mini_auspex documentation

Version 1.5

LASSIP Team

Jun 15, 2024

CONTENTS

CHAPTER

ONE

ABSTRACT

This research and innovation project aims to develop and thoroughly evaluate the performance of advanced ultrasound signal processing algorithms to assist in underwater inspections. The expected results are a better use of acquired signals, more accurate detection and sizing of defects, reduced off-shore inspection time, shipping costs, and operating ROVs (Remotely Operated underwater Vehicles).

The project works on four work fronts:

1. Corroded Internal Surface Reconstruction: intelligently combine ultrasound signals to form maps of the internal surface of the equipment.
2. Arbitrary External Surface Identification: identify the external geometry of equipment from ultrasound signals and correct sonic trajectories.
3. Online Identification of Inspection Parameters: automatically correct and adjust inspection parameters, such as sound propagation speed, to improve image reconstruction.
4. Organization of Data and Interfaces with Instruments: study and propose an organization of ultrasound inspection data to facilitate the storage, retrieval, and use of data in treatments and analyses based on signals from ultrasound instruments and ultrasound simulators.

Each work front is responsible for the following tasks:

1. Bibliographic survey, identification, and selection of the most relevant state-of-the-art works.
2. Implementação dos algoritmos e reprodução dos resultados descritos nos trabalhos selecionados.
3. Proposal for modifications and new algorithms to adapt the methods to the reality of underwater inspections in the oil industry.
4. Documentation and publication of results in scientific dissemination vehicles.

CHAPTER

TWO

THEORETICAL REVIEW

2.1 Non-destructive ultrasound testing

A *Non-Destructive Test* (NDT) is defined as an examination, test, or evaluation carried out on any object without any alteration to it [Hel03]. The purpose of END is to determine the presence of conditions that may harm the usability of that object. However, for this objective to be achieved, the following are necessary: (I) a definition of appropriate measurement procedures for detecting failure conditions; (II) the design and construction of the instrumentation used to make the measurements; and (III) the development of techniques for analyzing the measurements obtained. It indicates that ENDs belong to a multidisciplinary study area:cite:*Thompson1985*.

There are several techniques used for NDTs, each with advantages and disadvantages, and the choice of the most appropriate method depends on the failure conditions sought and the object inspected. The main techniques, cited by the Brazilian Association of Non-Destructive Testing (ABENDI) [ABE14] are:

- visual inspection;
- radiography, radioscopy, and gammagraphy;
- penetrating liquids;
- ultrasound;
- eddy currents, and;
- acoustic emission.

Among these techniques, ultrasound NDT is one of the most used due to three reasons [TT85]: (I) the ease in generating and receiving ultrasonic signals, which simplifies the development of measuring instruments; (II) the characteristic of deep penetration of ultrasound waves inside the parts, without excessive attenuation and; (III) the ability of return signals (echoes) to carry information related to the characteristics of the material and discontinuities found. Based on this, the ultrasound NDT technique is applied when desired to find discontinuities inside parts and classify and characterize them by their sizes, shapes, orientations, and locations.

The measuring instruments used in ultrasound NDT emit ultrasonic waves into the inspected object and then receive any signals reflected by discontinuities internal to the object, as shown in Fig. 2.1. Then, these echo signals are digitized by an acquisition system and made available for analysis. Such signals are called *amplitude scanning signals* (*A-scan – amplitude scanning*) [Sch98]. Trained and qualified inspectors can analyze such signs and obtain the necessary information to characterize the discontinuities found.

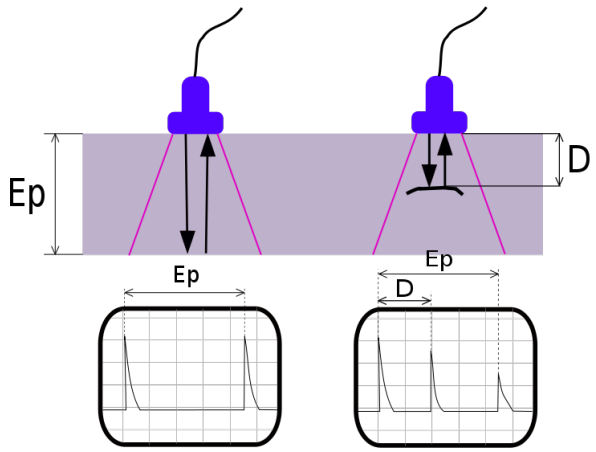


Fig. 2.1: Ultrasound NDT principle.

2.2 Ultrasonic NDT inspection system

Performing an NDT using ultrasound requires an appropriate measurement and acquisition system. Fig. 2.2 presents the basic block diagram of an inspection system of this type. Such a system can be divided into two parts: measurement system and acquisition system. The measurement system is responsible for generating and transmitting ultrasonic waves that affect the inspected part, the reception of echoes emitted by discontinuities found in the part, and their conversion into electrical signals. The acquisition system digitizes the electrical signals of received echoes and makes this data available to computers, where the necessary processing is carried out for subsequent analysis of the signals.

Within the measurement system, the pulser generates electrical pulses of short duration (≈ 0.1 us) and amplitude in the order of hundreds of Volts [Sch98]. These pulses excite a piezoelectric transducer, which emits high-frequency sound waves (ultrasound). As the transducer is in contact with the inspected part, these sound waves propagate through the part material.

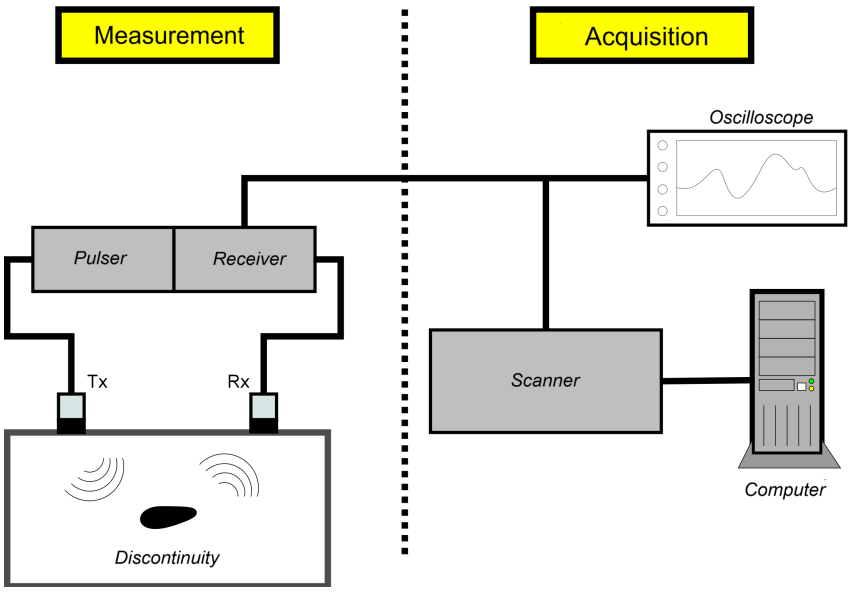


Fig. 2.2: Block diagram of an ultrasound NDT system.

When reaching a discontinuity within the piece, the incident sound waves interact with it. This interaction causes the incident waves to spread so that they are reflected as echoes in different directions [BMS73, Kin79]. The reflected waves can be received by a receiving piezoelectric transducer and converted into electrical signals. After

being amplified, these electrical signals represent in their amplitude the instantaneous energy of the received echoes (at the position of the reception transducer) as a function of time. Such signals are called *A-scan* [Sch98]. An example of an *A-scan* signal is presented in Fig. 2.3.

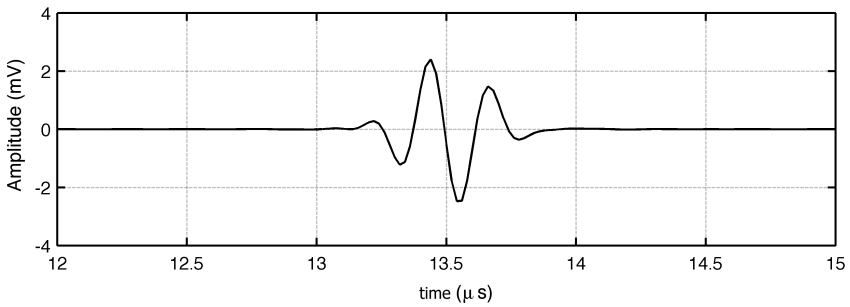


Fig. 2.3: Example of an *A-scan* signal captured by an inspection system.

There are three different configurations for contact inspection using ultrasound: pulse-echo, pitch-catch, and transparency (through-transmission) [Sch98]. These settings are shown in Fig. 2.4.

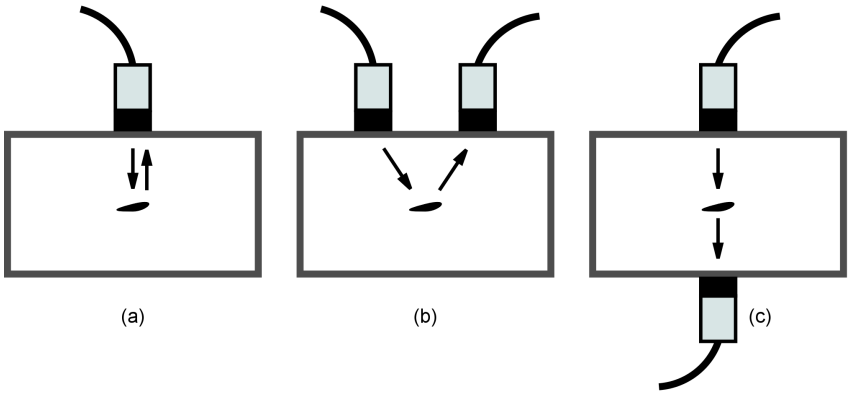


Fig. 2.4: Settings for contact inspection with ultrasound: (a) pulse-echo, (b) pitch-catch, and (c) transparency.

In the pulse-echo configuration, the same transducer emits the ultrasonic pulse and receives the reflected response from any discontinuity in the part. In this configuration, the transducer is in contact with only one of the surfaces of the inspected part, allowing inspection even on parts that have a surface that is difficult to access (e.g., the internal surface of oil and oil product storage tanks). In addition to being possible to detect discontinuitiesinternal to the part, the pulse-echo configuration also allows the measurement of the thickness of the part, with the detection of the echo produced by the reflection of the pulse emitted on the opposite surface [And11].

When two different transducers are used to emit the pulse and receive the echo, but both are in contact with the same surface of the part, there is a configuration called pitch-catch or tandem [MMLK90, SRDillhofer+12]. With this configuration, it is possible to detect, in a more appropriate way, some types of discontinuities existing inside the part, taking advantage of specular reflection and the diffraction of these discontinuities [RLS+05].

An *A-scan* signal contains information about the discontinuity that generated the echo signal. With the time delay between the electrical pulse emitted by the pulser and the pulse observed in the echo signal, it is possible to determine the distance covered by the sound waves from the point of their emission to the end of their reception. In the case of inspections with the pulse-echo configuration, this delay is given by $\Delta t = 2z/c$, where z is the distance from the inspected surface to the discontinuity and c is the speed of sound propagation in the inspected material. The amplitude of the *A-scan* signal depends on the type of discontinuity and its size [DS78, Kin79].

2.3 Phased-array ultrasound

The use of phased-array in NDT has grown in recent times as it presents better performance compared to the use of just a single transducer. Its main advantages are the ability to electronically control the focusing and direction of the beam and the possibility of conducting different inspections in the exact location. Thus, it allows a faster visualization of the object's internal structure to be inspected [DW06].

The phased-array transducer consists of small, individually connected piezoelectric elements, where each element is driven separately, and each response is received independently. Fig. **??** shows a configuration in which the array elements are connected to a circuit and all excited similarly.

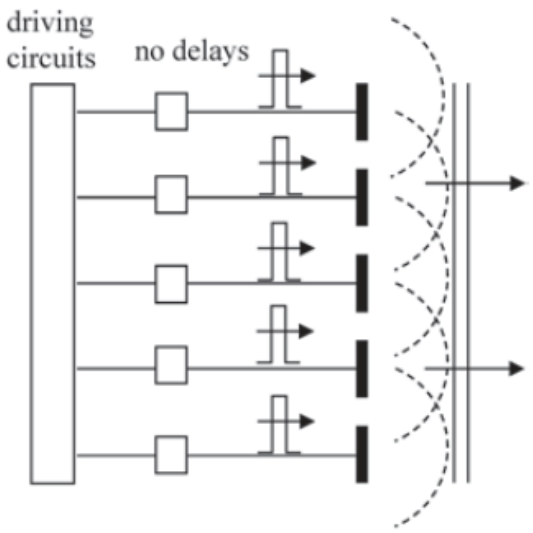


Fig. 2.5: Configuration in which electrical pulses are generated sequentially and without delays [SJ15].

Then, the electrical conduction pulses propagate simultaneously, i.e., without delay. Therefore, each tiny element of the array acts as a source point and radiates a spherical wave; waves formed by each element combine to create a wave pulse, as shown by the dotted lines in Fig. **??**. Another possible configuration for phased-array is demonstrated in Fig. **??**.

This configuration varies the relative time delays of the electrical pulses propagated to the small elements. The set of relative time delays is called the *delay law* [CCSR00]. These delays make the phased-array capable of guiding and focusing the sound beam in different directions without moving the transducer. Relative time delays can also modify the characteristics of signals received into the array. Several pulses are generated as the wave reaches each element of the array. If delays are applied to the received signals, all signals coincide and can be summed.

Because a phased-array can transmit and receive with each element independent of the other elements, it is possible to apply amplitude weights to the elements in the generation and reception of signals. The set of amplitude weights is called *apodization law*. Fig. **??** shows how these amplitude weights are applied to the pulses.

Phased-array transducers have different geometries; according to [DW06], they can be classified as one-dimensional (1-D), two-dimensional (2-D), or annular and are illustrated in Fig. **??**. In the case of 1-D arrays, the elements are distributed in a single direction (x axis). In comparison, 2-D arrays are distributed in two directions ($x - y$), presenting a grid pattern. Annular arrays differ from other arrays in design and do not allow beam guiding.

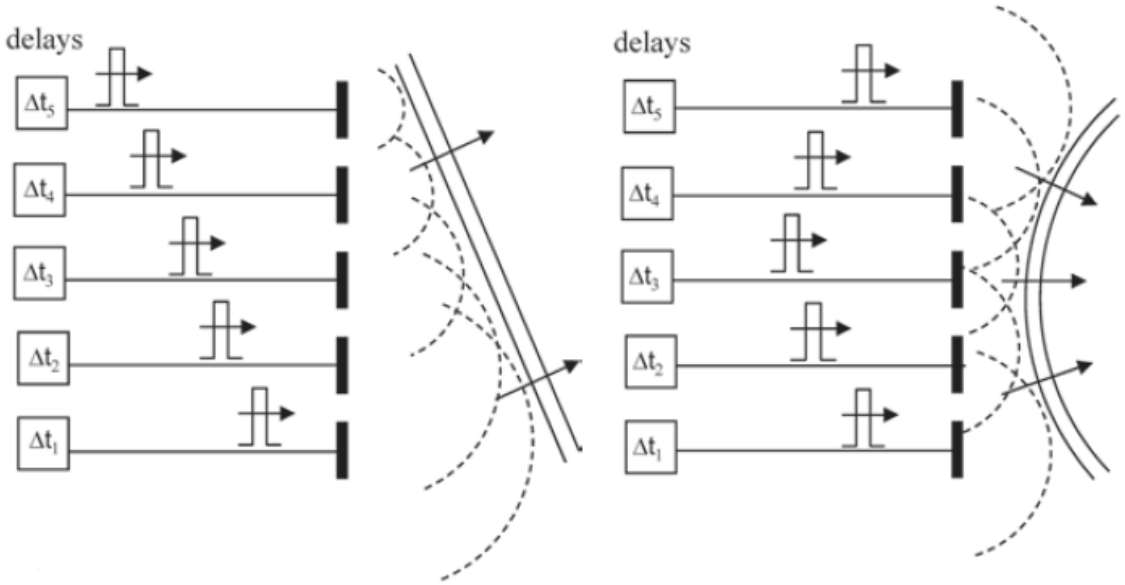


Fig. 2.6: Phased-array configured with time delays, causing the sound beam to be focused and directed [SJ15].

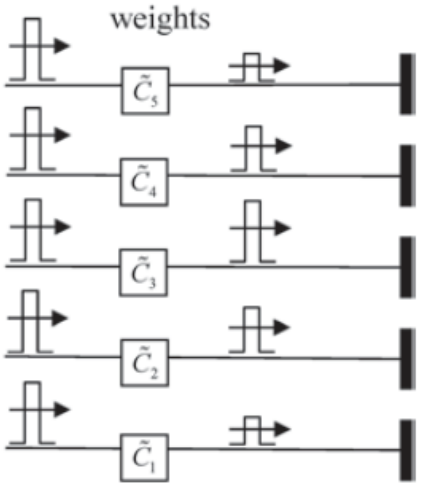


Fig. 2.7: Amplitude weights are applied to the pulses [SJ15].

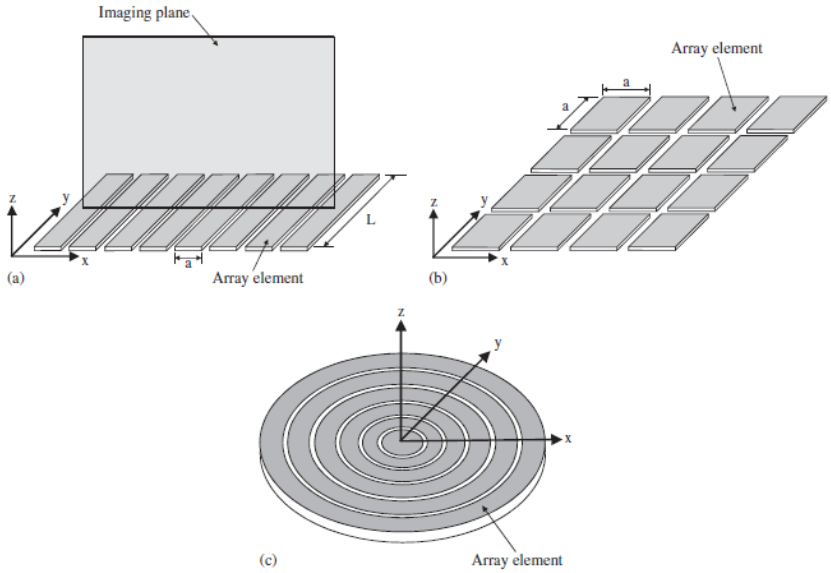


Fig. 2.8: Typical geometries of a phased array. a) 1-D. b) 2-D. and c) annular [SJ15].

CHAPTER

THREE

PROJECT PACKAGES

3.1 Package `framework`

A *framework* is a tool that aims to facilitate and accelerate the development of specific applications. Consider the case of a writer who wants to publish a book. The writer can use a text editor (LibreOffice, TeXworks) that automatically numbers the pages, sections, and equations. With this, the writer directs his focus to developing the content instead of accounting for all the figures in his publication. Furthermore, the writer can quickly create a model with the text editor (framework) used and use it again in other publications.

In software engineering, a framework aims to provide the user with the development of specific applications using ready-made and reusable tools. The literature offers several definitions of framework [eBF88, eDCS97, Fir94, Mat96]. A framework can be considered as an architecture developed to maximize reuse and with the potential for specialization [Mat96]. Furthermore, a framework can be viewed as an abstract project designed to solve a family of problems [eBF88, eDCS97].

3.1.1 Structure of the framework in the AUSPEX project

The AUSPEX project framework has tools that aim to simplify the development of new applications, facilitating the use of data from different sources and enabling quick visualization of results. In this way, greater focus can be directed to developing data processing algorithms and analysis of results.

Fig. **??** illustrates, in a block diagram, the framework as the basis for a new application. The framework has modules for reading data from different simulators or inspection systems. These modules are responsible for converting the specific data format of each source to a standardized data structure, resulting in greater transparency between the application and the data source.

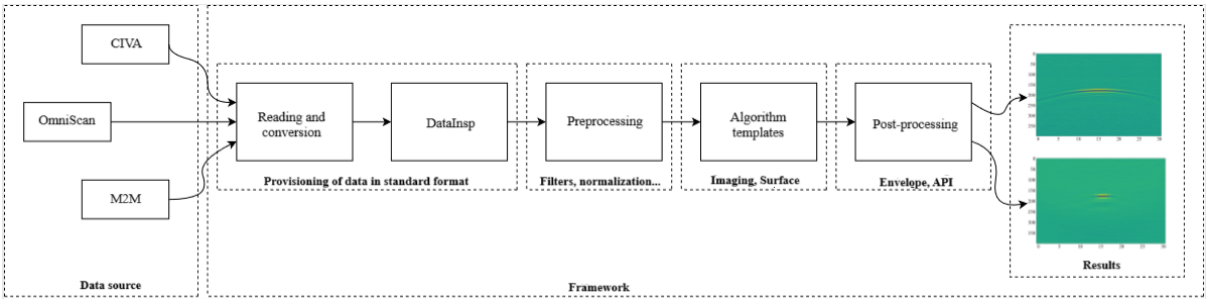


Fig. 3.1: Diagram showing the organization of the proposed framework for theintegration of inspection data from the CIVA simulator andM2M and OmniScan inspection systems.

The framework is organized as a `package` of the language``Python``, where each `module` encapsulates specific functionality. The `data_types` module contains the framework's definitions of all data structures. All modules with functions for importing inspection data from different sources are identified by the prefix `file_`.

Other modules in this `package` encapsulate functions used in implementing the framework's signal-processing algorithms. Each module contains its own documentation with specific information.

3.2 Module `data_types`

The *`data_types`* module contains classes that define data structures used by the *`framework`*. Data structures store inspection data (such as inspection, part, and transducer parameters), generate regions of interest (ROI), and store results from imaging algorithms.

3.2.1 Inspection parameters

The *`InspectionParams`* class defines a data structure to store the various parameters related to an inspection procedure.

The inspection process can be categorized in two ways: contact testing and immersion testing. The *`InspectionParams`* class allows you to define the test type through the *`InspectionParams.type_insp`* attribute, which can be `contact` or `immersion`. Furthermore, the test can be performed with different types of capture, defined by the *`InspectionParams.type_capt`* attribute. Capture types can be `sweep`, for mono transducers; `FMC`, for acquisition with a linear array; and `PWI`, for tests with plane waves.

In the contact test, the transducer can be positioned close to the part or coupled using a wedge. Fig. **??** shows the first case. As indicated in the figure, the transducer is positioned parallel to the surface of the part under inspection, and the waves emitted by the transducer fall on the part at an angle normal to its surface.

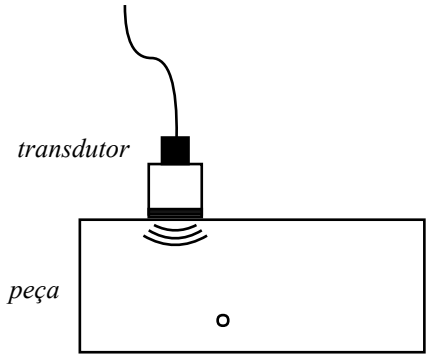


Fig. 3.2: Contact inspection.

Fig. **??** shows where the transducer is coupled to the object under inspection via a wedge. The wedge aims to conduct the sound waves emitted by the transducer, transmitting them to the part. From Fig. **??**, it is possible to notice that, in the case of contact through the wedge, the waves incident on the part have an angle with the surface normal. Furthermore, the wave carried by the wedge undergoes refraction when passing through the part due to the difference in material between the part and the wedge. The *`InspectionParams`* class allows you to define the incidence angle when the inspection is carried out through a wedge, with the *`InspectionParams.impact_angle`* attribute.

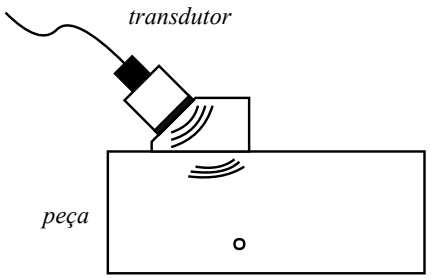


Fig. 3.3: Contact inspection with a wedge.

On the other hand, in immersion inspection, the transducer is coupled to the part by a coupling medium, commonly water, as indicated in Fig. **??**. The figure shows an object and the transducer submerged in water, with the transducer positioned at a height h from the part's surface.

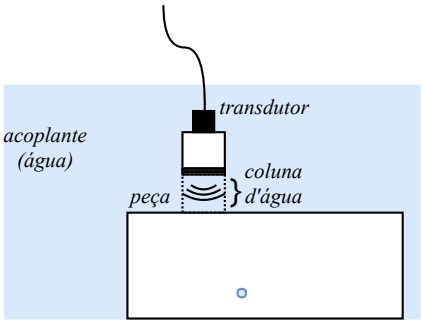


Fig. 3.4: Immersion inspection.

In the case where the test is of the water immersion type, it is possible to define two additional parameters: the speed of sound in water and the size of the water column that separates the transducer from the surface of the part. These parameters are defined in the attributes of attr:*InspectionParams.coupling_cl* and `InspectionParams.water_path`.

The `InspectionParams` class allows storing the coordinates of the initial position of the transducer in the inspection process through the `InspectionParams.point_origin` attribute, as indicated in Fig. **??**. The figure shows the initial position of a transducer under a part, which originates in the upper left corner. In this case, the transducer is positioned at a position (x, y, z) at the origin of the part, which is used as a reference point.

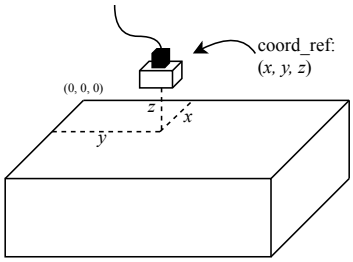


Fig. 3.5: Reference coordinates for inspection.

The inspection positions are defined by the `InspectionParams.step_points` attribute, which is an array with the displacements traveled by the transducer during the inspection process. Each line of the matrix represents a displacement to the initial position of the transducer.

This class also allows you to store electronic aspects of the inspection process. It is possible to define the sampling frequency with which data acquisition from the transducer is performed. Furthermore, the class describes the sampling period as the inverse of the sampling frequency. These parameters can be accessed from the `InspectionParams.sample_freq` and `InspectionParams.sample_time` attributes.

Gate information can be accessed from the `InspectionParams.gate_start`, `InspectionParams.gate_end`, and `InspectionParams.gate_samples` attributes. The gate parameters define the beginning and end of transducer data acquisition and the number of samples taken in this time interval. This way, it is possible to define a window for acquiring data from the transducer, which allows data considered irrelevant for processing to be rejected.

Fig. **??** shows an example of data acquisition from a transducer. When the transducer emits an ultrasonic signal, part of the wave is reflected by the surface of the part under inspection. This reflection can be seen in the A-scan signals, as illustrated in Fig. **??**, where the data in the first microseconds of the acquisition comes from the reflection from the part's surface. This data may not carry relevant information for data processing and subsequent reconstruction of an image. This way, it is possible to define an initial and end time at which the transducer signal will be sampled. In the case of Fig. **??**, this data range can be between 10 us and 25 us. Therefore, the initial and final gate values could be 10 us and 25 us, respectively.

Although the gate has three parameters, it is possible to determine the third from two parameters. For example, given the initial value and end of the gate, obtaining the number of samples is possible since the sampling frequency

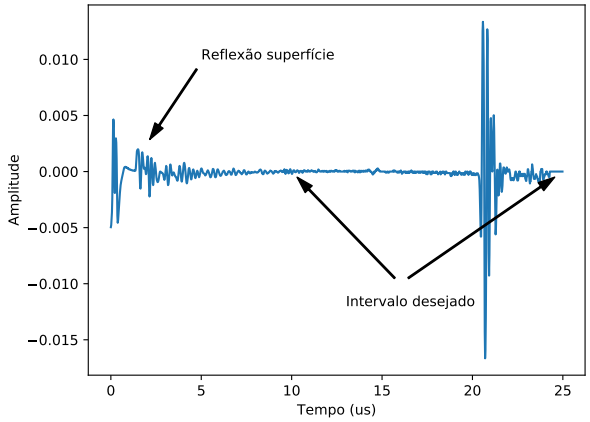


Fig. 3.6: Reference coordinates for inspection.

is fixed. Likewise, with the initial gate value and the number of samples, it is possible to determine the final value. In the current implementation of *framework*, automatic handling of gate parameters is being defined.

For tests using plane waves, the *InspectionParams.angles* attribute is used to store the firing angles used in the test. This attribute is only used when the capture type is PWI, with a value of zero being assigned to other capture types.

3.2.2 Specimen parameters

The *SpecimenParams* class allows you to define the parameters of the part under inspection.

Currently, it is possible to store the speed of longitudinal and transverse waves and the roughness of the part with the attributes *SpecimenParams.cl*, *SpecimenParams.cs*, and *SpecimenParams.roughness*, respectively.

3.2.3 Transducer parameters

The transducers used in NDTs have several parameters that define the geometric and electrical aspects. These parameters are determined and stored with the *ProbeParams* class.

Among the different types of transducers available, the current implementation allows defining a transducer as being of the *mono* or *array* type. The transducer type can be stored and accessed from the *ProbeParams.type_probe* attribute.

The mono transducer consists of just one piezoelectric material element capable of emitting waves and receiving echo signals. Among the various possible shapes, the current implementation considers that the transducer can have a rectangular or circular shape, defined using the *ProbeParams.shape* attribute. Depending on the shape of the transducer, the dimensions to characterize it are different. If the transducer is rectangular, its characterization is based on its length and width. If the element is circular, the radius is sufficient to characterize it. The *ProbeParams.elem_dim* attribute defines the transducer dimensions, which can be a tuple or a number, depending on the transducer geometry.

A linear array transducer comprises several elements arranged side by side. Although the arrangement of elements can take different forms, the current implementation only considers array-type transducers with rectangular elements. The Fig. **??** illustrates a linear array-type transducer composed of rectangular elements, indicating its main parameters. All elements are assumed to have the same length L and width d, while thickness is disregarded. A distance separates elements:math:g, while the distance from the center of one element to the center of the next element is p. The center-to-center distance is also known as *pitch*.

The number of *array* transducer elements is defined by the *ProbeParams.num_elem* attribute. The spacing between elements, the center-to-center distance, and the width of each element can be determined with the attributes *ProbeParams.inter_elem*, *ProbeParams.pitch*, and *ProbeParams.elem_dim*, respectively.

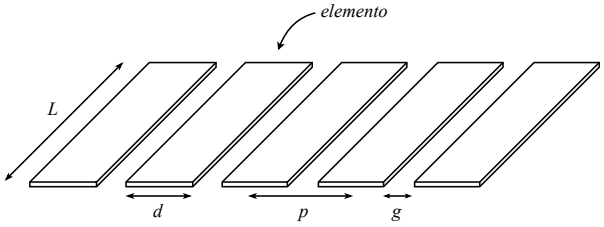


Fig. 3.7: Example of linear *array* transducer.

For imaging algorithms, working with coordinates relative to the transducer position may be more convenient. Therefore, the `ProbeParams` class allows you to define the coordinates of the geometric center of the transducer to the specimen using the `ProbeParams.elem_center` attribute. In the case of the mono transducer, the coordinates refer to its geometric center, as indicated in Fig. **??**, which shows the top view of a mono transducer positioned under a specimen. The geometric center of the transducer is located at a position (x, y, z), which is used as a reference point in imaging algorithms.

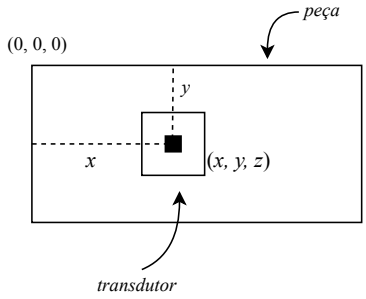


Fig. 3.8: `ProbeParams.elem_center` attribute for mono transducer.

In the case of an array transducer, `ProbeParams.elem_center` is an array containing the coordinates of the geometric centers of all elements, where these coordinates are relative to the center transducer geometric. Fig. **??** illustrates a linear *array* type transducer under a piece in which the geometric center of the first element is at a position (x, y, z).

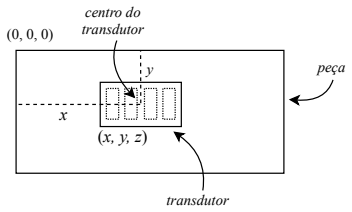


Fig. 3.9: `ProbeParams.elem_center` attribute for linear *array* transducer.

The `ProbeParams` class allows you to define electrical aspects of the transducer, such as center frequency, passband, and type of excitation pulse.

The transducer frequency refers to the resonance frequency of the piezoelectric crystal used. When excited with a pulse, the piezoelectric element will vibrate, emitting sound waves at the resonant frequency. The transducer frequency can be defined using the `ProbeParams.central_freq` parameter.

The bandwidth of the transducer is related to its sensitivity. Transducers with a greater bandwidth have greater sensitivity, as they can detect a broader range of frequencies. The bandwidth of a transducer is generally defined as a percentage of the center frequency. The `ProbeParams.bw` attribute specifies the transducer passband.

The transducer's excitation signal defines what the emitted sound wave will be like forming part of its model. In the current implementation, it is possible to specify the excitation pulse as `Gaussian`, `cossquare`, `hanning` and `hamming`, using the `ProbeParams.pulse_type` attribute.

3.2.4 Inspection data

The *DataInsp* class presents all data related to a complete non-destructive testing process. The class contains the procedure parameters, storing the specimen (*SpecimenParams*), transducer (*ProbeParams*), and inspection (*InspectionParams*) parameters.

In addition to the parameters related to the inspection procedure, the class also has A-scan data, a time grid of A-scan data, and the results of the imaging algorithms.

The A-scan data is generally represented by a 4-dimensional matrix, considering the amplitude data, firing and reception sequence of the transducer elements, and the transducer positions on the part. Data can be accessed from the *DataInsp.ascan_data* attribute.

Consider the data acquisition of the case where the system is pulse-echo, as indicated in Fig. **??**. In this case, it is possible to emit a sound wave and monitor the reflected waves, obtaining a data vector of the transducer signal amplitude over time, the A-scan. By moving the transducer and carrying out the acquisition process again, a new vector of A-scan data is obtained, making it possible to combine the two vectors obtained into a two-dimensional matrix. In this case, the matrix has as many rows as there are samples of the transducer amplitude signal and as many columns as the number of positions in which the signal acquisition was performed. Fig. **??** indicates this process, in which the transducer performs signal acquisition in n positions and, as a result, a two-dimensional matrix is obtained, with each column of the matrix representing a position of the transducer.

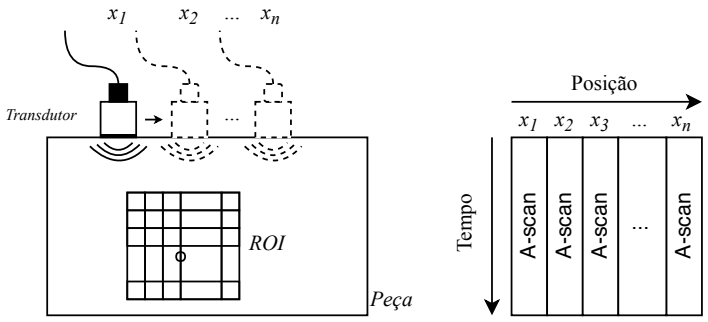


Fig. 3.10: Acquisition with the pulse-echo system.

It is possible to extend this acquisition matrix to the case where the transducer is composed of not a single element but an array of elements. Fig. **??** indicates the acquisition process for a transducer that has several elements and performs data acquisition in a single position. When the transducer triggers activating one or more of its elements, obtaining a data matrix similar to the matrix indicated in Fig. **??** is possible. However, in Fig. **??**, each matrix column represents a different transducer position; in this case, each matrix column represents a different element arranged in various positions. In this way, the matrix indicated in Fig. **??** is obtained from the transducer movement. In contrast, a similar matrix can be obtained with just one transducer shot in the case where the transducer is composed of several elements. A new data matrix is obtained if a new shot is performed, making it possible to form a cube with the data from each shot, as indicated in Fig. **??**.

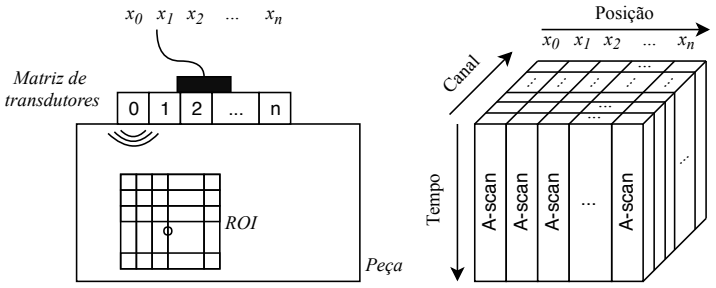


Fig. 3.11: Acquisition with a transducer composed of several elements.

The transducer containing multiple elements can also move and perform a new data acquisition procedure. In this case, it is possible to condense the acquisition data into a 4-dimensional matrix, where the last dimension represents

the position of the transducer.

The *DataInsp* class also has a vector containing the time instants in which the A-scan signals were obtained. As the sampling frequency is fixed, the transducers are sampled simultaneously; therefore, the time vector is common for all A-scan acquisitions. The time vector can be accessed from the *DataInsp.time_grid* attribute.

The imaging results are saved with the inspection A-scan data and accessed through the *DataInsp.imaging_results* attribute.

3.2.5 Region of interest

The *ImagingROI* class allows you to create and store parameters referring to the region of interest (ROI) for image reconstruction. In the current implementation, the class allows the creation of two-dimensional ROIs.

Considering an inspection process, the ROI is a region in which data acquisition and processing are desired, and the ROI of the inspection process may differ from the ROI of data processing.

Fig. **??** shows an inspection process with a mono transducer and defined ROI. The transducer performs data acquisition to cover the entire ROI. In the case of Fig. **??**, the ROI comprises only a part of the specimen, implying the movement of the transducer, which will be restricted, and the data acquisition window, which must ignore the echo signals received before and after the region of interest.

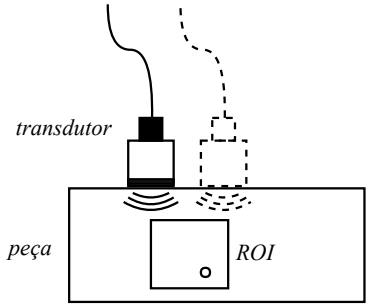


Fig. 3.12: Region of interest for an inspection.

ROI can be defined for image reconstruction and may differ from data acquisition. In the Fig. **??** example, a side drill role (SDH) to be detected is only in a part of the acquisition ROI. Once the point of interest for image processing and reconstruction has been identified, it is possible to define a new ROI exclusive to the imaging algorithms. Fig. **??** illustrates a new ROI used to reconstruct the image in that region of the part. As data processing is performed to generate an image, the ROI for data processing consists of a grid, where each grid point represents the position of a pixel.

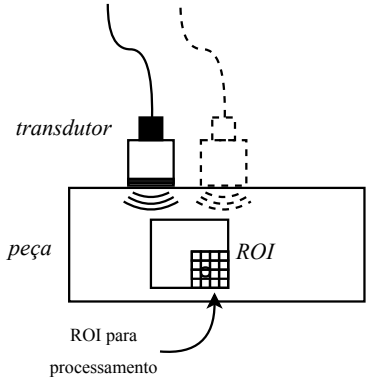


Fig. 3.13: Region of interest for processing.

A region of interest is defined by the height h and the width w and the number of points in each dimension. Fig. **??** illustrates how the ROI is formed, indicating the grid and the arrangement of pixels in the grid. On the vertical

axis, each ROI point (or pixel) is separated by a h/m distance, just as each point on the horizontal axis is separated by a w/n distance.

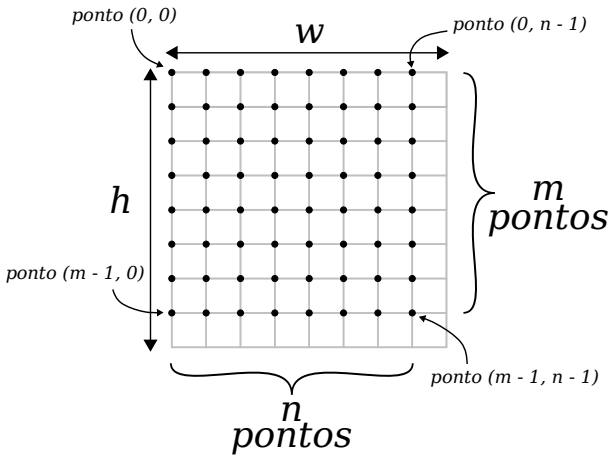


Fig. 3.14: Region of interest for processing.

The *ImagingROI* class allows you to create an ROI, informing the parameters `height`, `h_len`, `width` and `w_len`. The `height` and `h_len` parameters define the h height and the number of pixels in this dimension. Similarly, the parameters `width` and `w_len` define the w width and the number of pixels in the direction of the width dimension. These parameters are stored in the ROI object and can be accessed through the attributes *ImagingROI.height*, *ImagingROI.h_len*, *ImagingROI.width* and :attr:`ImagingROI.w_len `.

Another necessary parameter for defining an ROI is its position on the specimen, represented by coordinates. This parameter is informed when creating an ROI object and can be accessed later via the *ImagingROI.coord_ref* attribute.

After defining the ROI, with its dimensions and position on the part, the absolute coordinates of each point on the mesh can be accessed using the *ImagingROI.get_coord()* method, which returns a 2-dimensional matrix with the coordinates of each point of ROI.

3.2.6 Imaging results

The *ImagingResult* class is used to store images reconstructed from imaging algorithms. The reconstruction results can be summarized in the generated image and the ROI.

The *ImagingResult.image* attribute provides an array of type `np.ndarray` to store the reconstructed image. The size of the image depends on the size of the ROI.

The ROI in which the image was reconstructed is also stored in the class object, in the *ImagingResult.roi* attribute.

In addition to the image and ROI, it is possible to store a description of the result in text form with the *ImagingResult.description* attribute.

class framework.data_types.**InspectionParams**(*type_insp='immersion'*, *type_capt='FMC'*, *sample_freq=100.0*, *gate_start=0.0*, *gate_end=30.0*, *gate_samples=3000*, ***kwargs*)

Class contains the parameters relating to the inspection process.

Parameters

- **type_insp** (*str*) – Type of inspection. It presents two possible values:`immersion` or `contact`. The default value is `immersion`.
- **type_capt** (*str*) – Capture type. Indicates the type of signal capture `A-scan`. Here, it is necessary to check the types configured in CIVA. The possible values are: `sweep`, `FMC`, and `PWI`. The default value is `FMC`.

- **sample_freq** (*int, float*) – Sampling frequency of A-scan signals, in MHz. By default, it is 100 MHz.
- **gate_start** (*int, float*) – Initial value of the *gate*, in microseconds. By default, it is 0.
- **gate_end** (*int, float*) – Final value of the *gate*, in microseconds. By default, it is 30.
- **gate_samples** (*int*) – Number of samples per acquisition channel. By default, it is 3000.

type_insp

Type of inspection. It presents two possible values:immersion or contact. The default value is immersion.

Type
str

type_capt

Capture type. Indicates the type of signal capture A-scan. Here, it is necessary to check the types configured in CIVA. The possible values are: sweep, FMC, and PWI. The default value is FMC.

Type
str

point_origin

Position in space indicating the origin of the coordinate system for the inspection. All other point positions are relative to this point in space. Cartesian points are row vectors, where the first column is the x coordinate, the second column is the y coordinate, and the third column is the z coordinate.

Type
np.ndarray

step_points

Matrix with transducer coordinates during an inspection. Each line of this matrix corresponds to the position of the transducer and is equivalent to an element in the step dimension of the array *DataInsp.ascan_data* in *DataInsp*.

Type
np.ndarray

water_path

Water column length. Exclusive to inspections of type immersion.

Type
int, float

coupling_cl

Sound propagation velocity in the couplant, in m/s. Exclusive for ''immersion" type inspections.

Type
int, float

impact_angle

Incidence angle. Exclusive to contact type inspections.

Type
int, float

sample_freq

Sampling frequency of the A-scan signals, in MHz.

Type
int, float

sample_time

Sampling period of the A-scan signals, in microseconds.

Type
int, float

gate_start

Initial value of the gate, in microseconds.

Type
int, float

gate_end

Final value of the gate, in microseconds.

Type
int,float

gate_samples

Number of samples per acquisition channel.

Type
int, float

angles

Angles of incidence for experiments with plane waves, in degrees.

Type
np.ndarray

class framework.data_types.**ElementGeometry**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

CIRCULAR = 1

RECTANGULAR = 2

class framework.data_types.**SpecimenParams**(*cl=5900*, *cs=3230*, *roughness=0.0*)

Class containing the parameters of the inspected specimen.

Parameters

- **cl** (*int, float*) – Longitudinal wave propagation velocity in the specimen, in m/s. By default, it is 5900 m/s.
- **cs** (*int, float*) – Transverse wave propagation velocity in the specimen, in m/s. By default, it is 3230 m/s.
- **roughness** (*int, float*) – Roughness. By default, it is 0.0.

cl

Longitudinal wave propagation velocity in the specimen, in m/s.

Type
int, float

cs

Transverse wave propagation velocity in the specimen, in m/s.

Type
int, float

roughness

Roughness.

Type
int, float

class framework.data_types.**ProbeParams**(*tp='linear'*, *num_elem=32*, *pitch=0.6*, *dim=0.5*, *inter_elem=0.1*, *freq=5.0*, *bw=0.5*, *pulse_type='gaussian'*, *elem_list=None*)

Class containing the parameters of the transducer.

In the current implementation, the supported types are mono and linear.

Parameters

- **tp** (*str*) – Transducer type. Possible types are: `mono` (single element) and `linear` (linear array). By default, it is of type `linear`.
- **num_elem** (*int*) – Number of elements. Exclusive for transducers of type `linear`. By default, it is 32.
- **pitch** (*int, float*) – Gap between the centers of the elements, in mm. Exclusive for transducers of type `linear`. By default, it is 0.6 mm.
- **dim** (*int, float*) – The dimensions of the transducer elements are in mm. If the active element is circular, the value represents the diameter. If the active element is rectangular, the value is a tuple in the form `(dim_x, dim_y)`. If the active element is rectangular for a linear array, the value is the smallest dimension of the active element. By default, it is 0.5 mm.
- **inter_elem** (*int, float*) – Gap between elements, in mm. Exclusive for transducers of type ''linear". By default, it is 0.1 mm.
- **freq** (*int, float*) – Central frequency, in MHz. By default, it is 5 MHz.
- **bw** (*int, float*) – Bandwidth, as a percentage of the central frequency. By default, it is 0.5 (50%).
- **pulse_type** (*str*) – Excitation pulse type. Possible types are: `gaussian`, `cossquare`, `hanning`, and `hamming`. By default, it is `gaussian`.

`type_probe`

Transducer type. Possible types are: `mono` (single element) and `linear` (linear array).

Type
str

`num_elem`

Number of elements. Exclusive for transducers of type `linear`.

Type
int

`inter_elem`

Gap between elements, in mm. Exclusive for transducers of type `linear`.

Type
int, float

`pitch`

Gap between the centers of the elements, in mm. Exclusive for transducers of type `linear`.

Type
int, float

`elem_center`

If the transducer is of the `linear` type, it is a matrix with the Cartesian coordinates of the geometric center of each element in mm. These coordinates are relative to the geometric center of the transducer. If the transducer is of the `mono` type, it is the central position of the active element of the transducer, in mm.

Type
`np.ndarray`

shape

Transducer format. Possible values are `circle` and `rectangle`. The default value is `circle`. Exclusive for transducers of type `mono`.

Type
str

elem_dim

Dimension of the transducer elements, in mm. If the active element is circular, the value represents the diameter. If the active element is rectangular, the value is a tuple in the format `(dim_x, dim_y)`. If the active element is rectangular for a linear array, the value is the smaller dimension of the active element.

Type
int, float

central_freq

Central frequency, in MHz.

Type
int, float

bw

Bandwidth, as a percentage of the central frequency.

Type
int, float

pulse_type

Excitation pulse type. Possible types are: `gaussian`, `cossquare`, `hanning`, and `hamming`.

Type
str

class framework.data_types.**ImagingROI**(*coord_ref=array([[0., 0., 0.]])*, *height=20.0*, *h_len=200*, *width=20.0*, *w_len=200*, *depth=0.0*, *d_len=1*)

Class that stores the parameters of the region of interest (ROI) for image reconstruction.

Objects of this type are used as parameters for image reconstruction algorithms and should be stored together with the results of these algorithms.

Parameters

- **coord_ref** (np.ndarray) – Cartesian point indicating the reference coordinate of the ROI, in mm. By default, it is (0.0, 0.0, 0.0) mm.
- **height** (*int, float*) – Height of the ROI, in mm. By default, it is 20.0 mm.
- **h_len** (*int*) – Number of points in the ROI height dimension. By default, it is 200.
- **width** (*int, float*) – Width of the ROI, in mm. By default, it is 20.0 mm.
- **w_len** (*int*) – Number of points in the ROI width dimension. By default, it is 200.
- **depth** (*int, float*) – Depth of the ROI (in a linear transducer, typically corresponds to the passive direction). By default, it is 0.0 mm (two-dimensional ROI).
- **d_len** (*int*) – Number of points in the ROI depth dimension. By default, it is 1 (two-dimensional ROI).

coord_ref

Cartesian point indicating the reference coordinate of the ROI, in mm.

Type
np.ndarray

h_points

Vector with the coordinates of the ROI in the height direction (dimension 1) of the image, in mm.

Type
np.ndarray

h_len

Number of points of the ROI in the height direction.

Type
int

h_step

Step size of the ROI points in the height direction, in mm.

Type
float

height

Height of the ROI, in mm.

Type
float

w_points

Vector with the coordinates of the ROI in the width direction (dimension 2) of the image.

Type
np.ndarray

w_len

Number of points of the ROI in the width direction.

Type
int

w_step

Step size of the ROI points in the width direction, in mm.

Type
float

width

Width of the ROI, in mm.

Type
float

d_points

Vector with the coordinates of the ROI in the depth direction (dimension 1) of the image.

Type
np.ndarray

d_len

Number of points of the ROI in the depth direction.

Type
int

d_step

Step size of the ROI points in the depth direction, in mm.

Type
float

depth

Depth of the ROI, in mm.

Type
float

Raises
TypeError – Raises a `TypeError` exception if `coord_ref` is not of type `np.ndarray` and/or does not have 1 row and three columns.

Notes

This class applies to ROIs in two and three dimensions.

get_coord()

Method that returns all coordinates of the ROI (*mesh*) in vectorized format.

Returns
Matrix M x 3, where M is the number of points in the ROI. Each row of this matrix is the Cartesian coordinate of a point in the ROI.

Return type
`np.ndarray`

class framework.data_types.**ImagingResult**(*roi=<framework.data_types.ImagingROI object>*, *description=''*)

Class storing the results obtained from executing imaging algorithms.

Parameters

- **roi** (*ImagingROI*) – Image ROI
- **description** (*str*) – Text describing the result.

image

Array for storing the reconstructed image. This *array* has two dimensions (image). By default, the image should be composed of raw values, without any post-processing. If any post-processing is required, it should be performed by the data visualization methods.

Type
`np.ndarray`

roi

Image ROI

Type
ImagingROI

description

Text describing the result.

Type
str

name

Name of the algorithm used for image reconstruction.

Type
str

class framework.data_types.**DataInsp**(*inspection_params=<framework.data_types.InspectionParams object>*, *specimen_params=<framework.data_types.SpecimenParams object>*, *probe_params=<framework.data_types.ProbeParams object>*)

Class containing all the necessary data for a non-destructive test.

Parameters

- **inspection_params** (*InspectionParams*) – Object containing the inspection parameters.
- **specimen_params** (*SpecimenParams*) – Object containing the sample parameters.
- **probe_params** (*ProbeParams*) – Object containing the transducer parameters.

inspection_params

Object containing the inspection parameters.

Type
InspectionParams

specimen_params

Object containing the sample parameters.

Type
SpecimenParams

probe_params

Object containing the transducer parameters.

Type
ProbeParams

ascan_data

Array for storing the *A-scan* signals. This is a four-dimensional array. The first dimension represents the time scale of the *A-scan* signals (*time*). The second dimension represents the transducer firing sequence (*sequence*). This dimension will always be unitary for transducers of type *mono*. The third dimension represents the transducer reception channels (*channel*). This dimension will always be unitary for transducers of type *mono*. The fourth dimension represents the transducer steps (*step*). Each index of this dimension is directly associated with the number of coordinates in the list *InspectionParams.step_points*.

Type
np.ndarray

ascan_data_sum

Array for storing the sum of received *A-scan* signals in a test. This is a three-dimensional *array*. The first dimension represents the time scale of the *A-scan* signals (*time*). The second dimension represents the transducer firing sequence (*sequence*). This dimension will always be unitary for transducers of type *mono*. The third dimension represents the transducer steps (*step*). Each index of this dimension is directly associated with the number of coordinates in the list *InspectionParams.step_points*. This array will only exist if the inspection test is configured to collect the sum of the channels. Otherwise, it will have a value of *None*.

Type
np.ndarray

time_grid

Array for storing the time grid for all *A-scan* signals. Like the A-scan signals, this array is a column vector.

Type
np.ndarray

imaging_results

Dictionary containing objects of type *ImagingResult* containing the results of the execution of the image reconstruction algorithms.

Type
ImagingResult

dataset_name

String containing the name of the dataset. This is information provided by Panther.

Type
str

3.3 Module file_civa

The *file_civa* module is dedicated to reading files from the CIVA simulator.

CIVA is developed by CEA and partners in simulation Non-Destructive Testing (NDTs).

It is a software platform comprising six modules with multiple knowledge intended for developing and optimizing NDT methods and probe design. Its objectives are to improve the quality of NDT techniques and assist in the interpretation of complex inspection results.

Simulations are essential in identifying potential defects from part design during testing, qualifying methods, optimizing parameters, and analyzing disturbance factors to developing samples for NDTs of sample geometries or similar structures to initial designs.

As the simulation data output format is in text files, which makes data reading excessively slow, we chose to read the files in the *.CIVA* format.

These *.CIVA* files have a series of directories called *procN*, where *N* corresponds to the simulation gating number. Only gating 0 is considered in this framework module, and the others are disregarded.

In the *proc0* directory, the simulation settings (part, transducer, and inspection parameters) are in a *model.xml* file. Because *XML* is an inherently hierarchical data format, the most natural way to represent it is with a tree. The *etree* library is used to read these files. The *etree* library has two classes: the first is *ElementTree*, which represents the entire *XML* document as a tree; the second is *Element* which means a single node in this tree.

Já os *A-scan* são salvos no arquivo *channels_signal_Mephisto_gate_1* quando a inspeção é feita com transdutor linear, e *sum_signal_Mephisto_gate_1*, para inspeções com transdutor mono.

The *A-scan* signals are saved in the file *channels_signal_Mephisto_gate_1* when the inspection is carried out with a linear transducer, and *sum_signal_Mephisto_gate_1* for inspections with a mono transducer.

framework.file_civa.**read**(*filename*, *sel_shots=None*, *read_ascan=True*)

Open and parse a .civa file, returning the simulation data.

The data is returned as an object of the class *DataInsp*, containing the inspection parameters, transducer parameters, piece parameters, and simulation data.

Parameters

- **filename** (*str*) – Path to the .civa file.
- **sel_shots** (*NoneType, int, list ou range*) – *Shots* for reading. If it is None, reads all available *shots*. If int, reads the specified index. If list or range, reads the specified indices.

Returns
Data from the read file, containing inspection parameters, transducer parameters, piece parameters, and simulation data.

Return type
DataInsp

Raises

- **TypeError** – Raises a TypeError exception if the parameter sel_shots is not of type NoneType, int, list, or range.

- **IndexError** – Raises an IndexError exception if the specified shot does not exist.
- **FileNotFoundError** – Raises a FileNotFoundError exception if the file does not exist.

3.4 Module file_m2k

The file_m2k module is responsible for reading files with the .m2k extension. Currently, the module can read .m2k files generated by inspections with Multix++ and Panther equipment.

A-scan data is obtained from decoding binary files generated by inspection with the equipment. The inspection parameters are obtained from processing the .xml files that accompany the binary files, containing information such as sampling frequency, gate information, and capture type, among others.

The module also supports reading files the Panther equipment produces with multiple acquisitions and capture types. If saved, the images produced by the equipment are also found in binary files and are automatically detected and made available in the data_types attribute.DataInsp.imaging_results.

class framework.file_m2k.**DataDescSave**

Class containing information regarding the position of the data stored in the binary file acq_data.bin.

id

Identifier

Type
numpy.uint32

file_pointer

List of pointers in the binary data file.

Type
list

index

Indexer (we are not yet sure of this field's function).

Type
numpy.uint64

carto

Information about the mapping of the motion captured by the encoders (mechanical or time).

Type
numpy.array

pad

Pad (these bytes always store the number 4).

Type
numpy.uint32

bytes_per_channel

Number of bytes stored in each channel.

Type
list

type50

Information "type 50". Number of bytes in a TFM image.

Type
numpy.uint32

type51

Information "type 51". Half of the offset quantity in a TFM image.

Type
numpy.uint32

type52

Information "type 52". Half of the offset quantity in a TFM image.

Type
numpy.uint32

total_bytes()

Method that returns the total number of bytes stored for all channels.

Returns
Sum of all bytes contained in the list bytes_per_channel.

Return type
int

has_recep_ascan()

This method returns whether the total number of bytes stored for received *A-scan* signals was stored.

Returns
True if there was storage of the received *A-scan* signals.

Return type
bool

has_sum_ascan()

This method returns whether the summed A-scan signal of all received A-scans was stored.

Returns
True if there was storage of the summed *A-scan* signal.

Return type
bool

has_tfm()

This method returns whether Acquire stored a TFM image.

Returns
True if there was storage of TFM images by Acquire.

Return type
bool

has_encoders_info()

This method returns whether encoder information was stored.

Returns
True if there was storage of encoder information.

Return type
bool

framework.file_m2k.**read**(*filename*, *freq_transd*, *bw_transd*, *tp_transd*, *sel_shots=None*, *read_ascan=True*, *type_insp='contact'*, *water_path=0.0*)

Open a .m2k file and populate a *DataInsp* object.

It is considered that the amplitudes of the A-scan data are 2 bytes.

Parameters

- **filename** (str) – Path to the .m2k. file.

- **sel_shots** (NoneType, int, list ou range) – *Shots* for reading. If it is None, reads all available *shots*. If int, reads the specified index. If list or range, reads the specified indices. By default, it is None.
- **read_ascan** (bool) – Flag indicating the reading of A-scan signals. It is True by default.
- **type_insp** (str) – Type of inspection. It can be immersion or contact. It is contact by default.
- **water_path** (float) – If the inspection is of type immersion, water_path defines the size of the water column separating the transducer from the piece, in mm. By default, it is 0 mm.
- **freq_transd** (float) – Transducer nominal frequency, in MHz. By default, it is 5.0 MHz.
- **bw_transd** (float) – Transducer bandwidth, as a percentage of the central frequency. By default, it is 0.5%.
- **tp_transd** (str) – Transducer excitation pulse type. By default, it is gaussian.

Returns
Test data performed, which may contain inspection parameters, of the transducer and part, in addition to *A-scan* data.

Return type
DataInsp

Raises

- **TypeError** – Raises a TypeError exception if the parameter sel_shots is not of type NoneType, int, list, or range.
- **IndexError** – Raises an IndexError exception if the specified shot does not exist.

3.5 Module pre_proc

This module implements algorithms for processing the data loaded by the framework. All functions receive an object of type *data_types.DataInsp* and an array of type numpy.ndarray named *shots*.

framework.pre_proc.**remove_media**(*data_insp*, *shots=array([0])*)

Removes the mean from a signal.

Parameters

- **data_insp** (*data_types.DataInsp*) – Object containing the data loaded by the *framework*.
- **shots** (np.ndarray) – Vector with the shots to be processed.

Returns
Data processed.

Return type
numpy.ndarray

framework.pre_proc.**add_noise**(*data_insp*, *snr=50*, *shots=array([0])*)

Adds noise with desired SNR.

Parameters

- **data_insp** (*data_types.DataInsp*) – Object containing the data loaded by the *framework*.
- **snr** (float) – Desired SNR.

- **shots** (np.ndarray) – Vector with the shots to be processed.

Returns
Noisy data.

Return type
numpy.ndarray

framework.pre_proc.**sum_shots**(*data_insp*, *shots=array([0])*)

Summarizes several different shots. Saves the result in the first shot of the list.

Parameters

- **data_insp** (*data_types.DataInsp*) – Object containing the data loaded by the *framework*.
- **shots** (np.ndarray) – Vector with the shots to be summed.

Returns
Shot with the summed data.

Return type
numpy.ndarray

framework.pre_proc.**matched_filter**(*data_insp*, *shots=array([0])*)

Filters the signal with a matched filter to improve the SNR [Turin60] (reference).

It is assumed that the echoes have the shape of a Gaussian pulse with the characteristics of the transducer, central frequency, and bandwidth. The transducer characteristics are the same as those present in the DataInsp structure. This Gaussian pulse is the frequency response of the filter applied to each A-scan.

Parameters

- **data_insp** (*data_types.DataInsp*) – Object containing the data loaded by the *framework*.
- **shots** (np.ndarray) – Vector with the shots to be summed.

Returns
Data processed.

Return type
numpy.ndarray

framework.pre_proc.**hilbert_transforms**(*data*, *shots=array([0])*, *N=2*)

Hilbert transform for very heavy files, aiming to reduce computational costs. The fastest approach found was performing the transform 2 shots at a time in an FMC (Full Matrix Capture).

Parameters

- **data** (*o arquivo a ser aplicada a transformada, deve ser tipo data_insp ou FMC;*)
- **shots** (*os shots nos quais será realizada a transformada*)

Returns
data

Return type
retorna um ponteiro do data.ascan_data

3.6 Module post_proc

This module implements functions that apply *post-processing* operations to the results provided by image reconstruction algorithms. All functions in this module take an image as the main parameter (in the form of a two-dimensional *array*) and apply necessary operations to facilitate the user's analysis of the images.

framework.post_proc.**envelope**(*image*, *axis=-2*)

This function calculates the envelope in an image created from some reconstruction algorithm. The image envelope is calculated using the Hilbert Transform, taking only one axis of the image.

Parameters

- **image** (np.ndarray) – Image like a two-dimensional *array*.
- **axis** (*int*) – Axis to apply the envelope.

Returns
Image envelope.

Return type
numpy:ndarray

Raises
TypeError – Raises a TypeError exception if the parameter image is not of type np.ndarray.

framework.post_proc.**normalize**(*image*, *final_min=0*, *final_max=1*, *image_min=None*, *image_max=None*)

This function normalizes the values of an image, always placing them between [final_min; final_max]. It places all values in the interval [0; 1] if only the image is passed as an argument.

Parameters

- **image** (np.ndarray) – Image like a two-dimensional *array*.
- **final_min** (float) – Minimum value of the final image.
- **final_max** (float) – Maximum value of the final image.
- **image_min** (float) – Minimum value to be considered for the input image. By default, the lowest value present in the image is considered.
- **image_max** (float) – Maximum value to be considered for the input image. By default, the highest value present in the image is considered.

Returns
Normalized image.

Return type
numpy:ndarray

Raises
TypeError – Raises a TypeError exception if the parameter image is not of type np.ndarray.

framework.post_proc.**api**(*image*, *roi*, *wavelength=0.00118*)

This function calculates the API index of an image. This index, defined in [HDW05], indicates the area of the image that is above -6 dB. The 0 dB threshold is relative to the highest absolute value of the image.

Parameters

- **image** (np.ndarray) – Image like a two-dimensional *array*.
- **roi** (*framework.data_types.ImagingROI*) – Region of interest (ROI) of the image
- **wavelength** (float) – Wavelength of the ultrasonic pulse used in the inspection process.

Returns
API index.

Return type
`float`

Raises
TypeError – Raises a `TypeError` exception if the parameter `image` is not of type `np.ndarray`.

`framework.post_proc.cnr`(*foreground*, *background*)

This function calculates the API index of an image. This index, defined in [HDW05], indicates the area of the image that is above -6 dB. The 0 dB threshold is relative to the highest absolute value of the image.

Parameters

- **foreground** (`np.ndarray`) – Foreground image in the form of a two-dimensional *array*.
- **background** (`np.ndarray`) – Background image in the form of a two-dimensional array.

Returns
Noise contrast ratio.

Return type
float

Raises
TypeError – Raises a `TypeError` exception if the parameter `foreground` or `background` is not of type `np.ndarray`.

3.7 Package `imaging`

Image-based techniques are undoubtedly the most used among all the available methods for analyzing ultrasonic signals. Several authors, such as [CT94, DBS96, DHR86, MullerSSchafer86, SRDillhofer+12, vBMS93], indicate that the presentation of an image improves the performance of inspectors when interpreting ultrasound inspection data. Thus, the problem in question is how to create the image of a discontinuity, initially unknown, from a set of *A-scan* signals measured by the measurement system and possibly distorted by noise. This type of problem is defined as *image reconstruction* [Bov00].

Images provided by an ultrasound inspection system represent the acoustic reflectivity within an object [vBMS93]. They are created by applying an appropriate reconstruction algorithm to a set of *A-scan* signals. There are several algorithms for reconstructing images in ultrasound NDT. The most straightforward algorithm, called *B-scan*, assembles an image as a matrix of dots. In it, each column represents the spatial position of the transducer, and each line corresponds to the propagation time of the ultrasonic waves from the transducer to a position within the inspected object. The intensity of each point in the image is proportional to the amplitude of the *A-scan* signal related to the position of the transducer and the propagation time. A *B-scan* image shows the inspected object's profile representation (side section). Although the *B-scan* algorithm is simple and fast in image reconstruction, it has a low lateral resolution. Furthermore, the transducer diameter and discontinuity depth affect the quality of the reconstructed image [SCMuller00, SYHY12] due to the effects of diffraction and beam scattering [Kin87].

In the early 1970s, the technique of *Synthetic Aperture Focusing Technique* (SAFT) [BGH74, Pri72, Sey82] was developed to improve the lateral resolution of reconstructed images. This technique was inspired by the concepts of *Synthetic Aperture* (SA) used in airborne radar mapping systems [SRR62] In general, SAFT is implemented by sum and shift operations directly on *A-scan* signals [CGK78, FSF76, KCBP80]. However, it can also be implemented in other ways, such as matrix-vector multiplication [LOS03]; Stolt migration [Sto78] applied to *A-scan* signals in the frequency domain (algorithm wk) [CC00, GH97, MMLK90, Ste07]; and using distributed processing in *Graphics Processing Units* (GPU) [MartinARLMartinezG+12].

The *B-scan* and SAFT algorithms were initially developed for measurement systems containing monostatic transducers (with a single active element) in pulse-echo configuration. However, there has been a significant increase

in measurement systems that use transducers with multiple active elements, so-called array transducers [HDW05]. With these transducers, measurement systems can electronically control the ultrasound beam's opening, direction, and focus over discontinuities [DW06]. They can also emulate the behavior of a monostatic transducer by sequentially triggering each element of the array. Thus, it provides two advantages: (I) it avoids physical movement of the transducer to scan the region of interest, and (II) it allows the acquisition of *A-scan* signals for all combinations of transmitting and receiving elements. This signal acquisition mode is called *Full Matrix Capture* (FMC) [HDW05]. FMC allows image reconstruction using other algorithms, such as the *Total Focusing Method* (TFM) [HDW05]; *Inverse Wave Field Extrapolation* (IWEX) [PGB07]; a version of the *wk-SAFT* algorithm for FMC [HDW08], an invertible backpropagation algorithm [VW09] and an adaptive beamforming algorithm [LH11]. In all these algorithms, the image resolution of a point reflector is improved when compared to *B-scan* and SAFT [HDW05, LH11, VW10]. Despite the advantages of array transducers, measurement systems with monostatic transducers are still widely used, especially in portable and embedded systems.

3.7.1 Package `imaging` in the AUSPEX project

The *imaging* package contains `Python` implementations of algorithms for image reconstruction within the AUSPEX project. All algorithms in this package must have the same interface, as they can be used both in developing *script* applications and with graphical human-machine interfaces.

The interface standard for image reconstruction algorithms adopted in the AUSPEX project is as follows:

- Each algorithm must be implemented as a module in the *imaging* package.
- Module names must identify the algorithm.
- Modules must contain two public access functions, `xxxx_kernel` and `xxxx_params`, where `xxxx` is necessarily the name of the module.
- The `xxxx_kernel` function contains the implementation of the algorithm, while the `xxxx_params` function is for use in applications with graphical human-machine interfaces.

All `xxxx_kernel` functions must receive at least five mandatory parameters:

- An instance of the *framework.data_types.DataInsp* class contains all data from an inspection section.
- An instance of the *framework.data_types.ImagingROI* class, which defines the region of the reconstructed image.
- An *identification key* for the dictionary that stores the result of the algorithm within the *framework.data_types.DataInsp* object.
- A *string* to identify the result.
- In the case of multiple inspections, the *step* index selector.

In addition, each algorithm may require other parameters, such as propagation speed, regularization parameters, and threshold levels. The return of the `xxxx_kernel` functions is done by inserting an instance of the *framework.data_types.ImagingResult* class into the *framework.data_types.DataInsp.imaging_results* dictionary. The key of this object in the dictionary is returned to the caller of `xxxx_kernel`.

The `xxxx_params` functions do not need any parameters. They return a dictionary whose elements are the default values of the parameters of the `xxxx_kernel` function. The keys of the elements must be the name of each parameter. All `xxxx_kernel` parameters must have a default value, except the *framework.data_types.DataInsp* instance.

3.7.2 Algorithms and examples

Currently, the algorithms available in the *imaging* package are:

- B-scan
- SAFT
- _
- _
- TFM
- _
- _
- _
- CPWC
- _
- _
- _
- _
- _

Each algorithm is documented separately, accompanied by an example of use. The examples use synthetic inspection data from the CIVA simulator. The algorithms use the *framework.file_civa* module to read and process the file generated by the simulator.

The specimen used for the simulation can be seen in Fig. **??**. The piece is 80 mm wide, 60 mm high, and 25 mm deep (not shown). A 1 mm diameter side drilled hole is located 40 mm from the top and 40 mm from the left corner, with the hole going through the part.

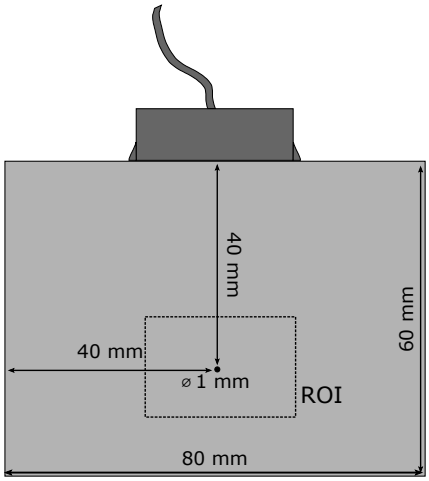


Fig. 3.15: Part used to simulate the test at CIVA.

A linear array transducer with 32 elements and a central frequency of 5 MHz was used for the simulation. The inspection data was obtained using the FMC capture method.

3.8 Module bscan

B-scan is an image reconstruction algorithm for non-destructive testing that receives all elements of the ultrasonic signals from the aperture. In the time domain, these signals are then combined to produce the final B-Scan image.

3.8.1 Example

The following *script* shows the use of the B-scan algorithm to reconstruct an image from synthetic data from the CIVA simulator. (The data is assumed to be in the same folder where the *script* is run).

The *script* shows the procedure for reading a simulation file using the *framework.file_civa* module; data processing using the *imaging.bscan* module; and data post-processing using the *framework.post_proc* module.

The result of the *script* is an image displaying the result of the algorithm and the result with post-processing.

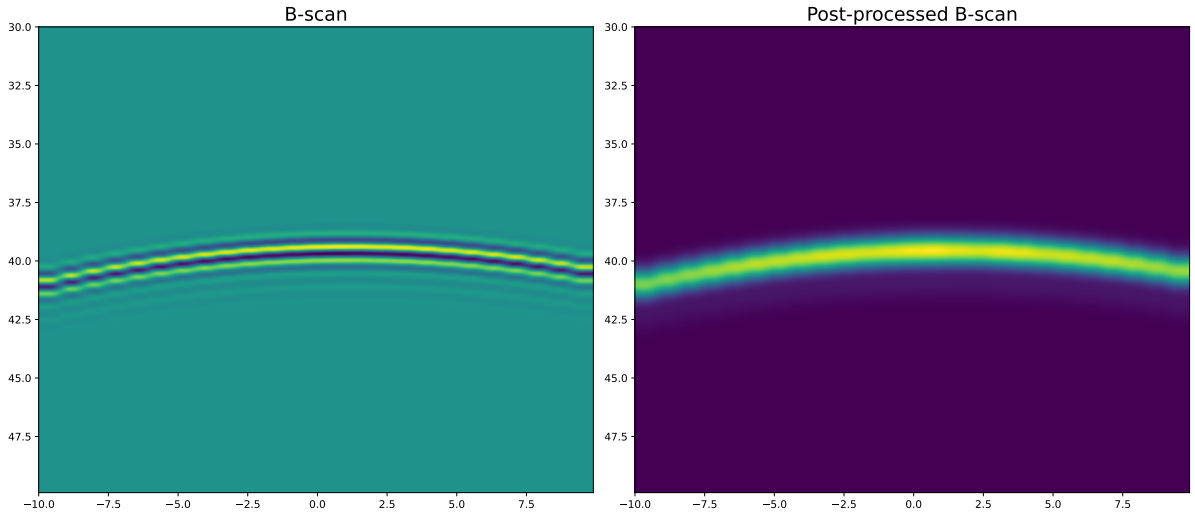
```
import numpy as np
import matplotlib.pyplot as plt
from framework import data_types, file_civa
from imaging import bscan
from framework.post_proc import envelope, normalize

# --- Data ---
# Loads inspection data from the CIVA simulation file.
data = file_civa.read("Furo40mmPA_FMC_Contact_new.civa")

# --- ROI ---
# Defines a 20 mm x 20 mm ROI.
height = 20.0
width = 20.0

# Defines the ROI, starting in (-10, 0, 30).
corner_roi = np.array([[-10.0, 0.0, 30.0]])
roi = data_types.ImagingROI(corner_roi, height=height, width=width)

# --- Processing ---
# Gets the B-scan image. Note that the algorithm only returns
# the identification key, and the result is saved in the
# "data" variable. Furthermore, the algorithm obtains the
# image at the ROI defined above.
bscan_key = bscan.bscan_kernel(data, roi)

# --- Images ---
plt.figure(figsize=(16, 7))

# Shows the B-scan image.
plt.subplot(1, 2, 1)
plt.imshow(data.imaging_results[bscan_key].image, aspect='auto',
           extent=[roi.w_points[0], roi.w_points[-1], roi.h_points[-1], roi.h_points[0]])
plt.title('B-scan', fontsize=18)

# Displays the result of the B-scan algorithm with a normalized envelope.
plt.subplot(1, 2, 2)
plt.imshow(normalize(envelope(data.imaging_results[bscan_key].image, -2)), aspect='auto',
           extent=[roi.w_points[0], roi.w_points[-1], roi.h_points[-1], roi.h_points[0]])
plt.title('Post-processed B-scan', fontsize=18)

plt.tight_layout()
plt.show()
```

imaging.bscan.**bscan_kernel**(*data_insp*, *roi=<framework.data_types.ImagingROI object>*, *output_key=None*, *description=''*, *sel_shot=0*, *c=None*)

Processes A-scan data using the B-scan algorithm.

Parameters

- **data_insp** (*data_types.DataInsp*) – Inspection data, containing inspection parameters, piece parameters, transducer parameters, and the structure to save the obtained results.



- **roi** (*data_types.ImagingROI*) – Region of interest where the algorithm will be executed. The dimensions of the ROI should be in mm.
- **output_key** (*None ou int*) – Identifier key of the processing result. The attribute *data_types.DataInsp.imaging_results* is a dictionary capable of storing multiple processing results. The key is a numeric value representing the ID of the result, while the value is the processing result itself. If `output_key` is `None`, a new random key is generated, and the result is stored in the dictionary. If `int`, the result is stored under the specified key, creating a new entry if the key does not exist in the dictionary or overwriting the previous results if the key already exists. By default, it is `None`.
- **description** (*str*) – Descriptive text for the result. By default, it is an empty string.
- **sel_shot** (*int*) – Parameter referring to the shot if the transducer has been displaced. By default, it is 0.
- **c** (*int ou float*) – Propagation velocity of the wave in the object under inspection. By default, it is None, and in this case, the value from data_insp is obtained.

Returns
Identification key of the result (`output_key`).

Return type
int

Raises

- **TypeError** – If `data_insp` is not of type *data_types.DataInsp*.
- **TypeError** – If `roi` is not of type *data_types.ImagingROI*.
- **TypeError** – If `output_key` is not of type `NoneType` or if it cannot be converted to `np.int32`.
- **TypeError** – If `description` is not of type `str` or if it cannot be converted to `str`.
- **TypeError** – If `c` is not of type `float` or if it cannot be converted to `float`.
- **NotImplementedError** – If the capture type (*data_types.InspectionParams.type_capt*) is not `sweep` or `FMC`.

imaging.bscan.**bscan_params**()

Returns the parameters of the B-scan algorithm.

Returns
Dictionary, where the key `roi` represents the region of interest used by the algorithm, the key `output_key` represents the identification key of the result, the key `description` represents

the description of the result, the key `sel_shot` represents the transducer shot, and the key `c` represents the velocity of wave propagation in the piece.

Return type
dict

3.9 Module saft

SAFT (Synthetic Aperture Focusing Technique) is a tool to restore ultrasonic images obtained from *B-scans* with focus distortion. Using SAFT, an improvement in image resolution can be obtained without traditional ultrasonic lenses.

Synthetic focusing is based on geometric reflection or acoustic ray model.

The algorithm model considers that the focus of the ultrasonic transducer is assumed to be a point of constant phase through which all sound rays pass before diverging into a cone whose angle is determined by the transducer diameter and focal length.

If a reflective target is located below the focal point and within the cone, the path length and transit time for a signal traveling along the beam will be calculated. The cone's width in a given range corresponds to the aperture width that can be synthesized, and the path length that the signal must travel corresponds to the phase shift seen in the signal for that transducer position.

3.9.1 Example

The following *script* shows the use of the SAFT algorithm to reconstruct an image from synthetic data from the CIVA simulator. (The data is assumed to be in the same folder where the *script* is run).

The *script* shows the procedure for reading a simulation file using the *framework.file_civa* module; data processing using the *imaging.bscan* and *imaging.saft* modules; and data post-processing using the *framework.post_proc* module.

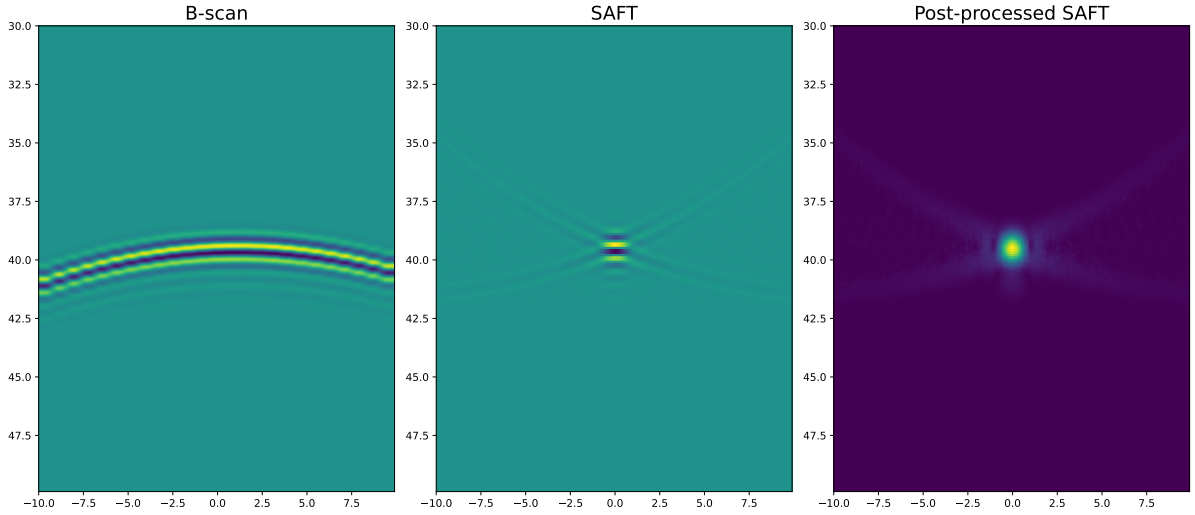
The result of the *script* is an image comparing the reconstructed image with the B-scan algorithm and the SAFT algorithm. Furthermore, the image shows the result of SAFT with post-processing.

```
import numpy as np
import matplotlib.pyplot as plt
from framework import data_types, file_civa
from imaging import saft, bscan
from framework.post_proc import envelope, normalize

# --- Data ---
# Loads inspection data from the CIVA simulation file.
data = file_civa.read("Furo40mmPA_FMC_Contact_new.civa")

# --- ROI ---
# Defines a 20 mm x 20 mm ROI.
height = 20.0
width = 20.0

# Defines the ROI, starting in (-10, 0, 30).
corner_roi = np.array([[-10.0, 0.0, 30.0]])
roi = data_types.ImagingROI(corner_roi, height=height, width=width)

# --- Processing ---
# Gets the B-scan image. Note that the algorithm only returns
# the identification key, and the result is saved in the
# "data" variable. Furthermore, the algorithm obtains the
# image at the ROI defined above.
bscan_key = bscan.bscan_kernel(data, roi)

# Gets the SAFT image.
saft_key = saft.saft_kernel(data, roi)
```

(continues on next page)

(continued from previous page)

```
# --- Images ---
plt.figure(figsize=(16, 7))

# Shows the B-scan image.
plt.subplot(1, 3, 1)
plt.imshow(data.imaging_results[bscan_key].image, aspect='auto',
           extent=[roi.w_points[0], roi.w_points[-1], roi.h_points[-1], roi.h_points[0]])
plt.title('B-scan', fontsize=18)

# Shows the SAFT image.
plt.subplot(1, 3, 2)
plt.imshow(data.imaging_results[saft_key].image, aspect='auto',
           extent=[roi.w_points[0], roi.w_points[-1], roi.h_points[-1], roi.h_points[0]])
plt.title('SAFT', fontsize=18)

# Displays the result of the SAFT algorithm with a normalized envelope.
plt.subplot(1, 3, 3)
plt.imshow(normalize(envelope(data.imaging_results[saft_key].image, -2)), aspect='auto',
           extent=[roi.w_points[0], roi.w_points[-1], roi.h_points[-1], roi.h_points[0]])
plt.title('Post-processed SAFT', fontsize=18)

plt.tight_layout()
plt.show()
```



imaging.saft.**saft_oper_direct**(*image*, *roi*, *coord_transd*, *nt*, *nu*, *c=5900.0*, *dt=1e-08*, *tau0=0.0*)

Calculates Kirchhoff modeling (Claerbout, 2004, p. 108)

Calculates how each point in the image influences the A-scan signals. It performs each ROI point's *scattering* operation on the A-scan signals.

imaging.saft.**saft_oper_adjoint**(*data*, *roi*, *coord_transd*, *c=5900.0*, *dt=1e-08*, *tau0=0.0*)

Calculates Kirchhoff migration (Claerbout, 2004, p.108).

Equivalent to the SAFT algorithm. It takes a set of A-scan signals and performs the *Delay-and-Sum* operation to obtain an image.

imaging.saft.**saft_kernel**(*data_insp*, *roi=<framework.data_types.ImagingROI object>*, *output_key=None*, *description=''*, *sel_shot=0*, *c=None*, *scattering_angle=None*)

Processes A-scan data using the SAFT algorithm.

Parameters

- **data_insp** (*data_types.DataInsp*) – Inspection data, containing inspection, part and transducer parameters, as well as the structure to save the results obtained.

- **roi** (*data_types.ImagingROI*) – Region of interest in which the algorithm will be executed. ROI dimensions must be in mm.
- **output_key** (*None ou int*) – Key identifier for the processing result. The attribute *data_types.DataInsp.imaging_results* is a dictionary capable of storing various processing results. The key (*key*) is a numeric value representing the ID of the result, while the value (*value*) is the processing result. If output_key is None, a new random key is generated, and the result is stored in the dictionary. If it is an integer, the result is stored under the specified key, creating a new entry if the key does not exist in the dictionary or overwriting previous results if the key already exists. By default, it is None.
- **description** (*str*) – A descriptive text for the processing result. By default, it's an empty *string*.
- **sel_shot** (*int*) – Parameter referring to the shot in case the transducer has been displaced. By default, it's 0.
- **c** (*int ou float*) – Propagation velocity of the wave in the object under inspection. By default, it's None, and in this case, the value from data_insp is obtained.
- **scattering_angle** (*float ou nd-array*) – Filter to consider the width of the beam emitted by each transducer element. By default it is None to be defined later.

Returns
Key identifier of the processing result (output_key).

Return type
int

Raises

- **TypeError** – If data_insp is not of type *data_types.DataInsp*.
- **TypeError** – If roi is not of type *data_types.ImagingROI*.
- **TypeError** – If output_key is not of type NoneType or if it cannot be converted to np.int32.
- **TypeError** – If description is not of type str or if it cannot be converted to str.
- **TypeError** – If c is not of type float or if it cannot be converted to float.
- **NotImplementedError** – If the capture type (*data_types.InspectionParams.type_capt*) is not sweep or FMC.

imaging.saft.**saft_params**()

Returns the SAFT algorithm parameters.

Returns
Dictionary, where the key roi represents the region of interest used by the algorithm, the key output_key represents the result identification key, the key description represents the result description, the key sel_shot represents the transducer shot, and the key c represents the wave propagation velocity in the piece.

Return type
dict

3.10 Module `tfm`

TFM (*Total Focusing Method*) is an image reconstruction algorithm for non-destructive testing. The inspection system uses ultrasonic transducers *phased array*, and the capture system is FMC (*Full Matrix Capture*). In TFM, the beam is focused on all points of the `ROI`. The first step of the algorithm consists of discretizing the ROI in the (x, z) plane on a defined grid. Then, the signals from all matrix elements are summed to synthesize a focus at all grid points. The image intensity $I(x, z)$ is calculated at any point in the scan using the Equation (**??**):

$$I(x,z) = \left| \sum h_{tx,rx} \left(\frac{\sqrt{(x_{tx}-x)^2+z^2} + \sqrt{(x_{rx}-x)^2+z^2}}{c} \right) \right|, \tag{3.1}$$

where c is the speed of sound in the medium, x_{tx} and x_{rx} is the lateral positions of the transmitting and receiving elements, respectively [HDW05].

Due to the need to perform linear interpolation of previously discretely sampled time domain signals, summing is performed for each possible transmitter-receiver pair. It uses the maximum amount of information available for each point.

The main limitation of this technique is computing time.

3.10.1 Example

The following *script* shows the use of the TFM algorithm to reconstruct an image from synthetic data from the CIVA simulator. (The data is assumed to be in the same folder where the *script* is run).

The *script* shows the procedure for reading a simulation file using the *`framework.file_civa`* module; data processing using the *`imaging.bscan`* and *`imaging.tfm`* modules; and data post-processing using the *`framework.post_proc`* module.

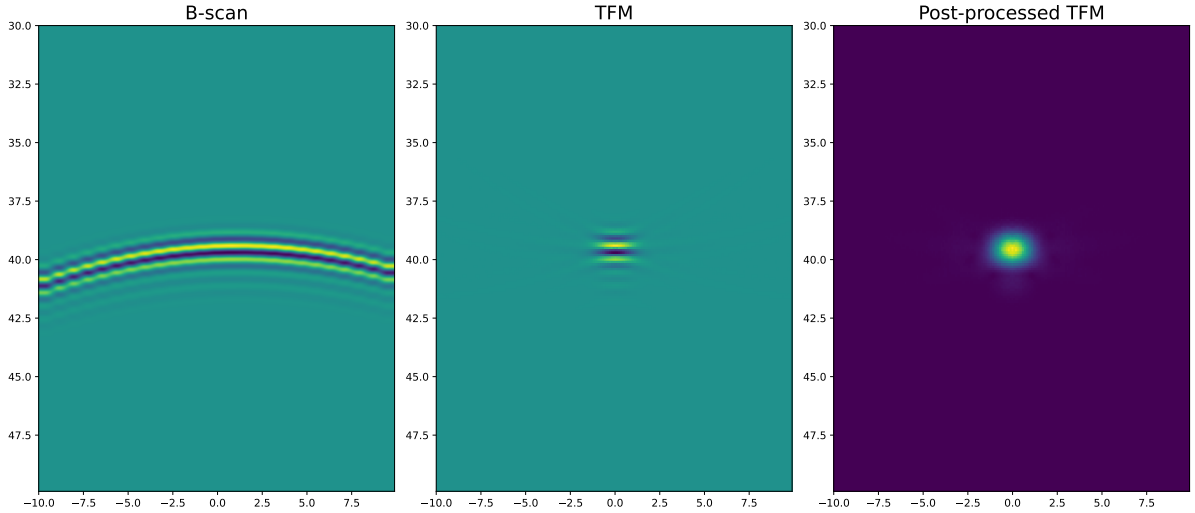
The result of the *script* is an image comparing the reconstructed image with the B-scan algorithm and the TFM algorithm. Furthermore, the image shows the result of TFM with post-processing.

```
import numpy as np
import matplotlib.pyplot as plt
from framework import data_types, file_civa
from imaging import tfm, bscan
from framework.post_proc import envelope, normalize

# --- Data ---
# Loads inspection data from the CIVA simulation file.
data = file_civa.read("Furo40mmPA_FMC_Contact_new.civa")

# --- ROI ---
# Defines a 20 mm x 20 mm ROI.
height = 20.0
width = 20.0

# Defines the ROI, starting in (-10, 0, 30).
corner_roi = np.array([[-10.0, 0.0, 30.0]])
roi = data_types.ImagingROI(corner_roi, height=height, width=width)

# --- Processing ---
# Gets the B-scan image. Note that the algorithm only returns
# the identification key, and the result is saved in the
# "data" variable. Furthermore, the algorithm obtains the
# image at the ROI defined above.
bscan_key = bscan.bscan_kernel(data, roi)

# Gets the TFM image.
tfm_key = tfm.tfm_kernel(data, roi)

# --- Images ---
plt.figure(figsize=(16, 7))

# Shows the B-scan image.
```

(continues on next page)

(continued from previous page)

```
plt.subplot(1, 3, 1)
plt.imshow(data.imaging_results[bscan_key].image, aspect='auto',
           extent=[roi.w_points[0], roi.w_points[-1], roi.h_points[-1], roi.h_points[0]])
plt.title('B-scan', fontsize=18)

# Shows the TFM image.
plt.subplot(1, 3, 2)
plt.imshow(data.imaging_results[tfm_key].image, aspect='auto',
           extent=[roi.w_points[0], roi.w_points[-1], roi.h_points[-1], roi.h_points[0]])
plt.title('TFM', fontsize=18)

# Displays the result of the TFM algorithm with a normalized envelope.
plt.subplot(1, 3, 3)
plt.imshow(normalize(envelope(data.imaging_results[tfm_key].image, -2)), aspect='auto',
           extent=[roi.w_points[0], roi.w_points[-1], roi.h_points[-1], roi.h_points[0]])
plt.title('Post-processed TFM', fontsize=18)

plt.tight_layout()
plt.show()
```



imaging.tfm.**tfm_params**()

Returns the parameters of the TFM algorithm.

Returns

Dictionary, where the key `roi` represents the region of interest used by the algorithm, the key `output_key` represents the identification key of the result, the key `description` represents the result description, the key `sel_shot` represents the transducer shot, the key `c` represents the propagation velocity of the wave in the part, and `trcomb` represents the combinations of transmitters and receivers used.

Return type

dict

imaging.tfm.**tfm2D_kern**(*data_insp*, *roi=<framework.data_types.ImagingROI object>*, *output_key=None*, *description=''*, *sel_shot=0*, *c=None*, *trcomb=None*, *scattering_angle=None*, *elem_geometry=ElementGeometry.RECTANGULAR*, *analytic=False*)

Processes A-scan data using the TFM algorithm.

Parameters

- **data_insp** (*data_types.DataInsp*) – Inspection data, containing inspection parameters, piece parameters, transducer parameters, and the structure to save the obtained results.
- **roi** (*data_types.ImagingROI*) – Region of interest (ROI) in which the algorithm will be executed. The dimensions of the ROI should be in millimeters.

- **output_key** (*None ou int*) – Key identifier for the processing result. The attribute *data_types.DataInsp.imaging_results* is a dictionary capable of storing various processing results. The key (*key*) is a numeric value representing the ID of the result, while the value (*value*) is the processing result. If output_key is None, a new random key is generated, and the result is stored in the dictionary. If it is an integer, the result is stored under the specified key, creating a new entry if the key does not exist in the dictionary or overwriting previous results if the key already exists. By default, it is None.
- **description** (*str*) – Descriptive text for the result. By default, it is an empty *string*.
- **sel_shot** (*int*) – Parameter that refers to triggering if the transducer has been moved.
- **c** (*int ou float*) – Propagation velocity of the wave in the object under inspection. By default, it's None, and in this case, the value is obtained from data_insp.
- **trcomb** (*None ou 2d-array int*) – Specify which combinations of Transmitter and Receiver elements to use.
- **scattering_angle** (*None, float, ou 2d-array bool*) – Provides an angle from which a map of points influencing the A-scan is generated. Optionally, the map can be provided directly.
- **elem_geometry** (framework.data_types.ElementGeometry) – Geometry of the transducer elements, which is used in the calculation of the set of elements framed in the scattering_angle. By default, it is RECTANGULAR.
- **analytic** (*bool*) – If True, the TFM calculation is performed on the analytical signal, generating an image with complex values. If False, the TFM calculation is performed on the raw data, generating an image with real values.

Returns
Key identifier for the result (output_key).

Return type
int

Raises

- **TypeError** – If data_insp is not of type *data_types.DataInsp*.
- **TypeError** – If roi is not of type *data_types.ImagingROI*.
- **TypeError** – If output_key is not of type NoneType or if it cannot be converted to np.int32.
- **TypeError** – If description is not of type str or if it cannot be converted to str.
- **TypeError** – If sel_shot is not of type int or if it cannot be converted to int.
- **TypeError** – If c is not of type float or if it cannot be converted to float.
- **NotImplementedError** – If the capture type (*data_types.InspectionParams.type_capt*) is not sweep or FMC.

imaging.tfm.**tfm3d_kern**(*data_insp*, *roi=<framework.data_types.ImagingROI object>*, *output_key=None*, *description=''*, *sel_shot=0*, *c=None*, *trcomb=None*, *scattering_angle=None*, *elem_geometry=ElementGeometry.RECTANGULAR*, *analytic=False*)

Processes A-scan data using the TFM algorithm.

Parameters

- **data_insp** (*data_types.DataInsp*) – Inspection data, containing inspection parameters, piece parameters, transducer parameters, and the structure to save the obtained results.
- **roi** (*data_types.ImagingROI*) – Region of interest (ROI) in which the algorithm will be executed. The dimensions of the ROI should be in millimeters.

- **output_key** (*None ou int*) – Key identifier for the processing result. The attribute *data_types.DataInsp.imaging_results* is a dictionary capable of storing various processing results. The key (*key*) is a numeric value representing the ID of the result, while the value (*value*) is the processing result. If output_key is None, a new random key is generated, and the result is stored in the dictionary. If it is an integer, the result is stored under the specified key, creating a new entry if the key does not exist in the dictionary or overwriting previous results if the key already exists. By default, it is None.
- **description** (*str*) – Descriptive text for the result. By default, it is an empty *string*.
- **sel_shot** (*int*) – Parameter that refers to triggering if the transducer has been moved.
- **c** (*int ou float*) – Propagation velocity of the wave in the object under inspection. By default, it's None, and in this case, the value is obtained from data_insp.
- **trcomb** (*None ou 2d-array int*) – Specify which combinations of Transmitter and Receiver elements to use.
- **scattering_angle** (*None, float, ou 2d-array bool*) – Provides an angle from which a map of points influencing the A-scan is generated. Optionally, the map can be provided directly.
- **elem_geometry** (framework.data_types.ElementGeometry) – Geometry of the transducer elements, which is used in the calculation of the set of elements framed in the scattering_angle. By default, it is RECTANGULAR.
- **analytic** (*bool*) – If True, the TFM calculation is performed on the analytical signal, generating an image with complex values. If False, the TFM calculation is performed on the raw data, generating an image with real values.

Returns
Key identifier for the result (output_key).

Return type
int

Raises

- **TypeError** – If data_insp is not of type *data_types.DataInsp*.
- **TypeError** – If roi is not of type *data_types.ImagingROI*.
- **TypeError** – If output_key is not of type NoneType or if it cannot be converted to np.int32.
- **TypeError** – If description is not of type str or if it cannot be converted to str.
- **TypeError** – If sel_shot is not of type int or if it cannot be converted to int.
- **TypeError** – If c is not of type float or if it cannot be converted to float.
- **NotImplementedError** – If the capture type (*data_types.InspectionParams.type_capt*) is not sweep or FMC.

3.11 Module cpwc

CPWC (*Coherent Plane Wave Compounding*) is an algorithm used to reconstruct images when the type of inspection is by *plane waves*. In this method, all linear *array* transducer elements are fired simultaneously, creating a single wavefront that illuminates the object under inspection, as illustrated in Fig. **??**.

It can transmit plane waves with inclinations, applying *delay* to trigger the transducer elements, as indicated in Fig. **??**.

The CPWC algorithm produces a final image by summing the images obtained from waves with different angles. Each image is formed by applying *delay*-and-sum to the A-scan signals, with the *delay* applied depending on the

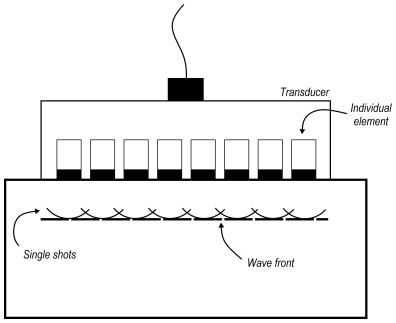


Fig. 3.16: Plane waves inspection.

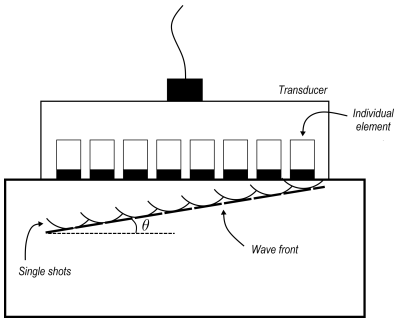


Fig. 3.17: Plane waves inspection.

position of the image point and the angle of the plane wave. Fig. **??** illustrates the distances traveled by a wave emitted with an inclination θ to the surface of a part.

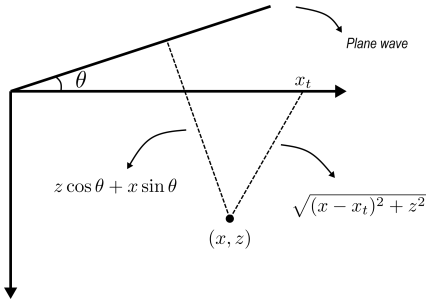


Fig. 3.18: Plane waves inspection.

The distance d_i that the wave travels until it reaches a point (x, z) is a function of the position of the point and the slope of the wave:

$$d_\mathrm{i} = z\cos\theta + x\sin\theta.$$

After reaching the (x, z) point, the wave can be reflected. The distance d_v traveled by the wave, from the point (x, z) to a transducer positioned at $(x_t, 0)$ is:

$$d_\mathrm{v} = \sqrt{(x - x_t)^2 + z^2}.$$

The *delay* τ applied to the transducer signal in x_t is obtained from the total distance traveled by the wave and its speed c in the middle:

$$\tau_{x_t} = \frac{d_i + d_v}{c}.$$

3.11.1 Example

The following *script* shows the use of the CPWC algorithm to reconstruct an image from synthetic data from the CIVA simulator. (The data is assumed to be in the same folder where the *script* is run).

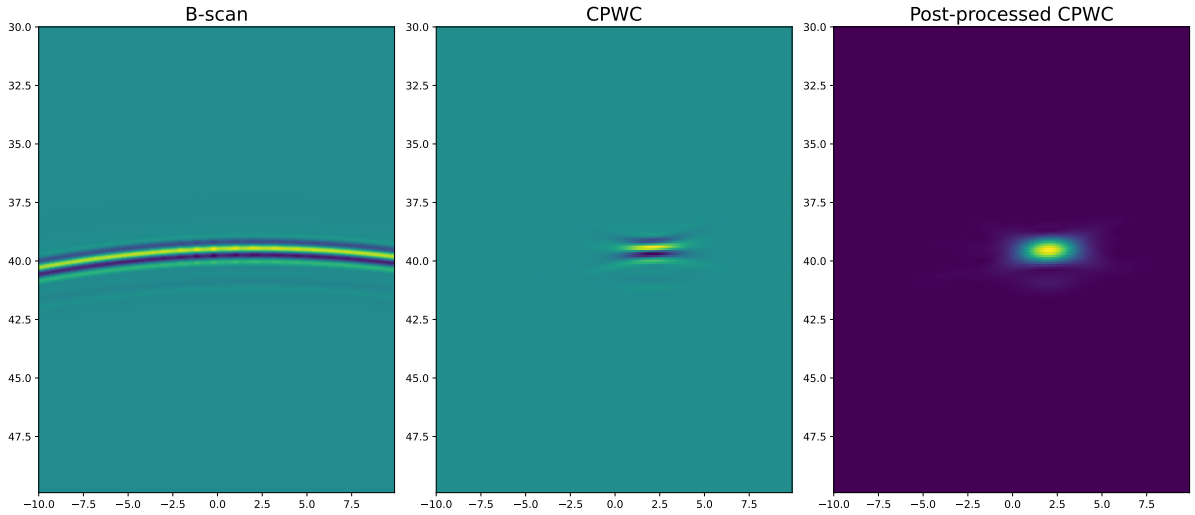
The *script* shows the procedure for reading a simulation file using the *framework.file_civa* module; data processing using the *imaging.bscan* and *imaging.cpwc* modules; and data post-processing using the *framework.post_proc* module.

The result of the *script* is an image comparing the reconstructed image with the B-scan algorithm and the CPWC algorithm. Furthermore, the image shows the result of CPWC with post-processing.

```
import numpy as np
import matplotlib.pyplot as plt
from framework import data_types, file_civa
from imaging import cpwc, bscan
from framework.post_proc import envelope, normalize

# --- Data ---
# Loads inspection data from the CIVA simulation file.
data = file_civa.read("../../../data/peca_80_60_25_ensaio_pw_validation.civa")

# --- ROI ---
# Defines a 20 mm x 20 mm ROI.
height = 20.0
width = 20.0

# Defines the ROI, starting in (-10, 0, 30).
corner_roi = np.array([[-10.0, 0.0, 30.0]])
roi = data_types.ImagingROI(corner_roi, height=height, width=width)

# --- Processing ---
# Gets the B-scan image. Note that the algorithm only returns
# the identification key, and the result is saved in the
# "data" variable. Furthermore, the algorithm obtains the
# image at the ROI defined above.
bscan_key = bscan.bscan_kernel(data, roi)

# Gets the CPWC image.
cpwc_key = cpwc.cpwc_kernel(data, roi)

# --- Images ---
plt.figure(figsize=(16, 7))

# Shows the B-scan image.
plt.subplot(1, 3, 1)
plt.imshow(data.imaging_results[bscan_key].image, aspect='auto',
           extent=[roi.w_points[0], roi.w_points[-1], roi.h_points[-1], roi.h_points[0]])
plt.title('B-scan', fontsize=18)

# Shows the CPWC image.
plt.subplot(1, 3, 2)
plt.imshow(data.imaging_results[cpwc_key].image, aspect='auto',
           extent=[roi.w_points[0], roi.w_points[-1], roi.h_points[-1], roi.h_points[0]])
plt.title('CPWC', fontsize=18)

# Displays the result of the TFM algorithm with a normalized envelope.
plt.subplot(1, 3, 3)
plt.imshow(normalize(envelope(data.imaging_results[cpwc_key].image, -2)), aspect='auto',
           extent=[roi.w_points[0], roi.w_points[-1], roi.h_points[-1], roi.h_points[0]])
plt.title('Post-processed CPWC', fontsize=18)

plt.tight_layout()
plt.show()
```

imaging.cpwc.**cpwc_kernel**(*data_insp*, *roi=<framework.data_types.ImagingROI object>*, *output_key=None*, *description=''*, *sel_shot=0*, *c=None*, *cmed=None*, *angles=array([-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])*)

Processes A-scan data using the CPWC algorithm.



Parameters

- **data_insp** (*data_types.DataInsp*) – Inspection data, containing inspection parameters, piece parameters, transducer parameters, and the structure to save the obtained results.
- **roi** (*data_types.ImagingROI*) – Region of interest where the algorithm will be executed. The dimensions of the ROI should be in mm.
- **output_key** (None ou int) – Identification key of the processing result. The attribute *data_types.DataInsp.imaging_results* is a dictionary capable of storing multiple processing results. The key is a numeric value representing the ID of the result, while the value is the processing result itself. If output_key is None, a new random key is generated, and the result is stored in the dictionary. If int, the result is stored under the specified key, creating a new entry if the key does not exist in the dictionary or overwriting the previous results if the key already exists. By default, it is None.
- **description** (*str*) – Descriptive text for the result. By default, it is an empty string.
- **sel_shot** (*int*) – Parameter referring to the shot if the transducer has been displaced.
- **c** (*int ou float*) – Propagation velocity of the wave in the object under inspection. By default, it is None, and in this case, the value from data_insp is obtained.
- **cmed** (*int ou float*) – Propagation velocity of the wave in the coupling medium. By default, it is None, and in this case, the value from data_insp is obtained.
- **angles** (np.ndarray) – Vector with angles to execute the CPWC algorithm from FMC data. By default, it is a vector [-10, -9, ..., 10].

Returns

Identification key of the result (output_key).

Return type

int

Raises

- **TypeError** – If data_insp is not of type *data_types.DataInsp*.
- **TypeError** – If roi is not of type *data_types.ImagingROI*.
- **TypeError** – If output_key is not of type NoneType or if it cannot be converted to np.int32.
- **TypeError** – If description is not of type str or if it cannot be converted to str.
- **TypeError** – If sel_shot is not of type int or if it cannot be converted to int.

- **TypeError** – If `c` is not of type `float` or if it cannot be converted to `float`.
- **TypeError** – If `angles` is not of type `np.ndarray`.
- **NotImplementedError** – If the capture type (*data_types.InspectionParams.type_capt*) is not `PWI` or `FMC`.

imaging.cpwc.**cpwc_roi_dist**(*xr*, *zr*, *xt*, *theta*, *c*, *ts*, *tgs*)

Calculates the delays for the DAS of the CPWC algorithm.

The *delays* are converted to indices based on the sampling period. The delays are calculated according to the trajectory of the plane wave, from the transducer to the point of the ROI and back to the transducer.

Parameters

- **xr** (`np.ndarray`) – Vector with the values of x from the ROI, in meters.
- **zr** (`np.ndarray`) – Vector with the values of z from the ROI, in meters.
- **xt** (`np.ndarray`) – Vector with the values of x of the transducer elements, in meters.
- **theta** (`int`, `float`) – Angle of inclination of the plane wave, in radians.
- **c** (`int`, `float`) – Propagation velocity of the wave in the medium.
- **ts** (`int`, `float`) – Sampling period of the transducer.
- **tgs** (`int`, `float`) – Initial gate time.

Returns

A matrix of integer numbers $M_r \cdot N_r$ by N, where M_r is the number of elements in the vector x, N_r is the number of elements in the vector z, and N is the number of elements in the transducer.

Return type

`np.ndarray`

imaging.cpwc.**cpwc_roi_dist_immersion**(*xr*, *zr*, *xt*, *theta*, *c*, *cmed*, *ts*, *tgs*, *surf*)

Calculates the delays for the DAS of the CPWC algorithm.

The *delays* are converted to indices based on the sampling period. The delays are calculated according to the trajectory of the plane wave, from the transducer to the point of the ROI and back to the transducer.

Parameters

- **xr** (`np.ndarray`) – Vector with the values of x from the ROI, in meters.
- **zr** (`np.ndarray`) – Vector with the values of z from the ROI, in meters.
- **xt** (`np.ndarray`) – Vector with the values of x of the transducer elements, in meters.
- **theta** (`int`, `float`) – Angle of inclination of the plane wave, in radians.
- **c** (`int`, `float`) – Propagation velocity of the wave in the medium.
- **ts** (`int`, `float`) – Sampling period of the transducer.
- **tgs** (`int`, `float`) – Initial gate time.
- **surf** (`Surface`) – Object with information about the external surface.

Returns

A matrix of integer numbers $M_r \cdot N_r$ by N, where M_r is the number of elements in the vector x, N_r is the number of elements in the vector z, and N is the number of elements in the transducer.

Return type

`np.ndarray`

imaging.cpwc.**cpwc_sum**(*data*, *img*, *j*)

Performs the summation for the DAS of the CPWC algorithm.

Parameters

- **data** (np.ndarray) – Matrix M by N containing the acquisition data.
- **img** (np.ndarray) – Vector N_r to accumulate the data.
- **j** (np.ndarray) – Matrix with the delays for each point of the ROI. It should be a matrix $M_r \cdot N_r$ by N, where M_r is the number of elements in the vector x, N_r is the number of elements in the vector z, and N is the number of elements in the transducer.

Returns

Vetor 1 por $M_r \cdot N_r$ contendo a soma no eixo 1 da matriz.

Return type

np.ndarray

imaging.cpwc.**cpwc_params**()

Returns the CPWC algorithm settings.

Returns

Dictionary, where the key `roi` represents the region of interest used by the algorithm, the key `output_key` represents the identification key of the result, the key `description` represents the description of the result, the key `sel_shot` represents the transducer shot, and the key `c` represents the velocity of the wave propagation in the piece.

Return type

dict

3.12 Pacote surface

In the inspection of objects with arbitrary surfaces, flexible array transducers [HDW10, MHG09, TNCD08] can be used, or immersion testing can be performed, where the medium in which the system is immersed is responsible for the acoustic coupling between the transducer and the object. The tests within the scope of this project will be carried out using a robotic arm, without a human operator on site, which complicates the accommodation of flexible array systems. Additionally, the tests will be conducted in a submarine environment, making the immersion technique a natural choice.

In an immersion test, to access the internal surface of the inspected object, it is necessary to know its external surface, since the sound waves traveling from the transducer to the internal surface and back undergo refraction and attenuation at the external surface interface, as shown in Fig. **??** (a). Refraction at the interfaces follows Snell's law.

$$\frac{\text{sen}(\theta_2)}{\text{sen}(\theta_2)} = \frac{c_2}{c_1},$$

as shown in Fig. **??** (b), where θ_1 and θ_2 are the angles from the surface normal and c_1 and c_2 are the sound velocities in media 1 and 2, respectively.

3.12.1 Calculation of delays using Snell's Law and Fermat's Principle.

In delay-and-sum algorithms like SAFT and TFM, it is crucial to know the propagation time of a sound pulse between a specific transducer element A and a particular position F in the ROI, as the time difference between various element/pixel combinations is compensated through the definition of focal laws [SJ15]. In contact tests, typically only one medium is considered (the material itself) with a single velocity, so the propagation time between transducer element A and position F in the ROI is given by the Euclidean distance between them, divided by the sound velocity in the medium. However, in immersion tests, two media must be considered: the coupling medium

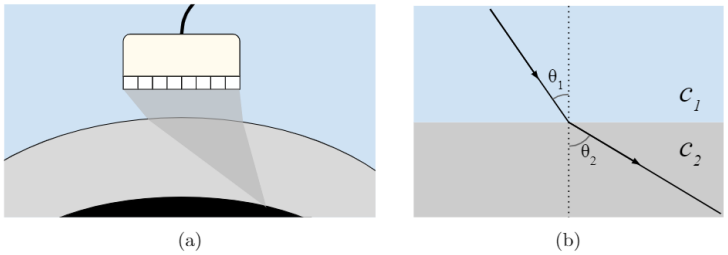


Fig. 3.19: (a) Representação da refração sofrida na superfície externa pelas ondas sonoras que se deslocam do transdutor até a superfície interna do objeto inspecionado e de volta ao transdutor. Conhecer a superfície externa é necessário para a correta geração dos pulsos e interpretação dos dados de eco. (b) A refração nas interfaces entre dois meios se dá segundo a Lei de Snell.

(e.g., water) with velocity c_1 and the material with velocity c_2. In this case, the elapsed time on the path of a pulse from A to F or vice versa is given by

$$T_{AF} = \frac{1}{c_1}\sqrt{(x_A - x_S)^2 + (z_A - z_S)^2} + \frac{1}{c_2}\sqrt{(x_F - x_S)^2 + (z_F - z_S)^2},$$

where (x_A, z_A) are the (two-dimensional) coordinates of transducer element A, (x_F, z_F) are the coordinates of position F in the ROI, (x_S,z_S) are the coordinates of the point where the pulse reaches the surface and is refracted, c_1 is the speed of sound in the coupling medium, and c_2 is the speed of sound in the material.

The entry point (x_S, z_S) is where Snell's law is respected. Fig. **??** shows an example where the trajectory of a pulse from an element A to a position F within the material must be determined. Several candidate trajectories are shown, with only one being the true trajectory. According to Fermat's principle, the trajectory that respects Snell's law is also the fastest trajectory [Sch98, Sch04]. Therefore, the problem of defining the entry point on the surface corresponds to determining the point (x_S, z_S) belonging to the surface that minimizes the time T_{AF} in Eq. (**??**), that is,

$$(\hat{x}_S, \hat{z}_S) = \underset{(x_S, z_S)}{\arg\min} \frac{1}{c_1}\sqrt{(x_A - x_S)^2 + (z_A - z_S)^2} + \frac{1}{c_2}\sqrt{(x_F - x_S)^2 + (z_F - z_S)^2}. \tag{3.2}$$

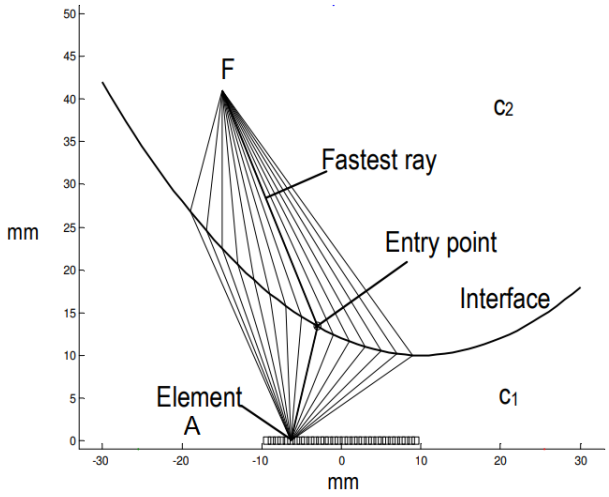


Fig. 3.20: Application of Fermat's principle with $c_2 > c_1$. (Source: [PIbanezCF07])

3.12.2 Modules and examples

The *surface* package contains Python implementations of algorithms that perform the following tasks:

- Identification of the material surface (interface with water) based on pulse-echo or FMC data.
- Calculation of trajectories and travel times between each transducer element and each pixel defined in the ROI.

The `surface` package is essentially used through the methods of the *`surface.surface.Surface`* class.

The `surface.nonlinearopt.NewtonMultivariate` class is used by the *`surface.surface.Surface`* class to implement surface parameter search algorithms based on the Multivariate Newton-Raphson method and can also be used independently.

3.13 Module surface

The *`surface.surface.Surface`* class is the main class of the module and performs the functions of identifying the surface and providing the T_{AF} times for a set of transducer elements and a set of ROI points. From the point of view of using the class (in the SAFT and TFM algorithms), the process is transparent, requiring only the acquisition data, the coordinates of the elements and the ROI, and the speed of sound in the coupling medium.

3.13.1 Example

The *script* below demonstrates the use of the Surface class to calculate the transit times between the central element of an array transducer and all points within a specific region of interest. This usage is typically employed by image reconstruction algorithms based on time of flight, such as SAFT and TFM.

```
from surface.surface import Surface, SurfaceType
from framework import file_civa
import matplotlib.pyplot as plt
import numpy as np
from framework.data_types import ImagingROI

# --- Dados ---
# Carrega os dados de inspeção do arquivo de simulação do CIVA. A simulação
# considera um bloco de aço-carbono distante 15 mm da superfície do transdutor.
data = file_civa.read('block_sdh_immersion_close.civa')

# --- Surface ---
# Instancia um objeto Surface a partir dos dados. Na chamada
# do construtor Surface(), a superfície é identificada a partir
# dos dados e pode ser utilizada daqui em diante.
mySurf = Surface(data, 0, c_medium=1498, keep_data_insp=False)

# --- ROI ---
# Define uma ROI de 40 mm x 40 mm.
heigth = 40.0
width = 40.0
h_len = 100
w_len = 64
corner_roi = np.array([-20.0, 0.0, 0.0])[np.newaxis, :]
roi = ImagingROI(corner_roi, height=heigth, width=width, h_len=h_len, w_len=w_len)

# --- Distâncias ---
# Calcula as distâncias percorridas na água e no meio para cada par elemento-pixel.
# Como as distâncias são dadas em mm, multiplica-se o resultado por 1e-3 para se
# obterem os valores em m.
[dist_water, dist_material] = mySurf.cdist_medium(
                data.probe_params.elem_center, roi.get_coord()) * 1e-3

# --- Tempos de percurso ---
# Calcula o tempo de percurso entre cada pixel e o elemento central do trandutor.
# O tempo de percurso em cada meio é dado pelas distâncias em cada meio divididas
```

(continues on next page)

(continued from previous page)

```
# pelas respectivas velocidades de propagação das ondas longitudinais.
center_elem = int(data.probe_params.num_elem / 2)
travel_time_center_elem = dist_water[center_elem, :] / data.inspection_params.coupling_cl + \
              dist_material[center_elem, :] / data.specimen_params.cl


# --- Exibição dos tempos de persurso ---
# Exibe os tempos de percurso entre cada pixel e o elemento central do trandutor.
travel_time_center_elem = travel_time_center_elem.reshape(w_len, h_len)
travel_time_center_elem = travel_time_center_elem.transpose()
plt.imshow(travel_time_center_elem, aspect='auto',
           extent=[roi.w_points[0], roi.w_points[-1], roi.h_points[-1], roi.h_points[0]])
plt.colorbar()
plt.title('Tempos de percurso [s] (elemento central)', fontsize=14)

# --- Exibição da superfície ---
# A superfície encontrada pelo construtor da classe é exibida como uma linha vermelha.
plt.plot(mySurf.x_discr, mySurf.z_discr, 'r')
plt.axis([roi.w_points[0], roi.w_points[-1], roi.h_points[-1], roi.h_points[0]])

plt.show()
```

class surface.surface.**Surface**(*data_insp*, *xdczerototal=0*, *c_medium=0*, *keep_data_insp=False*, *surf_type=None*, *surf_param=None*)

Class containing information about the surface identified in the immersion test

Parameters

- **data_insp** (framework.DataInsp) – Inspection dataset.
- **xdczerototal** (*int*) – Delay (in number of samples) imposed by the transducer on received signals, regardless of the distances traveled by the pulse in the coupling medium. It is the difference between which sample contains the maximum value of an ultrasonic echo (considering the envelope) and which sample "should" contain such maximum value if the transducer imposed no delay. The default value is 0.
- **c_medium** (*float*) – Propagation velocity in the coupling medium. If not provided or the value is 0, the class uses the value contained in data_insp. The default value is 0.
- **keep_data_insp** (*bool*) – If True, retains the attribute data_insp after the constructor method execution. If False, removes the attribute data_insp after the constructor method execution to reduce memory usage. The default value is False.

Return type
None

surfacetype

Type of surface encountered, as well as the regression method used to define the corresponding parameters.

Type
surface.surface.SurfaceType

x_discr

Horizontal coordinates of the set of discretized points derived from the analytical description of the surface obtained by regression methods.

Type
numpy.array

z_discr

Vertical coordinates of the set of discretized points derived from the analytical description of the surface obtained by regression methods.

Type
numpy.array

sumsqudistbscancylinder(*x1*, *z1*, *x2*, *z2*, *r*)

Calculates the sum of squared differences between the points belonging to the outer surface of the object, as measured by the transducer, and the cylinder described by the parameters [x1, a, z1]^T, [x2, -a, z2]^T, and r. The pair of points [x1, a, z1]^T and [x2, -a, z2]^T define a line passing through the center of the cylinder.

self: surface

x1
[float] X-coordinate of vector 1 describing the line passing through the center of the cylinder.

z1
[float] Z-coordinate of vector 1 describing the line passing through the center of the cylinder.

x2
[float] X-coordinate of vector 2 describing the line passing through the center of the cylinder.

z2
[float] Z-coordinate of vector 2 describing the line passing through the center of the cylinder.

r
[float] Radius of the cylinder

float
Result of the sum of squared errors.

sumsqudistbscanplane(*coef_a*, *coef_b*, *coef_c*)

Calculates the sum of squared differences between points belonging to the outer surface of the object, as measured by the transducer, and the plane described by the parameters coef_a, coef_b, and coef_c.

self: surface

coef_a
[float] Coefficient a in : z = a*x + b*y + c

coef_b
[float] Coefficient b in : z = a*x + b*y + c

coef_c
[float] Coefficient c in : z = a*x + b*y + c

float
Result of the sum of squared errors.

fit3D(*surf_type=None*, *surf_param=None*, *shot=0*, *roi=None*, *sel_shots=None*)

Calculates the parameters related to the desired or recognized three-dimensional outer surface.

Parameters

- **surf_type** – Value for which we want to find the next higher power of 2.
- **surf_param** – Initialization parameters of the surface type
- **shot** (*int*)
- **roi** (*data_types.ImagingROI*) – Region of interest (ROI) where the algorithm will be executed. The dimensions of the ROI should be in millimeters (mm).
- **sel_shots** – Parameter referring to triggering in case the transducer has been displaced.

Return type
None

Raises

- **TypeError** – If `surf_type` is not one of the types implemented by the algorithm.
- **TypeError** – If `surf_param` is not of type *SurfaceType* or `NoneType`.
- **TypeError** – If `shot` is not of type int and cannot be converted to int.
- **TypeError** – If `roi` is not of type *data_types.ImagingROI* or `NoneType`.
- **TypeError** – If `sel_shots` is not of type int and cannot be converted to int, or `NoneType`.

`planenewtonfit()`

Calculates the parameters a, b, and c of the surface.

Through initial estimates for parameters a, b, and c, the function, through the method
Newton-Raphson tries to minimize the function of

`cdist`(*coordelem*, *coordroi*)

Calculates all distances between two-dimensional `coordelem` positions and `coordroi` positions. The method assumes that all `coordroi` positions are found inside the material so that all trajectories pass through the surface undergoing refraction according to Snell's Law. The point on the surface at which the trajectory is refracted is calculated using the discretized Newton-Raphson method [PIbanezCF07].

Parameters

- **coordelem** (`numpy.array`) – Spatial coordinates (in mm) of the transducer elements.
- **coordroi** (`numpy.array`) – Spatial coordinates of the pixels defined for the region of interest.

Returns
Array with two elements. The first element is the matrix of distances between the `coordelem` positions and the surface positions where the beams undergo refraction on the way to the `coordroi` positions. The second element is the matrix of distances between those surface positions and the `coordroi` positions. Both matrices have the number of rows equal to the number of `coordelem` positions and the number of columns equal to the number of `coordroi` positions and follow the pattern of the matrices returned by the `scipy.spatial.distance.cdist()`.

Return type
`numpy.array`

`cdist_medium`(*coordelem*, *coordroi*, *roi=None*, *sel_shot=0*)

Calculates all distances between the two-dimensional positions `coordelem` and the positions `coordroi`. The method checks, for each `coordroi` position, whether it is located inside the material or in the coupling medium.

Parameters

- **coordelem** (`numpy.array`) – Spatial coordinates (in mm) of the transducer elements.
- **coordroi** (`numpy.array`) – Spatial coordinates of the pixels defined for the region of interest.

Returns
For positions of `coordroi` located inside the material, the format of the returned data is the same as the method `surface.surface.cdist()`. For positions of `coordroi` located in the coupling medium, the corresponding value in the first matrix contains the distance from the element to the pixel, and the value in the second matrix is zero.

Return type
`numpy.array`

get_water_path()

Returns the size of the water column estimated by the estimation method that returned the smallest SSE. if the

If the surface is a line described by z = ax+b, the water column corresponds to the coefficient b. If the surface is a circle with center (x, z) and radius r, the water column corresponds to the difference z-r.

Returns
Water column in mm

Return type
int

class surface.surface.**SurfaceType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Enum class that lists the possible types of surfaces, as well as the corresponding regression methods:

- CIRCLE_MLS: Circle with parameters defined by Modified Least Squares method.
- CIRCLE_QUAD: Circle with parameters defined by Newton-Raphson one-dimensional method.
- LINE_LS: Line with parameters defined by least squares.
- LINE_OLS: Line with parameters defined by orthogonal least squares.
- CIRCLE_NEWTON: Circle with parameters defined by Newton-Raphson multivariable
- LINE_NEWTON: Line with parameters defined by Newton-Raphson multivariable.
- ARBITRARY: Non-parametric surface determined by the ARBITRARY method
- ARBITRARY: Non-parametric surface determined by the HECTOR method

3.14 Module nonlinearopt

TODO

3.15 Package parameter_estimation

This package provides methods to estimate the parameters of an inspection such as propagation speed and roughness.

3.16 Module cl_estimators.py

TODO

parameter_estimation.cl_estimators.**cl_estimator_tenenbaum**(*image: ndarray[Any, dtype[_ScalarType_co]]*) → float

Calculate the Tenenbaum gradient as a metric to image contrast. :param image: Input image. :return: Tenenbaum contrast.

parameter_estimation.cl_estimators.**cl_estimator_normalized_variance**(*image: ndarray[Any, dtype[_ScalarType_co]]*) → float

Calculate the Normalized Variance as a metric to image contrast. :param image: Input image. :return: Normalized variance.

parameter_estimation.cl_estimators.**cl_estimator_contrast**(*image: ndarray[Any, dtype[_ScalarType_co]]*) → float

Calculate the Contrast as a metric to image contrast. :param image: Input image. :return: Contrast.

parameter_estimation.cl_estimators.**gs**(*data:* DataInsp, *roi:* ImagingROI, *sel_shot: int*, *img_func: Callable[[ndarray[Any, dtype[_ScalarType_co]]], ndarray[Any, dtype[_ScalarType_co]]]*, *a: float*, *b: float*, *tol: float*, *metric_func*) → Tuple[float, dict[float, float]]

Estimates propagation speed. :param data: DataInsp object. :param roi: ROI object. :param sel_shot: Selected shot. :param img_func: Reference to imaging function. :param a: Start of interval. :param b: End of interval. :param tol: Tolerance. :param metric_func: Metric function. :return: Estimated propagation speed.

3.17 Module intsurf_estimation.py

TODO

BIBLIOGRAPHY

[ABE14] ABENDI. *Ensaios Não Destrutivos e Inspeção*. Associação Brasileira de Ensaios Não Destrutivos e Inspeção — ABENDI, 09 2014. URL: http://www.abendi.org.br/abendi/default.aspx?mn=709\T1\textbackslash{}&c=17\T1\textbackslash{}&s=\T1\textbackslash{}&friendly=.

[And11] R. Andreucci. *Ensaio por Ultrassom*. Associação Brasileira de Ensaios Não Destrutivos e Inspeção — ABENDI, São Paulo, 2011.

[BMS73] VM Baborovsky, DM Marsh, and EA Slater. Schlieren and computer studies of the interaction of ultrasound with defects. *Non-Destructive Testing*, 6(4):200–207, 1973. doi:10.1016/0029-1021(73)90033-9.

[Bov00] Alan C. Bovik, editor. *Handbook of Image and Video Processing*. Academic Press, Inc., Orlando, FL, USA, 1a. edition, 2000.

[BGH74] CB Burckhardt, P-A Grandchamp, and H Hoffmann. Methods for increasing the lateral resolution of b-scan. In *Acoustical holography*, pages 391–413. Springer, 1974. doi:10.1007/978-1-4757-0827-1_22.

[CC00] Young-Fo Chang and Chir-Cherng Chern. Frequency-wavenumber migration of ultrasonic data. *Journal of nondestructive evaluation*, 19(1):1–10, 2000. doi:10.1023/A:1006671706818.

[CCSR00] Sylvain Chatillon, Gérard Cattiaux, Marc Serre, and Olivier Roy. Ultrasonic non-destructive testing of pieces of complex geometry with a flexible phased array transducer. *Ultrasonics*, 38(1-8):131–134, 2000. doi:10.1016/S0041-624X(99)00181-X.

[CT94] Richard Y Chiao and Lewis J Thomas. Analytic evaluation of sampled aperture ultrasonic imaging techniques for nde. *Ultrasonics, Ferroelectrics and Frequency Control, IEEE Transactions on*, 41(4):484–493, 1994. doi:10.1109/58.294109.

[CGK78] P. D. Corl, P. M. Grant, and G.S. Kino. A digital synthetic focus acoustic imaging system for nde. In *1978 Ultrasonics Symposium*, 263–268. 1978. doi:10.1109/ULTSYM.1978.197042.

[DBS96] R.J. Ditchburn, S.K. Burke, and C.M. Scala. Ndt of welds: state of the art. *NDT & E International*, 29(2):111 – 117, 1996. doi:10.1016/0963-8695(96)00010-2.

[DHR86] S.R. Doctor, T.E. Hall, and L.D. Reid. Saft — the evolution of a signal processing technology for ultrasonic testing. *NDT International*, 19(3):163 – 167, 1986. doi:10.1016/0308-9126(86)90105-7.

[DS78] PA Doyle and CM Scala. Crack depth measurement by ultrasonics: a review. *Ultrasonics*, 16(4):164–170, 1978. doi:10.1016/0041-624X(78)90072-0.

[DW06] Bruce W. Drinkwater and Paul D. Wilcox. Ultrasonic arrays for non-destructive evaluation: a review. *NDT & E International*, 39(7):525 – 541, 2006. doi:10.1016/j.ndteint.2006.03.006.

[eBF88] R.E. Jonhson e B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 06 1988.

[eDCS97] M.E. Fayad e D. C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

[Fir94] D.G. Firesmith. Frameworks: the golden path to object nirvana. *Journal of Object-Oriented Programming*, 1994.

[FSF76] JR Frederick, JA Seydel, and RC Fairchild. Improved ultrasonic nondestructive testing of pressure vessels first annual report. *Nuclear Regulatory Commission Report NUREG-O007*, 1976.

[GH97] Peter T Gough and David W Hawkins. Unified framework for modern synthetic aperture imaging algorithms. *International journal of imaging systems and technology*, 8(4):343–358, 1997. doi:10.1002/(SICI)1098-1098(1997)8:4<343::AID-IMA2>3.0.CO;2-A.

[Hel03] C. Hellier. *Handbook of Nondestructive Evaluation*. McGraw-Hill handbooks. Mcgraw-hill, New York, NY, USA, 2003.

[HDW05] Caroline Holmes, Bruce W Drinkwater, and Paul D Wilcox. Post-processing of the full matrix of ultrasonic transmit–receive array data for non-destructive evaluation. *NDT & E International*, 38(8):701–711, 2005. doi:10.1016/j.ndteint.2005.04.002.

[HDW08] A. J. Hunter, B. W. Drinkwater, and P. D. Wilcox. The wavenumber algorithm for full-matrix imaging using an ultrasonic array. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 55(11):2450–2462, 2008. doi:10.1109/TUFFC.952.

[HDW10] Alan J Hunter, Bruce W Drinkwater, and Paul D Wilcox. Autofocusing ultrasonic imagery for non-destructive testing and evaluation of specimens with complicated geometries. *NDT & E International*, 43(2):78–85, 2010.

[KCBP80] G.S. Kino, D. Corl, S. Bennett, and K. Peterson. Real time synthetic aperture imaging system. In *1980 Ultrasonics Symposium*, 722 – 731. 1980. doi:10.1109/ULTSYM.1980.197494.

[Kin79] Gordon S Kino. Acoustic imaging for nondestructive evaluation. *Proceedings of the IEEE*, 67(4):510–525, 1979. doi:10.1109/PROC.1979.11280.

[Kin87] Gordon S Kino. *Acoustic waves: devices, imaging, and analog signal processing*. Volume 107. Prentice-Hall Englewood Cliffs, NJ, Englewood Cliffs, NJ, USA, 1987.

[LH11] Minghui Li and Gordon Hayward. Ultrasound nondestructive evaluation (nde) imaging with transducer arrays and adaptive processing. *Sensors*, 12(1):42–54, 2011. doi:10.3390/s120100042.

[LOS03] Fredrik Lingvall, Tomas Olofsson, and Tadeusz Stepinski. Synthetic aperture imaging using sources with finite aperture: deconvolution of the spatial impulse response. *The Journal of the Acoustical Society of America*, 114:225, 2003. doi:10.1121/1.1575746.

[MHG09] JW Mackersie, Gerald Harvey, and Anthony Gachagan. Development of an efficient conformable array structure. In *AIP Conference Proceedings*, volume 1096, 785–791. AIP, 2009.

[MartinARLMartinezG+12] C.J. Martín-Arguedas, D. Romero-Laorden, O. Martínez-Graullera, M. Pérez-López, and L. Gomez-Ullate. An ultrasonic imaging system based on a new saft approach and a gpu beamformer. *Ultrasonics, Ferroelectrics, and Frequency Control, IEEE Transactions on*, 59(7):1402–1412, 07 2012. doi:10.1109/TUFFC.2012.2341.

[Mat96] M. Mattsson. Object-oriented frameworks: a survey of methodological issues. Master's thesis, Department of Computer Science and Business Administration, University College of Karlskrona/Ronneby, 1996.

[MMLK90] K Mayer, R Marklein, KJ Langenberg, and T Kreutter. Three-dimensional imaging system based on fourier transform synthetic aperture focusing technique. *Ultrasonics*, 28(4):241–255, 1990. doi:10.1016/0041-624X(90)90091-2.

[MullerSSchafer86] W Müller, V Schmitz, and G Schäfer. Reconstruction by the synthetic aperture focussing technique (saft). *Nuclear Engineering and Design*, 94(3):393–404, 1986. doi:10.1016/0029-5493(86)90022-1.

[PIbanezCF07] M Parrilla, Alberto Ibáñez, J Camacho, and Carlos Fritsch. Fast focal law computing for non-destructive testing with phased arrays. In *International Congress on Ultrasonics*, 1–4. 2007.

[PGB07] N Portzgen, Dries Gisolf, and Gerrit Blacquiere. Inverse wave field extrapolation: a different ndi approach to imaging defects. *Ultrasonics, Ferroelectrics and Frequency Control, IEEE Transactions on*, 54(1):118–127, 2007. doi:10.1109/TUFFC.2007.217.

[Pri72] DW Prine. Synthetic aperture ultrasonic imaging. In *Proceedings of the engineering applications of holography symposium*, volume 287. 1972.

[RLS+05] R Raillon, M Lozev, R Spencer, E Kerbrat, and S Mahaut. Application of tandem techniques with contact mono-elements or phased array probes: simulation and experiments. In *AIP Conference Proceedings*, volume 760, 914. 2005. doi:10.1063/1.1916771.

[Sch98] Lester W Schmerr. *Fundamentals of ultrasonic nondestructive evaluation: a modeling approach*. Plenum Press, New York, NY, USA, 1998.

[SJ15] Lester W Schmerr Jr. *Fundamentals of ultrasonic phased arrays*. Volume 215 of Solid mechanics and its applications. Springer, 2015. doi:10.1007/978-3-319-07272-2.

[SCMuller00] V Schmitz, S Chakhlov, and W Müller. Experiences with synthetic aperture focusing technique in the field. *Ultrasonics*, 38(1):731–738, 2000. doi:10.1016/S0041-624X(99)00219-X.

[Sch04] Arthur Schuster. *An introduction to the theory of optics*. E. Arnold, 1904.

[Sey82] J Seydel. Ultrasonic synthetic-aperture focusing techniques in ndt. *Research techniques in nondestructive testing.*, 6:1–47, 1982.

[SRR62] C. W. Sherwin, J. P. Ruina, and R. D. Rawcliffe. Some early developments in synthetic aperture radar systems. *Military Electronics, IRE Transactions on*, MIL-6(2):111–115, 04 1962. doi:10.1109/IRETMIL.1962.5008415.

[SYHY12] Hsin M. Shieh, Hsin-Chun Yu, Yu-Ching Hsu, and Ruei Yu. Resolution enhancement of nondestructive testing from b-scans. *International Journal of Imaging Systems and Technology*, 22(3):185–193, 2012. doi:10.1002/ima.22021.

[SRDillhofer+12] Martin Spies, Hans Rieder, Alexander Dillhöfer, Volker Schmitz, and Wolfgang Müller. Synthetic aperture focusing and time-of-flight diffraction ultrasonic imaging—past and present. *Journal of Nondestructive Evaluation*, pages 1–14, 2012. doi:10.1007/s10921-012-0150-z.

[Ste07] Tadeusz Stepinski. An implementation of synthetic aperture focusing technique in frequency domain. *Ultrasonics, Ferroelectrics and Frequency Control, IEEE Transactions on*, 54(7):1399–1408, 2007. doi:10.1109/TUFFC.2007.400.

[Sto78] RH Stolt. Migration by fourier transform. *Geophysics*, 43(1):23–48, 1978. doi:10.1190/1.1440826.

[TT85] R.B. Thompson and D.O. Thompson. Ultrasonics in nondestructive evaluation. *Proceedings of the IEEE*, 73(12):1716–1755, 12 1985. doi:10.1109/PROC.1985.13367.

[TNCD08] G Toullelan, A Nadim, O Casula, and P Dumas. Flexible arrays for the ultrasonic inspection of complex parts. In *Review of Progress in Quantitative NDE*. 2008.

[VW09] A. Velichko and P.D. Wilcox. Reversible back-propagation imaging algorithm for postprocessing of ultrasonic array data. *Ultrasonics, Ferroelectrics, and Frequency Control, IEEE Transactions on*, 56(11):2492–2503, 11 2009. doi:10.1109/TUFFC.2009.1336.

[VW10] Alexander Velichko and Paul D Wilcox. An analytical comparison of ultrasonic array imaging algorithms. *The Journal of the Acoustical Society of America*, 127:2377, 2010. doi:10.1121/1.3308470.

[vBMS93] L von Bernus, F Mohr, and T Schmeidl. Sizing and characterization of ultrasonic indications using imaging techniques. *Nuclear engineering and design*, 144(1):177–198, 1993. doi:10.1016/0029-5493(93)90019-6.

[Turin60] G. Turin. An introduction to matched filters. *IRE Transactions on Information Theory*, 6(3):311–329, 6 1960. doi:10.1109/TIT.1960.1057571.

PYTHON MODULE INDEX

INDEX