

Interface com o Utilizador Baseada em *Threads*

Pedro Silva¹, Ésau Cardoso², Fernando Carvalho¹, and António Sousa^{1,3}

¹ ISEP - IPP, Porto, {f1160635, fjc}@isep.ipp.pt

² IEP, Porto, esau.cardoso@iep.pt

³ LEMA, ISEP-IPP, Porto, ats@isep.ipp.pt

Resumo. A Compatibilidade Eletromagnética traduz-se pela capacidade de todo o equipamento elétrico funcionar corretamente num ambiente onde ocorrem perturbações eletromagnéticas suscetíveis de gerar interferências em si mesmo, noutros equipamentos, ou receber essas interferências de equipamentos externos. Baseia-se em ensaios de imunidade eletromagnética que se regem pela norma IEC/EN61000-4-6, constituindo um dos requisitos para atribuição da marcação CE.

No Instituto Eletrotécnico Português, tem-se vindo a desenvolver uma aplicação computacional em Python para auxiliar os especialistas na realização deste ensaio, nomeadamente para controlo de dispositivos experimentais e registo de observações. Uma vez em funcionamento, na primeira versão da aplicação computacional, observaram-se bloqueios intermitentes da interface gráfica com o utilizador que foram reportados pelos seus utilizadores. Assim, no sentido de se superar o referido constrangimento, neste trabalho apresenta-se uma nova versão da aplicação computacional baseada no uso de *threads* e a implementação de novas tarefas que haviam sido apontadas na versão inicial como limitações e que agora são processadas em paralelo com a aplicação.

Palavras-chave: GUI, *Threads*, PyQt5, IEC/EN61000-4-6.

1 Introdução

A Compatibilidade Eletromagnética (EMC) de equipamentos elétricos assegura em parte que os mesmos chegam ao mercado de acordo com os requisitos que norteiam os seus processos de fabrico [1–4]. É finalmente a marcação CE que garante que foram devidamente avaliados pelo fabricante e que cumprem todos os requisitos normativos da UE em termos de segurança e de compatibilidade com o ambiente eletromagnético a que se destinam. A EMC considera como um dos requisitos para a certificação de equipamentos elétricos, a realização de ensaios de imunidade de tensões rádio frequência (RF) conduzidas, em conformidade com a norma IEC/EN61000-4-6 [3]. O ensaio baseia-se no esquema da Fig. 1 e avalia o comportamento de aparelhos com um cabo de alimentação ou de comunicação superior a 3 metros, quando expostos a distúrbios eletromagnéticos em RF transmitidos por sinais sinusoidais de frequência variável entre 150 kHz e 80 MHz [8]. A montagem é constituída por um gerador de sinais ligado ao computador e a um amplificador e este a um atenuador que está ligado a um

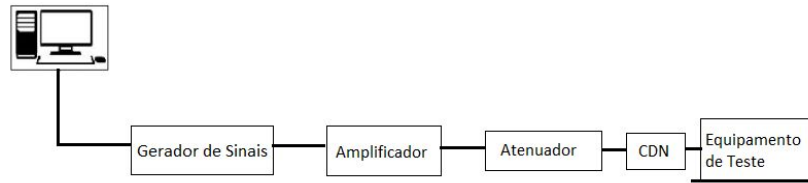


Fig. 1. Esquema da montagem dos instrumentos no ensaio de imunidade eletromagnética de dispositivos elétricos.

instrumento de acoplamento (CDN). O equipamento de teste é acoplado a um CDN específico que garante uma impedância em modo comum de $150\ \Omega$ ao longo do seu circuito - condição necessária para garantir a integridade dos seus componentes e impedir que os sinais de teste afetem outros instrumentos próximos. Esta montagem encontra-se instalada em [5] e os procedimentos relativos aos ensaios encontram-se descritos com detalhe em [8].

No sentido de auxiliar a realização de ensaios de imunidade tem-se vindo a desenvolver uma aplicação computacional em Python [8], a partir do *framework* PyQt5 [6] para a interface com o utilizador (GUI) e *pySerial* [7] para comunicações, para estabelecer a ligação entre o computador e o gerador de sinais via porta série RS232. A aplicação permite que o utilizador controle o gerador de sinais pela porta série e realize os ensaios de imunidade eletromagnética com registo de observações e parâmetros de teste. No entanto, verificou-se que esta primeira versão da aplicação bloqueia a GUI durante a realização de cada ensaio, impedindo o utilizador de aceder a outros recursos desta aplicação, assim como a aplicações diversas. Este efeito impede em cada ensaio e em tempo real, a sua suspensão e repetição sempre que se mostre discrepante com os requisitos da norma IEC/EN61000-4-6, assim como o registo de notas por parte do utilizador durante a realização do ensaio e processadas pela aplicação, a incluir no relatório final do ensaio.

Em [8], esta limitação foi minimizada a partir de um evento que decorria em intervalos de tempo muito curtos, através da classe `QTimer` do PyQt5. Dado que a primeira versão não é uma solução ótima, este trabalho apresenta uma versão melhorada da aplicação, recorrendo ao uso da metodologia baseada em *threads*, descrita nas secções dois e três deste documento. Na quarta secção mostram-se resultados obtidos e termina-se na secção seguinte com algumas conclusões da abordagem proposta e perspetivas de trabalho futuro.

2 Eventos, *threads* e processos

As aplicações com GUI são orientadas por eventos. Os eventos são ações do utilizador sobre a interface, às quais a aplicação deve responder. Por vezes esses eventos podem ter outra origem, sendo gerados por ligações ao meio exterior, por cronómetros, etc. Para que uma GUI seja funcional, deve responder ao utilizador sempre que este atua na interface, mas quando um procedimento leva mais

tempo a ser executado, a GUI pode ficar pendente e o utilizador vê-se impedido de realizar outras tarefas. Assim, a GUI deve ser planeada no sentido de prever quais as tarefas que a podem bloquear e correr essas tarefas de forma paralela em conjunto com a tarefa principal dedicada à sua gestão. Para o efeito, existem dois modos de realizar este processamento paralelo: por lançamento de processos independentes (*process forks*) e por lançamento de novos *threads*. Funcionalmente ambos os modos executam tarefas em paralelo, mas só os *threads* permitem que o mesmo código corra em diferentes sistemas operativos e é mais fácil proceder à comunicação entre diferentes *threads* da mesma aplicação, dada a possibilidade de partilharem variáveis globais no programa principal que podem ser alteradas por qualquer um deles [9].

Uma GUI desenvolvida em Python tem ainda o problema adicional de correr apenas sob um *thread* e por isso não permite o lançamento de processos (limitação em algumas implementações do Python para o Windows) [10].

No presente trabalho as soluções possíveis para resolver o problema do bloqueio da GUI exigem o processamento paralelo de eventos, ou seja, é necessário paralelizar a gestão da GUI com a gestão e controlo dos dispositivos experimentais e também com o lançamento dos ensaios de teste.

2.1 Implementação de *threads* em Python usando a biblioteca padrão *threading*

A biblioteca padrão *threading* do Python implementa os métodos necessários para criar *threads*. Depois de se criar uma nova instância da classe, recorrendo à função *Thread* - cujos parâmetros são o nome da função que vai correr em paralelo, o nome do *thread* e os argumentos de chamada da função - deve-se iniciar com o método *start()* [10]. Na Fig. 2 mostra-se um exemplo básico de aplicação com o lançamento de vários *threads* para executar cópias da função *runFunction* em paralelo, durante intervalos de tempo diferentes.

2.2 Implementação em PyQt5

O PyQt5 possui a classe *QThread* para a implementação baseada em *threads* de forma similar à classe *Thread* da biblioteca *threading*. Se a função a correr em paralelo for um método de um objeto PyQt5, é possível mover essa função para uma instância *QThread* previamente criada, utilizando o método *moveToThread(.)* do objeto.

Apesar da haver várias vias de implementação, o modo recomendado baseia-se nas classes *QRunnable* e *QThreadPool* do módulo *QtCore* da biblioteca PyQt5 [11, 12]. A classe *QThreadPool* controla a fila de execução dos *threads* e recupera os resultados. A classe *QRunnable* permite criar subclasses, designadas por *Workers*, personalizadas pela função a executar que se deverá colocar no método *run()*. Para executar a função em paralelo, basta criar uma nova instância do *Worker* e passar à instância do *QThreadPool* para ser colocada na fila de execução. Na Fig. 3 mostra-se o mesmo exemplo da Fig. 2 implementado em PyQt5, no qual os resultados permitem observar a fila de execução dos *threads*.

```

1 # -*- coding: utf-8 -*-
2 from threading import Thread
3 import time
4 from random import seed, randint
5
6 seed(5)
7 N= randint(2,10)
8 A=[randint(5,20) for i in range(N)]
9 B=[0 for i in range(N)]
10 # função a correr em paralelo pelos threads
11 def runFunction(i, nome):
12     """A cada segundo incrementa B[i] até terminar, passando B[i]=0"""
13     print(nome, "vive", A[i], "segundos")
14     for j in range(A[i]+1):
15         time.sleep(1)
16         B[i]=B[i]+1
17     B[i]=0
18     print(nome, "terminou\n")
19
20 # Bloco principal do programa
21 if __name__ == "__main__":
22     # Lançar os threads
23     for i in range(N):
24         nome="ThreadN"+str(i)
25         # Novo thread com a função a executar, nome e argumentos
26         x = Thread(target=runfunction, name=nome,
27                   args=(i,nome))
28         x.start()
29         time.sleep(0.1)
30     print("\n\nTodos os threads lançados\n")
31     # Avaliar a cada segundo o estado de cada thread
32     for t in range(max(A)+5):
33         print("t =", t, "- Estado:", B)
34         time.sleep(1)
35     print("\nFIM")

```

Fig. 2. Exemplo baseado no uso da biblioteca padrão `threading`.

Nesta abordagem, é necessário criar uma classe diferente sempre que for necessário criar um *thread* com uma tarefa diferente daquela especificada na função `run()`. No entanto, utilizando as funcionalidades da classe `QThreadPool` é possível criar *threads* de uma função específica sem necessitar de colocar a tarefa pedida na função `run()`. Desta forma só é necessário especificar uma classe `Worker` e será a partir desta que todos os *threads* serão criados.

3 Metodologia

Dadas as vantagens e limitações de cada uma das metodologias, expostas na secção anterior, utilizaram-se *threads* para a resolução do problema do bloqueio da interface. Como a GUI está desenvolvida em PyQt5, os *threads* são criados, recorrendo à abordagem baseada nas classes `QRunnable` e `QThreadPool`.

Na GUI inicial, a função que gere os ensaios e a comunicação com os dispositivos externos é a causa do bloqueio da aplicação, mas como necessita de ser um método da Interface - classe que gere a GUI - foram utilizadas instâncias da classe `QThreadPool` para controlar a implementação e lançamento de *threads* a partir da Interface. Na Fig. 4 mostra-se o *thread* principal, responsável pelo

```

1 # -*- coding: utf-8 -*-
2 import time,sys
3 from random import seed, randint
4 from PyQt5.QtCore import QApplication, QRunnable, QThreadPool
5
6 seed(0)
7 N= randint(2,10) # Semente para números aleatórios
8 A=[randint(5,20) for i in range(N)] # Selecionar número de threads
9 B=[0 for i in range(N)] # Tempo de vida de cada thread
10 # variável alterada por cada thread
11
12 class Worker(QRunnable): # Criar a subclasse para threads
13     def __init__(self,j):
14         super(Worker, self).__init__()
15         self.j=j # Número do thread
16     def run(self): # Função a correr em cada thread
17         app = QApplication.instance()
18         print("Worker", self.j, "vive", A[self.j], "segundos")
19         for t in range(A[self.j]+1):
20             B[self.j] +=1 # Incrementar B[j] em cada segundo
21             time.sleep(1)
22         B[self.j]=0
23         print("Worker", self.j, "terminou")
24         app.exit()
25
26 if __name__ == "__main__":
27     app = QApplication([])
28     threadpool=QThreadPool() # Nova instância da fila de threads
29     print("Número máximo de threads:", threadpool.maxThreadCount())
30     for j in range(N):
31         threadpool.start(Worker(j)) # Colocar um novo thread na fila
32         time.sleep(0.05)
33     for i in range(21):
34         print("t =",i,"\tB = ",B) # Escrever estado de B a cada segundo
35         time.sleep(1)
36     print("Aplicação terminou...")
37     sys.exit(app.exec_())

```

Fig. 3. Exemplo baseado no uso do PyQt5 com QRunnable e QThreadPool.

funcionamento da GUI e dois *threads* que correm em paralelo com aplicação: o *thread* de gestão de dispositivos e o *thread* de gestão de ensaios.

É de salientar que como estes dois *threads* foram criados como objetos da Interface, então a classe **Worker** também terá acesso aos métodos e atributos definidos na Interface. Isto permite que a gestão de dispositivos bem como a gestão de ensaios (lançamento de ensaios e controlo de comunicações) sejam executadas por *threads* diferentes do da GUI.

Para além da resolução do problema apresentado, a mais-valia desta nova versão da aplicação reside na implementação da função que gere os dispositivos necessários à realização dos ensaios, correndo sempre em paralelo com a GUI. Isto foi possível graças às funcionalidades fornecidas pela classe **QThreadPool**. Esta nova função gere as portas que estão conectadas a dispositivos externos a partir do protocolo de comunicação RS-232 ou dispositivos GPIB (*General Purpose Interface Bus*), usando as bibliotecas **pySerial** e **pyVisa**, respetivamente. Os dispositivos detetados nas portas do computador são registados em listas que os outros *threads* acedem sempre que necessário, sendo atualizada a cada segundo. No caso de um dispositivo ser desconectado será lançado um alerta para avisar

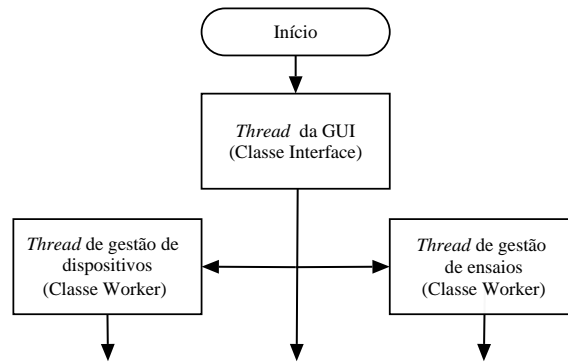


Fig. 4. Esquema do funcionamento dos *threads* na aplicação.

o utilizador do ocorrido e o ensaio não poderá ser realizado. Na Fig. 5 mostra-se o fluxograma completo desta função de gestão de dispositivos.

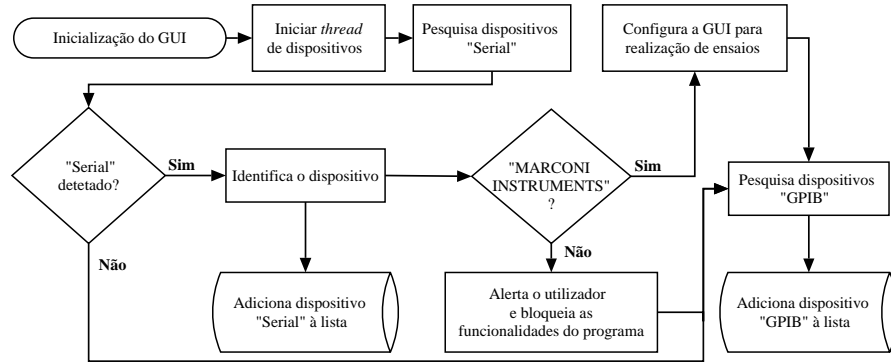
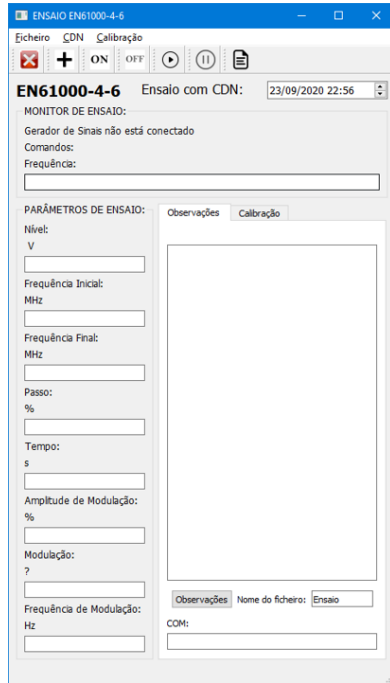


Fig. 5. Fluxograma da função de gestão de dispositivos.

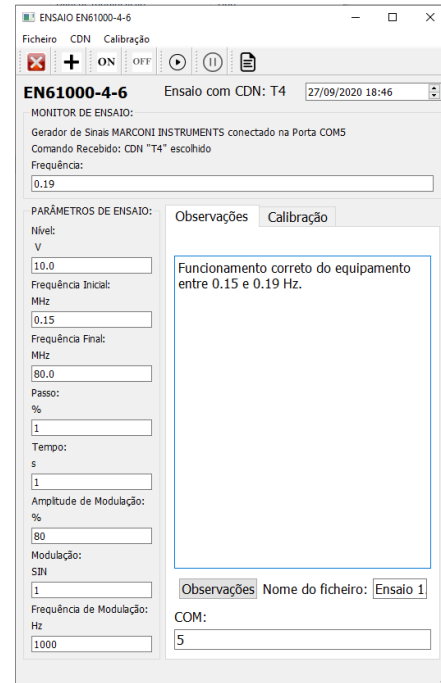
4 Resultados - protótipo da GUI

A nova versão da aplicação computacional encontra-se em funcionamento em [5] e satisfaz, de acordo com as opiniões recolhidas, os requisitos dos vários utilizadores. A interface é gerida pelo *thread* principal ilustrado na Fig. 6(a), onde se localizam os campos relativos aos parâmetros do ensaio e os botões que permitem a sua monitorização. Na Fig. 6(b) apresenta-se a GUI da janela de ensaios controlada pelo *thread* de gestão de ensaios.

Finalmente, na Fig. 7 apresentam-se imagens do funcionamento da aplicação sobre o estado conectado/não conectado de dispositivos necessários à realização do ensaio e de alertas ao utilizador sobre o estado da conexão, controlado pelo *thread* de gestão de dispositivos.

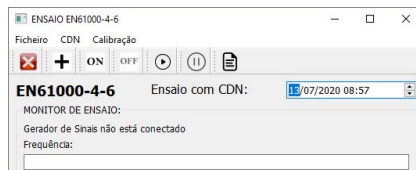


(a) GUI sem gerador de sinais.

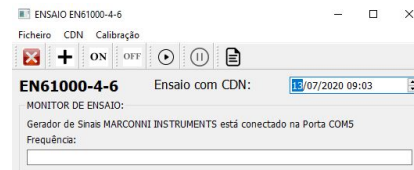


(b) GUI em ensaio.

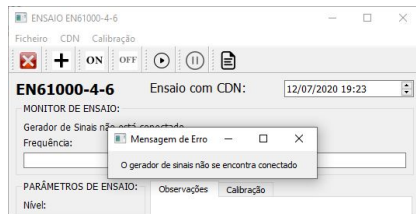
Fig. 6. GUI final da aplicação suportada pelo *thread* principal em (a) e pelo *thread* de gestão de ensaios em (b).



(a) Gerador não detetado.



(b) Gerador detetado.



(c) Gerador desconectado.

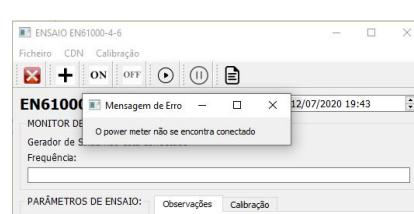
(d) Outro dispositivo desconectado.
(Power meter).

Fig. 7. A aplicação procura os dispositivos e alerta o utilizador de anomalias, usando as funcionalidades suportadas pelo *thread* de gestão de dispositivos.

5 Conclusão

Este trabalho foi desenvolvido na linguagem de programação Python, com recurso à framework PyQt no sentido de melhorar a primeira versão da aplicação, recorrendo nesta nova versão ao uso de *threads* para ultrapassar a falta de responsividade da GUI durante os ensaios. Para o efeito apresentam-se duas abordagens de implementação baseadas no uso de *threads*, a primeira usa a biblioteca padrão `threading` e a segunda as classes `QRunnable` e `QThreadPool` do PyQt5. No entanto, foi usada a segunda, dado o facto da GUI se encontrar desenvolvida em PyQt5, pese embora, a primeira abordagem também fosse facilmente implementável. Definiu-se o uso de três *threads*: o principal, dedicado à GUI e outros dois, um dedicado à gestão dos dispositivos e o outro à gestão dos ensaios.

De referir que a GUI inicial apenas comunicava com dispositivos série ligados a uma porta USB pré-definida na aplicação, tendo esta nova versão introduzido um *thread* para gestão de dispositivos, que permite controlar as portas que estão conectadas a dispositivos externos a partir do protocolo de comunicação RS-232 ou dispositivos GPIB usando as bibliotecas `pySerial` e `pyVisa`.

Este projeto pode ainda ser melhorado futuramente nas vertentes de implementação de processos automáticos para execução de tarefas de rotina, que correm no *thread* de gestão de ensaios, de modo a agilizar os procedimentos.

Referências

1. L. Muriel, Introdução à Compatibilidade Eletromagnética em Conversores Estáticos, (1999)
2. C. Sartori, Aspecto de Compatibilidade Electromagnética em Estruturas Atingidas por Descarga Atmosféricas, (1999)
3. IEC, EMC The Role and Contribution of IEC standards, (2001)
4. International Electrotechnical Commission, Electromagnetic compatibility (EMC) – Part 4-6: Testing and measurement techniques – Immunity to conducted disturbances induced by radio-frequency fields, Edição 4.0, Outubro 2013
5. Instituto Electrotécnico Português, Os Nossos Serviços, <https://www.iep.pt/servicos/>, Acedido em 18 de Maio de 2020
6. PyQt5 - Python bindings for the Qt cross platform application toolkit <https://pypi.org/project/PyQt5/> Acedido em 18 de setembro de 2020
7. pySerial - Python Serial Port Extension <https://pypi.org/project/pyserial/> Acedido em 18 de setembro de 2020
8. C. Duarte, E. Cardoso, F. Carvalho e A. Sousa, Automatização do Ensaio de Imunidade a Tensões Rádio Frequências Conduzidas, apresentado ao SEI'19 a 4 de dezembro de 2019, Porto, Portugal
9. N. Matloff e F. Hsu, Tutorial on Threads Programming with Python, University of California, (2009)
10. Mark Lutz, Programming Python, 4th Edition, O'Reilly Media, Inc. ISBN: 9780596158101 (2010)
11. M. Fitzpatrick, Create Simple GUI Applications with Python & Qt5, (2016)
12. M. Fitzpatrick, Multithreading PyQt applications with QThreadPool, <https://www.learnpyqt.com/courses/concurrent-execution/multithreading-pyqt-applications-qthreadpool/>, Acedido em 18 de setembro de 2020