUI_SMPLCNT, f=GUI_F1,
24             amp=GUI_AMP)
25         elif analog_mod == GUI_FSK:
26             amsg = cf.FSK_modulation(dmsg, (GUI_F1, GUI_F2), smplcnt=
27             GUI_SMPLCNT, amp=GUI_AMP)
28         elif analog_mod == GUI_BPSK:
29             amsg = cf.BPSK_modulation(dmsg, smplcnt=GUI_SMPLCNT, f=GUI_F1,
30             amp=GUI_AMP)
31
32         # Adicionar ruído
33         noise = np.random.normal(0, 0.01, len(amsg))
34         amsg += noise
35
36         # Decodificar por portadora
37         sinal_decodificado = self.decoder.decode(amsg)
38
39         # Analisar resultado
40         if sinal_decodificado is None:
41             print("Erro na codificação")
42             return False
43         else:
44             print("Mensagem decodificada com sucesso!")
45             return True
46
47     def on_button_clicked(self, button):
48         if button.get_label() == "Enviar":
49             self.run()
50
51     def on_error(self, message):
52         print(message)
53
54     def on_message(self, message):
55         print(message)
56
57     def on_decoder_error(self, error):
58         print(error)
59
60     def on_decoder_message(self, message):
61         print(message)
62
63     def on_decoder_error(self, error):
64         print(error)
65
66     def on_decoder_message(self, message):
67         print(message)
68
69     def on_decoder_error(self, error):
70         print(error)
71
72     def on_decoder_message(self, message):
73         print(message)
74
75     def on_decoder_error(self, error):
76         print(error)
77
78     def on_decoder_message(self, message):
79         print(message)
80
81     def on_decoder_error(self, error):
82         print(error)
83
84     def on_decoder_message(self, message):
85         print(message)
86
87     def on_decoder_error(self, error):
88         print(error)
89
90     def on_decoder_message(self, message):
91         print(message)
92
93     def on_decoder_error(self, error):
94         print(error)
95
96     def on_decoder_message(self, message):
97         print(message)
98
99     def on_decoder_error(self, error):
100        print(error)
101
102    def on_decoder_message(self, message):
103        print(message)
104
105    def on_decoder_error(self, error):
106        print(error)
107
108    def on_decoder_message(self, message):
109        print(message)
110
111    def on_decoder_error(self, error):
112        print(error)
113
114    def on_decoder_message(self, message):
115        print(message)
116
117    def on_decoder_error(self, error):
118        print(error)
119
120    def on_decoder_message(self, message):
121        print(message)
122
123    def on_decoder_error(self, error):
124        print(error)
125
126    def on_decoder_message(self, message):
127        print(message)
128
129    def on_decoder_error(self, error):
130        print(error)
131
132    def on_decoder_message(self, message):
133        print(message)
134
135    def on_decoder_error(self, error):
136        print(error)
137
138    def on_decoder_message(self, message):
139        print(message)
140
141    def on_decoder_error(self, error):
142        print(error)
143
144    def on_decoder_message(self, message):
145        print(message)
146
147    def on_decoder_error(self, error):
148        print(error)
149
150    def on_decoder_message(self, message):
151        print(message)
152
153    def on_decoder_error(self, error):
154        print(error)
155
156    def on_decoder_message(self, message):
157        print(message)
158
159    def on_decoder_error(self, error):
160        print(error)
161
162    def on_decoder_message(self, message):
163        print(message)
164
165    def on_decoder_error(self, error):
166        print(error)
167
168    def on_decoder_message(self, message):
169        print(message)
170
171    def on_decoder_error(self, error):
172        print(error)
173
174    def on_decoder_message(self, message):
175        print(message)
176
177    def on_decoder_error(self, error):
178        print(error)
179
180    def on_decoder_message(self, message):
181        print(message)
182
183    def on_decoder_error(self, error):
184        print(error)
185
186    def on_decoder_message(self, message):
187        print(message)
188
189    def on_decoder_error(self, error):
190        print(error)
191
192    def on_decoder_message(self, message):
193        print(message)
194
195    def on_decoder_error(self, error):
196        print(error)
197
198    def on_decoder_message(self, message):
199        print(message)
200
201    def on_decoder_error(self, error):
202        print(error)
203
204    def on_decoder_message(self, message):
205        print(message)
206
207    def on_decoder_error(self, error):
208        print(error)
209
210    def on_decoder_message(self, message):
211        print(message)
212
213    def on_decoder_error(self, error):
214        print(error)
215
216    def on_decoder_message(self, message):
217        print(message)
218
219    def on_decoder_error(self, error):
220        print(error)
221
222    def on_decoder_message(self, message):
223        print(message)
224
225    def on_decoder_error(self, error):
226        print(error)
227
228    def on_decoder_message(self, message):
229        print(message)
230
231    def on_decoder_error(self, error):
232        print(error)
233
234    def on_decoder_message(self, message):
235        print(message)
236
237    def on_decoder_error(self, error):
238        print(error)
239
240    def on_decoder_message(self, message):
241        print(message)
242
243    def on_decoder_error(self, error):
244        print(error)
245
246    def on_decoder_message(self, message):
247        print(message)
248
249    def on_decoder_error(self, error):
250        print(error)
251
252    def on_decoder_message(self, message):
253        print(message)
254
255    def on_decoder_error(self, error):
256        print(error)
257
258    def on_decoder_message(self, message):
259        print(message)
260
261    def on_decoder_error(self, error):
262        print(error)
263
264    def on_decoder_message(self, message):
265        print(message)
266
267    def on_decoder_error(self, error):
268        print(error)
269
270    def on_decoder_message(self, message):
271        print(message)
272
273    def on_decoder_error(self, error):
274        print(error)
275
276    def on_decoder_message(self, message):
277        print(message)
278
279    def on_decoder_error(self, error):
280        print(error)
281
282    def on_decoder_message(self, message):
283        print(message)
284
285    def on_decoder_error(self, error):
286        print(error)
287
288    def on_decoder_message(self, message):
289        print(message)
290
291    def on_decoder_error(self, error):
292        print(error)
293
294    def on_decoder_message(self, message):
295        print(message)
296
297    def on_decoder_error(self, error):
298        print(error)
299
300    def on_decoder_message(self, message):
301        print(message)
302
303    def on_decoder_error(self, error):
304        print(error)
305
306    def on_decoder_message(self, message):
307        print(message)
308
309    def on_decoder_error(self, error):
310        print(error)
311
312    def on_decoder_message(self, message):
313        print(message)
314
315    def on_decoder_error(self, error):
316        print(error)
317
318    def on_decoder_message(self, message):
319        print(message)
320
321    def on_decoder_error(self, error):
322        print(error)
323
324    def on_decoder_message(self, message):
325        print(message)
326
327    def on_decoder_error(self, error):
328        print(error)
329
330    def on_decoder_message(self, message):
331        print(message)
332
333    def on_decoder_error(self, error):
334        print(error)
335
336    def on_decoder_message(self, message):
337        print(message)
338
339    def on_decoder_error(self, error):
340        print(error)
341
342    def on_decoder_message(self, message):
343        print(message)
344
345    def on_decoder_error(self, error):
346        print(error)
347
348    def on_decoder_message(self, message):
349        print(message)
350
351    def on_decoder_error(self, error):
352        print(error)
353
354    def on_decoder_message(self, message):
355        print(message)
356
357    def on_decoder_error(self, error):
358        print(error)
359
360    def on_decoder_message(self, message):
361        print(message)
362
363    def on_decoder_error(self, error):
364        print(error)
365
366    def on_decoder_message(self, message):
367        print(message)
368
369    def on_decoder_error(self, error):
370        print(error)
371
372    def on_decoder_message(self, message):
373        print(message)
374
375    def on_decoder_error(self, error):
376        print(error)
377
378    def on_decoder_message(self, message):
379        print(message)
380
381    def on_decoder_error(self, error):
382        print(error)
383
384    def on_decoder_message(self, message):
385        print(message)
386
387    def on_decoder_error(self, error):
388        print(error)
389
390    def on_decoder_message(self, message):
391        print(message)
392
393    def on_decoder_error(self, error):
394        print(error)
395
396    def on_decoder_message(self, message):
397        print(message)
398
399    def on_decoder_error(self, error):
400        print(error)
401
402    def on_decoder_message(self, message):
403        print(message)
404
405    def on_decoder_error(self, error):
406        print(error)
407
408    def on_decoder_message(self, message):
409        print(message)
410
411    def on_decoder_error(self, error):
412        print(error)
413
414    def on_decoder_message(self, message):
415        print(message)
416
417    def on_decoder_error(self, error):
418        print(error)
419
420    def on_decoder_message(self, message):
421        print(message)
422
423    def on_decoder_error(self, error):
424        print(error)
425
426    def on_decoder_message(self, message):
427        print(message)
428
429    def on_decoder_error(self, error):
430        print(error)
431
432    def on_decoder_message(self, message):
433        print(message)
434
435    def on_decoder_error(self, error):
436        print(error)
437
438    def on_decoder_message(self, message):
439        print(message)
440
441    def on_decoder_error(self, error):
442        print(error)
443
444    def on_decoder_message(self, message):
445        print(message)
446
447    def on_decoder_error(self, error):
448        print(error)
449
450    def on_decoder_message(self, message):
451        print(message)
452
453    def on_decoder_error(self, error):
454        print(error)
455
456    def on_decoder_message(self, message):
457        print(message)
458
459    def on_decoder_error(self, error):
460        print(error)
461
462    def on_decoder_message(self, message):
463        print(message)
464
465    def on_decoder_error(self, error):
466        print(error)
467
468    def on_decoder_message(self, message):
469        print(message)
470
471    def on_decoder_error(self, error):
472        print(error)
473
474    def on_decoder_message(self, message):
475        print(message)
476
477    def on_decoder_error(self, error):
478        print(error)
479
480    def on_decoder_message(self, message):
481        print(message)
482
483    def on_decoder_error(self, error):
484        print(error)
485
486    def on_decoder_message(self, message):
487        print(message)
488
489    def on_decoder_error(self, error):
490        print(error)
491
492    def on_decoder_message(self, message):
493        print(message)
494
495    def on_decoder_error(self, error):
496        print(error)
497
498    def on_decoder_message(self, message):
499        print(message)
500
501    def on_decoder_error(self, error):
502        print(error)
503
504    def on_decoder_message(self, message):
505        print(message)
506
507    def on_decoder_error(self, error):
508        print(error)
509
510    def on_decoder_message(self, message):
511        print(message)
512
513    def on_decoder_error(self, error):
514        print(error)
515
516    def on_decoder_message(self, message):
517        print(message)
518
519    def on_decoder_error(self, error):
520        print(error)
521
522    def on_decoder_message(self, message):
523        print(message)
524
525    def on_decoder_error(self, error):
526        print(error)
527
528    def on_decoder_message(self, message):
529        print(message)
530
531    def on_decoder_error(self, error):
532        print(error)
533
534    def on_decoder_message(self, message):
535        print(message)
536
537    def on_decoder_error(self, error):
538        print(error)
539
540    def on_decoder_message(self, message):
541        print(message)
542
543    def on_decoder_error(self, error):
544        print(error)
545
546    def on_decoder_message(self, message):
547        print(message)
548
549    def on_decoder_error(self, error):
550        print(error)
551
552    def on_decoder_message(self, message):
553        print(message)
554
555    def on_decoder_error(self, error):
556        print(error)
557
558    def on_decoder_message(self, message):
559        print(message)
560
561    def on_decoder_error(self, error):
562        print(error)
563
564    def on_decoder_message(self, message):
565        print(message)
566
567    def on_decoder_error(self, error):
568        print(error)
569
570    def on_decoder_message(self, message):
571        print(message)
572
573    def on_decoder_error(self, error):
574        print(error)
575
576    def on_decoder_message(self, message):
577        print(message)
578
579    def on_decoder_error(self, error):
580        print(error)
581
582    def on_decoder_message(self, message):
583        print(message)
584
585    def on_decoder_error(self, error):
586        print(error)
587
588    def on_decoder_message(self, message):
589        print(message)
590
591    def on_decoder_error(self, error):
592        print(error)
593
594    def on_decoder_message(self, message):
595        print(message)
596
597    def on_decoder_error(self, error):
598        print(error)
599
600    def on_decoder_message(self, message):
601        print(message)
602
603    def on_decoder_error(self, error):
604        print(error)
605
606    def on_decoder_message(self, message):
607        print(message)
608
609    def on_decoder_error(self, error):
610        print(error)
611
612    def on_decoder_message(self, message):
613        print(message)
614
615    def on_decoder_error(self, error):
616        print(error)
617
618    def on_decoder_message(self, message):
619        print(message)
620
621    def on_decoder_error(self, error):
622        print(error)
623
624    def on_decoder_message(self, message):
625        print(message)
626
627    def on_decoder_error(self, error):
628        print(error)
629
630    def on_decoder_message(self, message):
631        print(message)
632
633    def on_decoder_error(self, error):
634        print(error)
635
636    def on_decoder_message(self, message):
637        print(message)
638
639    def on_decoder_error(self, error):
640        print(error)
641
642    def on_decoder_message(self, message):
643        print(message)
644
645    def on_decoder_error(self, error):
646        print(error)
647
648    def on_decoder_message(self, message):
649        print(message)
650
651    def on_decoder_error(self, error):
652        print(error)
653
654    def on_decoder_message(self, message):
655        print(message)
656
657    def on_decoder_error(self, error):
658        print(error)
659
660    def on_decoder_message(self, message):
661        print(message)
662
663    def on_decoder_error(self, error):
664        print(error)
665
666    def on_decoder_message(self, message):
667        print(message)
668
669    def on_decoder_error(self, error):
670        print(error)
671
672    def on_decoder_message(self, message):
673        print(message)
674
675    def on_decoder_error(self, error):
676        print(error)
677
678    def on_decoder_message(self, message):
679        print(message)
680
681    def on_decoder_error(self, error):
682        print(error)
683
684    def on_decoder_message(self, message):
685        print(message)
686
687    def on_decoder_error(self, error):
688        print(error)
689
690    def on_decoder_message(self, message):
691        print(message)
692
693    def on_decoder_error(self, error):
694        print(error)
695
696    def on_decoder_message(self, message):
697        print(message)
698
699    def on_decoder_error(self, error):
700        print(error)
701
702    def on_decoder_message(self, message):
703        print(message)
704
705    def on_decoder_error(self, error):
706        print(error)
707
708    def on_decoder_message(self, message):
709        print(message)
710
711    def on_decoder_error(self, error):
712        print(error)
713
714    def on_decoder_message(self, message):
715        print(message)
716
717    def on_decoder_error(self, error):
718        print(error)
719
720    def on_decoder_message(self, message):
721        print(message)
722
723    def on_decoder_error(self, error):
724        print(error)
725
726    def on_decoder_message(self, message):
727        print(message)
728
729    def on_decoder_error(self, error):
730        print(error)
731
732    def on_decoder_message(self, message):
733        print(message)
734
735    def on_decoder_error(self, error):
736        print(error)
737
738    def on_decoder_message(self, message):
739        print(message)
740
741    def on_decoder_error(self, error):
742        print(error)
743
744    def on_decoder_message(self, message):
745        print(message)
746
747    def on_decoder_error(self, error):
748        print(error)
749
750    def on_decoder_message(self, message):
751        print(message)
752
753    def on_decoder_error(self, error):
754        print(error)
755
756    def on_decoder_message(self, message):
757        print(message)
758
759    def on_decoder_error(self, error):
760        print(error)
761
762    def on_decoder_message(self, message):
763        print(message)
764
765    def on_decoder_error(self, error):
766        print(error)
767
768    def on_decoder_message(self, message):
769        print(message)
770
771    def on_decoder_error(self, error):
772        print(error)
773
774    def on_decoder_message(self, message):
775        print(message)
776
777    def on_decoder_error(self, error):
778        print(error)
779
780    def on_decoder_message(self, message):
781        print(message)
782
783    def on_decoder_error(self, error):
784        print(error)
785
786    def on_decoder_message(self, message):
787        print(message)
788
789    def on_decoder_error(self, error):
790        print(error)
791
792    def on_decoder_message(self, message):
793        print(message)
794
795    def on_decoder_error(self, error):
796        print(error)
797
798    def on_decoder_message(self, message):
799        print(message)
800
801    def on_decoder_error(self, error):
802        print(error)
803
804    def on_decoder_message(self, message):
805        print(message)
806
807    def on_decoder_error(self, error):
808        print(error)
809
810    def on_decoder_message(self, message):
811        print(message)
812
813    def on_decoder_error(self, error):
814        print(error)
815
816    def on_decoder_message(self, message):
817        print(message)
818
819    def on_decoder_error(self, error):
820        print(error)
821
822    def on_decoder_message(self, message):
823        print(message)
824
825    def on_decoder_error(self, error):
826        print(error)
827
828    def on_decoder_message(self, message):
829        print(message)
830
831    def on_decoder_error(self, error):
832        print(error)
833
834    def on_decoder_message(self, message):
835        print(message)
836
837    def on_decoder_error(self, error):
838        print(error)
839
840    def on_decoder_message(self, message):
841        print(message)
842
843    def on_decoder_error(self, error):
844        print(error)
845
846    def on_decoder_message(self, message):
847        print(message)
848
849    def on_decoder_error(self, error):
850        print(error)
851
852    def on_decoder_message(self, message):
853        print(message)
854
855    def on_decoder_error(self, error):
856        print(error)
857
858    def on_decoder_message(self, message):
859        print(message)
860
861    def on_decoder_error(self, error):
862        print(error)
863
864    def on_decoder_message(self, message):
865        print(message)
866
867    def on_decoder_error(self, error):
868        print(error)
869
870    def on_decoder_message(self, message):
871        print(message)
872
873    def on_decoder_error(self, error):
874        print(error)
875
876    def on_decoder_message(self, message):
877        print(message)
878
879    def on_decoder_error(self, error):
880        print(error)
881
882    def on_decoder_message(self, message):
883        print(message)
884
885    def on_decoder_error(self, error):
886        print(error)
887
888    def on_decoder_message(self, message):
889        print(message)
890
891    def on_decoder_error(self, error):
892        print(error)
893
894    def on_decoder_message(self, message):
895        print(message)
896
897    def on_decoder_error(self, error):
898        print(error)
899
900    def on_decoder_message(self, message):
901        print(message)
902
903    def on_decoder_error(self, error):
904        print(error)
905
906    def on_decoder_message(self, message):
907        print(message)
908
909    def on_decoder_error(self, error):
910        print(error)
911
912    def on_decoder_message(self, message):
913        print(message)
914
915    def on_decoder_error(self, error):
916        print(error)
917
918    def on_decoder_message(self, message):
919        print(message)
920
921    def on_decoder_error(self, error):
922        print(error)
923
924    def on_decoder_message(self, message):
925        print(message)
926
927    def on_decoder_error(self, error):
928        print(error)
929
930    def on_decoder_message(self, message):
931        print(message)
932
933    def on_decoder_error(self, error):
934        print(error)
935
936    def on_decoder_message(self, message):
937        print(message)
938
939    def on_decoder_error(self, error):
940        print(error)
941
942    def on_decoder_message(self, message):
943        print(message)
944
945    def on_decoder_error(self, error):
946        print(error)
947
948    def on_decoder_message(self, message):
949        print(message)
950
951    def on_decoder_error(self, error):
952        print(error)
953
954    def on_decoder_message(self, message):
955        print(message)
956
957    def on_decoder_error(self, error):
958        print(error)
959
960    def on_decoder_message(self, message):
961        print(message)
962
963    def on_decoder_error(self, error):
964        print(error)
965
966    def on_decoder_message(self, message):
967        print(message)
968
969    def on_decoder_error(self, error):
970        print(error)
971
972    def on_decoder_message(self, message):
973        print(message)
974
975    def on_decoder_error(self, error):
976        print(error)
977
978    def on_decoder_message(self, message):
979        print(message)
980
981    def on_decoder_error(self, error):
982        print(error)
983
984    def on_decoder_message(self, message):
985        print(message)
986
987    def on_decoder_error(self, error):
988        print(error)
989
990    def on_decoder_message(self, message):
991        print(message)
992
993    def on_decoder_error(self, error):
994        print(error)
995
996    def on_decoder_message(self, message):
997        print(message)
998
999    def on_decoder_error(self, error):
1000       print(error)
1001
1002    def on_decoder_message(self, message):
1003       print(message)
1004
1005    def on_decoder_error(self, error):
1006       print(error)
1007
1008    def on_decoder_message(self, message):
1009       print(message)
1010
1011    def on_decoder_error(self, error):
1012       print(error)
1013
1014    def on_decoder_message(self, message):
1015       print(message)
1016
1017    def on_decoder_error(self, error):
1018       print(error)
1019
1020    def on_decoder_message(self, message):
1021       print(message)
1022
1023    def on_decoder_error(self, error):
1024       print(error)
1025
1026    def on_decoder_message(self, message):
1027       print(message)
1028
1029    def on_decoder_error(self, error):
1030       print(error)
1031
1032    def on_decoder_message(self, message):
1033       print(message)
1034
1035    def on_decoder_error(self, error):
1036       print(error)
1037
1038    def on_decoder_message(self, message):
1039       print(message)
1040
1041    def on_decoder_error(self, error):
1042       print(error)
1043
1044    def on_decoder_message(self, message):
1045       print(message)
1046
1047    def on_decoder_error(self, error):
1048       print(error)
1049
1050    def on_decoder_message(self, message):
1051       print(message)
1052
1053    def on_decoder_error(self, error):
1054       print(error)
1055
1056    def on_decoder_message(self, message):
1057       print(message)
1058
1059    def on_decoder_error(self, error):
1060       print(error)
1061
1062    def on_decoder_message(self, message):
1063       print(message)
1064
1065    def on_decoder_error(self, error):
1066       print(error)
1067
1068    def on_decoder_message(self, message):
1069       print(message)
1070
1071    def on_decoder_error(self, error):
1072       print(error)
1073
1074    def on_decoder_message(self, message):
1075       print(message)
1076
1077    def on_decoder_error(self, error):
1078       print(error)
1079
1080    def on_decoder_message(self, message):
1081       print(message)
1082
1083    def on_decoder_error(self, error):
1084       print(error)
1085
1086    def on_decoder_message(self, message):
1087       print(message)
1088
1089    def on_decoder_error(self, error):
1090       print(error)
1091
1092    def on_decoder_message(self, message):
1093       print(message)
1094
1095    def on_decoder_error(self, error):
1096       print(error)
1097
1098    def on_decoder_message(self, message):
1099       print(message)
1100
1101    def on_decoder_error(self, error):
1102       print(error)
1103
1104    def on_decoder_message(self, message):
1105       print(message)
1106
1107    def on_decoder_error(self, error):
1108       print(error)
1109
1110    def on_decoder_message(self, message):
1111       print(message)
1112
1113    def on_decoder_error(self, error):
1114       print(error)
1115
1116    def on_decoder_message(self, message):
1117       print(message)
1118
1119    def on_decoder_error(self, error):
1120       print(error)
1121
1122    def on_decoder_message(self, message):
1123       print(message)
1124
1125    def on_decoder_error(self, error):
1126       print(error)
1127
1128    def on_decoder_message(self, message):
1129       print(message)
1130
1131    def on_decoder_error(self, error):
1132       print(error)
1133
1134    def on_decoder_message(self, message):
1135       print(message)
1136
1137    def on_decoder_error(self, error):
1138       print(error)
1139
1140    def on_decoder_message(self, message):
1141       print(message)
1142
1143    def on_decoder_error(self, error):
1144       print(error)
1145
1146    def on_decoder_message(self, message):
1147       print(message)
1148
1149    def on_decoder_error(self, error):
1150       print(error)
1151
1152    def on_decoder_message(self, message):
1153       print(message)
1154
1155    def on_decoder_error(self, error):
1156       print(error)
1157
1158    def on_decoder_message(self, message):
1159       print(message)
1160
1161    def on_decoder_error(self, error):
1162       print(error)
1163
1164    def on_decoder_message(self, message):
1165       print(message)
1166
1167    def on_decoder_error(self, error):
1168       print(error)
1169
1170    def on_decoder_message(self, message):
1171       print(message)
1172
1173    def on_decoder_error(self, error):
1174       print(error)
1175
1176    def on_decoder_message(self, message):
1177       print(message)
1178
1179    def on_decoder_error(self, error):
1180       print(error)
1181
1182    def on_decoder_message(self, message):
1183       print(message)
1184
1185    def on_decoder_error(self, error):
1186       print(error)
1187
1188    def on_decoder_message(self, message):
1189       print(message)
1190
1191    def on_decoder_error(self, error):
1192       print(error)
1193
1194    def on_decoder_message(self, message):
1195       print(message)
1196
1197    def on_decoder_error(self, error):
1198       print(error)
1199
1200    def on_decoder_message(self, message):
1201       print(message)
1202
1203    def on_decoder_error(self, error):
1204       print(error)
1205
1206    def on_decoder_message(self, message):
1207       print(message)
1208
1209    def on_decoder_error(self, error):
1210       print(error)
1211
1212    def on_decoder_message(self, message):
1213       print(message)
1214
1215    def on_decoder_error(self, error):
1216       print(error)
1217
1218    def on_decoder_message(self, message):
1219       print(message)
1220
1221    def on_decoder_error(self, error):
1222       print(error)
1223
1224    def on_decoder_message(self, message):
1225       print(message)
1226
1227    def on_decoder_error(self, error):
1228       print(error)
1229
1230    def on_decoder_message(self, message):
1231       print(message)
1232
1233    def on_decoder_error(self, error):
1234       print(error)
1235
1236    def on_decoder_message(self, message):
1237       print(message)
1238
1239    def on_decoder_error(self, error):
1240       print(error)
1241
1242    def on_decoder_message(self, message):
1243       print(message)
1244
1245    def on_decoder_error(self, error):
1246       print(error)
1247
1248    def on_decoder_message(self, message):
1249       print(message)
1250
1251    def on_decoder_error(self, error):
1252       print(error)
1253
1254    def on_decoder_message(self, message):
1255       print(message)
1256
1257    def on_decoder_error(self, error):
1258       print(error)
1259
1260    def on_decoder_message(self, message):
1261       print(message)
1262
1263    def on_decoder_error(self, error):
1264       print(error)
1265
1266    def on_decoder_message(self, message):
1267       print(message)
1268
1269    def on_decoder_error(self, error):
1270       print(error)
1271
1272    def on_decoder_message(self, message):
1273       print(message)
1274
1275    def on_decoder_error(self, error):
1276       print(error)
1277
1278    def on_decoder_message(self, message):
1279       print(message)
1280
1281    def on_decoder_error(self, error):
1282       print(error)
1283
1284    def on_decoder_message(self, message):
1285       print(message)
1286
1287    def on_decoder_error(self, error):
1288       print(error)
1289
1290    def on_decoder_message(self, message):
1291       print(message)
1292
1293    def on_decoder_error(self, error):
1294       print(error)
1295
1296    def on_decoder_message(self, message):
1297       print(message)
1298
1299    def on_decoder_error(self, error):
1300       print(error)
1301
1302    def on_decoder_message(self, message):
1303       print(message)
1304
1305    def on_decoder_error(self, error):
1306       print(error)
1307
1308    def on_decoder_message(self, message):
1309       print(message)
1310
1311    def on_decoder_error(self, error):
1312       print(error)
1313
1314    def on_decoder_message(self, message):
1315       print(message)
1316
1317    def on_decoder_error(self, error):
1318       print(error)
1319
1320    def on_decoder_message(self, message):
1321       print(message)
1322
1323    def on_decoder_error(self, error):
1324       print(error)
1325
1326    def on_decoder_message(self, message):
1327       print(message)
1328
1329    def on_decoder_error(self, error):
1330       print(error)
1331
1332    def on_decoder_message(self, message):
1333       print(message)
1334
1335    def on_decoder_error(self, error):
1336       print(error)
1337
1338    def on_decoder_message(self, message):
1339       print(message)
1340

```

```
28     amp=GUI_AMP)
29         elif analog_mod == GUI_QPSK:
30             amsg = cf.QPSK_modulation(dmsg, smplcnt=GUI_SMPLCNT, f=GUI_F1,
31             amp=GUI_AMP)
32             elif analog_mod == GUI_16QAM:
33                 amsg = cf.QAM_modulation(dmsg, smplcnt=GUI_SMPLCNT, f=GUI_F1,
34                 amp=GUI_AMP)
35             else:
36                 return False
37             if amsg is None:
38                 return False
39
40             # ...
41
42             # Adicionar ruido
43             amsg = utils.samples_addnoise(amsg, average=errorcurve_mean, spread=
44             errorcurve_variance)
45             if amsg is None:
46                 return False
47
48             # ...
49
50             # Enviar mensagem
51             if not tm.send_to_server(amsg):
52                 return False
53
54             # Receber mensagem
55             receiver.flag_ready.wait(timeout=5)
56             rmsg: npt.NDArray = receiver.interpreted_data
57             if rmsg is None:
58                 return False
59
60             # ...
61
62             # Decodificar mensagem
63             if analog_mod == GUI_ASK:
64                 rmsg = cf.ASK_demodulation(rmsg, smplcnt=GUI_SMPLCNT, f=GUI_F1,
65                 amp=GUI_AMP)
66                 elif analog_mod == GUI_FSK:
67                     rmsg = cf.FSK_demodulation(rmsg, (GUI_F1, GUI_F2), smplcnt=
68                     GUI_SMPLCNT, amp=GUI_AMP)
69                     elif analog_mod == GUI_BPSK:
70                         rmsg = cf.BPSK_demodulation(rmsg, smplcnt=GUI_SMPLCNT, f=GUI_F1
71                         , amp=GUI_AMP)
72                         elif analog_mod == GUI_QPSK:
73                             rmsg = cf.QPSK_demodulation(rmsg, smplcnt=GUI_SMPLCNT, f=GUI_F1
74                             , amp=GUI_AMP)
75                             elif analog_mod == GUI_16QAM:
76                                 rmsg = cf.QAM_demodulation(rmsg, smplcnt=GUI_SMPLCNT, f=GUI_F1
77                                 , amp=GUI_AMP)
78                                 else:
79                                     return False
80                                 if rmsg is None:
81                                     return False
82
83             # ...
```

2.2 Camada de Enlace

A camada de enlace foi estruturada logicamente em dois componentes: o transmissor e o receptor. O transmissor recebe um trem de bits representado por uma lista de valores booleanos (False para 0 e True para 1), contendo a mensagem original a ser enviada. A etapa inicial do processamento consiste em segmentar o fluxo de bits em blocos de carga útil, tarefa realizada pela função `split_bitstream_into_payloads`. Como resultado, obtém-se uma lista de listas de booleanos, em que cada lista interna representa um bloco. Cada bloco possui, por padrão, no máximo 32 bits, valor que pode ser ajustado pelo usuário na interface gráfica. O último bloco pode conter menos de 32 bits, porém seu tamanho deve obrigatoriamente ser par, um requisito necessário para a aplicação do *checksum*.

Para garantir essa condição, caso o último bloco tenha um número ímpar de bits, a função `add_padding_and_padding_size` insere um bit 0 ao final do bloco. Depois, ela acrescenta um bloco adicional de 2 bits que indica se houve padding: “00” quando nenhum bit foi inserido e “01” quando um bit 0 foi adicionado ao pacote anterior.

Em seguida, aplica-se um protocolo de detecção de erros (EDC) por meio da função `add_EDC`. O método específico é definido pelo usuário na interface gráfica e pode ser: bit de paridade, *checksum* ou CRC-32. A função `add_EDC` opera de maneira iterativa: para cada bloco de carga útil, ela invoca uma função correspondente ao protocolo selecionado, aplicando-o individualmente a cada bloco. Esse processo é repetido sequencialmente até que todos os blocos tenham sido processados e estejam devidamente protegidos pelo código de detecção de erros.

A título de exemplo, segue, na Listagem 4, a definição da função que adiciona o protocolo CRC-32 a um bloco de carga útil. O processo inicia com o acréscimo de 32 bits zero ao payload original. Utiliza o polinômio gerador `0x104C11DB7` (padrão IEEE 802.3). O algoritmo executa então uma divisão polinomial bit a bit: percorre todo o payload estendido, realizando operações XOR com o polinômio gerador quando o bit mais significativo (MSB) do resto atual é 1, e efetua deslocamentos sucessivos.

Listagem 4: Implementação do protocolo CRC-32

```

1 def crc32_insert(payload: List[bool]) -> List[bool]:
2
3     # Adiciona 32 bits 0 ao padding
4     padding: List[bool] = [False] * 32
5     dividend: List[bool] = payload + padding
6
7     # Converte polinomio gerador em lista de booleanos
8     divisor: List[bool] = int_to_bool_list(num=GENERATOR_POLYNOMIAL, size
9     =33)
10
11    # Inicia divisor com primeiros 33 bits do dividendo
12    remainder: List[bool] = dividend[:33]
13
14    # Divisao polinomial
15    for i in range(len(divisor), len(dividend)):
16        if remainder[0]:
17            remainder = [a ^ b for a, b in zip(remainder, divisor)]
18            remainder = remainder[1:] + [dividend[i]]
19
20        if remainder[0]:
21            remainder = [a ^ b for a, b in zip(remainder, divisor)]
22            remainder = remainder[1:] # Tira overflow do resto
23
24    crc_bits: List[bool] = remainder

```

```

24     payload.extend(crc_bits)
25
26     return payload
27

```

Após a inserção dos códigos de detecção de erros (EDC), aplica-se a camada de correção através da função `add_ECC`. Esta função invoca iterativamente a função `hamming_insert` para cada bloco de dados. A função `hamming_insert`, cuja definição encontra-se na Listagem 5, implementa o protocolo Hamming para correção de erros, seguindo quatro etapas principais:

1. **Calcula o número de bits de paridade** necessários com base no tamanho dos dados, utilizando a relação $2^r \geq m + r + 1$, onde m é o número de bits de dados e r o número de bits de paridade.
2. **Insere placeholders para os bits de paridade** nas posições correspondentes a potências de 2 (1, 2, 4, 8, ...). Estes *placeholders* são inicializados com o valor `False` (bit 0). Esta inicialização é crucial porque:
 - O bit `False` (0) atua como **elemento neutro** da operação XOR
 - Permite que os bits de dados sejam processados sem interferência
3. **Calcula cada bit de paridade** usando operações XOR. É fundamental compreender que:
 - A operação XOR (^) implementa a **soma binária módulo 2**
 - Esta operação equivale à **paridade par**: resultado é 1 se houver número ímpar de 1's, 0 se houver número par de 1's
 - Cada bit de paridade cobre um padrão específico de posições:
 - **P1 (posição 1)**: bits 1, 3, 5, 7, 9, 11, ...
 - **P2 (posição 2)**: bits 2, 3, 6, 7, 10, 11, 14, 15, ...
 - **P4 (posição 4)**: bits 4, 5, 6, 7, 12, 13, 14, 15, ...
 - **P8 (posição 8)**: bits 8, 9, 10, 11, 12, 13, 14, 15, 24, ...
4. **Substitui os placeholders** iniciais pelos valores de paridade calculados, completando a palavra-código.

O resultado final é uma palavra-código de Hamming completa, onde cada bit de paridade protege um subconjunto específico de bits, permitindo detectar e corrigir erros individuais na transmissão.

Listagem 5: Implementação do código de Hamming para correção de erros

```

1 def hamming_insert(data_bits: List[bool]) -> List[bool]:
2
3     data_size: int = len(data_bits)
4     parity_bits_count: int = floor(log2(data_size)) + 1
5     # Ajusta numero de bits de paridade ate encontrar
6     # a condicao 2^r >= m + r + 1
7     while log2(data_size + parity_bits_count) > parity_bits_count:
8         parity_bits_count += 1
9
10    # Insere placeholder nas posicoes 1, 2, 4, 8, ...
11    position: int = 1
12    for _ in range(parity_bits_count):

```

```

13     data_bits.insert(position - 1, False)
14     position *= 2
15
16     # Calcula bits de paridade
17     step: int = 1
18     for _ in range(parity_bits_count):
19         parity: bool = False
20
21         for j in range(step, len(data_bits) + 1, step * 2):
22             for bit in data_bits[j - 1 : j - 1 + step]:
23                 parity ^= bit
24
25         # Armazena bits de paridade
26         data_bits[step - 1] = parity
27
28         # Move para proxima posicao (2, 4, 8, ...)
29         step *= 2
30
31     return data_bits

```

A função `add_framing_protocol` aplica o método de enquadramento selecionado, suportando contagem de caracteres, *flags* com inserção de bits e de bytes. Para protocolos baseados em bytes, a função `apply_8bit_padding` garante o alinhamento a bytes através da adição de bits de preenchimento e um campo de 8 bits indicando sua quantidade. A Listagem 6 apresenta as implementações das funções de enquadramento por contagem de caracteres e alinhamento.

Listagem 6: Funções de enquadramento por contagem de caracteres e alinhamento de dados

```

1 def apply_8bit_padding(frame_body: List[bool]) -> List[bool]:
2
3     # Se len(frame_body) ja for multiplo de 8
4     # queremos que padding_size = 0 no lugar de 8
5     # Por isso, adicionamos um modulo extra
6     padding_size:int = (8 - len(frame_body) % 8) % 8
7
8     frame_body += [False] * padding_size
9
10    # Codifica padding size como uma lista de 8 booleanos
11    padding_bits: List[bool] = int_to_bool_list(padding_size, 8)
12
13    frame_body += padding_bits
14
15    return frame_body
16
17 def add_char_count_flag_to_frame(frame_body: List[bool]) -> List[bool]:
18
19     # Aplica padding
20     frame_body = apply_8bit_padding(frame_body)
21
22     # Calcula numero de bytes
23     num_of_bytes: int = len(frame_body) // 8
24
25     # Codifica numero de bytes como um cabecalho
26     header: List[bool] = int_to_bool_list(num_of_bytes, 8)
27
28     frame_body = header + frame_body
29
30     return frame_body

```

Por fim, a função `add_padding_for_4bit_alignment` assegura que o número total de bits seja múltiplo de quatro antes que os quadros sejam enviados à camada física. Esse alinhamento é necessário para permitir a correta utilização de protocolos de modulação como QPSK e 16-QAM, cujos símbolos representam múltiplos fixos de 2 ou 4 bits. A função adiciona também um bloco final de quatro bits que codifica a quantidade de preenchimento aplicada.

No receptor, o processo inverso é realizado. O trem de bits recebido pode conter erros devido ao ruído inserido na camada física. A primeira etapa, realizada pela função `remove_padding_for_4bit_alignment`, remove os bits que foram utilizados para ajustar o tamanho da mensagem a um múltiplo de quatro. Na segunda, `remove_framing_protocol` retira os bits relacionados ao enquadramento e devolve a lista de pacotes correspondentes. Nessa etapa, também é removido o preenchimento inserido nos métodos de enquadramento baseados em bytes.

Posteriormente, a função `ECC_fix_corrupted_bits` aplica o código de Hamming para corrigir erros de um único bit presentes nos pacotes, e a função `remove_ECC` descarta os bits de redundância utilizados pelo ECC. A função `find_corrupted_frames` percorre os pacotes e gera uma lista indicando quais deles foram identificados como corrompidos pelos protocolos de detecção de erros.

Em seguida, a função `remove_EDC` remove os bits de detecção de erros dos pacotes já corrigidos. Por fim, a função `remove_paddings` verifica no último pacote quantos bits de preenchimento haviam sido adicionados no transmissor. Caso o último pacote esteja corrompido, ele é simplesmente descartado; caso contrário, a função descarta o último pacote e utiliza o valor armazenado nesse último pacote para remover os bits excedentes do penúltimo pacote. O resultado final é o trem de bits que representa a mensagem recebida pelo destinatário.

2.3 Interface de Usuário

A Interface de Usuário consiste em uma janela principal a qual permite a inserção de todos os parâmetros desejados de uma simulação (presentes no arquivo `InterfaceGrafica.py`). Seja a escolha de modulação, protocolo de enlace, códigos de erro, quantidade de samples da onda, amplitude, frequência, etc.. Todos os blocos estão devidamente descritos. Para realizar a simulação, basta clicar no botão "Executar simulação", e janelas extras — cada qual com um gráfico relevante — são abertas para análise da simulação.

Por dentro do código, a janela principal consiste apenas de uma janela simples do GTK com um único objeto, uma grade. Outros campos de entrada são, então, organizados na grade, a fim de facilitar a organização e visualização para o usuário. Os outros campos consistem de texto, entrade de texto, dropdown e checkboxes. Segue um exemplo de construção da grade e do dropdown de escolhas para codificação de portadora na Listagem 7

Listagem 7: Construção do simulador

```

1 import gi
2 gi.require_version('Gtk', '3.0')
3 from gi.repository import Gtk
4
5 # ...
6
7 class Programa(Gtk.Window):
8     def __init__(self):
9         Gtk.Window.__init__(self, title="Simulador de Camadas Física e"
10                           Enlace")
11         # self.set_default_size(1600, 900)
12         # Layout básico

```

```

13     self.grid = Gtk.Grid()
14     self.grid.set_column_spacing(15)
15     self.grid.set_row_spacing(10)
16     self.grid.set_border_width(10)
17     self.add(self.grid)
18
19     # Entradas de input simples
20     self.input_section()
21     self.digmod_section()
22     self.algmod_section()
23     self.framing_section()
24     self.errordetct_section()
25     self.errorcurve_section()
26     self.extraparam_smpls()
27
28     # Botao Run()
29     self.run_button = Gtk.Button(label="Executar Simulação")
30     self.run_button.connect("clicked", self.on_run_clicked)
31     self.grid.attach(self.run_button, 9, 0, 2, 2)
32
33     #
34
35 def algmod_section(self):
36     # Analog modulation options
37     analog_label = Gtk.Label(label="Modulação por Portadora ->")
38     self.grid.attach(analog_label, 3, 1, 1, 1)
39
40     self.algmod_option = Gtk.ComboBoxText()
41     self.algmod_option.append_text(GUI_ASK)
42     self.algmod_option.append_text(GUI_FSK)
43     self.algmod_option.append_text(GUI_BPSK)
44     self.algmod_option.append_text(GUI_QPSK)
45     self.algmod_option.append_text(GUI_16QAM)
46     self.algmod_option.set_active(0)
47     self.grid.attach(self.algmod_option, 4, 1, 2, 1)
48
49 #

```

As janelas dos gráficos são ainda mais simples. Usando do `FigureCanvas` disponibilizado pelo backend GTK do `matplotlib`, as figuras são colocadas em janelas individuais, para permitir a movimentação e comparação entre gráficos. Tanto a versão digital como analógica da visualização de gráficos, como também o display de texto, está implementada na Listagem 8.

Listagem 8: Output de texto e gráficos

```

1 import gi
2 gi.require_version('Gtk', '3.0')
3 from gi.repository import Gtk
4 import matplotlib.pyplot as plt
5 from matplotlib.backends.backend_gtk3agg import FigureCanvasGTK3Agg as
6   FigureCanvas
7
8
9 class Programa(Gtk.Window):
10
11     #
12
13     def analog_plot(self, signal, title):

```

```

14     mod = self.algmod_option.get_active_text()
15     graph = plt.Figure()
16     ax = graph.add_subplot()
17     ax.plot(signal)
18     ax.set_title(f"Sinal Analógico")
19     ax.set_xlabel("Amostras")
20     ax.set_ylabel("Leitura de tensão")
21
22     canvas = FigureCanvas(graph)
23     canvas.set_size_request(800, 400)
24     win = Gtk.Window(title=f"Analógico - {title}")
25     win.add(canvas)
26     win.show_all()
27
28 def digital_plot(self, dsignal, title):
29     graph = plt.Figure()
30     ax = graph.add_subplot()
31
32     mod = self.digmod_option.get_active_text()
33
34     if title == GUI_MAN:
35         amnt_smpl = len(dsignal) // 2
36     else:
37         amnt_smpl = len(dsignal)
38
39     x = np.linspace(0, amnt_smpl, amnt_smpl, endpoint=False)
40
41     ax.plot(x, dsignal, drawstyle="steps-pre")
42     ax.set_title(f"Sinal Digital")
43     ax.set_xlabel("Amostras")
44     ax.set_ylabel("Leitura de tensão")
45     ax.set_xticks(np.arange(0, amnt_smpl + 1))
46     ax.set_xlim(0, amnt_smpl)
47
48     canvas = FigureCanvas(graph)
49     canvas.set_size_request(800, 400)
50     win = Gtk.Window(title=f"Digital - {title}")
51     win.add(canvas)
52     win.show_all()
53
54 def text_show(self, message):
55     # Display de texto
56     label = Gtk.Label(label=message)
57     scrlb = Gtk.ScrolledWindow()
58     scrlb.add(label)
59     scrlb.set_size_request(600, 600)
60     win = Gtk.Window(title=f"Output de texto")
61     win.add(scrlb)
62     win.show_all()
63
64 # ...

```

Por fim, o funcionamento interno consiste do uso de sinais, funcionalidade própria do GTK. Um sinal é enviado ao requisitar a execução da simulação, e então os valores providenciados são verificados e, por fim, a simulação é realizada.

3 Membros

3.1 Henrique França

As contribuições realizadas iniciam-se pela camada física, com a implementação das modulações digitais NRZ-Polar, Manchester e Bipolar. Foi desenvolvida tanto a modulação quanto a desmodulação de cada técnica, assegurando que os sinais digitais fossem representados corretamente.

Na camada de enlace, foram implementados os métodos de enquadramento por Contagem de Caracteres e por FLAGS com inserção de bytes. Esses protocolos foram integrados de forma consistente ao restante da estrutura do simulador.

3.2 José Antônio de Andrade

Suas contribuições consistem, principalmente, da implementação da interface gráfica. Realizou todo o sistema de funcionamento e interação com o usuário, desde cheques relacionados à atribuição de valores até a projeção de gráficos na tela. Também, neste escopo, construiu o sistema de conversa via sockets presente no trabalho.

Adicionalmente, realizou os protocolos de (de)codificação de portadora ASK, FSK e BPSK, e ajudou na produção deste relatório.

3.3 Rafael Sapienza

Na camada física, implementou os esquemas de modulação QPSK e 16-QAM. Também implementou os algoritmos de demodulação desses protocolos.

No desenvolvimento da camada de enlace, desenvolveu uma função que recebe o trem bruto de bits e o segmenta em payloads de 32 bits, adicionando um pacote de padding quando o último bloco não possui o tamanho necessário. Além disso, implementou os protocolos de detecção de erros de checksum e de CRC32, bem como o mecanismo de correção de erros (ECC) baseado no código de Hamming, incluindo as rotinas de cálculo e inserção dos bits de redundância. Para o enquadramento, implementou o método baseado em flags com inserção de bits, abrangendo as funções de enquadramento e de desenquadramento.

4 Conclusão

A principal característica dessas simulações, mesmo com otimizações, é a demanda computacional excessiva quando implementada por software. Devidamente, esses protocolos foram feitos para serem realizados com circuitos físicos, principalmente em relação à Camada Física. Mesmo assim, o programa pode ser útil para a análise de sistemas sob influência de ruído constante, para testar os limites e configurações necessárias para a transmissão de mensagens.

A Camada Física não apresentou dificuldades em sua implementação, mas a Camada de Enlace forneceu diversos problemas durante o desenvolvimento. Foi necessário testar diversos casos extremos para identificar problemas que surgiram. Notavelmente, a aplicação de padding e sua devida remoção foram campos de tediosa implementação. Por sua vez, a Interface de Usuário foi outra área problemática, principalmente para introduzir os gráficos da biblioteca matplotlib de forma correta.

De todo modo, o trabalho providenciou uma visão mais ampla dos diversos protocolos e algoritmos por trás da comunicação comum do dia a dia, não só como são realizados e produzidos, mas também das diversas dificuldades presentes no estabelecimento de um protocolo tanto eficiente quanto prático.