UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

LUCAS HENRIQUE KELNIAR

MELHORIA DO DESEMPENHO DE PROCESSAMENTO DE ALGORITMOS DE RECONSTRUÇÃO DE IMAGENS EM ENSAIOS NÃO DESTRUTIVOS POR ULTRASSOM UTILIZANDO AS FERRAMENTAS CYTHON E NUMBA

PATO BRANCO

2021

LUCAS HENRIQUE KELNIAR

MELHORIA DO DESEMPENHO DE PROCESSAMENTO DE ALGORITMOS DE RECONSTRUÇÃO DE IMAGENS EM ENSAIOS NÃO DESTRUTIVOS POR ULTRASSOM UTILIZANDO AS FERRAMENTAS CYTHON E NUMBA

Improving the processing performance of image reconstruction algorithms in ultrasonic nondestructive testing using Cython and Numba tools

Trabalho de conclusão de curso de graduação apresentada como requisito para obtenção do título de Bacharel em Engenharia Elétrica da Universidade Tecnológica Federal do Paraná (UTFPR).

Orientador(a): Prof. Dr. Giovanni Alfredo Guarneri

PATO BRANCO

2021

LUCAS HENRIQUE KELNIAR

MELHORIA DO DESEMPENHO DE PROCESSAMENTO DE ALGORITMOS DE RECONSTRUÇÃO DE IMAGENS EM ENSAIOS NÃO DESTRUTIVOS POR ULTRASSOM UTILIZANDO AS FERRAMENTAS CYTHON E NUMBA

Trabalho de conclusão de curso de graduação apresentada como requisito para obtenção do título de Bacharel em Engenharia Elétrica da Universidade Tecnológica Federal do Paraná (UTFPR).

Data de aprovação: 30/novembro/2021

_____

Diogo Ribeiro Vargas
Doutor
Universidade Tecnológica Federal do Paraná – Câmpus Pato Branco

_____

Giovanni Alfredo Guarneri
Doutor
Universidade Tecnológica Federal do Paraná – Câmpus Pato Branco

_____

Gustavo Weber Denardin
Doutor
Universidade Tecnológica Federal do Paraná – Câmpus Pato Branco

PATO BRANCO

2021

Dedico este trabalho a todos que se lançam em jornadas árduas, e que, no percurso, começam a entender melhor quem são de verdade.
E a todos que tornam a jornada possível de percorrer.

AGRADECIMENTOS

Agradeço primeiramente a Deus, e que minha gratidão por tudo que me propicia nunca cesse.

Ao meus pais, Celso Kelniar e Loici Caumo Kelniar, que sempre moveram tudo, e até mais, que fosse preciso para que eu obtivesse todas as condições para ter um ensino de qualidade. E a todos familiares que também me deram seu apoio de alguma forma.

Ao meu orientador, Giovanni Alfredo Guarneri, por inicialmente ter me aceito e confiado no meu potencial. Agradeço pela sugestão do tema, que foi do meu gosto, pela disposição de ajudar, pela paciência, pelas contribuições exaustivas que teve para o cumprimento deste trabalho, e por tudo que me ensinou.

Aos amigos, que também me ensinaram e foram veículos para o entendimento de elementos valiosos para o cumprimento deste trabalho.

Aos demais amigos que eternizei, a todos momentos e experiências que compartilhamos, e a todas que hão de vir.

A todos os professores que me conduziram e me formaram nas suas áreas de conhecimento. Obrigado por terem aceito essa verdadeira missão, tão valiosa pro mundo, de ensinar.

A todos os que de alguma forma contribuíram para o cumprimento desta etapa que vivenciei.

Há três caminhos para o fracasso: não ensinar o que se sabe, não praticar o que se ensina, e não perguntar o que se ignora.

– São Beda

RESUMO

KELNIAR, Lucas Henrique. **Melhoria do desempenho de processamento de algoritmos de reconstrução de imagens em ensaios não destrutivos por ultrassom utilizando as ferramentas Cython e Numba**. 2021. 71 f. Trabalho de Conclusão de Curso (Bacharelado em Engenharia Elétrica) – Universidade Tecnológica Federal do Paraná. Pato Branco, 2021.

Este trabalho tem o objetivo de avaliar o uso das ferramentas *Numba* e *Cython* na otimização do tempo de processamento de códigos *Python* utilizados para reconstrução de imagens com dados oriundos de inspeções não destrutivas por ultrassom. Foram avaliados somente os algoritmos de reconstrução de imagens baseados no princípio de atraso-e-soma e que são amplamente utilizados: *Synthetic Aperture Focusing Technique* (SAFT), o *Total Focusing Method* (TFM) e o *Coherent Plane Wave Compounding* (CPWC). Esse trabalho propõe utilizar técnicas de adaptação dos códigos, já desenvolvidos em um *framework*, para melhorar o desempenho no tempo de execução, utilizando as ferramentas *Numba* e *Cython*. É feita uma avaliação das implementações dessas técnicas pela medição e comparação dos tempos de execução. Foi observado uma melhora de aproximadamente 4 vezes para o SAFT, 8 vezes para o TFM e 388 vezes para o CPWC. Todos são os melhores resultados, e correspondem ao uso do *Numba*, porém o *Cython* também exibiu melhoras nos tempos de execução com aplicação das técnicas. Isso mostra que as ferramentas são eficazes para melhorar o desempenho de execução dos algoritmos de reconstrução de imagens.

Palavras-chave: Ensaios Não Destrutivos. Ultrassom. Reconstrução de Imagens. Python.

ABSTRACT

KELNIAR, Lucas Henrique. **Improving the processing performance of image reconstruction algorithms in ultrasonic nondestructive testing using Cython and Numba tools**. 2021. 71 p. Bachelor Thesis (Bachelor's Degree in Electrical Engineering) – Universidade Tecnológica Federal do Paraná. Pato Branco, 2021.

This work aims to evaluate the use of *Numba* and *Cython* tools in optimizing the processing time of *Python* codes used for image reconstruction from nondestructive ultrasound inspections data. Only the image reconstruction algorithms that are based on the principle of delay-and-sum and that are widely used were evaluated: *Synthetic Aperture Focusing Technique* (SAFT), the *Total Focusing Method* (TFM) and the *Coherent Plane Wave Compounding* (CPWC). This paper proposes to use techniques to adapt codes, already developed in a *framework*, to improve runtime performance using the *Numba* and *Cython* tools. An evaluation of the implementations of these techniques is made by measuring and comparing execution times. An improvement of approximately 4 times for SAFT, 8 times for TFM and 388 times for CPWC was observed. All are the best results, and correspond to the use of *Numba*, but *Cython* also exhibited improvements in execution times with application of the techniques. This shows that the tools are effective in improving the runtime performance of image reconstruction algorithms.

Keywords: Non-Destructive Testing. Ultrasound. Image Reconstruction. Python.

LISTA DE ALGORITMOS

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ABENDI	Associação Brasileira de Ensaios Não Destrutivos.
AOT	*Ahead of time* (À frente do tempo de execução)
API	*Application programming interface* (Interface de programação de aplicações)
AUSPEX	*Advanced Ultrasound Signal Processing for Equipment Inspections* (Técnicas Avançadas de Processamento de Sinais de Ultrassom para Inspeções de Equipamentos)
CPU	*Central processing unit* (Unidade central de processamento)
CPWC	*Coherent plane wave compounding* (Composição coerente de ondas planas)
DAS	*Delay-and-sum* (Atraso e soma)
END	Ensaio não destrutivo
FMC	*Full matrix capture* (Captura por matriz completa)
GPU	*Graphics processing unit* (Unidade de processamento gráfico)
JIT	*Just-in-time* (Em tempo de execução)
PO	*Python object* (Objeto Python).
PWI	*Plane wave imaging* (Imageamento por ondas planas)
ROI	*Region of interest* (Região de interesse)
SAFT	*Synthetic aperture focusing technique* (Técnica de abertura por foco sintético)
TFM	*Total focusing method* (Método de focalização total)

SUMÁRIO

1 INTRODUÇÃO

A humanidade está sempre em busca de técnicas e tecnologias que possibilitam trazer, cada vez mais, segurança, confiabilidade, utilidade e qualidade aos produtos e serviços destinados à sociedade. Uma das maneiras que essas técnicas e tecnologia podem atingir isso, é através da verificação dos produtos envolvidos no processo industrial. Os ensaios não destrutivos (ENDs), vem como uma resposta a isso. São técnicas que tornaram possível avaliar diversas características e as qualidades de um material sendo produzido ou que está sendo utilizado no processo de fabricação, e identificam características como: tamanho, composição, defeitos e suas dimensões, dentre outras. E, fazer isso sem comprometer as características, as condições de trabalho, as qualidades e sem corromper esse material de nenhuma forma (HELLIER, 2003). A aplicação dos ENDs trazem uma contribuição para a qualidade dos bens e serviços, redução de custos nos processos envolvidos e a preservação da vida e do meio ambiente onde são aplicados, é muito popular na indústria e tem uma vasta aplicação inclusive na área médica (ABENDI, 2014).

O Brasil possui uma grande matriz de extração de petróleo em alto mar (ou *off shore*), sendo que inspeções frequentes para dimensionar e monitorar falhas nessas tubulações são de extrema importância para garantir a integridade dos procedimentos em execução. Este trabalho faz parte do projeto AUSPEX (sigla em inglês para *Advanced Ultrasound Signal Processing for Equipment Inspections* – Técnicas Avançadas de Processamento de Sinais de Ultrassom para Inspeções de Equipamentos) que tem como objetivo principal desenvolver ferramentas de análise de ENDs por ultrassom para auxiliar nas inspeções de tubulações submarinas de petróleo.

O projeto tem diversas frentes de trabalho, e em uma dessas frentes foi desenvolvida uma ferramenta que disponibiliza uma interface gráfica para a leitura, manipulação e o processamento de dados de inspeções por ultrassom. Atualmente, essa ferramenta atende diversas funcionalidades, distribuídas nas etapas de leitura e conversão dos dados provenientes de diversas fontes, sejam elas transdutores comerciais ou simuladores, pré-processamento desses dados, algoritmos de reconstrução de imagens e pós-processamento das imagens.

Todas essas funcionalidades foram desenvolvidos na linguagem de programação *Python*. Essa linguagem é bastante usada em aplicações científicas (MAROWKA, 2018), e recomendada em aplicações que precisam de desenvolvimento rápido, implantação de produção (ou quando uma aplicação fica disponível para o usuário final) e sistemas escaláveis (GORELICK; OZSVALD, 2014). Mas, em aplicações que demandam grande quantidade de operações com-

putacionais, que é o caso dos algoritmos do projeto, tem um desempenho inferior quando comparado com linguagens compiladas como C, C++ e *Fortran* (LANARO, 2017). Felizmente existem diversas ferramentas disponíveis na linguagem *Python* para melhorar o desempenho do processamento, dentre elas, o *Cython* e o *Numba*.

Este trabalho tem como proposta utilizar o *Numba* e o *Cython* (que são ferramentas de otimização da execução de códigos desenvolvidos na liguagem *Python*) para a melhoria do desempenho de execução de algoritmos de reconstrução de imagem baseados no método atraso-e-soma (ou DAS, do inglês *delay-and-sum*) (BESSON *et al.*, 2016): SAFT, TFM e CPWC. Existem algoritmos baseados em outros métodos, sendo desenvolvidos dentro do projeto, porém os que usam o DAS, como esses três algoritmos indicados, são os que estão implementados na ferramenta do projeto AUSPEX.

1.1 Objetivos

O objetivo desse trabalho é implementar melhorias no desempenho do tempo de processamento em algoritmos de reconstrução de imagens baseados no método *delay-and-sum* utilizando as ferramentas de compilação *Cython* e *Numba*. Esse objetivo geral está subdividido nos seguintes objetivos específicos:

- Estudar o funcionamento das ferramentas computacionais *Cython* e *Numba*, com foco na adaptação de códigos previamente implementados sem esses recursos.
- Analisar os códigos dos algoritmos de reconstrução de imagens, verificando a viabilidade da aplicação das ferramentas *Cython* e *Numba* para a melhoria do desempenho.
- Projetar e implementar as adequações nos códigos dos algoritmos de reconstrução de imagens existentes no projeto *AUSPEX*.
- Realizar ensaios de reconstrução de imagens utilizando os algoritmos implementados nesse trabalho e medir os tempos de execução.
- Analisar e discutir os resultados obtidos, publicando-os na forma de Trabalho de Conclusão de Curso.

1.2 Organização do Trabalho

Existe um contexto para o qual o trabalho foi aplicado, e isso se constrói até o objetivo final na aplicação das ferramentas de otimização. Esse trabalho foi organizado com o intuito de explicar esse contexto e fundamentar os elementos necessários para o seu cumprimento. Seguindo essa proposta, o trabalho foi dividido como segue:

- **Capítulo 2** Apresenta a base do conhecimento necessário para executar o que foi proposto nesse trabalho, que é melhorar o desempenho de algoritmos de reconstrução de imagens de ENDs. Contém algumas das principais referências de cada tema abordado e situa o problema central.
- **Capítulo 3** É abordado o problema central desse trabalho e a forma com que pode ser solucionado. Será abordada uma maneira de verificar isso e em seguida atendê-lo.
- **Capítulo 4** São organizados os resultados obtidos em forma de tabelas comparativas e feita uma discussão do que isso significa.
- **Capítulo 5** Revisa todo o trabalho com a finalidade de discutir os impactos encontrados nos resultados. E também sugere possíveis continuações a partir do que foi feito aqui.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo é apresentado o que foi necessário estudar para o desenvolvimento desse trabalho. São mostrados os conceitos sobre os ensaios não destrutivos (ENDs) usando ultrassom, os algoritmos de reconstrução de imagens e as ferramentas de otimização utilizadas.

2.1 Ensaios não destrutivos por ultrassom

Ensaios não destrutivos são métodos de avaliar a integridade dos materiais, em busca de falhas internas, na sua superfície ou condições metalúrgicas, sem comprometer ou destruir o material analisado, de alguma forma, ou afetar sua capacidade de serviço (KUMAR; MAHTO, 2013). Existem diversas forma de se realizar ENDs. As principais técnicas utilizadas atualmente são (ABENDI, 2014): inspeção visual, radiografia, líquidos penetrantes, correntes parasitas, emissão acústica e ultrassom. Cada técnica apresenta pontos positivos e negativos dependendo da aplicação. Dentre todas elas, o ultrassom é mais utilizado. Isso, se deve ao fato das ondas ultrassônicas se propagarem profundamente no material analisado, e os sinais de retorno conterem informações importantes sobre características relacionadas a falhas ou defeitos nesse material. Além disso, a estrutura para gerar e detectar esse tipo de sinal é financeiramente mais viável (THOMPSON; THOMPSON, 1985).

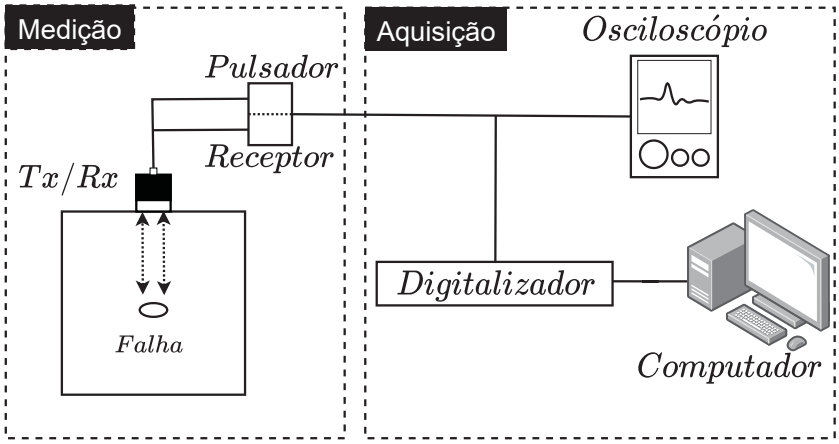
2.1.1 Princípio de funcionamento dos ensaios não destrutivos

O princípio da propagação e reflexão de ondas sonoras é o que baseia as técnicas de inspeção por ultrassom (SCHMERR, 2016; SHULL, 2002; BLITZ; SIMPSON, 1995). Um sistema de inspeção consiste em algumas unidades funcionais, e pode ser dividido em duas partes: medição e aquisição, como mostra a Figura 1.

Na medição ocorre a geração e a transmissão dos sinais de ultrassom, em que um pulsador gera pulsos elétricos de curta duração e com amplitude de centenas de Volts. Esses pulsos excitam um transdutor piezoelétrico, que emite ondas sonoras de alta frequência (ultrassom na ordem de 1 a 25 MHz).

Como forma de explicar o funcionamento de um END, é possível usar a configuração mais simples, o pulso-eco, em que um elemento transdutor envia e recebe os sinais de ultrassom

Figura 1 – Diagrama de blocos de um sistema END por ultrassom.



Fonte: Autoria própria.

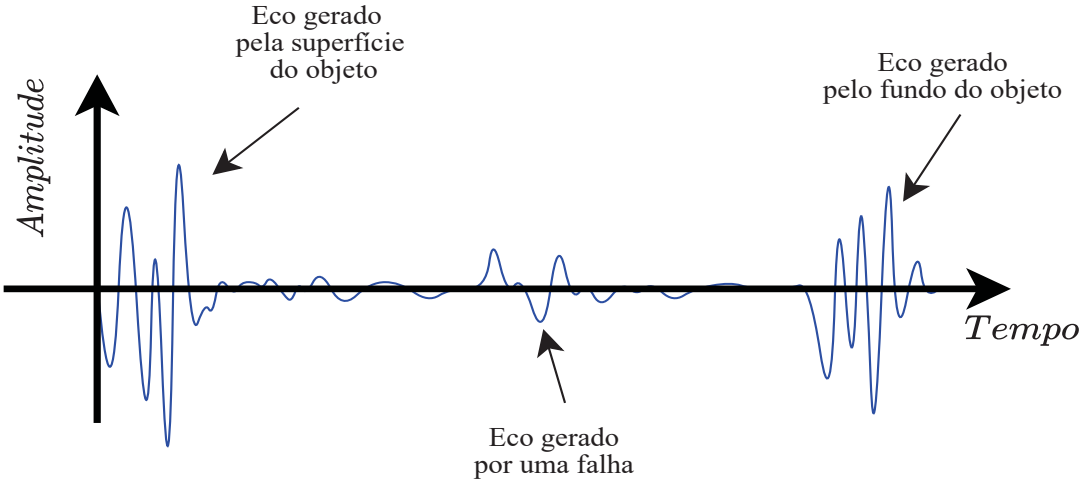
no objeto a ser inspecionado. Nessa configuração, após emitir a frente de onda, o transdutor se torna um receptor, convertendo ondas acústicas em sinais elétricos (SCHMERR, 2016). Quando a onda se propaga pelo material do objeto analisado, essa pode encontrar alguma descontinuidade no material (normalmente um defeito), e parte dessa onda emitida é refletida, gerando um sinal de eco. O transdutor receptor capta esses ecos e produz sinais chamados de *A-scan* (sinal de varredura de amplitude, ou, do original em inglês, *amplitude scanning*). O sistema de aquisição é responsável por digitalizar esses sinais e disponibilizá-los para análise (SCHMERR, 2016).

Em um procedimento de inspeção, os sinais de eco, refletidos pelo material, têm uma amplitude proporcional ao tamanho dos defeitos. Os ecos podem ser devido à alguma falha na estrutura do objeto, ou por conta de encontrarem a superfície oposta desse. Sendo que partir desse último, é possível descobrir a espessura do objeto analisada (ANDREUCCI, 2011). A Figura 2 mostra um sinal de A-scan, a partir dele é possível identificar algumas informações, nesse caso, sobre a superfície de contato, a descontinuidade, e o fundo da peça analisada. Conhecendo a velocidade com a qual a onda acústica se propaga no material do objeto, as posições dos defeitos podem ser estimadas.

2.2 Métodos de ensaio

Existem duas formas de realizar um ensaio de uma peça com a técnica de ultrassom, que está relacionado também com a aplicação final. Na primeira forma, o transdutor é posto em contato direto com o objeto, enquanto que na segunda, esse contato é indireto, havendo algum meio de propagação antes das ondas entrarem em contato com o material do objeto a ser

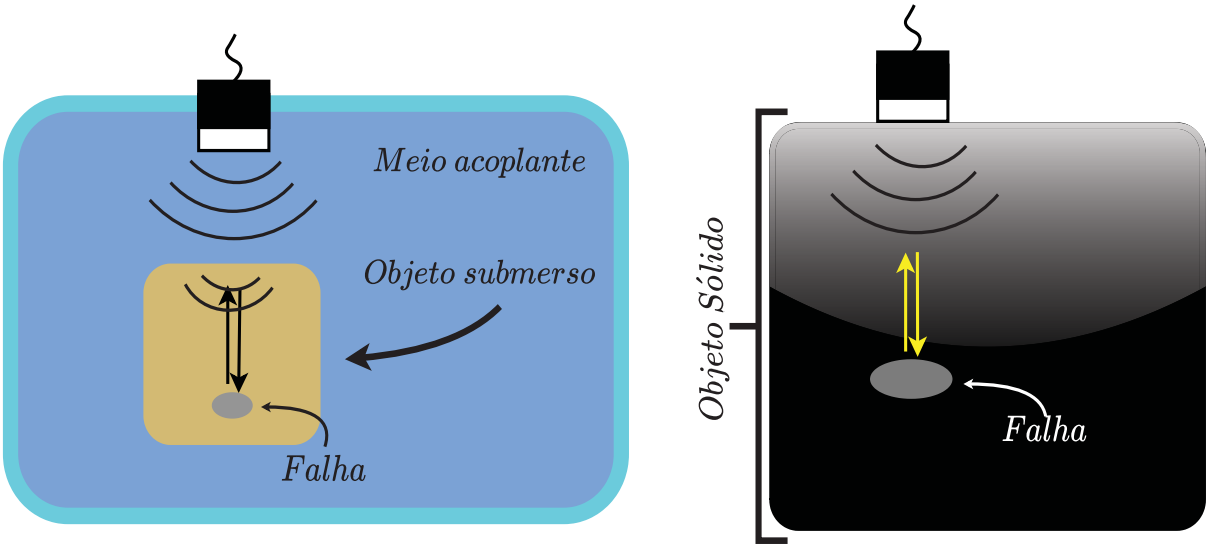
Figura 2 – Exemplo de um sinal A-scan e as informações contidas nele.



Fonte: Autoria própria.

analisado. Essa última forma recebe o nome de *ensaio por imersão*. A Figura 3 mostra o arranjo transdutor/material nos dois casos.

Figura 3 – Exemplos de ensaios sendo realizados por imersão (à esquerda) e por contato (à direita).



Fonte: Autoria própria.

No caso de ensaios por imersão tanto o transdutor quanto a peça estão imersos e separados por alguma substância acoplante, que normalmente é a água. A complexidade dos algoritmos de processamento dos dados desse tipo de ensaio também é maior, sendo que a onda precisa navegar por dois meios diferentes, ocasionando um maior efeito de refração.

Em ensaios por contato, o transdutor é colocado em contato com o objeto a ser inspecionado. Para haver compatibilização acústica e a onda emitida possa penetrar na peça, é utilizada uma camada fina de alguma substância acoplante (ANDREUCCI, 2011). Uma grande vantagem

dos ensaios por contato é a possibilidade de emitir ondas com maior intensidade no objeto devido a compatibilidade acústica produzida pelo acoplante, que conduz a onda do transdutor para a peça com perdas menores no processo. A desvantagem é que para ensaios envolvendo peças grandes pode ser inconveniente a aplicação de um acoplante em grandes áreas.

2.3 Transdutores

Os transdutores são os responsáveis por emitir e receber os sinas de ultrassom. São construídos utilizando materiais piezoelétricos, que são capazes de converter pulsos elétricos em ondas acústicas, e também de converter ondas acústicas em sinais elétricos. Fazem isso por meio do efeito piezoelétrico, em que materiais são capazes de mudar sua estrutura física mediante a aplicação de um campo elétrico e vice-versa (SHUNG; ZIPPURO, 1996).

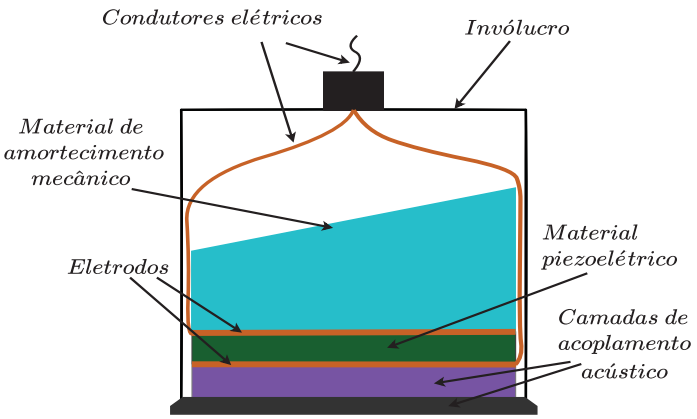
Pode-se classificar os transdutores de acordo com a quantidade de elementos que possuem e de como os elementos estão dispostos (DRINKWATER; WILCOX, 2006). Os tipos principais utilizados nesse trabalho são o transdutor monoelemento e o *array* linear.

2.3.1 Monoelemento

O transdutor monoelemento consiste em apenas um *elemento ativo* de material piezoelétrico, capaz de emitir ondas e receber os sinais de eco. Algumas informações sobre a sua construção podem ser vistas na Figura 4. Como, por exemplo, os eletrodos que conduzem os pulsos elétricos em torno do material piezoelétrico que vibra e geram as ondas ultrassônicas. Um material de amortecimento das vibrações assegura que as ondas sejam conduzidas até a extremidade oposta com os materiais acoplantes, por onde os sinais de ultrassom são enviados e recebidos.

Um ensaio comum para o tipo mono é o pulso-eco. Em que, o mesmo transdutor envia e recebe o sinal de ultrassom uma ou mais vezes na mesma posição. Em transdutores de elemento único o comportamento do feixe de ultrassom depende do tamanho do diâmetro do transdutor e da largura de onda produzido. Que, pode trazer algumas limitações quando comparados com outros tipos de transdutores, como o de *array* linear.

Figura 4 – Detalhes construtivos do transdutor de ultrassom monoelemento.

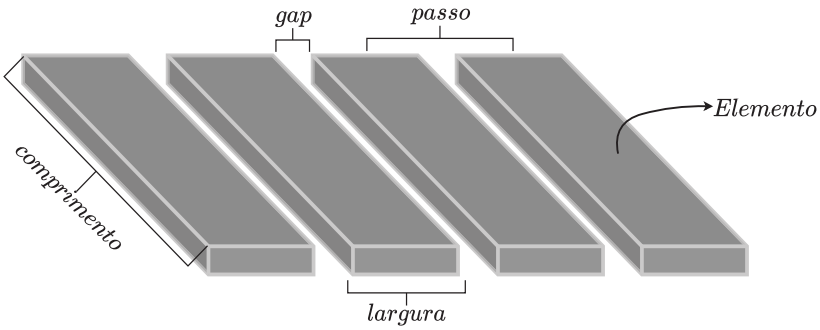


Fonte: Adaptado de (NAKAMURA, 2012).

2.3.2 *Array* linear

Os transdutores de tipo *array* linear são compostos por diversos elementos ativos capazes de emitir e receber ondas sonoras. A quantidade de elementos que transdutores comerciais possuem pode variar de 32 até 256. Um número maior permite uma maior flexibilidade da emissão das ondas, controlando os disparos dos elementos de forma individual durante a inspeção (DRINKWATER; WILCOX, 2006). Exemplo da disposição dos elementos em um transdutor linear na Figura 5.

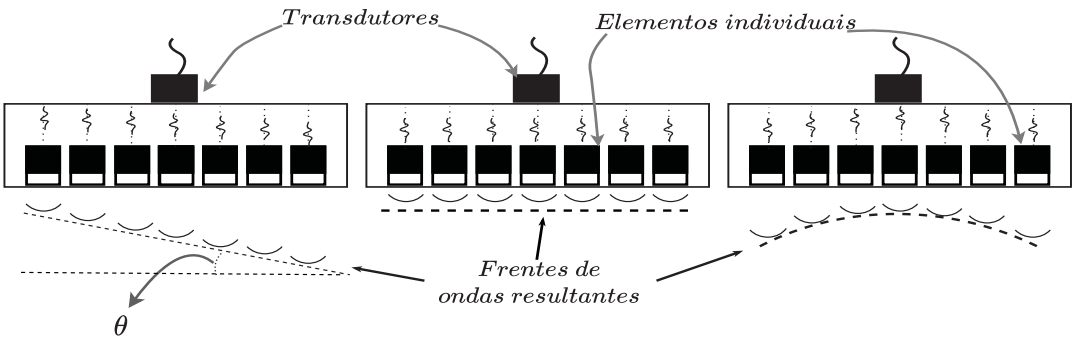
Figura 5 – Disposição dos elementos em um transdutor do tipo *array* linear.



Fonte: Autoria própria.

Tanto o comportamento do feixe, a focalização, a largura e o direcionamento da frente de onda são facilmente manipuláveis usando transdutores do tipo *array* linear (GURURAJA; PANDA, 1998). É possível perceber isso a partir da Figura 6, em que diferentes métodos de disparos, conseguem alterar o perfil da onda resultante. Os elementos desses transdutores são normalmente dispostos lado a lado, que é uma configuração que permite direcionar as frentes de onda em um plano bidimensional.

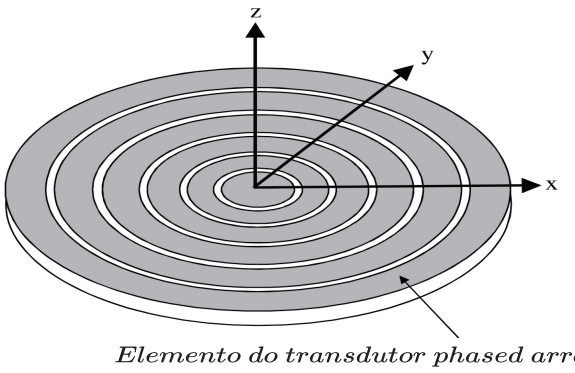
Figura 6 – Exemplos de transdutores do tipo *array* linear realizando diferentes tipos de disparos.



Fonte: Autoria própria.

Para um controle ainda maior do comportamento da frente de onda e, consequentemente, a composição de imagens tridimensionais, outras configurações de transdutores são utilizadas. Como exemplo, tem-se o *phased array* circular (Figura 7), que são vários elementos transdutores circulares dispostos em vários anéis concêntricos; e o *phased array* bidimensional, que são os elementos dispostos em uma matriz.

Figura 7 – Disposição dos elementos no transdutor do tipo *phased array* circular.



Fonte: Adaptado de (DRINKWATER; WILCOX, 2006)

2.4 Métodos de captura

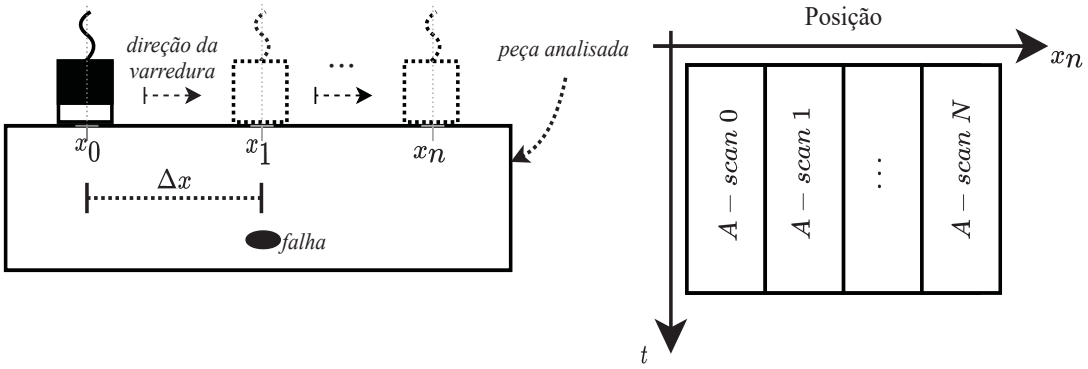
No início do uso dos ENDs por ultrassom, a captura de sinais A-scan era baseada no uso de transdutores de elemento único por varredura. Atualmente, os transdutores mais usados são *arrays* lineares, juntamente com o tipo mais comum de captura realizado com ele que é o FMC (ou captura por matriz completa, do inglês *full matrix capture*). É uma das técnicas mais utilizadas por conta de obter imagens de alta qualidade. Porém, os ensaios usando essa técnica, produzem uma grande quantidade de dados. Como forma de propor uma solução para

esse problema, foi proposto a PWI (ou imageamento por onda plana, do inglês *plane wave imaging*), técnica inicialmente utilizada na área médica, mas que consegue obter uma imagem com qualidade equivalente ao FMC com um número menor de disparos. Essa seção apresenta os métodos de captura por varredura, FMC e PWI.

2.4.1 Varredura

No método de captura por varredura, um transdutor monoelemento é movimentado lateralmente na superfície do objeto, em um passo conhecido, Δx, e constante. Em cada uma dessas posições, que o transdutor percorre, é efetuado um disparo e a captura dos sinais de eco e gerado um sinal A-scan. Ao longo de várias posições pode ser composto uma matriz, conforme a Figura 8, em que cada coluna é uma posição e cada linha é uma amostra do sinal no tempo.

Figura 8 – Ensaio sendo efetuado pela técnica de varredura (à esquerda) e a disposição dos dados adquiridos em forma de matriz (à direita).



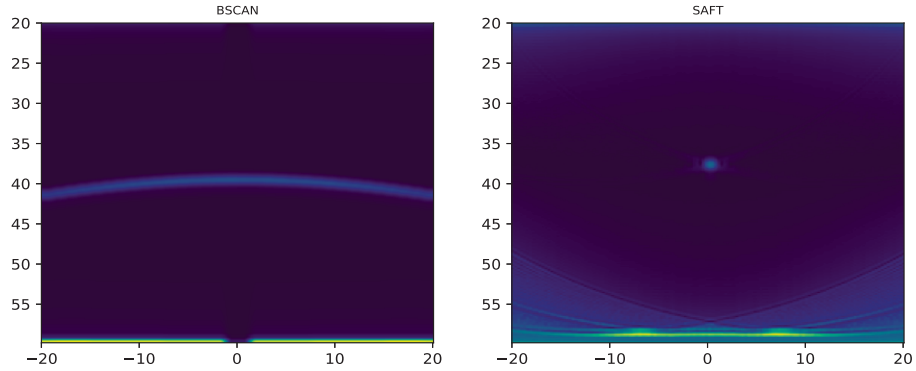
Fonte: Autoria própria.

Com essa matriz, é possível formar uma imagem chamada de B-scan (ou *brightness scan*, e também *brightness mode*). Nessa imagem a intensidade de cada pixel representa o sinal de A-scan em uma determinada posição. Na Figura 9 encontra-se um exemplo de uma imagem B-scan. Essa imagem pode ser filtrada e refinada por técnicas mais aprimoradas como a SAFT (ou *Synthetic Aperture Focusing Technique*) que está descrita na subseção 2.5.1 (LINGVALL *et al.*, 2003).

2.4.2 *Full matrix capture*

A FMC é uma técnica de aquisição de sinais A-scan comum em transdutores do tipo *array* linear. A captura dos dados é baseada em coletar todas as combinações possíveis entre

Figura 9 – Exemplo de conjunto de dados formando uma imagem B-scan (à esquerda) e o mesmo conjunto sendo filtrado pela SAFT (à direita).



Fonte: Autoria própria.

emissão e recepção dos elementos do transdutor (WESTON *et al.*, 2012).

Após o disparo de um elemento, todos os elementos do transdutor recebem os ecos de ultrassom gerando um sinal de A-scan em cada elemento. Cada disparo forma um conjunto 2-D de sinais A-scan, ao passo que, quando todos os transdutores tiverem efetuado seus disparos, será formada uma matriz 3-D.

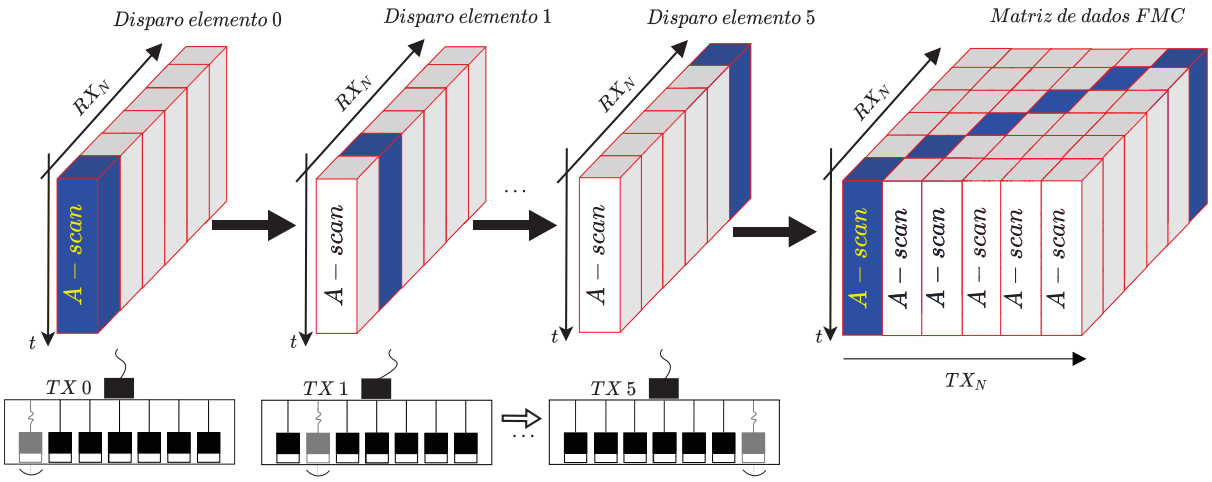
A Figura 10 mostra um exemplo de captura por FMC. Nela, é possível perceber que um dos elementos efetua um disparo, e todos os transdutores recebem os sinais de eco. Isso é feito para todos os elementos do transdutor. No final, forma o conjunto 3-D dos dados descrito. Os A-scans marcados em azul podem ser utilizados para realizar uma varredura eletrônica, sem que o transdutor precise ser movido. Nesse caso, os dados utilizado são apenas do disparo e captura de cada um desses trandutores. Ou seja, é um *pulso-echo* com vários elementos do trandutor de *array* linear.

2.4.3 *Plane wave imaging*

No método de captura por ondas planas, um conjunto de transdutores, como um *array* linear, é disparado de forma a gerar uma frente de onda plana que pode ser transmitida em diversas direções no meio de inspeção, como na figura 11. E para cada transmissão, as ondas de retorno são registradas por todos os elementos ativos que efetuaram o disparo. Os dados capturados são então processados por algoritmos específicos (JEUNE *et al.*, 2015).

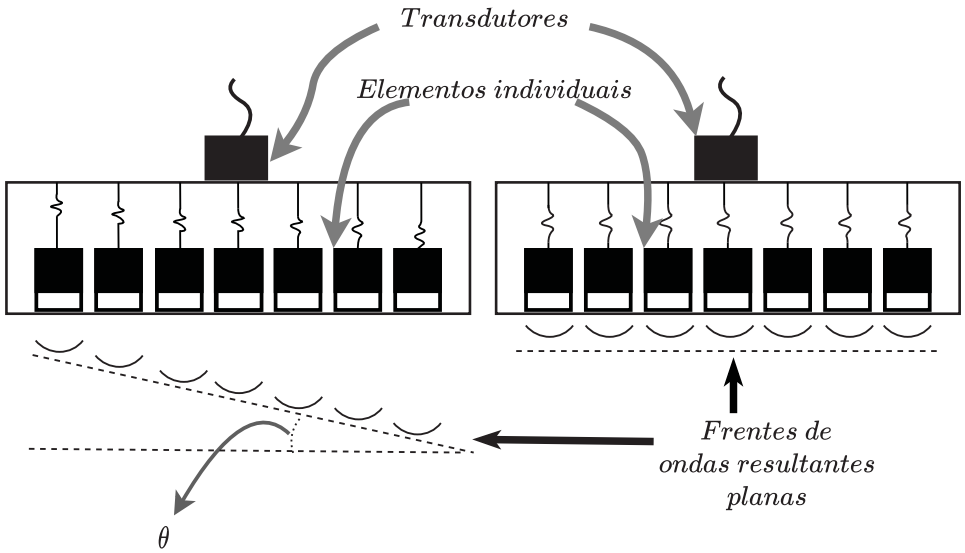
Inicialmente o PWI foi proposto para aplicações médicas por conta da alta qualidade das imagens com poucos disparos, e por conta de todos transdutores serem excitados ao mesmo tempo, causando uma maior energia sonora enviada ao meio. Percebeu-se que isso poderia ser

Figura 10 – Formação de um conjunto de dados FMC usando um transdutor do tipo *array* linear.



Fonte: Autoria Própria.

Figura 11 – Arranjo dos transdutores de *array* lineares para disparos de ondas planas.



Fonte: Autoria própria.

promissor aos ENDs por ultrassom (JEUNE *et al.*, 2015; JEUNE *et al.*, 2016). Frequentemente, o PWI é denominado na literatura como o próprio método de CPWC (composição coerente de ondas planas, ou do inglês*Coherent Plane Wave Compounding*). Mas faz-se essa distinção aqui para diferenciar o método de captura do algoritmo de reconstrução de imagens.

2.5 Algoritmos de reconstrução de imagens

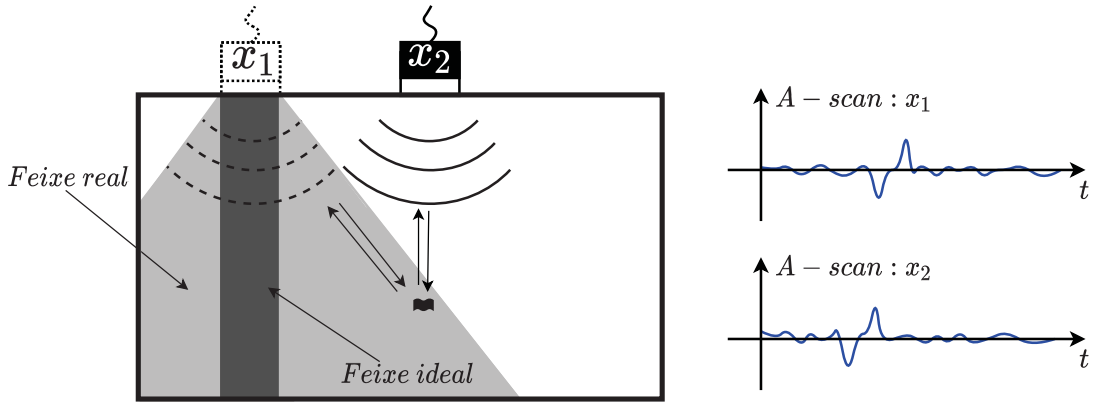
É possível formar uma imagem interna do objeto que está sendo analisado a partir dos dados coletados durante o procedimento de inspeção, por meio da combinação de informações de

vários A-scans obtidos ao movimentar o transdutor sobre o objeto em posições predeterminadas. Em cada uma das posições obtém-se um ou mais sinais A-scan ao realizar disparos e aquisições em cada uma das posições.

Combinar sinais A-scan, em sequência, para cada uma dessas posições, forma uma matriz. Cada coluna dessa matriz representa uma posição do transdutor, e cada linha representa uma amostra digital do A-scan no tempo. Dessa maneira é possível compor uma imagem chamada *B-scan* (varredura de brilho, ou, em inglês, *brightness-scan*). Na imagem B-scan, cada pixel representa a intensidade da *reflexividade acústica* do material inspecionado naquela posição, como mostra a Figura 8. O algoritmo computacional para compor imagens B-scan é mais simples de todos os algoritmos de reconstrução de imagens em END, e pode ser usado como uma avaliação prévia do objeto sob inspeção.

Idealmente, supõe-se que a propagação das ondas ultrassônicas seja longitudinal e alinhada com a posição de emissão do transdutor. Porém, as frentes de onda se dispersam de forma que um transdutor posicionado em posições diferentes pode captar o eco gerado por um mesmo defeito, conforme a Figura 12. Um transdutor posicionado na posição x_1, por exemplo, irá obter um sinal de eco, gerado pelo defeito, atrasado em relação ao da posição x_2. O arco que se vê na Figura 9 do B-scan é consequência dessa peculiaridade.

Figura 12 – Defeitos próximos ao transdutor em uma posição geram sinais de eco (esquerda) que aparecem em sinais de A-scan (direita), defasados em relação a outra posição.



Fonte: Autoria própria.

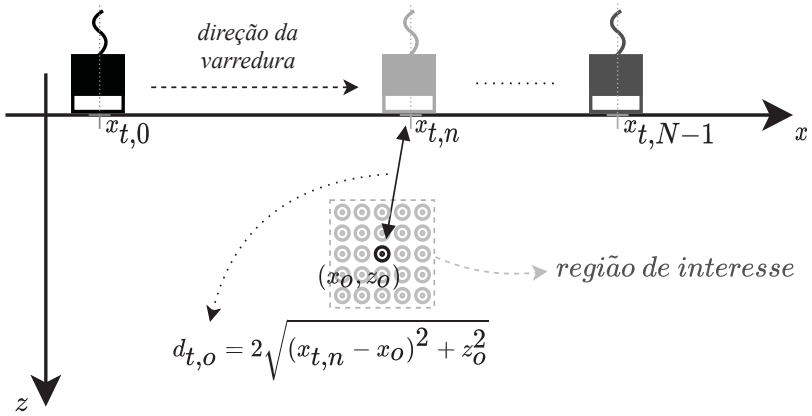
Somente a análise de sinais A-scan e imagens B-scan é insuficiente para permitir a avaliação dos objetos inspecionados. É necessário outras técnicas de reconstrução dessas imagens a partir desses sinais. A seguir são apresentados os algoritmos de reconstrução de imagens mais utilizados na área dos ENDs por ultrassom baseados no princípio do atraso-e-soma (ou *DAS, delay-and-sum*) de sinais A-scan no domínio do tempo.

2.5.1 SAFT

A técnica de focalização por abertura sintética (ou SAFT, do inglês *Synthetic Aperture Focusing Technique*) foi desenvolvida com intuito de melhorar a resolução lateral das imagens reconstruídas. O algoritmo SAFT, assim como o B-scan, foi desenvolvido inicialmente para atender aos sistemas de inspeção com transdutores exclusivamente monoestáticos (que contém apenas um elemento piezoelétrico ativo), usando o método de captura pulso-eco por varredura[1]. O algoritmo SAFT é baseado no conceito de DAS e forma a imagem combinando sinais de eco recebidos pelo transdutor em posições diferentes de varredura (THOMSON, 1984; FRAZIER; O'BRIEN, 1998; LINGVALL *et al.*, 2003).

Se o caminho de varredura e a geometria da superfície são conhecidos, é possível prever com precisão a forma do lugar dos ecos para cada ponto dentro do objeto de teste. Assumindo que a varredura seja linear, o algoritmo SAFT, realiza a focalização sintética por meio de somatórios coerentes ao longo de distâncias do transdutor como se vê na Figura 13. Essas distâncias simplesmente expressam o tempo da onda sonora se deslocar até a posição alvo no objeto (LINGVALL *et al.*, 2003).

Figura 13 – Distância entre o transdutor e um ponto no objeto analisado usando a técnica de varredura.



Fonte: Autoria própria.

Por conta da dispersão, que ocorre no feixe de propagação das ondas de ultrassom, como na Figura 12, é comum que para posições próximas de varredura onde haja um defeito, seja identificado o mesmo sinal de eco no A-scan, porém, defasado no tempo. No processo de varredura, os sinais detectados pelo transdutor em uma posição, podem ser confirmados

[1] Atualmente, sistemas de inspeção com transdutores do tipo *array* linear, capturando sinais A-scan por FMC, também permitem a execução dos algoritmos SAFT e B-scan. Basta utilizar somente os sinais A-scan capturados pelos elementos emissores, e descartar os sinais A-scan recebidos pelos outros elementos.

por sinais detectados quando esse está situado em outra posição. Isso é feito pelo processo da soma coerente, usado no algoritmo SAFT, que consiste em deslocar os sinais de A-scan pelo tempo de atraso estimado entre a saída do transdutor até o ponto espacial da *região de interesse* (ROI) [2] e somar os dados das amplitudes dos ecos nesse ponto. É feita, então, uma média com o valor dos A-scans em um número adequado de pontos, com base na abertura do transdutor, no tamanho da imagem, e da região do objeto que foi analisada no ensaio. Se o ponto imageado não corresponder a um local de origem dos ecos, então a interferência destrutiva dos A-scans nesse ponto fará com que a média espacial nesse ponto seja próxima a zero (DOCTOR *et al.*, 1986).

É possível perceber isso no exemplo da Figura 12, que mostra como funciona o princípio do DAS. As figuras geométricas coloridas interligadas indicam qual a parcela do sinal A-scan em cada umas das n posições x_n, que correspondem ao sinal reconstruído nesta posição. Por exemplo, o transdutor na posição x_0 captou um sinal A-scan correspondente a um defeito no objeto. Isso poderia indicar que o defeito está na posição X_0, porém os sinais A-scan das outras posições não confirmam isso. Ao deslocar o transdutor à posição x_1, x_2, e assim por diante, o pico no sinal, indicando o defeito, defasou na direção oposta em relação a contribuição desta posição. Indicando que o defeito não está na posição x_0. Do mesmo modo, a reconstrução da coluna x_2 apresenta um defeito confirmado pelas outras medições.

O sinal A-scan é formado capturando os sinais de eco em uma frequência de amostragem adequada. Sabendo que a onda percorre uma distância $d_{t,o} = 2\sqrt{(x_{t,n} - x_o)^2) + z_o^2}$ e a velocidade c de propagação das ondas de ultrassom no material do objeto, é possível saber o tempo $\tau_{n,o}$ de deslocamento da onda até algum ponto da região de interesse e o retorno do eco por:

$$\tau_{n,o}(x_{t,n}, x_o, z_o) = \frac{2\sqrt{(x_{t,n} - x_o)^2 + z_o^2}}{c}. \tag{1}$$

Sabendo esse tempo, é possível somar a parcela correspondente a esse tempo nos sinais de A-scan, somar as contribuições em cada posição x_n, como na figura 12. Sendo $s(x_{t,n}, t)$ o sinal de A-scan do transdutor na posição $x_{t,n}$, a amostra $s(x_{t,n}, \tau_{n,o})$ contém informações sobre a posição (x_o, z_o) da região de interesse do objeto. A imagem nesse ponto (x_o, z_o) é formada somando as contribuições do sinal A-scan de cada umas das N posições de varredura:

[2] Região especificada e delimitada dentro do objeto, como na Figura 13. Cada ponto hipotético nessa região tem um sinal de A-scan atrelado. Esse sinal será calculado por meio de algum algoritmo e formará um pixel na imagem reconstruída.

$$o(x_o, z_o) = \sum_{n}^{N-1} s(x_n, \tau_{n,o}). \quad (2)$$

As Equações 1 e 2 norteiam o princípio básico de funcionamento do DAS e do algoritmo SAFT, mostrado no Algoritmo 1. Para executar o algoritmo SAFT é necessário informar os vetores x e z (de tamanhos N_x e N_z, respectivamente) que compõe a região de interesse do objeto para formar a imagem, o vetor x_t (de tamanho N_{x_t}) com as posições percorridas pelo transdutor, a matriz s (de tamanho (N_{x_t}, N_s)) que contém cada os dados de A-scan em cada posição percorrida pelo transdutor, escalar c com a velocidade do som no material do objeto e o escalar t_s, que fornece o período de amostragem do transdutor. A saída é uma matriz o (de tamanho (N_x, N_z)) e contém a imagem reconstruída da região do objeto analisado.

Algoritmo 1 – Pseudocódigo do algoritmo SAFT.

inserir x (vetor de tamanho N_x), z (vetor de tamanho N_z), x_t (vetor de tamanho N_{x_t}),
s (matriz de tamanho (N_{x_t}, N_s)), c (escalar), t_s (escalar), o (matriz de tamanho (N_x, N_z))
para cada $i = 0,1, ..., N_x - 1$ **faça**
  para cada $j = 0,1, ..., N_z - 1$ **faça**
    para cada $n = 0,1, ..., N_{x_t} - 1$ **faça**
      $\tau \leftarrow 2\sqrt{(x_t[n] - x[i])^2 + z[j]^2}/c$
      $\tau_{amostra} \leftarrow arredonda(\tau/t_s)$
      $o[i, j] \leftarrow o[i, j] + s[x_t[n], \tau_{amostra}]$
    finaliza para
  finaliza para
finaliza para
retorna o

Fonte: Autoria própria.

2.5.2 TFM

O TFM (método de foco total, do inglês *Total Focusing Method*) é usado para processar dados de ensaios de inspeção usando a técnica FMC. O algoritmo tem o funcionamento muito similar ao do SAFT para a formação da imagem, também baseando-se no método de DAS.

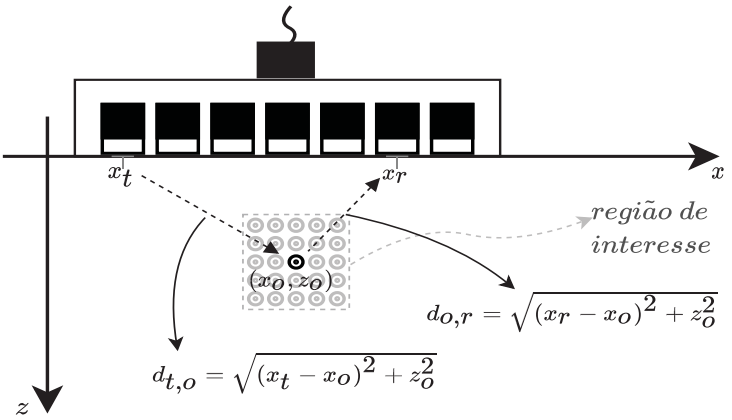
Em um primeiro momento, como na figura 14, um elemento do transdutor, na posição x_t, emite um pulso ultrassônico que se propaga pela peça por uma distância $d_{t,o} = \sqrt{(x_t - x_o)^2 + z_o^2}$ e atinge um ponto (x_o, z_o). Se houver uma descontinuidade, a onda é refletida e percorre uma distância $d_{r,o} = \sqrt{(x_r - x_o)^2 + z_o^2}$ atingindo um outro elemento transdutor, em uma posição x_r. Conhecendo a velocidade de propagação no material, é possível determinar tempo τ de deslocamento do eco:

$$\tau(x_t,x_r,x,z) = \frac{\sqrt{(x_t-x)^2+z^2}+\sqrt{(x_r-x)^2+z^2}}{c}. \tag{3}$$

Considerando um sinal A-scan, associado à emissão por um elemento T e o recebimento por um elemento R de um transdutor, representado por s($x_{T,R}, t$) a amostra $s(x_{T,R}, \tau(x_t, x_r, x, z))$ carrega informações do ponto (x, z) do objeto. Esse procedimento é feito para todas as combinações possíveis de transdutores, e as contribuições, de cada uma dessas combinações, são somadas para cada ponto (x, z) da região de interesse do objeto, dessa forma é possível obter a imagem do objeto por meio da seguinte equação:

$$o(x,z) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} s(x_{(i,j)}, \tau(x_{t_i}, x_{r_j}, x, z)), \tag{4}$$

Figura 14 – Distância entre um elemento transmissor, um ponto do objeto e um receptor de um transdutor do tipo *array* linear, no método FMC.



Fonte: Autoria própria.

em que N é a quantidade de elementos do transdutor usado no ensaio. O Algoritmo 2 mostra o pseudo-código para o algoritmo do TFM. É possível perceber a semelhança com o algoritmo da SAFT, diferindo na variável de saída, o, que é uma soma das contribuições dos sinais de A-scan em todos os transdutores, pois a matriz s é um conjunto de dados FMC.

Para executar o algoritmo TFM é necessário informar os vetores x e z (de tamanhos N_x e N_z, respectivamente) que compõe a região de interesse do objeto para formar a imagem, o vetores x_t (de tamanho N_{x_t}) com as posições de cada elemento do transdutor de *array* linear, a matriz s (de tamanho (N_{x_t}, N_{x_t}, N_s)) que contém cada os dados de A-scan extraídos por FMC, o escalar c com a velocidade do som no material do objeto e o escalar t_s, que fornece o período de amostragem do transdutor. A saída é uma matriz o (de tamanho (N_x,N_z)) e contém a imagem reconstruída da região do objeto analisado.

Algoritmo 2 – Pseudocódigo do algoritmo TFM.

inserir x (vetor de tamanho N_x), z (vetor de tamanho N_z), x_t (vetor de tamanho N_{x_t}),
s (matriz de tamanho (N_{x_t}, N_{x_t}, N_s)), c (escalar), t_s (escalar), o (matriz de tamanho (N_x, N_z))

para cada $i = 0,1, ..., N_x - 1$ **faça**
  para cada $j = 0,1, ..., N_z - 1$ **faça**
    para cada $n = 0,1, ..., N_{x_t} - 1$ **faça**
      para cada $k = 0,1, ..., N_{x_t} - 1$ **faça**
        $\tau \leftarrow (\sqrt{(x_t[k] - x[i])^2 + z[j]^2} + \sqrt{(x_t[n] - x[i])^2 + z[j]^2})/c$
        $\tau_{amostra} \leftarrow arredonda(\tau/t_s)$
        $o[i, j] \leftarrow o[i, j] + s[x_t[k], x_t[n], \tau_{amostra}]$
      finaliza para
    finaliza para
  finaliza para
finaliza para
retorna o

Fonte: Autoria própria.

O algoritmo consiste em percorrer todos os pontos da região de interesse do objeto, e cada combinação de transdutor transmissor e receptor. Para cada um desses é calculado um tempo de deslocamentos dos sinais, que é arredondado e usado como índice na matriz que contém os A-scans da inspeção. Somar todas as contribuições para todos os pontos forma a imagem completa da região de interesse do objeto analisado.

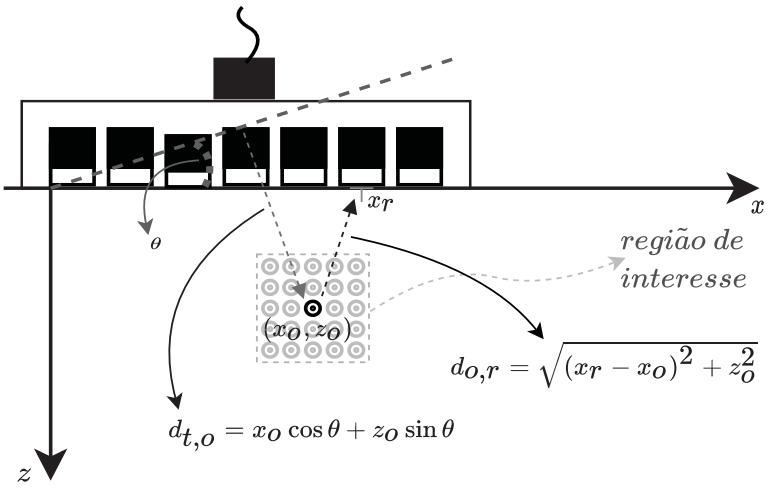
2.5.3 CPWC

Apesar de também ser chamado de PWI na literatura, que é referente à imageamento, é feito essa distinção para diferenciar a técnica de aquisição dos dados, usando ondas planas, para o algoritmo de reconstrução de imagem, pelo método CPWC (composição coerente de ondas planas, ou, do inglês *Coherent Plane-Wave Compounding*).

Também utiliza o método DAS para composição dos pontos da imagem a partir de dados de A-scans. Portanto, se assemelha aos algoritmos SAFT e TFM, com a diferença que os sinais A-scan da CPWC são conjuntos de dados de ensaios usando a técnica PWI, ou seja, em que todos os elementos do transdutor do tipo *array* linear são acionados para formar uma única frente de onda, como na Figura 15. Dependendo da sequência e do tipo de disparo, essa frente de onda pode viajar com um angulo θ de inclinação. Ao encontrar um defeito parte do eco é recebido por um transdutor em x_r. O tempo de deslocamento desde a emissão da onda até o recebimento do eco é dado por (MONTALDO *et al.*, 2009):

$$\tau(\theta, x_r, x_o, z_o) = x_o \cos\theta + z_o \sin\theta + \frac{\sqrt{(x_r - x_o)^2 + z_o^2}}{c}. \quad (5)$$

Figura 15 – Distância entre a onda plana, um ponto do objeto e um receptor de um transdutor do tipo *array* linear, usando o método PWI.



Fonte: Autoria própria.

A imagem reconstruída, da região de interesse, do objeto é formada a partir dos dados adquiridos de cada onda emitida. Usando o método DAS soma-se as contribuições dos sinais para cada ponto referente ao objeto e, com isso, formar a imagem final através da seguinte equação:

$$o(x,z) = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} s(\theta_j, \tau(\theta_j, x_{r_i}, x, z)), \tag{6}$$

em que N é a quantidade de elementos transdutores, M é a quantidade de ondas disparadas em diferentes ângulos θ. E, de forma similar ao SAFT e o TFM, no CPWC os vetores x e z delimitam os pontos de posição da região de interesse no objeto. O pseudocódigo da CPWC pode ser visto no Algoritmo 3 a seguir:

Algoritmo 3 – Pseudocódigo do algoritmo CPWC

inserir x (vetor de tamanho N_x), z (vetor de tamanho N_z), x_r (vetor de tamanho N_{x_r}), θ (vetor de tamanho N_θ), s (matriz de tamanho (N_θ,N_{x_r}, N_s)), c (escalar), t_s (escalar), o (matriz de tamanho (N_x, N_z))
para cada $i = 0,1,...,N_x - 1$ **faça**
  para cada $j = 0,1,...,N_z - 1$ **faça**
    para cada $n = 0,1,...,N_{x_r} - 1$ **faça**
      para cada $k = 0,1,...,N_\theta - 1$ **faça**
        $\tau \leftarrow x\cos\theta[k] + z\sin\theta[k] + \sqrt{(x_r[n] - x[i])^2 + z[j]^2}/c$
        $\tau_{amostra} \leftarrow arredonda(\tau/t_s)$
        $o[i,j] \leftarrow o[i,j] + s[\theta[k], x_r[n], \tau_{amostra}]$
      finaliza para
    finaliza para
  finaliza para
finaliza para
retorna o

Fonte: Autoria própria.

Para executar o algoritmo CPWC é necessário informar os vetores x e z (de tamanhos

N_x e N_z, respectivamente) que compõe a região de interesse do objeto para formar a imagem, o vetor x_r (de tamanho N_{x_r}) com as posições de cada elemento do transdutor de *array* linear, o vetor θ (de tamanho N_θ), a matriz s (de tamanho (N_θ, N_{x_r}, N_s)) que contém cada um dos dados de A-scan extraídos por PWI, o escalar c com a velocidade do som no material do objeto e o escalar t_s, que fornece o período de amostragem do transdutor. A saída é uma matriz o (de tamanho (N_x, N_z)) e contém a imagem reconstruída da região do objeto analisado.

O algoritmo consiste em percorrer todos os pontos da região de interesse do objeto, cada ângulo de disparo, e cada transdutor receptor. Para cada um desses é calculado um tempo de deslocamentos dos sinais, que é arredondado e usado como índice na matriz que contém os A-scans da inspeção. Somar todas as contribuições para todos os pontos forma a imagem completa da região de interesse do objeto analisado.

2.6 Linguagem de programação *Python*

A linguagem de programação *Python* começou a ser idealizada em 1980 por Guido Van Rossum, a partir da necessidade de uma linguagem *script* que fosse extensível, e que funcionasse em outros sistemas operacionais além do UNIX (STEWART, 2002). Ao longo dos anos, a linguagem *Python* permaneceu com essa essência, incluindo a possibilidade de integrar componentes escritos em linguagens de nível mais baixo (principalmente *C* e *C++*), atendendo necessidades diferentes em várias aplicações computacionais. Isso a difere de outras linguagens *script* que se limitam a situações específicas. Todos esses motivos, tornam a linguagem *Python* uma ferramenta poderosa que pode tornar sistemas mais flexíveis, de desenvolvimento mais rápido, e sem necessariamente abrir mão da velocidade de execução onde realmente importa (LUTZ, 2010).

A linguagem *Python*, é, em 2021, a linguagem de programação mais popular pelo índice TIOBE, que mede a popularidade das linguagens de programação (TIOBE, 2021). Também é classificada na primeira colocação pela *IEEE Spectrum* de 2021 (CASS, 2021). Isso pode ser devido, dentre outros vários aspectos, ao fato de ser uma linguagem intuitiva, ter uma sintaxe agradável, e de ser a linguagem introdutória mais escolhida pela academia em todo o mundo. Além de ser ideal para ser usada em áreas predominantemente matemáticas, como ciências em geral e engenharias, devido a uma extensa gama de *bibliotecas* auxiliares desenvolvidas pela comunidade *Python*. Seu uso reside principalmente em áreas que tiveram uma grande expansão com a revolução dos dados como *machine learning*, *scripts* de sistema e aplicações

WEB (LANARO, 2017).

No entanto há um preço a se pagar por toda essa versatilidade. O desempenho na velocidade de programas escritos em *Python* é inferior, quando comparado às linguagens de nível mais baixo compiladas, como o *C*, *C++* e o *Fortran* (MAROWKA, 2018). Isso vem em decorrência de ser uma linguagem interpretada e possuir tipificação dinâmica de atributos usando o próprio recurso de orientação a objetos para encapsular tipos nativos em objetos *Python*.

O *CPython*, que é o interpretador de referência, faz com que as instruções de um programa, que estão em um nível mais alto, sejam desconstruídas e convertidas em código mais próximo ao da máquina por uma série de etapas de interpretação. Na última etapa, esse interpretador cria um *bytecode*, que é um conjunto de instruções para a máquina virtual *Python*. Essa, por sua vez, gera o código de máquina que será executado. Essa sequência de etapas acaba impactando o desempenho de execução.

Esse impacto no desempenho fica ainda mais evidente na hora de executar estruturas iterativas e realizar operações matemáticas (SMITH, 2015). Contudo, há maneiras de contornar essa fragilidade, e combinar a usabilidade e a acessibilidade do código *Python* com a facilidade de processamento de linguagens de nível mais baixo (GORELICK; OZSVALD, 2014). Dispositivos físicos encaram limitações de poder computacional, mas existem diversas técnicas e ferramentas computacionais, no lado do desenvolvedor, que podem ser usadas para o melhoramento da eficiência de processamento dos mais diversos algoritmos.

Outro fator para que enaltece o uso dessa linguagem é a ampla disponibilidade de ferramentas, desenvolvidas pela comunidade, envolvendo linguagens compiladas, capazes de integrar códigos dessas linguagens ao *Python*. É possível, por exemplo, integrar o *Fortran* usando a ferramenta *F2PY*, o *Java* com o *Jython*, escrever interfaces em *C* para integrar código em *C* por meio do mecanismo do *CPython* ou alguma outra ferramenta que automatize isso, como o *SWIG* (que funciona para diversas linguagens de *script*) (LANGTANGEN *et al.*, 2008) e o *Cython* (que é focado apenas no *Python* e será abordado adiante).

Essas possibilidades citadas mostram o potencial de criação, extensibilidade e flexibilidade da linguagem. As bibliotecas, também chamadas de *pacotes*, desenvolvidas pela comunidade, são um conjuntos de *módulos* [3] que podem ser importados pela função primária `import` em qualquer sessão do *Python* e atender quaisquer necessidades que não sejam o foco do desenvolvimento em andamento. Algumas das principais bibliotecas que foram utilizadas

[3] Um módulo é um arquivo contendo definições e instruções *Python*. Pode conter definição de uma ou mais classes, e conter um conjunto de funções agrupados em um único arquivo.

nesse trabalho, são o *NumPy*, o *matplotlib* e o *SciPy*.

O *NumPy* é uma biblioteca usada na computação numérica, muito importante para a comunidade. Também é utilizado na composição de outras bibliotecas importantes que utilizam intercâmbio de dados numéricos, através de uma *interface de aplicação* em *C* que permite criação de extensões para o *Python*. Isso porque o *NumPy* fornece uma integração acessível com módulos escritos na linguagem *C* e *C++*, e que são pré-compilados.

Além disso, *NumPy* providencia estrutura de dados e algoritmos para aplicações no *Python* que envolvam dados numéricos. Além disso, fornece diversas outras utilidades, dentre todas elas, é possível destacar os objetos contendo *arrays* multidimensionais com acesso rápido e eficiente, operações matemáticas (envolvendo álgebra, manuseio de dados, transformadas de Fourier, operações com matrizes, geração de números aleatórios, entre outras) e realizar operações de leitura e escrita (usando vetores e matrizes) direto no disco (HARRIS *et al.*, 2020; MCKINNEY, 2012). Grande parte das funções implementadas nas ferramentas do *framework*[4] desenvolvida para o projeto AUSPEX, inclusive os algoritmos de reconstrução de imagem, se beneficiaram das funcionalidades do *NumPy*.

O *matplotlib* é a biblioteca mais popular para produzir visualizações de dados em duas dimensões. O *Python* possui varias bibliotecas semelhantes, mas a *matplotlib* é a mais utilizada. A biblioteca foi projetada para atender a qualidade de imagem requerida para publicações, desenvolvimento de aplicações e interação entre diversas interfaces e sistemas operacionais (HUNTER, 2007).

O *SciPy* é uma biblioteca contendo uma coleção de pacotes que aborda inúmeros problemas matemáticos padrões. Fornecendo uma série de algoritmos e funções convencionais desenvolvidas a partir do *NumPy*. Dessa forma, aumenta o poder de desenvolvimento do usuário, fornecendo comandos de nível alto e classes para a manipulação e visualização de dados. Podendo, inclusive, tornar o *Python* como uma ferramenta de processamento de dados e prototipação de sistemas. Assim, inúmeras aplicações científicas de nichos diferentes podem se beneficiar e adaptar suas soluções (VIRTANEN *et al.*, 2020).

2.7 *Numba*

O advento da *LLVM* (*Low Level Virtual Machine*) (LATTNER; ADVE, 2004) permitiu uma forma de compilação tanto estática, quanto dinâmica de linguagens *script*, como o *Ruby*, o

[4] Uma explicação mais detalhada do *framework* na seção 2.9

Python, o *PHP*, entre outras. E trouxe uma área de desenvolvimento de compiladores *just-in-time* (`JIT`), ou seja, que compilam o código em tempo de execução.

O *Numba* é uma das principais bibliotecas utilizadas nesse trabalho, e contém um desses compiladores `JIT`. Ele traduz funções escritas em *Python* para um código equivalente, mais próximo do nível da máquina, por meio de mecanismos internos, com rotas padrões de compilação. Primeiramente o *bytecode* gerado pelo interpretador padrão (*CPython*) é desconstruído, o *Numba* cria um código de referência intermediário, faz a leitura de tipos, executa operações de otimização ainda no nível alto, depois executa procedimentos de diminuição do nível do código, até a execução com a *LLVM* (LAM *et al.*, 2015). Faz isso de forma mais rápida que o mecanismo do interpretador padrão do *Python*, o *CPython*, e sem a necessidade de substituí-lo como no caso de outros projetos como o *Pyston*, também baseado na *LLVM*.

O fato de ser um compilador `JIT` propicia uma vantagem em relação a compiladores fora do tempo de execução, não havendo a necessidade de submeter o código por uma etapa separada de compilação usando compiladores externos. Bastando que o usuário utilize um dos *decoradores* na função que se deseja otimizar. Durante a execução a função decorada utiliza uma rota de interpretação otimizada, a depender do decorador utilizado. O *decorador* pode ser definido como uma função que retorna outra função. Tem esse nome pois é aplicado a uma função usando a sintaxe `@`, que na linguagem *Python* envolve a função e realiza alguma operação com ela. *Função* é um trecho de código que envolve uma tarefa ou um conjunto de tarefas e retorna um valor baseado nos argumentos que foram definidos para executar essas tarefas.

Códigos que possuam operações matemáticas, iterações por laços e funções e tipos da biblioteca *NumPy* são os recursos que tem mais potencial de se beneficiar do *Numba*. Esse é um dos grandes motivos de ter sido escolhido para a finalidade do projeto, em que grande parte das operações nos algoritmos de imageamento são funções que realizam operações matemáticas iterativas e fazendo uso da biblioteca *NumPy*.

Para permitir a otimização, em primeiro lugar, é necessário garantir que as operações dentro das funções, a serem otimizadas, estejam sendo executadas em modo nativo. Ou seja, todos os tipos dos argumentos, dessas funções, tenham sido mapeados pelo *Numba* e sejam tipos nativos (`int`, `float`, `char`, tipos nativos do *NumPy*), que implica em não conter nenhum objeto do *Python*. Caso isso ocorra, o *Numba* não executa a função decorada. Se o decorador sendo utilizado for o `@jit`, o programa é interrompido e mostrado a mensagem de erro informando o local da variável em modo objeto. No caso de usar o decorador `@njit`, isso fica ainda mais

crítico, o programa só exibe a mensagem de erro e não executa. No entanto, a otimização gerada pelo `@njit` é melhor, quando os requisitos do código são atendidos.

Se o código for adaptado para conter estas características abordadas, o desempenho de execução fornecido pelo uso do *Numba* pode ser melhorado. Nesse trabalho, além de se atentar a essas características, é proposto a utilização de variações, no uso e no modo de uso delas. Isso acaba entrando num campo mais subjetivo, por conta de requerer a observação do código e de como essas estruturas estão sendo usadas, mas que será abordado mais adiante no capítulo que descreve a metodologia.

O *Numba*, além da otimização da execução do processamento da CPU (unidade de processamento central, ou do inglês *central Processing Unit*), ele também permite a otimização se beneficiando do processamento da GPU (unidade de processamento gráfico, ou *graphics processing unit*). Esse trabalho, no entanto

2.8 *Cython*

O *Cython* (não confundir com *CPython*) é uma extensão para a linguagem *Python*, baseada no *Pyrex*, que é uma linguagem para escrever módulos de extensão em *Python* que tinha o mesmo princípio do *Cython* (EWING, 2010). O *Cython* é ao mesmo tempo uma linguagem de programação e também um compilador estático (BEHNEL *et al.*, 2010). O *Cython* pode ser visto como um incremento, ou superconjunto do *Python*, pois o código em *Python*, em si, já pode ser compilado pelo *Cython* (BEHNEL *et al.*, 2015). Além disso, é uma ferramenta que permite um acesso mais fácil às funções escritas em *C/C++* e a tipos nativos da linguagem *C*.

O compilador do *Cython* é capaz de traduzir o código (que à primeira vista pode ser visto como um código *Python* com anotação de tipos), e codificá-lo em *C* usando a interface de programação de aplicações (API) em *C* do *CPython*. O código do *Cython*, depois de compilado, passa a ser um módulo que pode ser utilizado em qualquer sessão do *CPython* usando a funcionalidade *import* no *Python*. Inclusive muitas bibliotecas do *Python*, importantes para a comunidade, fazem uso do *Cython* para melhorar o desempenho das suas funcionalidades, a exemplo do *skit-learn*, do *sci-py* (GORELICK; OZSVALD, 2020) e até mesmo do *line_profiler* (que é uma das ferramentas usadas nesse trabalho para verificar o tempo gasto para execução de cada linha de código).

A principal arma do *Cython* é abordar as estruturas de iteração (*for*, *while*) e as operações matemáticas. No *Python*, essas estruturas são notoriamente demoradas, pois o tipo de cada

argumento é um invólucro de um tipo nativo do C, e que pode conter qualquer valor (dada algumas restrições), por conta de ser uma linguagem tipificada de forma dinâmica. Na hora de executar uma operação entre duas variáveis, por exemplo `a + b`, poderia estar-se somando, teoricamente, qualquer tipo de dado. Isso impacta na hora da execução.

Diversos mecanismos de segurança e verificação, precisam ser executados, até se ter acesso ao tipo nativo em *C* de um argumento, e ao dado que ele possui em memória (SMITH, 2015). A linguagem *C* e o *Cython* compilam as estruturas de repetição, com a diferença que o *Cython* fornece a flexibilidade de ser um código mais amigável, e de fornecer diversos níveis de adaptação desde um código completamente em *Python* (só que compilado), até uma conversão completa ao *C*.

2.9 *Framework do projeto AUSPEX*

Esse trabalho foi aplicado ao projeto AUSPEX. Grande parte dos ensaios geram dados em grande quantidade dispostos de forma complexa. Muitos, desses dados ainda podem ser coletados a partir de diferentes fontes e técnicas variadas. Atualmente, o projeto conta com ferramentas dispostas em um *framework*, que está organizado como um pacote da linguagem *Python*, em que cada módulo encapsula funcionalidades específicas, visando simplificar o desenvolvimento de novas aplicações e facilitar o uso de dados de diferentes fontes, possibilitando uma rápida visualização de resultados. Dessa forma, um maior foco pode ser direcionado ao desenvolvimento dos algoritmos de processamento de dados e análise de resultados.

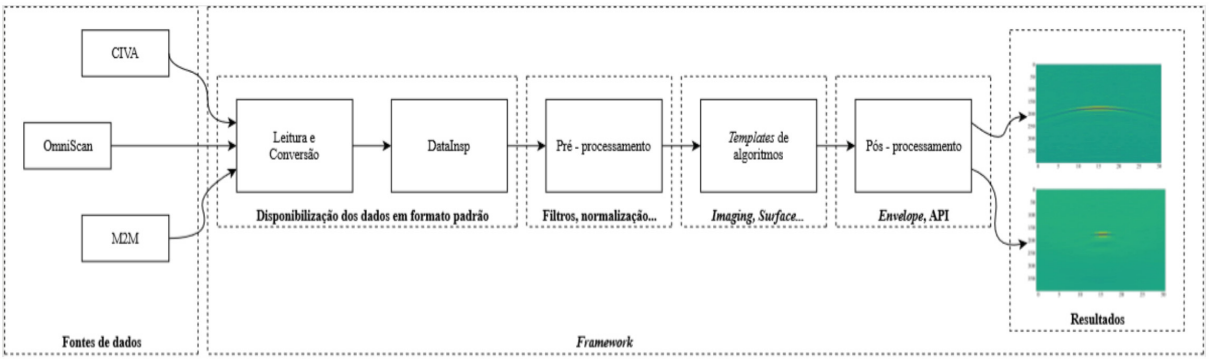
O *framework* segue a estrutura de organização disponível na Figura 16. Uma inspeção pode ser oriunda de diversas fontes de dados. É definido uma estrutura de dados para armazenar os diversos parâmetros relacionados aos instrumentos e as informações do procedimento de inspeção. Os dados coletados, depois de um ensaio, são lidos e convertidos em um formato que apresenta diversas informações na classe [5] `DataInsp`. Dentre elas parâmetros da peça, do transdutor utilizado, do tipo de inspeção, os dados de A-scan, entre outros.

No `Pré-processamento` é feito o processamento de todas as informações necessárias para alimentar os algoritmos de reconstrução. No bloco `Templates de algoritmos`, estão os algoritmos de imageamento, e dentre eles, os de reconstrução de imagens que serão utilizados nesse trabalho. Por fim, o `Pós-processamento`, contém ferramentas como filtros

[5] Classe é um estrutura de programação relacionado ao paradigma de orientação a objetos. Em uma classe são definidos os parâmetros e características de um objeto. O objeto é uma estrutura que carrega informações relacionadas à classe da qual se origina.

de imagem e `interface` com outras aplicações.

Figura 16 – Diagrama da organização da estrutura do *framework* para a integração dos dados de inspeção de fontes diferentes.



Fonte: Autoria própria.

Atualmente, um dos desafios enfrentados, é a necessidade de otimizar o desempenho de execução dos algoritmos de reconstrução de imagens. O processamento dos dados envolve uma grande quantidade de operações matemáticas e de forma iterativa. Os algoritmos estão escritos em *Python*, e sabe-se que seu desempenho é bastante impactado nesses casos citados. Apesar disso, a linguagem atende as necessidades do projeto, de ser extensível, trabalhar com diversas aplicações diferentes, e de tornar mais fácil o desenvolvimento de ferramentas novas.

2.10 Considerações finais

Este capítulo descreve de forma resumida o que são ensaios não destrutivos, sua finalidade e como são feitos utilizando sinais de ultrassom como forma de obter informações sobre a estrutura física de um objeto de análise. Foi abordado o princípio de funcionamento, a estrutura básica requerida para efetuar os ensaios, os principais elementos e métodos de aquisição, os sinais contendo as informações dos ensaios e os algoritmos de reconstrução utilizados no processo para compor a imagem da estrutura interna do objeto inspecionado.

Foi mostrado o funcionamento dos algoritmos baseados na SAFT, no TFM e na CPWC, usando, respectivamente, as técnicas de aquisição por varredura, FMC e PWI. Esses algoritmos de reconstrução fazem parte do setor de imageamento do *framework* do projeto AUSPEX que está todo escrito em *Python*. Nesse capítulo foram descritas as principais características dessa linguagem e dentre elas, a fraqueza relacionada ao desempenho de execução. E como forma de responder a isso, foram propostas as ferramentas *Cython* e *Numba*, que beneficiam principalmente os algoritmos de reconstrução que possuem códigos matemáticos e estruturas

de repetição. Porém para que seja alcançado as melhorias, os códigos dos algoritmos precisam atender à algumas condições. Essas condições têm algumas semelhanças, mas elas são diferentes para cada ferramenta.

3 METODOLOGIA

As ferramentas *Cython* e *Numba* são recomendadas para otimizar trechos mais relevantes nos códigos para os quais são destinadas, ou seja, que demandam maior tempo de execução. A partir de uma análise, com o auxílio de ferramentas de medição, é esperado que se consiga determinar esses trechos relevantes e assim propor uma maneira de implementar as otimizações em cada algoritmo.

3.1 Ferramentas de medição de tempo de execução

É esperado que a demanda computacional, na execução dos algoritmos, aumente em trechos e funções onde são realizadas operações matemáticas e tenha a presença de estruturas iterativas de repetição, que são algumas das fraquezas da linguagem *Python*. Para realizar uma verificação, foram utilizadas ferramentas de medição de tempo, com a intenção de procurar trechos relevantes que tenham potencial de serem otimizados nos códigos. Isso tem o objetivo de direcionar, as mudanças e adaptações sugeridas, de forma mais efetiva no objetivo de alcançar a melhora do desempenho de execução.

3.1.1 Perfiladores

Foram utilizadas ferramentas para construir um *perfil* [1] inicial dos algoritmos. A partir disso, é possível verificar os tempos de execução das funções internas utilizadas e encontrar os principais gargalos. Foram utilizadas as ferramentas *cProfile* (ROSSUM; DRAKE, 2011) e *line_profiler* (GORELICK; OZSVALD, 2014).

A ferramenta *cProfile* é uma extensão presente na biblioteca padrão da linguagem *Python*, tem um baixo custo de execução por estar implementada em C, mostra o tempo total para executar cada uma das funções que são chamadas durante a execução de um programa. A ferramenta *line_profiler* é uma extensão desenvolvida pela comunidade, implementada predominantemente em *Cython*, e mostra o tempo necessário para executar cada linha de um código *Python*.

O tempo de execução, mostrado nas ferramentas de perfil, ainda não são a medição de

[1] *Perfil* é o nome técnico dado a um conjunto de estatísticas que descrevem a frequência e a duração de várias partes de um programa executado.

tempo para a otimização. É apenas uma visão inicial da duração de execução das atividades, com o objetivo de verificar onde podem ser feitas melhorias.

A partir do resultado mostrado pela aplicação das ferramentas de perfil, construiu-se orientação para a aplicação dos métodos propostos. O objetivo foi observar como os algoritmos se transformaram nas suas versões otimizadas, os principais pontos de mudança, qual foi o impacto disso no desempenho e a comparações dos resultados entre as ferramentas.

3.1.2 Temporizadores

Medir o tempo não é uma tarefa simples, e envolve uma diversidade de fatores. Existem diversos temporizadores para medir tempos de execução. A finalidade pode ser a mesma: saber quanto tempo tempo leva para executar uma determinada operação. Mas objetivamente, é necessário saber o que está sendo medido, pois, se tratando de uma medição existem erros atrelados, e a referência de tempo pode ser um fator determinante nessa escolha.

Quando se está trabalhando com temporização, independentemente da plataforma, é necessário verificar o que realmente está sendo medido, para seja feita uma escolha adequada. A resolução de temporizador e a definição de tempo para cada plataforma, por exemplo, pode variar, e isso pode interferir negativamente, trazendo erros indesejáveis. Outro fator, é que o temporizador, precisa ter uma chamada sem muita demanda de processamento, para não entrar em conflito, e nem interferir com o processo que realmente interessa cronometrar.

Para a comparação dos códigos otimizados será utilizado o `perf_counter` da biblioteca `time`. O `perf_counter` é o contador de desempenho com a maior resolução disponível e com a maior precisão para curtas durações, comparado com outros contadores da biblioteca `time` (SIMPSON *et al.*, 2012). Não tem referência particular, mede somente o começo e o final de um intervalo e o que executou entre eles. Nessa temporização também é incluído o tempo total de interrupções e o tempo decorrido durante alguma espera. No entanto é o contador recomendado para a medição de tempo de referência (SIMPSON *et al.*, 2012).

3.2 Otimização com *Numba*

No caso do *Numba*, a maneira mais comum, e que também é usada nesse trabalho, é aplicá-lo em funções que se baseiam no uso de estruturas de repetição do *Python* (ex.: for..in, while, etc.), funções que contenham funcionalidades otimizáveis pelo *Numba* na biblioteca

NumPy, ou ainda uma combinação desses.

Além disso, existem duas variações de otimização principais: a compilação no tempo de execução usando o decorador `@jit` (just in time, ou em tempo de execução), ou o decorador `@njit` (em tempo de execução, mas sem modo objeto). O *Numba* analisa as funções em *Python*, implementa algoritmos inteligentes para identificar os tipos das variáveis, e as compila em uma representação intermediária de nível mais baixo, para uma execução mais rápida (LANARO, 2017).

No *Python* existe a representação *Objeto Python* (ou PO, *Python Object*) em que uma variável não é declarada com um tipo específico, podendo, assim, assumir tipos diferentes do inicial conforme ocorra uma atribuição que declare diferentes tipos de valores nessa variável. Quando se está otimizando, isso pode apresentar obstáculos para a melhoria do desempenho, então foram utilizados ambos os decoradores nos algoritmos, o `@jit` e o `@njit`. O `@jit` serviu apenas com o intuito de verificar se o programa estava executando fora do modo objeto. Se estiver tudo executando com tipos nativos aplicou-se o o decorador `@njit` e tornando isso definitivo. Esse decorador realiza uma rota ainda mais rápida que de otimização por conta de estar em modo nativo e não precisar verificar se existe o risco de executar código em modo objeto, como no caso do uso da rota de compilação do `@jit`.

Também foi usado o artifício de paralelismo, em que o compilador do *Numba* é habilitado para criar várias *threads*, para aproveitar a capacidade do núcleo da CPU do computador, que está sendo usada para executar as operações. Nesse trabalho, foi utilizado esse recurso para possibilitar que várias instâncias de uma estrutura de repetição, em uma função, que esteja aplicada o *Numba*, fossem processadas simultaneamente. Isso tende a melhorar o tempo de execução das funções que possuam iterações, e gerar resultados efetivos desde de que uma instância de uma estrutura de repetição seja descorrelacionada de outra, ou seja, que qualquer instância possa ser executada a qualquer momento, não dependendo da ordem.

Foi utilizado o *Numba* na versão 0.46.0, nos algoritmos baseados no SAFT, no TFM e no CPWC, em que foi analisado o tempo de execução desses algoritmos com a função `time` da biblioteca `time`, e usado um conjunto de dados de inspeção simulados em comum para os dois algoritmos. As técnicas de adaptação dos códigos dos algoritmos usadas nesse trabalho para o uso do *Numba* foram:

1. **Colocar o procedimento de cálculo correspondente à técnica de reconstrução usada em forma de função do *Python***. Esse é o quesito essencial para o uso do *Numba*;

2. **Usar laços de iteração para executar o procedimento respectivo ao procedimento de cálculo usado**. Como na Figura 17. O processo de compilação realizado pelo *Numba* otimiza estruturas iterativas (recomenda-se usar o `for...in` *Python*). Então transformar as operações matemáticas em iterações pode melhorar o desempenho;

Figura 17 – Exemplo de algoritmo de cálculo, à esquerda, colocado em forma de funções do *Python* e utilizando laços de iteração, à direita.

```
j = np.rint(samp_dist - gate_start
          sample_freq).astype(int)
o = g[j, nx]
f = o.sum(2)
```

```
def time_ref_ascan(...):
    for i in range(nb_elements):
        aux[:] = ((samp_dist[i, :])
                  - offset)
        j[i, :] = np.rint(aux)
    return j
def image_form(...):
    for h in range(h_len):
        for w in range(w_len):
            for i in range
                (nb_elements):
                a = j[h, w, i]
                b += g[a, i]
            f[h, w] = b
            b = 0
```

Fonte: Autoria própria.

3. **Usar variáveis com tipos conhecidos pelo *Numba* na função que se espera otimizar**. O *Numba* reconhece os tipos e implementa as rotas otimizadas de referência e compilação. O Quadro 1 mostra os tipos reconhecidos pelo *Numba*, do *Python* e da biblioteca *NumPy*;

4. **Adaptar todas funções do *NumPy* não reconhecidas pelo *Numba*.**[2] Quando se está utilizando funções da biblioteca *NumPy* não reconhecida pelo *Numba*, nos códigos dos algoritmos de reconstrução, elas precisam ser adaptadas. À exemplo do que é mostrado na Figura 18;

[2] As funções do *NumPy* suportadas pelo *Numba* podem ser vistas na documentação (NUMBA, 2020).

Quadro 1 – Tipos reconhecidos pelo *Numba*.

Tipo	Descrição
Inteiro	Todos inteiros de qualquer assinatura e de qualquer largura (até 64 `bits`).
Boolean	Que envolve os atributos `True` e `False` do *Python*.
Real	Precisão simples (`32-bit`) e precisão dupla (`64-bit`).
Complexo	Precisão simples (`2x32-bit`) e precisão dupla (`2x64-bit`).
Datas e timestamps	De qualquer unidade.
Sequência de caracteres	Porém, nenhuma operação é disponível neles.
Escalares estruturados	De qualquer um dos tipos acima e arrays dos tipos acima.
Arrays do NumPy	De qualquer um dos tipos acima independente da forma que estão dispostos.

Fonte: Adaptado de (NUMBA, 2020).

Figura 18 – Exemplo de algoritmo de cálculo, à esquerda, colocado em forma de funções do *Python* e utilizando laços de iteração à direita.

```
f = np.zeros((1, roi.h_len * roi.w_len),
    dtype=g.dtype)
combs = np.argwhere(trcomb.T)
tx_elements = combs[:, 0]
rx_elements = combs[:, 1]
nb_combs = combs.shape[0]
```

```
@njit(parallel=True)
def combinations(combs, size_trcomb):
    for j in range(size_trcomb):
        for i in range(size_trcomb):
            n = j * size_trcomb + i
            combs[n, 0] = j
            combs[n, 1] = i
    return combs

@njit(parallel=True)
def elements_comb (combs, selection, size_trcomb):
    for i in range(size_trcomb):
        elements[i] =
                combs[i, selection]
    return elements
(...)
f = np.zeros((roi.h_len, roi.w_len), dtype=g.dtype)
size_trcomb = trcomb.shape[0]
combs = np.zeros(((size_trcomb*size_trcomb), 2))
combs = combinations(combs, size_trcomb)
tx_elements = elements_comb(combs, 0, size_trcomb)
rx_elements = elements_comb(combs, 1, size_trcomb)
```

Fonte: Autoria própria.

5. **Informar, referir ou calcular um vetor ou matriz, do *NumPy*, de forma particularizada.** O *NumPy* permite que as variáveis recebam fatias [3] de outras matrizes, como no exemplo da Figura 19. Isso dispensa a utilização de estruturas de repetição, que inicialmente pode parecer algo bom. Nesse caso, o *NumPy* implementa as iterações necessárias, para realizar essa referência, dentro da sua programação interna usando sua própria otimização. Inclusive, isso facilita o desenvolvimento do código, no entanto, isso pode acabar impactando o desempenho. Então, para atender esta técnica proposta, cada uma das iterações, das estruturas de repetição, são feitas de forma a percorrer todas as posições de uma matriz ou vetor ponto a ponto. Quando se está realizando operações matemáticas ou de indexação, no *Numba*, espera-se que com isso haja uma melhora do desempenho;

6. **Tornar cada iteração de um laço descorrelacionada de outra, para possibilitar o paralelismo das iterações.** Na Figura 19 o código à esquerda utiliza estruturas de declaração acumuladas (exemplo: "`j[j < 0] = -1`"e "`j[j >= g.shape[0]] = -1`"). Isso cria uma condição de corrida [4] que impede a paralelização. Se uma ou mais instruções, dentro de uma mesma estrutura de repetição, dependerem de um valor gerado em uma outra iteração, dessa mesma estrutura, ou contém operações gerando condições de corrida, é necessário fazer modificações para que varias operações sejam executadas ao mesmo tempo por `threads` separadas e que são unidas depois da conclusão.

Figura 19 – À esquerda o código possui variável recebendo fatias de memória e declaração acumulada. À direita isso sendo feito de forma particularizada e com possibilidade de paralelização.

```
@njit
def kernel(f, g, rx, tx, nb_comb, samp_dist):
    for comb in range(nb_comb):
        j = samp_dist[rx[comb], :] +
            samp_dist[tx[comb], :]
        j[j < 0] = -1
        j[j >= g.shape[0]] = -1
        f += g[j, tx[comb], rx[comb]]
    return f
```

```
@njit(parallel=True)
def kernel(f, g, rx, tx, nb_comb, samp_dist):
    j = np.zeros(samp_dist.shape[1])
    for comb in range(nb_comb):
        for i in range(samp_dist.shape[1]):
            j[i] = samp_dist[rx[comb],i] +
                    samp_dist[tx[comb], i]
            if (j[i] < 0) or (j[i]
                        >= g.shape[0]):
                j[i] = -1
            f[0, i] += g[j[i], tx[comb],
                         rx[comb]]
    return f
```

Fonte: Autoria própria.

[3] Em uma matriz de duas dimensões, seria, por exemplo, o recebimento uma linha ou coluna dessa matriz.
[4] Condição de corrida é quando duas ou mais `threads`, em um mesmo processador, tentam acessar um mesmo conjunto de dados ao mesmo tempo e tentam modificá-lo.

Essas adaptações foram definidas estudando os materiais disponíveis na literatura (LAM *et al.*, 2015; LANARO, 2017; MAROWKA, 2018; GORELICK; OZSVALD, 2020) e que explicam condições recomendadas para realizar uma otimização. Diante disso, é de se esperar que quanto mais dessas técnicas fossem implementadas ao código, melhor seria o potencial de otimização. Analisando o procedimento de cálculo usado em cada algoritmo, foram propostas três versões diferentes para cada algoritmo. A intenção é verificar qual tipo de adaptação aproveita melhor o mecanismo de compilação do *Numba*, e discutir o motivo disso ter acontecido na seção dos resultados. Para facilitar as definições, eles foram nomeados pelo método que eles utilizam seguidos do número da versão, sendo que a primeira versão é a referência de velocidade de execução, pois é o algoritmo na forma atual.

3.3 Otimização com *Cython*

A utilização do *Cython* nos algoritmos de reconstrução foi um pouco diferente do *Numba*. Implementou-se apenas uma versão otimizada com o *Cython* para cada um dos algoritmos de reconstrução. Sendo isso, em decorrência de algumas das suas características:

- **Reconhecer apenas os tipos e não as funções da biblioteca *NumPy*.** Assim, quando se está utilizando alguma função do *NumPy*, não há a necessidade de testar sua velocidade, e sim, apenas escrever essa função, do zero, em *Cython*;
- **Mostrar, a partir de anotações no código compilado, em um relatório em *HTML*, o quanto esse código ainda pode se aproximar do *C*.** Quando está sendo feita verificação ou troca de tipos, ou sendo criado qualquer tipo de inferência dinâmica ao *Python*, o *Cython* informa, a partir dessa anotação, que ainda há espaço pra otimizações.

Existem inúmeras possibilidades de adaptação, até que todo o código esteja completamente em *C*, e utilizando tudo que o *C* pode fornecer. Porém, não é essa a intenção desse trabalho. O método utilizado, aqui, usa as principais ferramentas para um código matemático que utiliza o *NumPy* (BEHNEL *et al.*, 2015). Assim sendo, as técnicas utilizadas nas versões otimizadas são mostradas na Figura 20, e as suas definições são pontuadas a seguir:

1. **Tipificar todas as variáveis de forma correta.** Ou seja, os tipos do código *Cython*, da função adaptada do *Python*, precisam corresponder ao do código *Python*. Isso evita que o

Cython faça troca de tipos e utilize de mecanismos de transformação para objeto *Python* que prejudica o desempenho;

2. **Tipificar todas as funções não principais com base no valor que ela retorna, usando a definição `cdef`.** Informar a função usando essa definição e o tipo do valor de retorno, a aproxima do *C* e melhora a otimização;

3. **Usar tipos do *NumPy* disponíveis no *Cython*.** Eles possuem os mesmos atributos dos tipos do NumPy, porém tem a velocidade de acesso e a alocação de memória de tipos nativos *C*;

4. **Utilizar *memoryviews*.** Para iniciar um vetor *NumPy* é necessário informar o seu tipo e a quantidade de posições e dimensões. Isso, no *Cython*, o transforma em um objeto *Python*. É possível reservar um espaço em memória com o mesmo tipo desse vetor, e acessá-lo na velocidade do *C*;

5. **Informar que o acesso aos vetores é de forma contígua do *C*.** Existe duas disposições em memória de um vetor a do *C* e a do *Fortran*. Na hora da compilação é construída essa verificação. Então, informar que os vetores serão acessados na disposição do *C*, dá um acesso mais rápido aos dados durante a execução;

6. **Desabilitar a verificação dos limites dos vetores e da existência de índices negativos**. O *Python* permite usar essas duas funcionalidades, e o *Cython* precisa informar os mecanismos de verificação quando isso é feito no código.

Figura 20 – Código em *Python* (à esquerda). E código utilizando as técnicas de adequação ao *Cython* (à direita).

```python
# Definição das variávem em Python
xr = 1e-3 * roi.w_points #float64
zr = 1e-3 * roi.h_points #float64
m = pwdata.shape[1] #int
n = pwdata.shape[2] #int
m_r = zr.shape[0] #int
n_r = xr.shape[0] #int
xt_max = xt.shape[0] #int
ti_i = np.int64(tgs / ts) #int64
img = np.zeros((m_r * n_r, 1), dtype=np.
               float64) #float64
j = np.zeros((n_r * m_r, n),
              dtype=np.int64) #int64
j0 = j.shape[0] #int
j1 = j.shape[1] #int
data = np.vstack((pwdata[k], np.zeros((1, n))
       )).astype(np.float64) #float64
dv = np.zeros(xtmax,
       dtype=np.float64) #float64
size_data = data.shape[0] #int
d = np.float64(0.0) #float64
aux = np.zeros(j0, dtype=np.float64) #float64
# Chamada da função em Cython
img[:, 0] += cpwc_func(xr, zr, xt, thetak, c,
       ts, m_r, n_r, ti_i, xt_max, m, aux,
       data, dv, d, size_data, j, j0, j1)
```

```python
# Técnica 6
@cython.wraparound(False)
@cython.boundscheck(False)
# Definição inicial das variáveis da função
# Técnicas utilizadas: 1, 3, 4, 5
def cpwc_func(double[::1] xr, double[::1] zr,
    double[::1] xt, double thetak, double c,
    double ts, int m_r, int n_r, int ti_i,
    int xt_max, int m,
    np.ndarray[np.float64_t, ndim = 1] img,
    np.ndarray[np.float64_t, ndim = 2] data,
    double[np.float64_t, ndim = 1] dv,
    double d, int size_data,
    np.ndarray[np.int64_t, ndim = 2] j_,
    int j0, int j1)
  # Definição das variáveis Cython
  # Técnicas utilizadas: 1, 3, 4, 5
  cdef:
    long long [:, ::1] j = j_
    double[::1] im = im_
    double[:, ::1] data = data_
    np.ndarray[np.float64_t, ndim = 1] img
    double di
    double cos_t = cos(thetak)
    double sin_t = sin(thetak)
    double c_ts = c * ts
    Py_ssize_t i, jj, i_i, x
    int aux, aux2
  for i in range(n_r):
      for jj in range(m_r):
          i_i = i * m_r + jj
          (...)
  img = cpwc_sum(data_, im_, j_, j0, j1)
  return img
# Técnica 6
@cython.wraparound(False)
@cython.boundscheck(False)
# Definição inicial das variáveis da função
# secundária
# Técnica 1, 2, 3
cdef double[::1] cpwc_sum(np.ndarray
        [np.float64_t, ndim = 2] data_,
        np.ndarray[np.float64_t, ndim = 1]
        img_, np.ndarray[np.int64_t,
         ndim = 2] j_, int j0, int j1):
    # Técnicas utilizadas: 1, 3, 4, 5
    cdef:
        long long[:, ::1] j = j_
        double[::1] img = img_
        double[:, ::1] data = data_
        Py_ssize_t iii = j1
        Py_ssize_t jjj = j0
        Py_ssize_t jj, ii
        long long idx
    for jj in range(jjj):
        for ii in range(iii):
            (...)
    return img_
```

Fonte: Autoria própria.

O código em *Cython* depois de compilado pode ser configurado para mostrar uma anotação, e depois da compilação, mostra o quanto o código ainda pode se aproximar do *C*. Isso é uma forma de verificar o quanto o código ainda pode melhorar no seu desempenho de execução. Assim, diferentemente do *Numba*, apenas uma versão foi implementada de cada algoritmo de reconstrução para ser usado com o *Cython*. A ideia foi recriar a função matemática dos algoritmos, dentro de um método em *Cython*, de forma a aproveitar o conjunto de funcionalidades descritos. Isso é uma sugestão para fazer com que o código *Cython* tenha o mínimo de código de verificação junto ao *CPython* na hora da execução, consequentemente melhorando o desempenho de execução dos algoritmos.

3.4 Perfil dos algoritmos não otimizados

Inicialmente foi feito um perfil de execução dos algoritmos originais, que estão nomeados com o número "1". Esses algoritmos estavam implementados no *framework* AUSPEX, previamente ao começo desse trabalho, e que estão baseados nas técnicas SAFT, TFM e CPWC. A partir dos perfis, gerados pelas ferramentas de otimização, são verificados os principais gargalos de desempenho em cada algoritmo, e, a partir disso, são sugeridas as propostas de adaptação seguindo o que foi proposto para cada ferramenta de otimização.

3.4.1 Perfil do algoritmo SAFT

O código SAFT1 não possui uma função centralizada para o algoritmo SAFT. Dessa forma, foi aplicado o *line_profiler* e identificadas as instruções com maior tempo execução. É possível perceber, por meio da Figura 21, uma linha que usa 71,1% do tempo total de execução. Nas otimizações, isso precisa ser levado em consideração, para direcionar o foco na melhoria do que está sendo executado nessa linha.

3.4.2 Perfil do algoritmo TFM

O código TFM1 possui uma função para centralizar o algoritmo da TFM. A Figura 22 mostra a utilização do *line_profiler* no TFM1. É possível perceber que quase todo o tempo de execução é usado em uma única instrução, a função `kernel`. Coincidentemente é a função que centraliza o algoritmo do TFM e será o ponto para trabalhar a otimização.

Figura 21 – Resultado da aplicação do *line_profiler* no código do *SAFT1*.

```
saft1.py:
Timer unit: 1e-07 s
Total time: 0.810862 s
Line #     Time     % Time    Line Contents
==============================================================================
   142                         @profile
   143                         def saft_kernel(data_insp, roi=ImagingROI(), output_key=None)
                                 description="", sel_shot=0, c=None,
  (...)                        # DELAY: Calcula os índices dos sinais "A-scan" relativos a cada
   303  918196.0     11.3              dist = cdist(data_insp.probe_params.elem_center *
                                       1e-3, roi.get_coord() * 1e-3)
   304      45.0      0.0              dist_correction = 2.0 / (c *
                                          data_insp.inspection_params.sample_time * 1e-6)
   305  298585.0      3.7              samp_dist = dist * dist_correction
   306   23180.0      0.3      samp_dist[scatfilt] = g.shape[0]
   308 2883957.5     71.1      j = np.rint(samp_dist - data_insp.inspection_params.gate_start
  (...)                                 * data_insp.inspection_params.sample_freq).astype(int)
   310  114308.0      1.4      j[j < 0] = -1
   311  133128.0      1.6      j[j >= g.shape[0]] = -1
   312      31.0      0.0      nt = j.shape
   313      69.0      0.0      j = j.reshape(nt[0], roi.w_len, roi.h_len)
   314     171.0      0.0      j = np.transpose(j, (-1, 1, 0))
   315                         # SUM: soma das amplitudes dos sinais "A-scan" para cada ponto
   316     101.0      0.0      nx = np.arange(g.shape[-1], dtype=int)
   317  802094.0      9.9      o = g[j, nx]
   318   45918.0      0.6      f = o.sum(2)
  (...)                        # --- FIM DO ALGORITMO SAFT.
Process finished with exit code 0
```

Fonte: Autoria própria.

Além do *line_profiler*, foi executado o *cProfile*, pois apresenta um *overhead* [5] menor, e, por ter uma baixa influência no processo de medição, serve como um verificador. A Figura 23 confirma a demanda do tempo de execução da função `kernel`.

3.4.3 Perfil do algoritmo CPWC

Aplicando o *line_profiler* ao algoritmo original CPWC, é possível perceber dois pontos de trabalho, mostrados na Figura 23. Uma é a função `cpwc_roi_dist`, onde são calculadas as distâncias e os tempos de deslocamento da frente de onda entre cada ponto da região de interesse e os centros dos transdutores. E o outro ponto de trabalho é a função `cpwc_sum`, encarregada de efetuar a parcela da soma dos sinais A-scan nas referências calculadas. Ambos são pontos de melhora do desempenho de execução.

Na Figura 25, observa-se a utilização do *cProfile*. Confirma o que foi observado com o *line_profiler*, `cpwc_sum` e `cpwc_roi_dist` são as funções que apresentam o maior tempo acumulado. Ambas são executadas dentro da função `cpwc_kernel`.

[5] Aqui, *overhead*, ou sobrecarga, se refere a um custo computacional para executar uma atividade. Normalmente, envolve algum tempo que é gasto sem que seja executado a funcionalidade principal.

Figura 22 – Resultado da aplicação do *line_profiler* no código do TFM1.

```
tfm1.py:
Timer unit: 1e-07 s
Total time: 3.08098 s
Line #        Time    % Time       Line Contents
==============================================================================================
    92                              @profile
    93                              def tfm_kernel(data_insp, roi=ImagingROI(),
                                         output_key=None, description="", sel_shot=0, c=None,
                                         trcomb=None, scattering_angle=None):
  (...)                             # DELAY: Calcula a distância entre os pontos da ROI e os
                                         centros dos transdutores. E calcula os índices dos
                                         sinais "A-scan"
   257    943107.0      3.1        dist = cdist(data_insp.probe_params.elem_center * 1e-3,
                                             roi.get_coord() * 1e-3)
   258        47.0      0.0        dist_correction = 1.0 / (c *
                                         data_insp.inspection_params.sample_time * 1e-6)
   259    304979.0      1.0        samp_dist = dist * dist_correction
   261     22786.0      0.1    samp_dist[scatfilt] = g.shape[0]
   262    157168.0      0.5    samp_dist -= int(data_insp.inspection_params.gate_start*
                                             data_insp.inspection_params.sample_freq)//2
   263                           # SUM: soma das amplitudes dos sinais "A-scan" para cada ponto
                                     da ROI.
   264  29378822.0     95.4    f = kernel(f, g, rx_elements, tx_elements, nb_combs,
                                     samp_dist.astype(np.int32))
   265        88.0      0.0    f = (f.reshape((roi.w_len, roi.h_len))).T
   266                           # --- FIM DO ALGORITMO TFM.
Process finished with exit code 0
```

Fonte: Autoria própria.

Figura 23 – Resultado da aplicação do *cProfile* no código do TFM1.

```
tfm1.py
147 function calls (144 primitive calls) in 3.020 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    2.854    2.854    2.854    2.854 tfm1.py:315(kernel)
        1    0.079    0.079    0.079    0.079 {built-in method scipy.spatial._distance_wrap.
                                                 cdist_euclidean_double_wrap}
        1    0.057    0.057    3.020    3.020 tfm1.py:81(tfm_kernel)
        1    0.023    0.023    0.023    0.023 {method 'astype' of 'numpy.ndarray' objects}
        1    0.000    0.000    0.082    0.082 distance.py:2412(cdist)

Process finished with exit code 0
```

Fonte: Autoria própria.

3.5 Procedimentos efetuados para otimização

Mediante às informações relacionadas a cada uma das ferramentas de otimização, e usando como base o perfil inicial dos algoritmos, foi sugerida a criação de três versões novas para cada um dos algoritmos, com o objetivo de experimentar o impacto das técnicas e requisitos sugeridos para o aproveitamento de cada uma das ferramentas. Duas dessas versões propostas, juntamente da versão original (nomeada com número “1”) de cada algoritmo, que não precisa

Figura 24 – Resultado da aplicação do *line_profiler* no código *CPWC1*.

```
cpwc1:
Timer unit: 1e-07 s
Total time: 368.95 s
Line #          Time   % Time Line Contents
==============================================================================================
   126                          @profile
   127                          def cpwc_kernel(data_insp, roi=ImagingROI(), output_key=None,
                                                description="", sel_shot=0, c=None,
                                                      angles=np.arange(-10, 10 + 1, 1)):
  (...)                         # DELAY: Calcula a distância entre os pontos da ROI e os
                                            centros dos transdutores. E calcula os índices dos
                                            sinais "A-scan"
   279     1350948.0      0.0  j = np.empty((n_r * m_r, n), dtype=np.int64)
   280  1246743498.0     33.8  j = cpwc_roi_dist(j, xr, zr, xt, thetak, c, ts, tgs)
   281     3433647.0      0.1  j[j >= m] = -1
   282     3513132.0      0.1  j[j < 0] = -1
  (...)                         # SUM: soma das amplitudes dos sinais "A-scan" para cada ponto
                                  da ROI.
   285       29383.0      0.0  aux = np.empty(j.shape[0], dtype=pwdata.dtype)
   286  2433823828.0     66.0  img[:, 0] += cpwc_sum(data, aux, j)
   287         109.0      0.0  f = img.reshape((m_r, n_r), order='F')
   288                          # --- FIM DO ALGORITMO CPWC. ---
Process finished with exit code 0
```

Fonte: Autoria própria.

Figura 25 – Resultado da aplicação do *cProfile* no código *CPWC1*

```
cpwc1.py:
3966 function calls (3924 primitive calls) in 255.692 seconds

   Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    21  145.563    6.932  145.563    6.932 cpwc1.py:388(cpwc_sum)
    21  109.239    5.202  109.239    5.202 cpwc1.py:337(cpwc_roi_dist)
     1    0.832    0.832  255.692  255.692 cpwc1.py:126(cpwc_kernel)
     1    0.039    0.039    0.052    0.052 pre_proc.py:119(pwd_from_fmc)
   672    0.012    0.000    0.014    0.000 pre_proc.py:160(delay_signal)

Process finished with exit code 0
```

Fonte: Autoria própria.

ser adaptada, são direcionadas ao uso do *Numba*. Totalizando três versões para cada um dos algoritmos. Seus tempos de execução foram testados e depois feita uma comparação entre elas. A terceira versão nova, para cada algoritmo, foi direcionada ao uso do *Cython*. Foi feita uma comparação dos tempos de execução de cada um dos algoritmos implementados com *Cython* e a melhor versão desse algoritmo com *Numba*.

Para obter os resultados todas as versões foram executadas 10 vezes em uma máquina *Intel® Core*™ i5-93xxH, calculada a média e desvio padrão, e escolhido o menor resultado, conforme sugerido pela documentação do *Python* (ROSSUM; DRAKE, 2011). Quando se faz uso de várias amostras, é muito provável que a variação não seja por conta da execução do

Python, e sim por outras tarefas sendo executadas. Assim o menor valor de tempo, portanto, possui o resultado que indica melhor capacidade real da máquina.

3.5.1 Procedimentos para o uso do *Numba*

O código SAFT1 não possui o item 1, dos requisitos para uso do *Numba*. Esse é o requisito primordial, de forma que não é suficiente para o uso do *Numba*, e foi usado apenas como um parâmetro de comparação. Na versão SAFT2, isso foi adaptado, atendendo os requisitos 1 (função de cálculo), 2 (uso de laços para calcular), 3 (tipos conhecidos do *Numba*) e 6 (iterações descorrelacionadas, para efetuar paralelismo), possibilitando o uso do Numba. Na versão SAFT3 foi adicionado a técnicas 5 (trabalhar com vetores de forma particularizada).

O TFM1 possui as condições 1 (função de cálculo), 2 (uso de laços para calcular), 3 (tipos conhecidos do *Numba*) e 6 (iterações descorrelacionadas, para efetuar paralelismo). Depois, foi feito uma nova versão, TFM2 para aplicar a técnica 5 (trabalhar com vetores de forma particularizada). À versão TFM3, adicionada, acrescenta o requisito 4 (que adapta todas as funções não reconhecidas pelo *Numba* que estão sendo usadas dentro de uma função usando *Numba*).

O CPWC1 possui os requisitos 1 (função de cálculo), 2 (uso de laços para calcular), 3 (tipos conhecidos do *Numba*), 4 (que adapta todas as funções não reconhecidas pelo *Numba* que estão sendo usadas dentro de uma função usando *Numba*) e 6 (iterações descorrelacionadas, para efetuar paralelismo). O CPWC2 foi adaptado de forma a acrescentar parcialmente o requisito 5 (trabalhar com vetores de forma particularizada, porém nem todos foram particularizados). O CPWC3 foi acrescentado contendo todos os requisitos, incluindo o 5 (todos os vetores foram particularizados).

3.5.2 Procedimentos para o uso do *Cython*

A versão SAFT do *Cython*, utiliza as otimizações 1 (variáveis com tipos certos), 3 (usa tipos do *NumPy* disponíveis no *Cython*), 4 (utilizar *memory views*), 5 (acesso contíguo aos vetores e matrizes) foi feito parcialmente e 6 (desabilitar algumas verificações desnecessárias para a segurança dos dados.).

A versão TFM do *Cython*, utiliza as otimizações *Cython* 1 (variáveis com tipos certos), 3 (usa tipos do *NumPy* disponíveis no *Cython*), 4 (utilizar *memory views*), 5 (acesso contíguo aos

vetores e matrizes) foi feito parcialmente e 6 (desabilitar algumas verificações desnecessárias para a segurança dos dados.).

A versão CPWC do *Cython*, utiliza as otimizações *Cython* 1 (variáveis com tipos certos), 2 (tipificar funções não principais com a definição `cdef`), 3 (usa tipos do *NumPy* disponíveis no *Cython*), 4 (utilizar *memory views*), 5 (acesso contíguo aos vetores e matrizes) foi feito parcialmente e 6 (desabilitar algumas verificações desnecessárias para a segurança dos dados.). A Tabela 1, traz as implementações que foram descritas aqui de forma mais organizada.

Tabela 1 – As versões dos algoritmos e as implementações atendidas no uso de cada ferramenta.

Algoritmo	Requisitos atendidos do Numba	Otimizações implementadas do Numba	Otimizações do Cython
SAFT1	(nenhum)	- Original	1, 3, 4, parcialmente 5, 6
SAFT2	1, 2, 3, 6	- Convencional - @njit - @njit + parallel	
SAFT3	1, 2, 3, 5, 6	- Convencional - @njit - @njit + parallel	
TFM1	1, 2, 3	- Original - @njit	1, 3, 4, parcialmente 5, 6
TFM2	1, 2, 3, 5, 6	-Convencional - @njit - @njit + parallel	
TFM3	1, 2, 3, 4, 5, 6	- Convencional - @njit - @njit + parallel	
CPWC1	1, 2, 3, 4, 6	- Original - @njit - @njit + parallel	1, 2, 3, 4, 5, 6
CPWC2	1,2,3,4, 5 parcialmente, 6	- Convencional - @njit - @njit + parallel	
CPWC3	1, 2, 3, 4, 5, 6	- Convencional - @njit - @njit + parallel	

Fonte: Autoria própria.

3.6 Considerações finais

Aqui foi descrito o problema central desse trabalho. Acredita-se que exista um potencial de melhorar o desempenho dos algoritmos de reconstrução de imagens, utilizando as ferramentas

Numba e *Cython*. Elas servem, dentro de outras finalidades, no caso do Cython, o propósito de melhorar trechos de código *Python*. Estruturas de repetição e operações matemáticas são as mais beneficiadas por elas. É o caso desses algoritmos.

Foi apontado as ferramentas de medição: os perfiladores e os temporizadores. O primeiro cria um perfil dos códigos e exibe os principais gargalos. O segundo será utilizado nos algoritmos criados para gerar os resultados desse trabalho, por conta da maior precisão que eles fornecem.

A partir da informação dos perfiladores, adquiriu-se informações sobre os principais focos de melhoramento nos códigos. Assim foi proposto a metodologia para utilização das ferramentas *Numba* e *Cython*, e as versões de cada algoritmo aplicando o que foi proposto e para gerar os resultados.

4 RESULTADOS

As versões propostas na metodologia foram implementadas e testadas. Este capítulo apresenta os resultados obtidos através da medição de tempo da execução dessas versões, bem como uma comparação desses resultados.

4.1 Configuração dos ensaios para obtenção dos resultados

Todos os resultados apresentados nesse capítulo correspondem a reconstrução de uma imagem de 800 pixels de altura por 520 pixels de largura (416 mil pixels por imagem). A peça inspecionada é de aço carbono, com dimensões de 80 mm de largura, 60 mm de altura e 25 mm de profundidade, conforme mostrado na Figura 26. A imagem demonstra uma *região de interesse* (ROI – *Region of Interess*) de 40 mm de altura por 40 mm de largura, que contém um defeito do tipo *furo lateral passante* (SDH – *side drill hole*). Esse tipo de defeito é comumente utilizado para a avaliação de algoritmos de reconstrução.

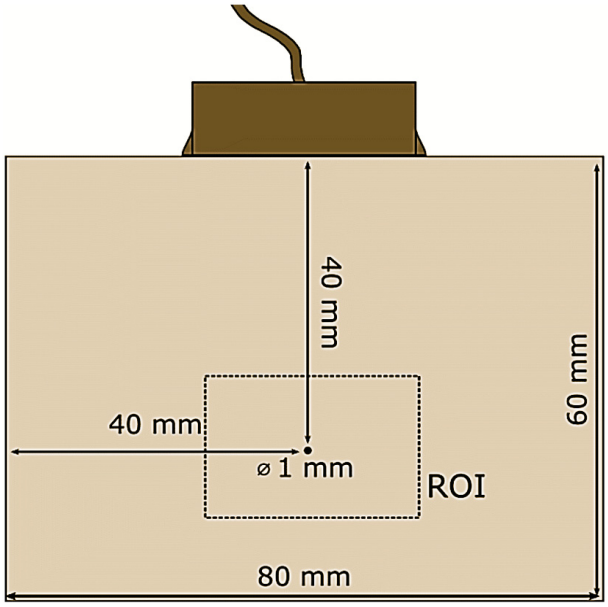
Os ensaios de inspeção por ultrassom foram realizados pela ferramenta de simulação CIVA (CALMON *et al.*, 2006; CALMON, 2012; RAILLON *et al.*, 2012), amplamente reconhecida pela comunidade dos ENDs por ultrassom. A Figura 27 mostra o resultado do processamento do conjunto de dados utilizado por cada um dos algoritmos. Essas são as imagens geradas por cada um dos algoritmos para todas suas respectivas versões. Isso significa que a utilização das técnicas de adaptação e o uso das ferramentas de otimização não interferiu nos resultado final da imagem gerada.

4.2 Resultado das otimizações

Seguindo a metodologia proposta, para cada ferramenta de otimização, obteve-se os resultados exibidos nas Tabelas 2 3 e 4, para o *Numba*, e Quadro 2, para o *Cython*.

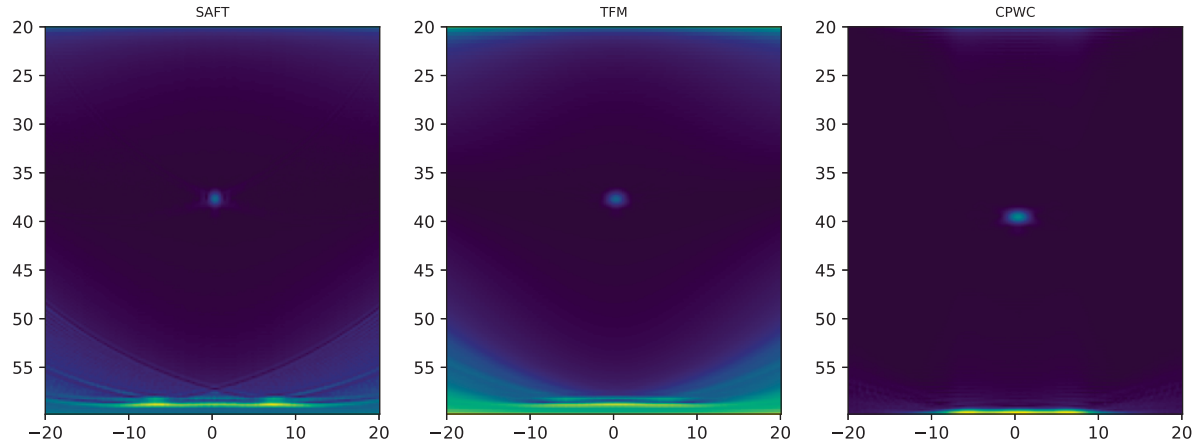
A versão 1 “convencional” é a original, dos algoritmos, que estavam implementadas no *framework* antes do começo desse trabalho. Todas as outras versões “convencionais” são novas versões do mesmo algoritmo com a aplicação das técnicas de adaptação para o uso do *Numba*. As versões convencionais mostram uma piora da velocidade de execução, mostrando que o processo de adaptação parece não ter sentido para o aumento de velocidade. Mas as adaptações

Figura 26 – Resultado da aplicação do *line_profiler* no núcleo do *SAFT1*



Fonte: Autoria própria.

Figura 27 – Exemplo de imagem reconstruída utilizando cada uma das técnicas.



Fonte: Autoria própria.

sugeridas tendem a serem executadas mais rápido em nível mais baixo, através das ferramentas, implicando em uma melhora significativa em relação à versão original.

O SAFT é um dos algoritmos usado para técnicas baseadas em transdutores monoelemento. Os dados usados para testar os programas são usando um transdutor *array linear* na técnica FMC. Mas para não alterar o conjunto de dados, e esse não ser um fator significante, é feito o uso de uma maneira parcial no SAFT, ao invés de analisar a informação de todos os sinais recebidos por cada disparo, aqui nesse modo é configurado da forma *pulso-echo* e apenas o transdutor que fez o disparo é lido, conforme explicado na subseção 2.4.2. O menor tempo de

Tabela 2 – Ganho de desempenho no tempo de execução das versões do algoritmo SAFT, para o uso do *Numba*, comparados com a versão original.

Implementação	Tempo médio (s)	Melhor (s)	Ganho
	SAFT1		
Convencional	0,866111 ± 0,012656	0,850989	**1,000000**
	SAFT2		
Convencional	8,500399 ± 0,205961	8,249054	0,103162
Numba(njit)	0,361709 ± 0,019886	0,340223	2,501268
Numba(njit, paralelo)	0,372743 ± 0,019954	0,347511	2,448811
	SAFT3		
Convencional	43,717187 ± 0,361706	43,000000	0,019790
Numba(njit)	0,218471 ± 0,015489	0,198352	**4,290297**
Numba(njit, paralelo)	0,257690 ± 0,016916	0,237389	3,584786

Fonte: Autoria própria.

execução no algoritmo SAFT é em decorrência disso.

Mesmo assim, como mostra a Tabela 2, percebeu-se a melhora no desempenho à medida que mais técnicas foram adicionadas aos programas. O SAFT3 contém todos os requisitos abordados para a aplicação do Numba e foi a versão com o maior ganho de desempenho. Isso mostra que as técnicas de adaptação fizeram com que houvesse maior aproveito da ferramenta. Era esperado que o resultado usando o paralelismo de `threads` do Numba trouxesse um benefício maior, todavia, isso não ocorreu. Isso pode ser devido ao custo computacional de criação dessas `threads`. E que, nesse caso, pode ter apresentado um impacto significativo comparado com a versão sem paralelismo.

A Tabela 3, mostra os resultados relacionados ao algoritmo TFM. Vê-se que não foi possível executar uma operação do algoritmo TFM1 usando o paralelismo de *threads* do *Numba*. Essa operação no entanto foi resolvida nas outras versões. Aqui a técnica 4 implementada no algoritmo TFM3 mostrou um ganho inferior de desempenho quanto comparada com o algoritmo TFM2. Isso se deve ao fato de que as funções da biblioteca *NumPy* possuem um acesso rápido à máquina, e quando a mesma operação é feita usando o *Python* e otimizadas com o *Numba*, isso pode não ser eficaz. Futuras implementações no *Numba* visam otimizar mais funções da biblioteca *NumPy*.

Aqui no caso do TFM o paralelismo trouxe benefícios e a versão com o maior ganho de desempenho foi o TFM2. Outra observação é que o paralelismo trouxe benefícios inclusive na versão 3, ressaltando que existe um nível de processamento em que o programa passa a se

Tabela 3 – Ganho de desempenho no tempo de execução das versões do algoritmo TFM, para o uso do *Numba*, comparados com a versão original.

Implementação	Tempo médio (s)	Melhor (s)	Ganho
	TFM1		
Convencional	3,186776 ± 0,065177	3,059655	1,000000
Numba(njit)	2,280291 ± 0,038234	2,253256	1,357882
	TFM2		
Convencional	9.556647 ± 0.179168	9,263672	0,330285
Numba(njit)	1,640911 ± 0,020394	1,615989	1,893364
Numba(njit, paralelo)	0,414937 ± 0,015078	0,392474	**7,795816**
	TFM3		
Convencional	59,990532 ± 0,586195	59,145780	0,051731
Numba(njit)	2,043793 ± 0,012244	2,027909	1,508773
Numba(njit, paralelo)	0,656657 ± 0,025823	0,617225	4,957114

Fonte: Autoria própria.

Tabela 4 – Ganho de desempenho no tempo de execução das versões do algoritmo CPWC, com o uso do *Numba*, comparados com a versão original.

Implementação	Tempo médio (s)	Melhor (s)	Ganho
	CPWC1		
Convencional	257,947722 ± 1,436780	253,985600	**1,000000**
Numba(njit)	5,064422 ± 0,021012	5,039447	50,399498
Numba(njit, paralelo)	2,858841 ± 0,384291	2,451953	103,585020
	CPWC2		
Convencional	1392.096065 ± 268.427457	1251.512240	0,202941
Numba(njit)	2,451081 ± 0,060586	2,382751	106,593429
Numba(njit, paralelo)	2,740873 ± 0,344172	2,332317	108,898404
	CPWC3		
Convencional	1847,114297 ± 427,052413	1582,048677	0,139649
Numba(njit)	1,530435 ± 0,007555	1,517882	167,328949
Numba(njit, paralelo)	0,674249 ± 0,011674	0,654667	**387,961513**

Fonte: Autoria própria.

beneficiar do paralelismo, em que o custo-benefício da criação de threads é viável e mostra performance melhor do que o normal.

O CPWC1 é o programa com o maior custo computacional. Isso se deve à técnica de PWI que adquire uma quantidade de dados maior que os outros dois algoritmos. É possível perceber, a partir dos resultados no Quadro 4, que a adesão das técnicas sugeridas na metodologia surtiu um efeito positivo no progresso das versões. Existe um fator importante na versão 3. Foi

observado no perfil do algoritmo CPWC que duas funções eram determinantes no impacto do desemnho. A adaptação no CPWC3 permitiu que fosse feita apenas uma chamada contendo as duas funções. Como as duas foram otimizadas pelo *Numba* isso implicou em um ganho maior de desempenho. Outra observação é que foi o maior ganho de desempenho absoluto, e o tempo de execução na melhor versão se equipara ao menor tempo das outras versões. Em grande parte foi por conta de adequar o requisito 5. Aumentar a quantidade de laços de repetição em *Python* pode parecer contraintuitivo, mas para o *Numba* é essencial e significativo e é o que os resultados mostram.

Na Tabela 2, foi feita uma comparação dos melhores resultados obtidos com o *Numba* em relação a versão *Cython* de cada algortimo. Percebe-se que o ganho proposto pelo *Cython* é inferior quando comparado com seus pares. Apesar de parecer que uma conversão do código *Python* ao *C* possa parecer bastante promissor, para um ganho de desempenho, não podemos deixar de levar em consideração que o próprio *NumPy* oferece mecanismo para indexar conjuntos de dados em nível mais baixo. O *Numba* é uma biblioteca que pretende formar uma integração completa com o *NumPy*, atualmete os tipos do *NumPy* usados nos algoritmos, são otimizados pelo *Numba*. O *Cython* oferece uma maneira de reconhecer à nível de memória alguns dos tipos da biblioteca *NumPy*, pois ambos partem da linguagem *C*, mas mesmo assim, durante uma chamada de uma função em *Cython* estão sendo feitas conversões ao *C* (das entradas) e de volta ao *Python* (no retorno da função). E que são fatores que podem acabar influenciando em um desempenho inferior.

Quadro 2 – Comparação entre a melhor versão implementada com o *Numba*, a versão *Cython* e a versão original de cada algoritmos.

Algoritmo	Implementação	Tempo médio (s)	Melhor (s)	Ganho
SAFT	Original	0,866111 ± 0,012656	0,850989	1,000000
	Versão 3 Numba(njit)	0,218471 ± 0,015489	0,198352	4,290297
	Cython	1,802950 ± 0,130443	1,698738	0,500954
TFM	Original	3,186776 ± 0,065177	3,059655	1,000000
	Versão 2 Numba (njit, paralelo)	0,414937 ± 0,015078	0,392474	7,795816
	Cython	1,753584 ± 0,016811	1,727852	1,770785
CPWC	Original	257,947722 ± 1,436780	253,985600	1,000000
	Versão 3 Numba (njit, paralelo)	0,674249 ± 0,011674	0,654667	387,961513
	Cython	1,893587 ± 0,007497	1,880113	135,090604

Fonte: Autoria própria.

4.3 Considerações finais

Neste capítulo foi apresentado o resultado do que foi proposto na metodologia. Ou seja, foram determinadas 4 versões, com exceção do SAFT1, para cada um dos algoritmos. Sendo que 3 dessas versões são relacionadas à aplicação da ferramenta *Numba* e a outra versão restante é aplica ao *Cython.* A partir dos resultados foi explicado o motivo de ter havido a melhora. Ou explicar porque onde era pra acontecer uma melhora não aconteceu. A partir dos resultados foi feito então uma tabela comparando os melhores resultados com o *Numba* com a versão *Cython*. E percebeu-se que, para esse trabalho, o *Numba* exibiu um desempenho superior.

5 CONCLUSÃO

Atualmente, grande parte do desenvolvimento computacional está sendo feito em *Python*. Este trabalho apesar de estar relacionado à área de ENDs, mostra essa relevância, tendo uma contribuição no sentido de mostrar que é possível aliar as qualidades da linguagem *Python*, relacionadas à facilidade de desenvolvimento e a sua flexibilidade, contornando suas fraquezas. E ainda, aproximando o desempenho de execução do código de linguagens muito mais complexas, de nível menor que, no lugar do uso do *Python*, poderiam ser inviáveis para um projeto com várias frentes de trabalho como esse.

As ferramentas *Numba* e *Cython* têm a proposta de fornecer uma resposta à necessidade de melhorar o desempenho de execução de código originalmente escrito em *Python*. A abordagem proposta para utilizar essas ferramentas e aplicá-las ao paradigma dos algoritmos de reconstrução de imagens de ENDs por ultrassom, se mostrou eficiente para obter uma melhoria no tempo de execução desses algoritmos. Isso, antes de mais nada, mostra o potencial dessas ferramentas para beneficiar o desenvolvimento em *Python*, principalmente nas aplicações que possuem um cunho mais científico, que envolvam procedimentos matemáticos e manipulação de dados numéricos, e onde o *Python*, por si só, ficaria devendo em desempenho por conta das suas características.

O *Cython* é recomendado normalmente em situações genéricas de uso extensivo de funcionalidades que são lentas de executar no *Python*, como operações matemáticas e laços de repetição. Por conta disso ser bastante eficiente de executar no *C*. Porém em códigos que envolvem a utilização extensiva dos atributos fornecidos através do *NumPy*, como é o caso desse trabalho, tendem a ser beneficiados pelo *Numba*, que tem justamente essa intenção específica. É possível perceber isso nos resultados. Era esperado que levar o nível do código ao *C*, através do uso do *Cython* melhoraria muito o desempenho de execução, porém durante da execução isso pode significar um constante acesso e conversão entre tipos nativos e objetos, que pode acabar impactando em uma aplicação como esta. Enquanto o *Numba* é limitado pelo quanto os seus desenvolvedores conseguem otimizar em uma rota de execução, o Cython é limitado pelo quanto o desenvolvedor sabe de *C* é uma ferramenta que reconhece grande parte dos mecanismos da biblioteca

Aplicando as ferramentas de otimização aos algoritmos SAFT, TFM e CPWC, seguindo a metodologia proposta, é possível concluir que as técnicas de adaptação e as sugestões de otimização propostas são um caminho para implementar melhorias de desempenho. Mas elas

não são definitivas e nem garantem que a melhoria sempre irá ocorrer. Mesmo funcionando em casos gerais, ainda é necessária a experiência do usuário tendo essa metodologia em mente. Pois cada código tem uma característica única e, portanto, subjetiva a aplicação estrita de alguma técnica em definitivo.

É importante ressaltar também que algumas vezes é necessário checar se o tempo necessário para adaptar um código para o uso dessas ferramentas é viável para o curso da aplicação final ou o projeto que está inserido. É comum pensar que tudo que mostra a oportunidade de reduzir o tempo de execução de uma aplicação é muito bem-vindo, independentemente da situação. Porém, muitas vezes isso é acompanhado de um gasto de tempo que poderia ser melhor investido em outras aplicações. Todavia, essas aqui, são ferramentas recomendada para melhoria de desempenho quando os fatores necessários para sua aplicação são satisfeitos. Com base nisso, para a situação desse trabalho o *Numba* se provou uma ferramenta favorável tanto na melhoria do desempenho, como no manuseio e na dificuldade relacionada à adaptação dos códigos para sua aplicação. O *Cython* tem uma complexidade maior, por conta de ser bem mais sensível em relação às operações e os tipos usados, e não entrega uma melhoria que torne isso viável de ser usado em uma finalidade parecida com esta do trabalho.

É natural concluir que o esforço de adaptação dos códigos pode ser poupado se desde o desenvolvimento inicial seja considerado um formato adequado para a aplicação do *Numba* em um código. Porém, ele não deve ser usado dessa maneira. E sim para melhorias pontuais de eficiência de processamento que impactam toda a execução. Já era bem sabido nesse projeto que as etapas que mais consumiam tempo e processamento eram os códigos que efetuavam os cálculos pelos métodos propostos, tanto que isso motivou o presente trabalho. Além disso, todos os algoritmos originais atendiam vários requisitos para o uso do *Numba*: código puramente em *Python*, uso da biblioteca *NumPy*, alto consumo de tempo e processamento em funções de cálculos por iteração.

Quanto ao *Cython*, é possível concluir que ele pode ser uma ferramenta recomendada para situações onde não esteja se utilizando a biblioteca *NumPy*, e ainda sim se queira realizar uma otimização significativa. Outra aplicação, seria para desenvolver alguma ferramenta para o *Python* que envolva a utilização das linguagens *C/C++*, ou algum tipo de relacionamento entre o *Python* e outras linguagens, que através do *Cython* conseguem uma integração bem mais acessível do que programar toda a integração ao *CPython* do zero.

Apesar de exigir um esforço de adaptação os códigos dos algoritmos, a metodologia

adotada foi essencial para aproveitar as ferramentas. Com isso já foi possível atingir o objetivo esperado de melhorar o tempo de execução desses algoritmos. Porém, a exemplo do *Numba*, ainda há espaço para utilização de outras atribuições simples do *Numba*, sem que sejam feitas qualquer adaptações no código. Dentre elas, reduzir a precisão dos valores calculados e habilitar que uma memória salve tipos repetidos em novas compilações, porém para esse trabalho a contribuição delas não foi significativa, por isso não foram utilizadas.

No caso do *Cython*, para aplicações semelhantes, o espaço para melhoria seria relacionado à utilização mais técnica da linguagem *C* nos algoritmos, mas que não era a intenção desse trabalho. Outro ponto é que o *Cython* permite o paralelismo, como o *Numba*. Porém é necessário garantir diversas medidas de segurança no código para que isso seja feito, o que no *Numba* é mais acessível. Outros trabalhos poderiam considerar outras funcionalidades não testadas do *Cython*, e sendo uma delas o paralelismo e a liberação do GIL do *Python* para o uso de diversos núcleos na hora de executar uma tarefa, entre outros.

A grande vantagem de se realizar otimizações que utilizam o processamento de uma *CPU*, como nesse trabalho, é que normalmente isso está atrelado a um bom escoamento dos dados processados à nível de barramento em um computador pessoal. Contudo, existe um grande potencial a ser aproveitado do processamento gráfico. Com o advento da computação heterogênea, mais sistemas estão tendo uma melhor integração do sistema com a central de processamento gráfico. O *Numba* tem uma possibilidade de fornecer a melhoria do desempenho usando processamento gráfica de uma GPU. O *Numba* tem uma integração com o CUDA, que é o *firmware* da fabricante Nvidia. E o sistema de outros fabricantes está começando a ser aderido pelos desenvolvedores do *Numba*. Uma sugestão de trabalho futuro é fazer a otimização dos algoritmos de reconstrução de imagens utilizando a otimização por processamento gráfico no *Numba*.

E por fim, existe uma ferramenta de perfil chamada *Scalene* (BERGER, 2020), porém estava em fase de desenvolvimento no período desse trabalho. Ela mostra o perfil, linha a linha do código executado, o uso do processador, uso de acesso à memória, quanto tempo o código foi executado no nível do *Python* e quanto tempo foi gasto em modo nativo, entre outras funções. A maioria das ferramentas realiza apenas uma dessas funções o que a torna uma ferramenta interessante. Ainda mais pelo fato de estar sendo desenvolvida em *C*, que implica em um impacto baixo para sua execução e tornando-a ainda mais confiável. É uma ferramenta promissora para o futuro da otimização em *Python*, e é colocada aqui como uma recomendação para uma

visualização mais apurada dos principais pontos com potencial de otimização em um código *Python* genérico sendo executado.

REFERÊNCIAS

ABENDI. **Ensaios Não Destrutivos e Inspeção**. 2014. Disponível em: http://www.abendi.org.br.

ANDREUCCI, R. **Ensaio por Ultrassom**. São Paulo: Associação Brasileira de Ensaios Não Destrituvos e Inspeção — ABENDI, 2011.

BEHNEL, Stefan; BRADSHAW, Robert; CITRO, Craig; DALCIN, Lisandro; SELJEBOTN, Dag Sverre; SMITH, Kurt. Cython: The best of both worlds. **Computing in Science & Engineering**, IEEE, v. 13, n. 2, p. 31–39, 2010.

BEHNEL, Stefan; BRADSHAW, Robert; SELJEBOTN, Dag Sverre; EWING, Greg; STEIN, William; GELLNER, Gabriel *et al.* Cython documentation. **http://docs. cython. org**, 2015.

BERGER, Emery D. Scalene: Scripting-language aware profiling for python. **arXiv preprint arXiv:2006.03879**, 2020.

BESSON, Adrien; CARRILLO, Rafael E; BERNARD, Olivier; WIAUX, Yves; THIRAN, Jean-Philippe. Compressed delay-and-sum beamforming for ultrafast ultrasound imaging. *In*: IEEE. **2016 IEEE International Conference on Image Processing (ICIP)**. *[S.l.]*, 2016. p. 2509–2513.

BLITZ, Jack; SIMPSON, Geoff. **Ultrasonic methods of non-destructive testing**. *[S.l.]*: Springer Science & Business Media, 1995. v. 2.

CALMON, Pierre. Trends and stakes of ndt simulation. **Journal of Nondestructive evaluation**, Springer, v. 31, n. 4, p. 339–341, 2012.

CALMON, P; MAHAUT, S; CHATILLON, S; RAILLON, R. Civa: An expertise platform for simulation and processing ndt data. **Ultrasonics**, Elsevier, v. 44, p. e975–e979, 2006.

CASS, Stephen. **The Top Programming Languages 2021**. 2021. Disponível em: https://spectrum.ieee.org/top-programming-languages-2021.

DOCTOR, SR; HALL, TE; REID, LD. Saft—the evolution of a signal processing technology for ultrasonic testing. **NDT international**, Elsevier, v. 19, n. 3, p. 163–167, 1986.

DRINKWATER, Bruce W; WILCOX, Paul D. Ultrasonic arrays for non-destructive evaluation: A review. **NDT & e International**, Elsevier, v. 39, n. 7, p. 525–541, 2006.

EWING, Greg. ***Pyrex: C-Extensions for Python***. 2010. Disponível em: https://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/.

FRAZIER, Catherine H; O'BRIEN, William D. Synthetic aperture techniques with a virtual source element. **IEEE transactions on ultrasonics, ferroelectrics, and frequency control**, IEEE, v. 45, n. 1, p. 196–207, 1998.

GORELICK, M.; OZSVALD, I. **High Performance Python: Practical Performant Programming for Humans**. Sebastopol, USA: O'Reilly Media, 2014. 370 p.

GORELICK, Micha; OZSVALD, Ian. **High Performance Python: Practical Performant Programming for Humans**. *[S.l.]*: O'Reilly Media, 2020.

GURURAJA, TR; PANDA, RK. Current status and future trends in ultrasonic transducers for medical imaging applications. *In*: IEEE. **ISAF 1998. Proceedings of the Eleventh IEEE International Symposium on Applications of Ferroelectrics (Cat. No. 98CH36245)**. *[S.l.]*, 1998. p. 223–228.

HARRIS, Charles R; MILLMAN, K Jarrod; WALT, Stéfan J van der; GOMMERS, Ralf; VIRTANEN, Pauli; COURNAPEAU, David; WIESER, Eric; TAYLOR, Julian; BERG, Sebastian; SMITH, Nathaniel J *et al.* Array programming with numpy. **Nature**, Nature Publishing Group, v. 585, n. 7825, p. 357–362, 2020.

HELLIER, C. **Handbook of Nondestructive Evaluation**. New York, NY, USA: Mcgraw-hill, 2003. (McGraw-Hill handbooks).

HUNTER, J. D. Matplotlib: A 2d graphics environment. **Computing in Science & Engineering**, IEEE COMPUTER SOC, v. 9, n. 3, p. 90–95, 2007.

JEUNE, Léonard Le; ROBERT, Sébastien; VILLAVERDE, Eduardo Lopez; PRADA, Claire. Multimodal plane wave imaging for non-destructive testing. **Physics procedia**, Elsevier, v. 70, p. 570–573, 2015.

JEUNE, Léonard Le; ROBERT, Sébastien; VILLAVERDE, Eduardo Lopez; PRADA, Claire. Plane wave imaging for ultrasonic non-destructive testing: Generalization to multimodal imaging. **Ultrasonics**, Elsevier, v. 64, p. 128–138, 2016.

KUMAR, Sanjay; MAHTO, DG. Recent trends in industrial and other engineering applications of non destructive testing: a review. **International Journal of Scientific & Engineering Research**, v. 4, n. 9, 2013.

LAM, Siu Kwan; PITROU, Antoine; SEIBERT, Stanley. Numba: A llvm-based python jit compiler. *In*: **Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC**. *[S.l.: s.n.]*, 2015. p. 1–6.

LANARO, Gabriele. **Python High Performance**. Birmingham, UK: Packt Publishing Ltd, 2017. 264 p.

LANGTANGEN, Hans Petter; BARTH, Timothy J; GRIEBEL, Michael. **Python scripting for computational science**. *[S.l.]*: Springer, 2008. v. 3.

LATTNER, Chris; ADVE, Vikram. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *In*: **Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)**. Palo Alto, California: *[s.n.]*, 2004.

LINGVALL, Fredrik; OLOFSSON, Tomas; STEPINSKI, Tadeusz. Synthetic aperture imaging using sources with finite aperture: Deconvolution of the spatial impulse response. **The Journal of the Acoustical Society of America**, Acoustical Society of America, v. 114, n. 1, p. 225–234, 2003.

LUTZ, Mark. **Programming Python: powerful object-oriented programming**. *[S.l.]*: "O'Reilly Media, Inc.", 2010.

MAROWKA, Ami. Python accelerators for high-performance computing. **The Journal of Supercomputing**, Springer, v. 74, n. 4, p. 1449–1460, 2018.

MCKINNEY, Wes. **Python for data analysis: Data wrangling with Pandas, NumPy, and IPython**. *[S.l.]*: "O'Reilly Media, Inc.", 2012.

MONTALDO, Gabriel; TANTER, Mickaël; BERCOFF, Jérémy; BENECH, Nicolas; FINK, Mathias. Coherent plane-wave compounding for very high frame rate ultrasonography and transient elastography. **IEEE transactions on ultrasonics, ferroelectrics, and frequency control**, IEEE, v. 56, n. 3, p. 489–506, 2009.

NAKAMURA, Kentaro. **Ultrasonic transducers: Materials and design for sensors, actuators and medical applications**. *[S.l.]*: Elsevier, 2012.

NUMBA. **User Manual**. *[S.l.]*, 2020. Disponível em: https://numba.readthedocs.io/en/latest/user/index.html.

RAILLON, Raphaële; TOULLELAN, Gwénaël; DARMON, Michel; CALMON, Pierre; LONNE, Sébastien. Validation of civa ultrasonic simulation in canonical configurations. **18th WCNDT**, 2012.

ROSSUM, Guido Van; DRAKE, Fred L. **The python language reference**. *[S.l.]*: Network Theory Ltd., 2011.

SCHMERR, Lester W. **Fundamentals of ultrasonic nondestructive evaluation**. *[S.l.]*: Springer, 2016.

SHULL, Peter J. **Nondestructive evaluation: theory, techniques, and applications**. *[S.l.]*: CRC press, 2002.

SHUNG, K.K.; ZIPPURO, M. Ultrasonic transducers and arrays. **IEEE Engineering in Medicine and Biology Magazine**, v. 15, n. 6, p. 20–30, 1996.

SIMPSON, Cameron; JEWETT, Jim; TURNBULL, Stephen J.; STINNER, Victor. **Add monotonic time, performance counter, and process time functions**. *[S.l.]*, 2012. Disponível em: https://www.python.org/dev/peps/pep-0008/.

SMITH, Kurt W. **Cython: A Guide for Python Programmers**. *[S.l.]*: "O'Reilly Media, Inc.", 2015.

STEWART, Bruce. **An Interview with Guido van Rossum,**. 2002. Disponível em: http://www.onlamp.com/pub/a/python/2002/06/04/guido.html?page=2.

THOMPSON, R Bruce; THOMPSON, Donald O. Ultrasonics in nondestructive evaluation. **Proceedings of the IEEE**, IEEE, v. 73, n. 12, p. 1716–1755, 1985.

THOMSON, RN. Transverse and longitudinal resolution of the synthetic aperture focusing technique. **Ultrasonics**, Elsevier, v. 22, n. 1, p. 9–15, 1984.

TIOBE. **Index for October 2021**. 2021. Disponível em: https://www.tiobe.com/tiobe-index/.

VIRTANEN, Pauli; GOMMERS, Ralf; OLIPHANT, Travis E; HABERLAND, Matt; REDDY, Tyler; COURNAPEAU, David; BUROVSKI, Evgeni; PETERSON, Pearu; WECKESSER, Warren; BRIGHT, Jonathan *et al.* Scipy 1.0: fundamental algorithms for scientific computing in python. **Nature methods**, Nature Publishing Group, v. 17, n. 3, p. 261–272, 2020.

WESTON, Miles; MUDGE, Peter; DAVIS, Claire; PEYTON, Anthony. Time efficient auto-focussing algorithms for ultrasonic inspection of dual-layered media using full matrix capture. **Ndt & E International**, Elsevier, v. 47, p. 43–50, 2012.