Published in Image Processing On Line on 2025–12–27.
Submitted on 2023–05–14, accepted on 2025–06–09.
ISSN 2105–1232 © 2025 IPOL & the authors CC–BY–NC–SA
This article is available online with supplementary materials,
software, datasets and online demo at
https://doi.org/10.5201/ipol.2025.485

CS-TRD: a Cross-Section Tree Ring Detection Method

Henry Marichal[1], Diego Passarella[2] and Gregory Randall[1]

[1]Instituto de Ingeniería Eléctrica, Facultad de Ingeniería, Universidad de la República, Uruguay(henry.marichal@fing.edu.uy / randall@fing.edu.uy)
[2]Sede Tacuarembó, CENUR Noreste, Universidad de la República, Uruguay(diego.passarella@cut.edu.uy)

Communicated by Gabriele Facciolo and Miguel Colom *Demo edited by* Henry Marichal and Cyril Voisard

Abstract

This work describes a Tree Ring Detection method for complete Cross-Sections of Trees (CS-TRD) that detects, processes, and connects edges corresponding to the tree's growth rings. The method relies on edge detection, and its parameters are set to default values and can be adjusted as needed. The only required input is the location of the biological center of the tree, the pith, which can be marked manually or using an automatic detection algorithm. CS-TRD achieves an F-Score of 91% in the UruDendro dataset (of Pinus taeda) and 97% in the Kennel dataset (of Abies alba) without specialized hardware requirements.

Source Code

A Python 3.11 implementation of CS-TRD is available on the web page of this article[1]. Usage instructions are included in the `README.md` file of the archive. The associated online demo is accessible through the website.

Keywords: image edge detection; dendrometry; tree ring detection

1 Introduction

Most of the available methods for dendrometry (the measurement of tree growth rings) use images taken from cores (small cylinders that cross all the tree growth rings) instead of complete transverse cross-sections. Figure 1 illustrates some core images. Using cores for analysis presents several advantages, as the rings are measured on a small portion of the trunk, which can be assumed to be a sequence of bands with repetitive contrast, simplifying the image analysis. However, this method provides limited information on annual tree growth because it results in a circular approximation based on a rectangular section. In many cases, ring growth is not uniform along different orientations, leading to significant errors. Some applications require the study of the entire cross-section, as in the case of examining the angular homogeneity of the ring-tree pattern to detect the so-called

[1]https://doi.org/10.5201/ipol.2025.485

Figure 1: Examples of core tree-ring images taken from a dataset with 239 images [6].

compression wood [4]. This is necessary because the lack of homogeneity in the growing pattern produces differential mechanical properties. On the downside, cross-section analysis implies the felling of the tree and includes the challenge of generating a pattern of (almost) concentric closed curves representing the tree rings. As Figure 2 shows, several factors increase the difficulty of the task. Among them are wood knots, fungi appearing as black spots with shapes following radial directions, and cracks that can be very wide.

This article presents a well-grounded method for automatically delineating tree rings on cross-section RGB images. The approach leverages the tree's cross-section structure and redundant information on a radial profile for different angles around the tree's pith.

The method is evaluated using two public datasets with different species and outperforms the other publicly available comparable method [8] in terms of accuracy.

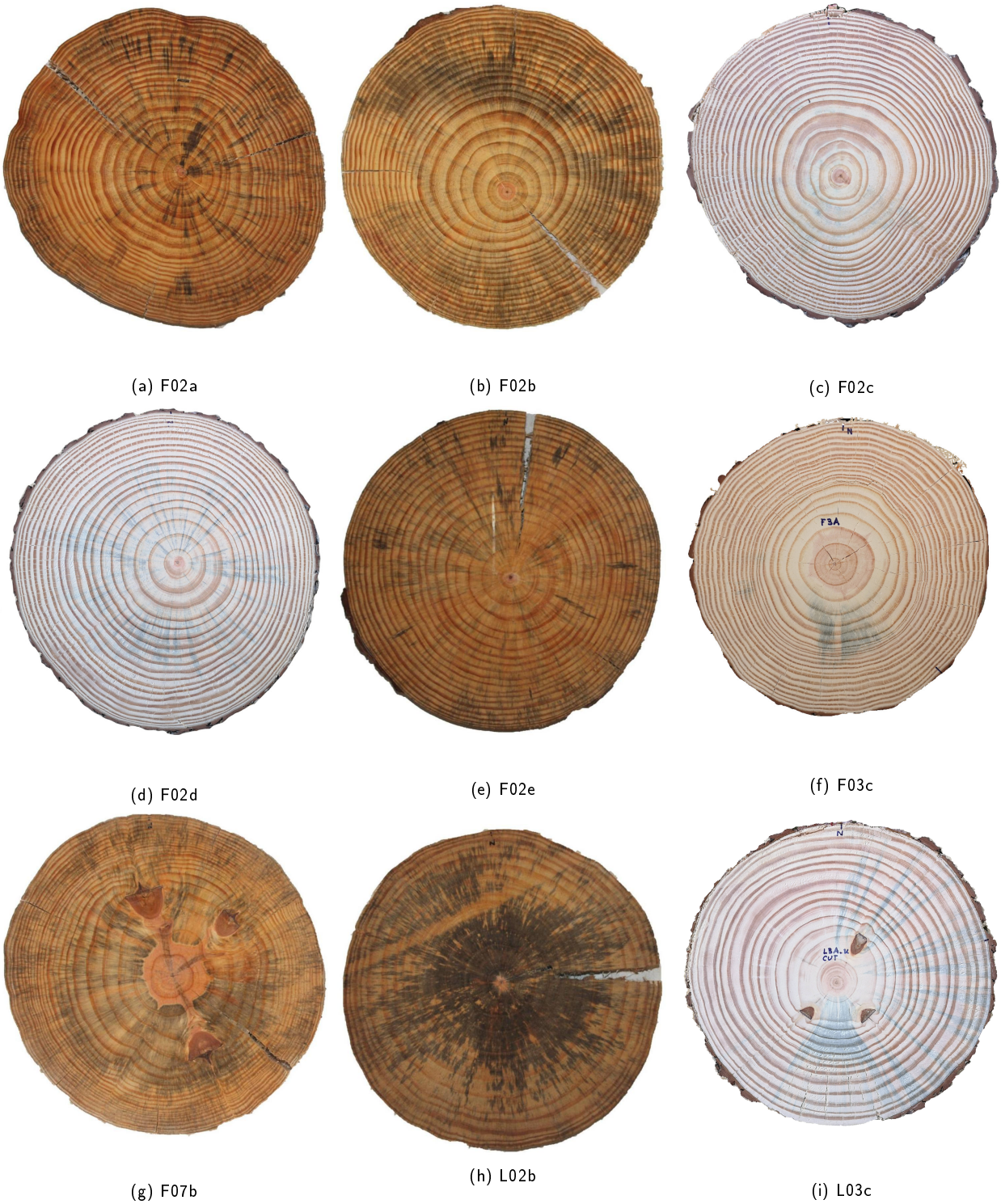
Section 2 briefly reviews previous work in the field. Section 3 describes the proposed automatic cross-section tree-ring detection algorithm (CS-TRD). Section 4 presents in more detail the algorithms, while Section 5 discusses implementation specifics. Section 6 presents the datasets used for developing and testing the proposed approach and discusses some experimental results. Section 7 concludes and propose future work. The supplementary material section includes additional experiments to demonstrate the method's relevance.

2 Previous Work

Tree ring detection is an old and essential problem in forestry. It has multiple applications in dendrometry, dendrochronology, ecology, forest management, and other fields. Due to species-specific idiosyncrasies, many practitioners still employ a manual approach, using a ruler or other manual tree-ring measurement systems. This is a tedious and time-consuming task that requires an expert operator.

Among the strategies proposed in recent years for the automation of tree ring delineation over wood cross-section images, we can mention the classic image-processing approaches that try to detect borders and then reconstruct the ring pattern [1, 16, 19, 10, 11, 12] and the algorithms based on deep learning methods which try to learn the solution from the data [8]. Most methods are designed to work with cores, for which a significantly greater number of datasets are available. Hence, the deep learning methods are generally designed for core dendrometry [5, 17].

Cerda et al. [1] proposed a classic image processing approach for detecting entire growth rings based on the Generalized Hough Transform. This work already suggests using the general geometrical structure of the tree rings, which we use in our approach, as illustrated in Figure 3. Norell et al. [16] use the Grey Weighted Polar Distance Transform to process end faces acquired in sawmill environments. Still, the method implies using rectangular sections, including the pith, and avoiding knots or other disturbances, diminishing the generality of the approach. Zhou et al. propose a much simpler method [19], which resembles the traditional manual procedure where two perpendicular lines across the slice are traced. The watershed method is applied to the profiles to obtain the peaks corresponding to each ring. Henkel et al. employ an active contours approach [10] alone or in conjunction with a Dual-Tree Complex Wavelet Transform [11], based on the evolution of a partial differential equation that incorporates terms related to both the image content and the curve itself. Many PDE-based algorithms are computationally intensive, making them challenging to implement



(a) F02a (b) F02b (c) F02c

(d) F02d (e) F02e (f) F03c

(g) F07b (h) L02b (i) L03c

Figure 2: Some examples of images from the UruDendro dataset. Note the variability of the images and the presence of fungus (L02b), knots (F07b and L03c), and cracks (F02a, F02e, and L02b). The first five images are from the same tree at different heights, as explained in Section 6.1.

in real-time applications. Makela et al. [12] proposed a method based on Jacobi Sets to detect the ring pattern and pith location. All these methods rely on detecting the edges corresponding to the tree rings and different strategies to reconstruct the pattern. In all cases, the pith is the center of a general structure. Most of these works were tested against a small number of images (ranging from 2 to 20). Unfortunately, the code and used images are unavailable in these cases for testing and comparing the results with other methodologies.

Deep learning approaches have become more prevalent in recent years and have naturally been applied to this problem. Still, the scarcity of labeled data for a given species is a significant problem in the area, as the methods must be tailored to the particularities of each species. Gillert et al. [8] proposed a method for cross-section tree-ring detection, named Iterative Next Boundary Detection Network (INBD), which was applied to high-resolution images of shrub cross-sections specially prepared for microscopy observation. In this case, the images are not only of a particular species, but also the microscopic resolution introduces specific characteristics. Starting from the pith, the method infers the annual ring at each iteration step, detecting rings one by one from the medulla to the tree's bark. A problem can arise if an intermediate ring is badly processed. In those cases, the error propagates and affects all the rings outward. The method was trained and tested on images obtained under the specific conditions of microscopic photos of shrub species. This is a unique method that allows us to access the code, allowing us to compare it with our approach. In Section 6.5, we illustrate the results of applying their model to our datasets, training their method for our particular images.

Besides the INBD method, most deep-learning-based approaches are applied to core images. For example, recently, Polek et al.[17] applied a deep learning approach to process cores of coniferous species. Fabijańska et al. proposed both a classic image-processing approach [6] (based on linking image gradient peak-detected pixels) and a convolutional neural network approach [5] for detecting tree rings in core images. Comparing both methods, they reported a precision of 43% and a recall of 51% for the classical approach and a precision of 97%, and a recall of 96% for the deep learning one. Without the code or the data, it is not possible to verify these claims with other species or datasets, as their experiments concern three ring-porous wood species.

In short, existing deep learning-based methods are almost all for cores, and the absence of labeled databases makes it difficult to use them on complete slices. Classical methods are generally based on edge detection and construct rings from them. Most reported methods lack the code or the data to verify their claims. Still, their analysis suggests that the presence of perturbations, such as fungus, cracks, and knots, significantly affects the performance because the construction of each ring depends on the previous ones. Therefore, an error in the pith or the rings closest to the center propagates to the rest of the structure.

3 Proposed Approach

In this section, we present the main ideas of the method for tree-ring delineation over RGB cross-section images, referred to as CS-TRD, the Cross-Section Tree-Ring Detector.

3.1 Assumptions

Our tree-ring detection algorithm is heavily based on the structural characteristics of the problem:

- The use of the whole horizontal cross-section of a tree (slice) instead of a wood dowel (or core), as most dendrometry approaches do.

- The following properties generally define the rings on a slice:

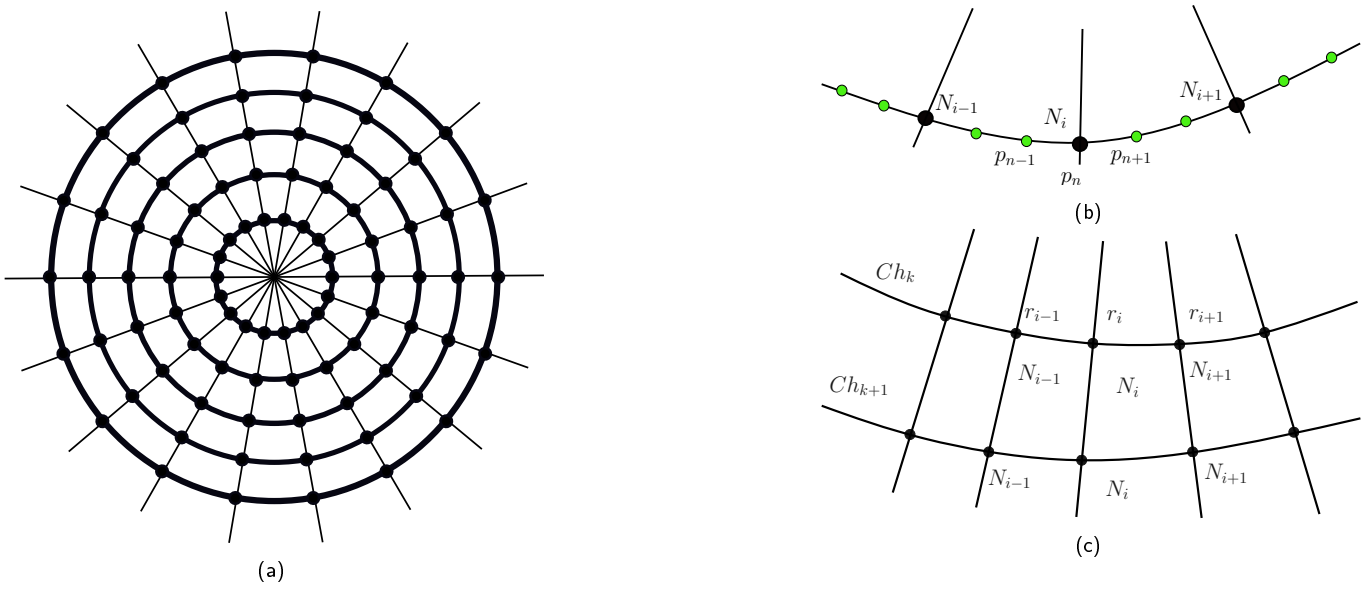


Figure 3: Cerda et al. [1] proposed a general geometrical structure of the tree rings, which we use in our approach. (a) The whole structure, called *spider web*, is formed by a *center* (which corresponds to the slice pith), N_r *rays* (in the drawing $N_r = 18$) and the *rings* (concentric curves). In the scheme, the *rings* are circles, but in practice, they can be (strongly) deformed as long as they don't intersect another *ring*. Each ray intersects a ring only once at a point called *node*. (b) A curve is a set of connected *points* (small green dots). Some of those *points* are the intersection with *rays*, named *nodes* (black dots). A chain is a set of connected *nodes*. In this case, the *node* N_i is the *point* p_n. (c) Each *Chain* Ch_k and Ch_{k+1}, intersects the *rays* r_{i-1}, r_i and r_{i+1} in *nodes* N_{i-1}, N_i and N_{i+1}.

1. The rings are roughly concentric, even if their shape is irregular. This means that two rings cannot cross.
2. Several rays can be traced outwards from the slice pith. Those rays will cross each ring only once.
3. We are interested only in the rings corresponding to the latewood (darker wood) to earlywood (lighter wood) transitions, namely the *annual rings*.

The principal idea of the method proposed in this work is the definition and use of a general structure formed by the rings, as explained in Section 3.2.

3.2 Definitions

To explain the approach, we need to define some key terms; see Figure 3. We call *spider web* the global structure of the tree rings we are searching for, depicted in Figure 3a. It comprises a *center*, associated with the slice's pith, the origin of several *rays*. The *rings* are concentric and closed curves that do not cross each other. Each *ring* is formed by a *curve* of connected points. Each *ray* crosses a *curve* only once. The *rings* can be viewed as a *curve* of points with *nodes* in the intersection with the *rays*. A *chain* is a set of connected *nodes*. As shown in Figure 3b, a *curve* is a set of chained dots (small green dots in the figure, noted p_n). Depending on the position of the *curve* concerning the *center*, some of those *points* are *nodes* (larger black dots in the figure, denoted N_i hereafter). The larger the number Nr of *rays*, the better the precision of the reconstruction of the *rings*. We fix $Nr = 360$. Note that this is the ideal setting. In actual images, *rings* can disappear without forming a closed curve, the *rings* can be strongly deformed, etc. Figure 3c illustrates the nomenclature used in this paper: *Chains* Ch_k and Ch_{k+1}, intersect the *rays* r_{i-1}, r_i and r_{i+1} at *nodes* N_{i-1}, N_i and N_{i+1}. The *rays* determine a sampling of the *curves*, producing *chains*, which merge to form a *ring*.

3.3 Approach

CS-TRD takes as input an image of the disk without background and the biological center of the disk (pith) (see Figure 4b) and returns the annual rings (see Figure 4i). Figure 4 illustrates the output of each stage of the method, which are fully described in Section 4. Briefly, it works as follows: from the center (the pith), rays are traced (see Figure 3a). A Canny edge filtering is then applied to the image (see Figure 4d), and by calculating the angle between the edge normals and the rays, most edges that do not belong to the latewood to earlywood transitions are eliminated. At the end of this stage, we obtain both noise and the ring edges of interest; additionally, some edges may be missing because they were not detected (see Figure 4e). Until now, the method is similar to the one proposed by Cerda et al. [1]; the more challenging step is grouping edges to form rings.

In the next stage, the obtained edges must enforce the *spider web* structure, which describes the general properties of the annual rings. To achieve this, all edges are subsampled using the rays: for each edge, we only retain the intersections between it and the respective rays (the nodes), forming what we call chains (see Figure 4f).

The final stage involves grouping all chains that belong to the same ring by imposing the *spider web* structure through a smoothness condition (see Figures 4g, 4h and Equation (5)). Chain grouping is performed iteratively to connect chains near areas with stronger edge information. Once no more chains are connected in the current neighborhood, the method moves on to the next region with strong edge information. This iterative process, which prioritizes edge information along with the smoothness condition, is the primary contribution of our method, which we describe in detail below.

4 Algorithm

In this section, the main algorithms of the method are described. Algorithm 1 describes the CS-TRD method and Figure 4 illustrates its intermediate results. The input is an image of a tree slice. First, we subtract the background, applying a deep learning-based approach [18] using a two-level nested U-structure (U^2Net).

Given the image without a background (shown in Figure 4b), we must identify the set of pixel chains that represent the annual rings, characterized by transitions from dark to clear. We need the *spider web* center $c = (cy, cx)$ (i.e., the disk's pith) as input. This fundamental point can be manually marked or can be automatically detected [2]. Here, we assume the point is known, since both options are available on the demo.

Algorithm 1: Tree-ring detection algorithm

Input: Im_{in}, // input image
pith_location // pith's coordinates, in pixels (cy, cx)
Output: tree-rings

1 *image* ← preprocessing(Im_{in}) // resizing, grayscale conversion and image equalization
2 *edges* ← canny_devernay_edge_detector(*image*) // use Canny-Devernay detector [9]
3 *edges* ← filter_edges(*edges*, *pith_location*) // preserve edges orthogonal to the rays
4 *chains* ← sampling_edges(*edges*, *pith_location*) // re-sample edges (Algorithm 2)
5 *chains* ← merge_chains(*chains*) // merge and complete neighboring chains (Algorithm 3)
6 *rings* ← postprocessing(*chains*) // see Algorithm 7
7 **return** *rings*

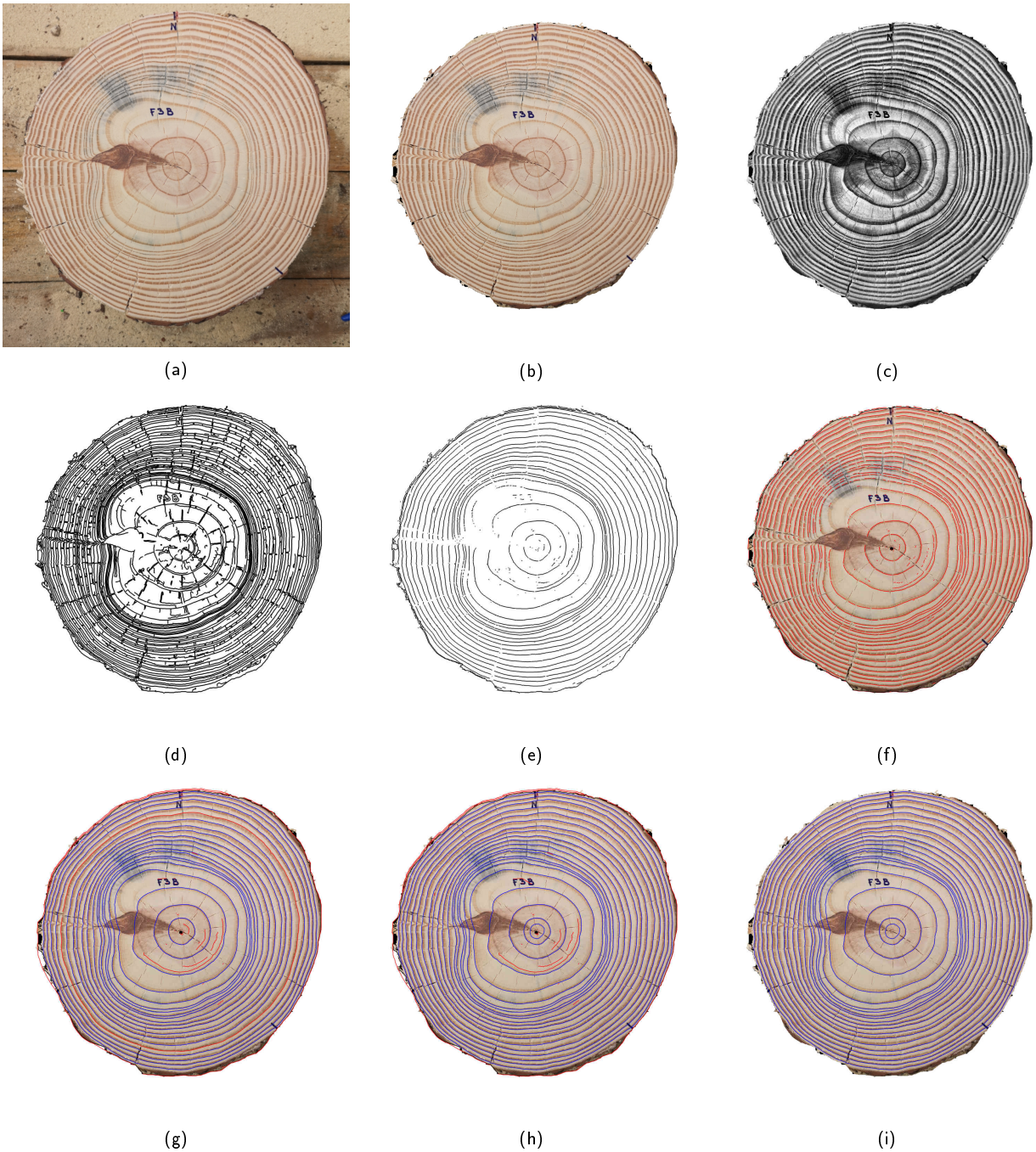


Figure 4: Principal steps of the CS-TRD algorithm: (a) Original image, (b) Background subtraction, (c) Pre-processed image (resized, equalized, grayscale conversion), (d) Canny Devernay edge detector, (e) Edges filtered by the direction of the gradient, (f) Detected chains, (g) Connected chains, (h) Post-processed chains and (i) Detected tree-rings.

4.1 Preprocessing

The first step in Algorithm 1 is to preprocess the input image to increase the method's performance. We resize the image to a fixed 1500×1500 pixels size via Lanczos interpolation [7], then convert it to grayscale, and a histogram equalization is applied (Figure 4c).

4.2 Canny-Devernay Edge Detector

Line 2 of Algorithm 1 corresponds to the edge detection stage. We apply the sub-pixel precision Canny Devernay edge detector [3, 9]. The output of this step is a list of pixel chains corresponding to the image edges. Besides some noise-derived ones, we can group those edges into three classes:

- $Edges_T$: produced by the tree growing process. It includes the edges that form the rings. Considering a pith outward direction, they can be of two types: those produced by earlywood to latewood transitions (clear to dark) and latewood to earlywood transitions (dark to clear). We are interested in detecting the latter ones, hereon called annual rings.
- $Edges_R$: radial edges produced by cracks, fungi, or other phenomena.
- Other edges produced by wood knots and noise.

The Canny Devernay edge detector has three parameters: the standard deviation of the Gaussian kernel σ, and the gradient modulus thresholds th_{low} and th_{high}, for the hysteresis filtering of the edge points. We only adjust the parameter σ; the other two are fixed (see Section 6.3.1). Figure 4d shows the output of this stage. We slightly modified the Canny-Devernay filter implementation [9] to obtain the matrices G_x and G_y, which contain the gradient components in the x and y directions, respectively. These matrices are used in the *filter_ edges* step.

4.3 Filtering the Edge Chains

Line 3 of Algorithm 1 corresponds to the edge filtering stage. Given the center $c = (cy, cx)$ and a point p_i of an edge *curve*, the angle $\delta(\vec{cp_i}, \vec{G_{p_i}})$ between vector $\vec{cp_i}$ and gradient vector $\vec{G_{p_i}}$ at p_i is

$$\delta\left(\vec{cp_i}, \vec{G_{p_i}}\right) = \arccos\left(\frac{\vec{cp_i} \cdot \vec{G_{p_i}}}{\|\vec{cp_i}\|\|\vec{G_{p_i}}\|}\right). \tag{1}$$

We filter out all *points* p_i for which $\delta(\vec{cp_i}, \vec{G_{p_i}}) \geq \alpha$. We fix $\alpha = 30$ degrees.

Two edge groups are filtered out: the earlywood transitions of the gradients $Edges_T$ set (pointing inward), and the gradients $Edges_R$ set, roughly normal to the *rays*. The algorithm produces a list of edges, primarily the ring's edges. The disk perimeter is placed in the last position. Figure 4e shows the output of this stage.

4.4 Sampling Edges

Line 4 of Algorithm 1, is the edge sampling stage which produces the *nodes* depicted in Figure 3b. We sample each *curve* of the *edges* list, computing the intersection with the rays defined in Figure 3a. Algorithm 2 shows the method's pseudocode. It has two parameters: N_r (360 by default) and m_c, the minimum number of nodes in a *chain*. As every *chain* has two endpoints, we fix $m_c = 2$. The algorithm produces a list *chains* (of *Chain* objects). This list includes two artificial *chains*, one of type *center* with N_r nodes and the exact pith coordinates but different angular orientations, and

one corresponding to the border (disk's perimeter). These artificial chains are advantageous for the connecting chains stage, discussed in the next section, as they impose an inward and outward boundary structure (see the supplementary material for more details). The attribute *type* of the *Chain* object indicates whether it is an artificial or a standard chain.

The rays defined in Figure 3a (l_rays) are computed in line 1 of Algorithm 2. Edge curves *edges* are sampled in the function *intersections_between_rays_and_devernay_curves* (line 2). To do so, we calculate the intersection between each curve and each ray in l_rays using the *intersection* method of the *shapely* library[2]. Finally, in line 4, two artificial chains (the pith and the bark) are added to the chain list.

Algorithm 2: Sampling Edges

Input: *edges*, // list of *curves*
pith_location, // pith coordinates in pixels: center of the *spider web*
Parameters:
m_c, // minimum length of a chain
Nr // number of total rays
Output: A list *chains* where each element is a *chain*

1 $l\_rays \leftarrow$ build_rays(Nr, $pith\_location$)
2 $chains \leftarrow$ intersections_between_rays_and_devernay_curves($pith\_location$, l_rays , $edges$, m_c, Nr)
3 $pith\_chain$, $border\_chain \leftarrow$ generate_virtual_chains($pith\_location$, Nr, $chains$)
4 $chains \leftarrow chains \cup pith\_chain \cup border\_chain$
5 **return** *chains*

Figure 4f shows the output of this step. Standard chains are in red, and center and border chains are in black. Due to the sampling, there are fewer chains than edges. Figure 4e has more (noisy) curves around the pith than Figure 4f. Some small curves with sampled lengths shorter than m_c are discarded. In that sense, this parameter filters out "short" chains.

Every chain has two endpoint nodes. Endpoint A is always the furthest node clockwise, while endpoint B is the most distant node counterclockwise. Given an endpoint, a chain has two attributes: *outward* and *inward* chains. Given the corresponding *ray* of a *chain* endpoint, we find the first *chain* that intersects it going from the chain to the center along the *ray* (named *inward*) and the first *chain* that intersects that *ray* moving away from the center (named *outward*). In Figure 5, chains are superposed over the gray-level image. The ray at endpoint A is blue, and the nodes are red. The *visible* chains for the black chain at endpoint A are in orange (outward) and yellow (inward).

Given $EndPoint_j$ for the current chain Ch_j, and $EndPoint_k$ for chain Ch_k, we can define:

- **Euclidean distance**

$$d_e = \sqrt{(x_j - x_k)^2 + (y_j - y_k)^2}, \tag{2}$$

  where (x_j, y_j) and (x_k, y_k) are the Cartesian coordinates of $EndPoint_j$ and $EndPoint_k$.

- **Radial Difference distance**

$$d_{rd} = \|r_j - r_k\|, \tag{3}$$

  r_j is the Euclidean distance between $EndPoint_j$ and the pith center, and r_k is the Euclidean distance between $EndPoint_k$ and the pith center.

[2]S. Gillies, C. van der Wel, J. van den Bossche, M.W. Taves, J. Arnott, B.C. Ward and others. Shapely. https://doi.org/10.5281/zenodo.5597138. Accessed October 2023.

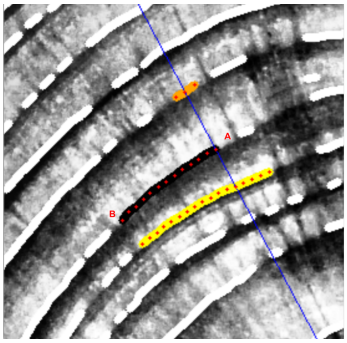


Figure 5: A given chain (in black) with two endpoints A and B. Its nodes (in red) appear at the intersection between the Canny Devernay curve and the rays. The ray at endpoint A is in blue. Other chains detected by Canny Devernay are in white. Endpoint A's inward and outward chains are in yellow and orange, respectively.

- **Angular distance**

$$d_a = (\theta_j - \theta_k + 360)\, mod\ 360, \tag{4}$$

where θ_j and θ_k are the ray's direction supporting $EndPoint_j$ and $EndPoint_k$ (both in degrees), respectively, *mod* refers to the modulus operation.

4.5 Merge Chains

We must now group the chains to form rings (Line 5 of Algorithm 1). As this section is a key part of the algorithm, we will divide it into two subsections: one that explains the general idea of merging chains and another that details the algorithms themselves.

4.5.1 General Logic of Chain Merging

The input of the algorithm is a set of chains. Some of these chains are spurious, produced by noise, small cracks, knots, etc., but most are part of the desired rings, as seen in Figure 4f.

In general, a ring is composed of multiple chains. To merge them, we must determine whether the endpoints of two given chains can be connected, as illustrated in Figure 6a. A support chain (denoted as Ch_0 in the figure) is used to decide whether the chains should be merged. In this context, we use the terms "connect chains" and "merge chains" interchangeably.

To group chains that belong to the same ring, we proceed as follows:

1. We order the chains by length and begin by processing the longest, called *Chain support*, Ch_i. Once we finish merging all the possible *candidate chains* related to that one, we do the same with the next longest *chain*, and so on. The reason for processing the longest chains first is that they contain more information about the tree ring structure, resulting in more robust results. Longer chains are more likely to be an edge belonging to an annual tree ring. Since consecutive rings have similar shapes, this iteration method propagates the shape of the longer chains.

2. We find the chains that are visible from the *Chain support* inwards (i.e., in the direction from *Chain support* to the center). Here, *visible* means that a *ray* that goes through one *candidate chain* endpoint crosses the *chain support* without crossing any other *chains* in between. The set of *candidate chains* of the *Chain support* Ch_i is named $candidates_{Ch_i}$. In Figure 7a this set is $candidates_{Ch_0} = \{Ch_1, Ch_2, Ch_4, Ch_5, Ch_6\}$. *Chain* Ch_3 is shadowed by Ch_1 and Ch_5 is not shadowed by Ch_6 because at least one of its endpoints are visible from Ch_0. The same process is applied to the chains visible from the *Chain support* outward.

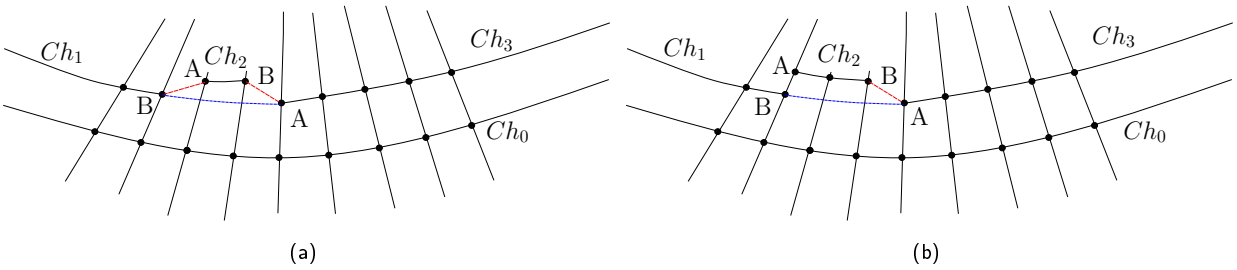


Figure 6: An illustration of the *connectivity* issue. (a) The question is if endpoint *A* of Ch_3 must be connected to endpoint *B* of Ch_2 (red line) or to endpoint *B* of Ch_1 (blue line). In Figure (b), the same question can be posed for the connection between endpoint B of Ch_1 and endpoint A of Ch_2. This connection is forbidden because Ch_1 and Ch_2 intersect (the endpoints are on the same *ray*). Note that we represent the connections by line segments for clarity, but these are curves in the image space, as we interpolate between *chain* endpoints in polar geometry. Each node is defined by an angle and radial distance from the pith (see Equation (3)). Given the endpoints of the chains to be connected, we perform a linear interpolation in polar space to link the chains. This approach ensures that the newly formed chains intersect with a radial line, allowing each connection to have a node at the radial intersection.

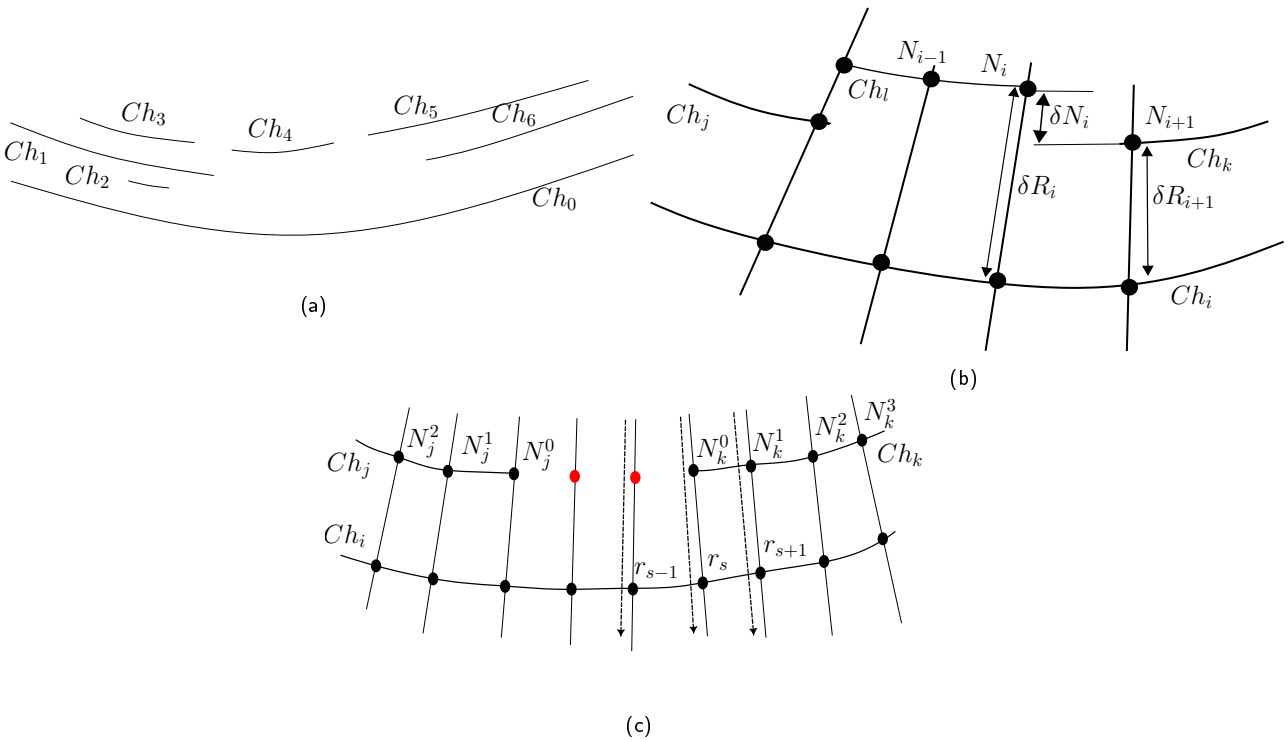


Figure 7: Connectivity nomenclature. (a) For the *chain support* Ch_0, the set of *chain candidates* is formed by Ch_1, Ch_2, Ch_4, Ch_5 and Ch_6. *Chain* Ch_3 is shadowed by Ch_1 but Ch_5 is not shadowed by Ch_6 because at least one endpoint of Ch_5 is visible from Ch_0. Note that a *chain* becomes part of the *candidate chains set* if at least one of its endpoints is visible from the *chain support*. (b) Quantities used to measure the connectivity between *chains*. δR_i is the radial difference between two successive *chains* along a *ray* r_i and δN_i is the radial difference between two successive *nodes* N_i and N_{i+1}. Note that these nodes can be part of the same *chain* or be part of two different *chains* that may be merged. Ch_i is the support chain. Ch_i visible chains are Ch_j, Ch_l and Ch_k. Chains Ch_j and Ch_k satisfy similarity conditions. (c) Chains Ch_j and Ch_k are candidates to be connected, and Ch_i is the support chain. N_j^n (N_k^n) are the nodes of Ch_j (Ch_k), with $n = 0$ for the endpoint to be connected, and r_s represents the radial distance to the pith. In red are the nodes created by an interpolation process between both endpoints.

3. We explore the set $candidates_{Ch_i}$ searching for connections between them. By construction, the *chain support* is not a candidate for merging in this step. From the endpoint of a chain, we move clockwise or counterclockwise depending on whether the endpoint is A or B. The next endpoint of a non-intersecting *chain* in $candidates_{Ch_i}$ is a candidate to be connected to the first one. We say that two *chains* intersect if there exists at least one *ray* that crosses both *chains*. For example, in Figure 7a, Ch_6 intersects with Ch_5 but not with Ch_4.

4. To decide if both chains must be connected, we must measure the *connectivity goodness* between them, combining four criteria:

   (a) *Radial tolerance for connecting chains.* The radial difference between the distance from each chain to be merged (measured at the endpoint to be connected) and the support chain must be small. For example, in Figure 7b, if we want to connect node N_i of Ch_l and node N_{i+1} of Ch_k, we must verify that

   $$\delta R_i * (1 - Th_{Radial\_tolerance}) \leq \delta R_{i+1} \leq \delta R_i * (1 + Th_{Radial\_tolerance}),$$

   where $Th_{Radial\_tolerance}$ is a parameter of the algorithm. We call this condition *RadialTol*.

   (b) *Similar radial distances of nodes in both chains.* For each chain, we define a set of nodes. For the chain Ch_j, this set is $N_j = \{N_j^0, N_j^1, \ldots, N_j^{n_{nodes}}\}$ where n_{nodes} is the number of nodes to be considered, fixed to $n_{nodes} = 20$ (see Figure 7c). We use the whole chain if it is shorter than n_{nodes}. We measure δR_i, the radial distance between each node in the given chain and the corresponding node (same ray) in the support chain, as illustrated in Figure 7b. This defines a set for each considered chain j and k: $Set_j = \{\delta R_j^0, \ldots, \delta R_j^{n_{nodes}}\}$ and $Set_k = \{\delta R_k^0, \ldots, \delta R_k^{n_{nodes}}\}$. We calculate their mean and standard deviation $Set_j(\mu_j, \sigma_j)$ and $Set_k(\mu_k, \sigma_k)$. This defines a range of radial distances associated with each chain: $Range_j = (\mu_j - Th_{Distribution\_size} * \sigma_j, \mu_j + Th_{Distribution\_size} * \sigma_j)$ and $Range_k = (\mu_k - Th_{Distribution\_size} * \sigma_k, \mu_k + Th_{Distribution\_size} * \sigma_k)$, where $Th_{Distribution\_size}$ is a parameter. There must be a non-null intersection between both distributions to connect both chains: $Range_j \cap Range_k \neq 0$. We call this condition *SimilarRadialDist*.

   (c) *Regularity of the derivative.* Given two chains Ch_j and Ch_k which can be connected, let's call Ch_{jk} the set of interpolated nodes between them (red nodes in Figure 7c). The new virtual chain created by the connection between Ch_j and Ch_k will encompass the nodes of those two chains and Ch_{jk}. The parameter *derivFromCenter* controls how are estimated the interpolated nodes between two chains, as the ones in red in Figure 7c. If $derivFromCenter = 1$, the ray angle and radial distance from the center are used to estimate the position of the interpolated nodes. If its value is 0, the estimation is made by measuring the radial distance to the support chain. To test the regularity of the derivative, we define a set of nodes for each concerned chain. For the chain Ch_j, this set is $\{N_j^0, N_j^1, \ldots, N_j^{n_{nodes}}\}$ where $n_{nodes} = 20$ is the number of nodes to be considered. If the chain is shorter, we use all nodes. We compute the centered derivative in each node for all chains, $\delta N^s = \frac{\|r_{s+1} - r_{s-1}\|}{2}$, where r_s is the radial distance of the node N^s to the center (i.e., the Euclidean distance between the node and the center). The set of the existing chains nodes is $Der(Ch_j, Ch_k) = \{\delta N_j^0, \ldots, \delta N_j^{n_{nodes}}, \delta N_k^0, \ldots, \delta N_k^{n_{nodes}}\}$. The condition *RegularDeriv* is asserted if the greatest derivative in the interpolated chain is less than or equal to the greatest derivative in the neighboring chains, with tolerance $Th_{Regular\_derivative}$

   $$max(Der(Ch_{jk})) \leq max(Der(Ch_j, Ch_k)) \times Th_{Regular\_derivative}.$$

	1	2	3	4	5	6	7	8	9
$Th_{Radial\_tolerance}$	0.1	0.2	0.1	0.2	0.1	0.2	0.1	0.2	0.2
$Th_{Distribution\_size}$	2	2	3	3	3	3	2	3	3
$Th_{Regular\_derivative}$	1.5	1.5	1.5	1.5	1.5	1.5	2	2	2
$NeighbourdhoodSize$	10	10	22	22	45	45	22	45	45
$derivFromCenter$	0	0	0	0	0	0	1	1	1

Table 1: Connectivity Parameters. Each column is the parameter set used on that iteration.

(d) *Non-Overlapping Chain.* Finally, no other chain must exist between the chains to be connected. If another chain exists in between, it must be connected to the closer one. For example, in Figure 7a, it is impossible to connect chains Ch_3 and Ch_5 because between them appears Ch_4. We call this condition *ExistChainOverlapping*.

Summarizing, in order to connect chains Ch_j and Ch_k, the following condition must be met

$$\textbf{not}\,ExistChainOverlapping \wedge RegularDeriv \wedge (SimilarRadialDist \vee RadialTol). \tag{5}$$

where $\vee$ and $\wedge$ stands for the logical *or* and *and* symbols, respectively, and the symbol **not** stands for the logical *not* operator.

The method iterates, searching for connectivity between chains over different neighborhood sizes. The parameter *NeighbourdhoodSize* defines the maximum allowed distance, measured in degrees, for connecting two chains. We iterate this process for the whole image nine times. In the first iteration, there are numerous small chains, but as the iterations progress, the relevant chains become more extensive and less noisy. As the merging process advances, we relax the parameters to connect more robust chains. Table 1 summarizes the parameters used in each iteration.

5. We proceed in the same manner in the outward direction.

4.5.2 Algorithms Description

Algorithm 3 describes the pseudocode for the *merge_ chains* method (line 5 in Algorithm 1). As previously described, the method iterates over nine parameter combinations (see Table 1), which is reflected in the for-loop from lines 1 to 16. For each parameter combination (represented by the variable *iteration*), the algorithm iterates over all chains in the list `chains`, using each one as a support chain to attempt merging with other chains, as described earlier. Once a full iteration is completed, the process continues with the next parameter set. This procedure is shown between lines 2 and 12, where Ch_i denotes the support chain in the current iteration.

Given a support chain, the first step is to identify the visible chains (line 4). These are determined in both outward and inward directions and stored in the lists `l_s_outward` and `l_s_inward`, respectively. The merging process iterates over both sets (line 5). The exploration of candidate chains, as detailed in Item 3, is performed within the for-loop from lines 6 to 11. For a given candidate chain Ch_j the algorithm seeks a chain Ch_k from the set $candidates_{Ch_i}$ to merge with. The first step (line 7) involves selecting the subset of non-intersecting chains concerning Ch_j from $candidates_{Ch_i}$; these are grouped in the list `valid_chains`. Among these, the algorithm selects the one with the smallest angular difference relative to Ch_j, within an angular neighborhood defined by the parameter `NeighbourhoodSize` (see Table 1), which corresponds to the current *iteration*. This selected chain is referred to as Ch_k. The endpoint of Ch_j to be connected is denoted as E_j.

If both chains satisfy the conditions specified in Equation (5) (line 9), they are merged (line 10). After the merge, Ch_k is removed from the `chains` list, and the updated version of Ch_j replaces the original in the list (line 11).

Once a complete iteration over `chains` for a given parameter set (*iteration*) has been performed, the algorithm exits the main while-loop (lines 3 to 12). As a final step in the *iteration*, the algorithm reviews all chains and completes those that contain more than $0.9 \times Nr$ nodes, a fixed threshold. To complete a chain, the algorithm finds the closest radial chain common to both of its endpoints (Ch_i) and calls the `close_chain` method (see Algorithm 5). Completing a chain involves adding new nodes via linear interpolation (see Section 4.5.3) until the chain reaches Nr nodes.

Algorithm 3: Merge Chains

Input: *chains*
Output: *chains*

```
1  for iteration in 1 to 9 do
2      Ch_i ← find_support_chain(chains)
3      while Ch_i do
4          l_s_outward, l_s_inward ← find_visible_chains(chains, Ch_i)
5          for candidates_Ch_i in [l_s_outward, l_s_inward] do
6              for Ch_j in candidates_Ch_i do
7                  valid_chains ← find_non_intersection(candidates_Ch_i, Ch_j)
8                  Ch_k, E_j ← find_closest(valid_chains, Ch_j, iteration)
9                  if connectivity_goodness_condition( Ch_j, Ch_k, iteration) then
                       /* Equation (5) is satisfied                              */
10                     Ch_j ← merge_two_chains(Ch_j, Ch_k, E_j, Ch_i) // see Algorithm 4
11                     chains ← update_chains_list( chains, Ch_j, Ch_k)
12         Ch_i ← find_support_chain(chains)
13     for chain in chains do
14         if chain_nodes(chain) ≥ 0.9 × Nr then
15             Ch_i ← get_common_chain_to_both_borders(chain)
16             close_chain(chain, Ch_i) // see Algorithm 5
17 return chains
```

Algorithm 4 describes the logic for connecting chains Ch_j and Ch_k, given a support chain Ch_i. Optionally, chains can also be connected using two support chains, as discussed in the following section (see Section 4.6).

First, given the endpoint E_j of chain Ch_j, the corresponding endpoint E_k of chain Ch_k is identified (line 1). Lines 2 to 6 describe the logic for generating new nodes based on one or two support chains. This procedure is detailed in Section 4.5.3. Finally, in line 8, a new chain is created by concatenating the nodes from Ch_j, the interpolated nodes, and the nodes from Ch_k.

Algorithm 5 describes the logic for completing the nodes of a chain using either one support chain (Ch_i) or two (Ch_i and *support*2). First, the endpoints of the target chain (E_A and E_B) are obtained. Then, new nodes are generated via linear interpolation, following the same procedure used in Algorithm 4. Finally, if no overlapping chain exists (see Algorithm 6), the chain is completed.

Algorithm 6 describes another key method, corresponding to the term *ExistChainOverlapping*[3] in Equation (5). This method checks whether any chains overlap with a given band (see Figure 8).

[3]The remaining terms in the equation involve straightforward calculations.

Algorithm 4: Merge Two Chains

Input: Ch_j, // current chain to be connected
Ch_k, // closest chain to be connected with Ch_j
E_j, //Ch_j endpoint to be connected
Ch_i, //support chain Ch_i
*support*2 // Optional. Second support chain.
Output: merged chain

```
E_k ← get_opposite_endpoint(Ch_j,E_j, Ch_k)
if support2 then
    interpolated ← interpolate_nodes(E_j, E_k, Ch_i. support2)
end
else
    interpolated ← interpolate_nodes(E_j, E_k, Ch_i)
end
merged ← Ch_j ∪ interpolated ∪ Ch_k
return merged
```

Algorithm 5: Close Chain

Input: *chain*, // Current chain to be connected
Ch_i, // Support chain Ch_i
*support*2, // Optional. Second support chain
Output: closed chain

```
E_A, E_B ← get_chain_endpoints(chain) //locate nodes belonging to
 non-angularly-consecutive rays
if support2 then
    interpolated ← interpolate_nodes(E_A, E_B, Ch_i. support2)
end
else
    interpolated ← interpolate_nodes(E_A, E_B, Ch_i)
end
/* See Algorithm 6                                                    */
if not exit_chain_overlapping(E_A ∪ interpolated ∪ E_B, chain, chain, Ch_i) then
    closed_chain ← chain ∪ interpolated
end
else
    closed_chain ← chain
end
return closed_chain
```

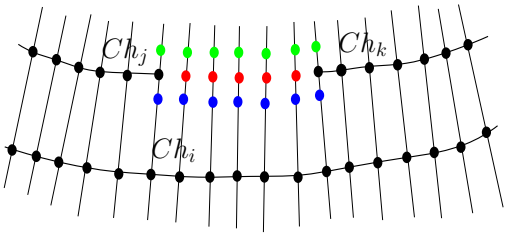


Figure 8: Red nodes are the interpolated ones between Ch_j and Ch_k chains. Blue nodes define the outer band (outward), while green nodes define the inward band. Ch_i is the support chain.

An overlapping chain is defined as any chain that has at least one node within the band that does not belong to either Ch_j or Ch_k.

The band itself is defined by a node list, *nodes*, which includes the (interpolated) red nodes as well as the endpoints of chains Ch_j and Ch_k (Figure 8). This band is constructed using the *InfoVirtualBand* class. The width of the band, denoted as $band\_width$, is defined as a percentage of the radial distance to the support chain Ch_i. If Ch_i is of type *center*, then $band\_width = 5\%$; otherwise, it is set to 10%. These values are fixed.

The algorithm iterates over *nodes*, and for each node it generates two new nodes located on the same ray but at different radial distances from the pith, as shown in Figure 8. For a given node $N_i \in nodes$ (red nodes), the radial distances of the corresponding band nodes are computed as $R\left(N_i^{green/blue}\right) \leftarrow \delta R_i * (1 \pm band\_width) + R(N_i)$ where $R(\cdot)$ denotes the radial distance to the pith, as defined in Equation (3). The nodes N_i (red), N_i^{green} and N_i^{blue} lie along the same ray. The green and blue nodes are stored in the `info_band` object.

The function `exist_chain_in_band_logic` (line 2) returns the list of chains that intersect the band defined by `info_band`. This is done by iterating over each chain and checking if any of its nodes fall between the corresponding blue and green band nodes. Chains that intersect the band are added to the list `l_chains_in_band`. If the length of this list is greater than zero, it indicates that at least one overlapping chain exists within the specified band.

Algorithm 6: Exist Chain Overlapping

Input: *nodes*, // list of interpolated nodes plus the endpoints
Ch_j, // source chain. Check Figure 8
Ch_k, // destination chain. Check Figure 8
Ch_i, // support chain
Output: Boolean. True if exists a chain belonging to *chains* in the band

1 $info\_band \leftarrow$ InfoVirtualBand(*nodes*, Ch_j, Ch_k, Ch_i)
2 $l\_chains\_in\_band \leftarrow$ exist_chain_in_band_logic($info\_band$)
3 $exist\_chain \leftarrow$ len($l\_chains\_in\_band$) > 0
4 **return** $exist\_chain$

4.5.3 Generating New Nodes

New nodes must be generated because the final chains cannot have holes. There are two situations where new nodes are generated. One case is when two chains are connected (Algorithm 4). In general,

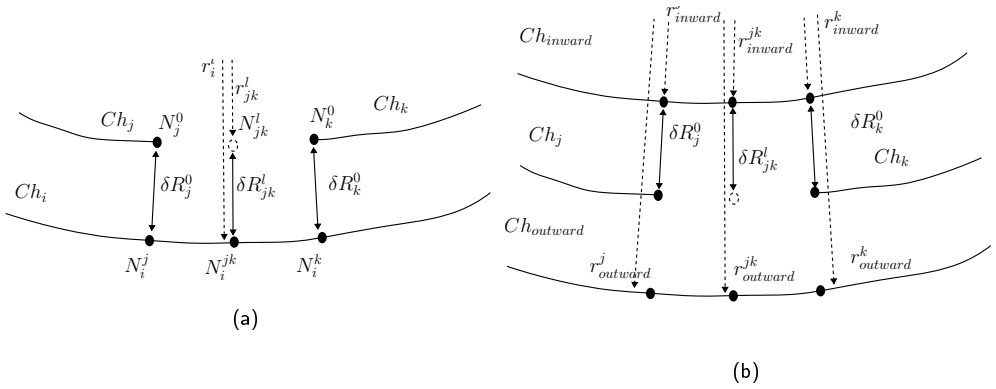


Figure 9: Nodes interpolation between chain endpoints. a) One support chain. b) Two support chains.

chains to be connected have an angular distance larger than two rays between the endpoints. A ray defines a node, which gives the direction and distance to the pith. We know the rays that lie between the endpoints that will be connected. Therefore, we linearly interpolate the position of the new nodes between the endpoints (for each ray direction, θ_i, the radial distance for the new nodes is generated). When a chain is completed, another similar situation is given (see Algorithm 5). In this case, node interpolation is made between chain endpoints. Figure 9a illustrates the situation described above. The angle of node N_{jk}^l is known, θ_i^l. The radial distance to node N_i^{jk} is r_i^l. We must compute the radial distance r_{jk}^l. As known variables we have: δR_j^0, which is the Euclidean distances between nodes N_j^0 and N_i^j , and δR_k^0 , which is the distance between nodes N_k^0 and N_i^k. Additionally, the chain endpoint angles are also known θ_j^0 and θ_k^0, and the radii. Therefore r_{jk}^l is computed as

$$\begin{cases} \delta R_{jk}^l &= sign \times \frac{\delta R_j^0 - \delta R_k^0}{d_a(\theta_j^0, \theta_k^0)} \theta_i^l \\ r_{jk}^l &= \delta R_{jk}^l + r_i^l \end{cases}, \tag{6}$$

where *sign* indicates if nodes are generated inward (-) or outward (+) from the support chain, and d_a is the angular distance defined in Equation (4).

The other case occurs when node interpolation is performed between two support chains, as in the Postprocessing step (see Section 4.6). In this case r_{jk}^l is computed as (see Figure 9b)

$$\begin{cases} \delta R_{j\_ratio}^0 &= \frac{\delta R_j^0}{r_{outward}^j - r_{inward}^j} \\ \delta R_{k\_ratio}^0 &= \frac{\delta R_k^0}{r_{outward}^j - r_{inward}^j} \\ \delta R_{jk}^l &= \left(\frac{\delta R_{j\_ratio}^0 - \delta R_{k\_ratio}^0}{d_a(\theta_j^0, \theta_k^0)} \theta_i^l + \delta R_{k\_ratio}^0 \right) \times \left(r_{outward}^{jk} - r_{inward}^{jk} \right) \\ r_{jk}^l &= \delta R_{jk}^l + r_{inward}^{jk} \end{cases}. \tag{7}$$

Finally, once the Euclidean distance to the pith (r_{jk}^l) is determined for direction θ_i^l, the node cardinal coordinates must be computed: $x_i \leftarrow x_{pith} + r_{jk}^l sin(\theta_i^l)$ and $y_i \leftarrow y_{pith} + r_{jk}^l cos(\theta_i^l)$.

4.6 Postprocessing

This final stage aims to complete the remaining chains, further relaxing the conditions. Many chains are completed at this stage (i.e., with $size = Nr$). We call them *rings*. We still have some non-closed chains that can be noisy or be part of a ring but have not been completed for some reason. We use neighborhood chains to close or discard these remaining chains. Figure 10a illustrates a typical situation, with the closed chains in blue and the non-closed ones in red. The method iterates from the

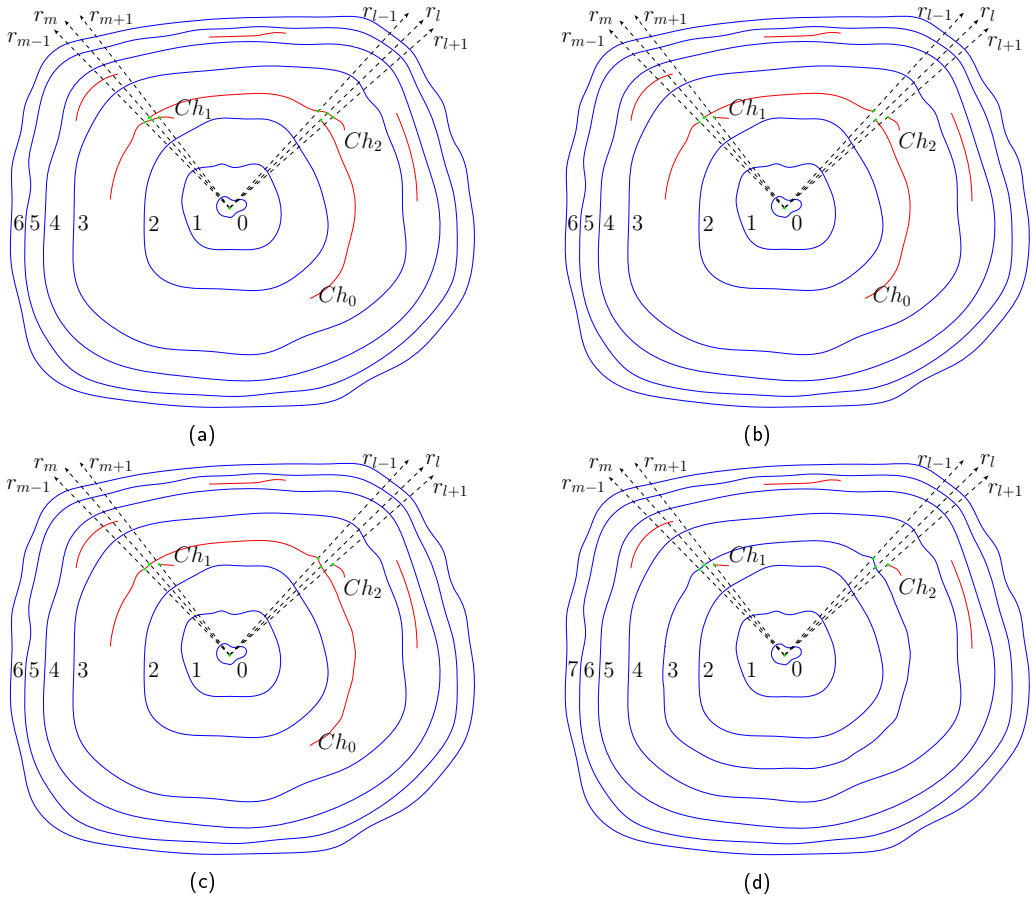


Figure 10: Postprocessing general logic. a) The input is a list of chains, with completed chains in blue and incomplete chains in red. Regions are labeled from 0 to 6. b) Chain Splitting. Chains Ch_1 and Ch_2 are split at the endpoints in rays r_m and r_l to avoid intersections. c) Chain Connection. Chains that satisfy the connectivity goodness condition are connected. d) Chain Completion and Region Addition. Chain Ch_0 is complete, with a new region added.

innermost to the outermost region. In this example, regions 0, 1, and 2 do not contain any candidate chains for connection. However, in region 3, intersecting chains, Ch_0, Ch_1, and Ch_2, are present. As shown in Figure 10b, chains are adjusted to eliminate intersections. Ch_2's endpoint intersects Ch_1 along ray r_m, leading to split Ch_1 into two subchains, with endpoints at rays r_{m-1} and r_{m+1}. A similar adjustment is applied to Ch_2, split along ray r_l. The next step consists of reconnecting chains if the connectivity goodness conditions are satisfied (Equation (5) with Table 1's last column parameters). Figure 10c illustrates the reconnection of chains.

Finally, in the last step of postprocessing, if the angular length of the chains within the region exceeds 180 degrees[4], as illustrated in Figure 10c, we consider that these incomplete chains contain sufficient information about the ring. The chains are completed in such cases, as shown in Figure 10d, by interpolating between the region's boundary rings in the positions of the existing chains (see Section 4.5.3).

Algorithm 7 describes the logic of the postprocessing stage (line 6 in Algorithm 1). In line 1, the regions are constructed. A region is defined by two complete chains (i.e., chains with a total number of nodes equal to Nr) and the set of incomplete chains between them. In line 2, the first region is extracted (the innermost one) to initiate the loop. The main loop, which iterates over the regions, spans lines 3 to 22.

In line 4, $ring1$, $ring2$, and $chains\_in\_region$, the inner and outer rings, and the chains within

[4]This parameter is estimated based on the biological structure of the disk.

the region, respectively, are extracted. From lines 5 to 16, the algorithm iterates over the internal chains in *chains_in_region*, attempting to connect them as described at the beginning of the section. In line 7, all candidate chains (*candidates*) that intersect *chain* at its endpoints are identified. In line 8, these intersecting chains are split. Then, in line 9, the subset of non-intersecting chains within *chains_in_region* is selected. In line 10, the closest chain is chosen from sets *candidates* and *non_intersecting_chains*.

If the candidate pair satisfies the connectivity conditions, the chains are merged (line 12). Unlike in Algorithm 3, in this postprocessing stage, the non-overlapping-chain condition is not required (i.e., the term **not** *ExistChainOverlapping* in Equation (5) is not enforced). In line 13, the list *chains_in_region* is updated. In line 14, if *chain* has more than $0.9 \times Nr$ nodes (a fixed threshold), it is completed.

Once all internal chains in the current region have been processed, the loop terminates. Finally, any chain with an angular span exceeding 180 degrees is considered complete. For this purpose, chains are first sorted in descending order (line 17). Then, the longest chain with more than $0.5 \times Nr$ nodes is completed (see Algorithm 5). When a chain is completed, the for-loop between lines 18 and 21 is exited, and the region is split into two subregions (line 22). The inner subregion will be processed in the next iteration. If no chain is completed, the algorithm proceeds to the next outer region relative to the current one.

4.7 Pith Detection

The pith position is an input for the method. Users can set it manually or using the method proposed by Decelle et al. [2] in the IPOL site or the method proposed by Marichal et al. [14].

5 Implementation Details

The implementation was made in Python 3.11. The `README.md` file in the code repository contains all the information needed to run the code. The demo requires, as input, an image of a tree slice and the pith position. Table 2 summarizes the method parameters, which the user can modify if needed. As output, the method returns a JSON file with the tree-ring positions in Labelme format[5]. Note that the values of parameters α, N_r, and m_c are fixed once and for all.

6 Experiments and Results

6.1 Datasets

We use two datasets to evaluate the CS-TRD method.

The first one (**UruDendro**) is an online database [13] featuring images of cross-sections from fourteen commercially grown 13- to 24-year-old *Pinus taeda* trees in northern Uruguay, collected in February 2020. The disks are between 5 and 20 cm thick and were dried at room temperature without further preparation, which resulted in the development of radial cracks and blue fungus stains. Surfaces were smoothed with a handheld planer and a rotary sander. The dataset comprises 64 images of varying resolutions, ranging from 1000 to 3000 pixels in width. It contains challenging features for automatic ring detection, including varying illumination and surface preparation, fungal infections (blue stains), knots, missing bark, interruptions in outer rings, and radial cracking. At

[5]K. Wada. Labelme: Image Polygonal Annotation with Python. https://doi.org/10.5281/zenodo.5711226

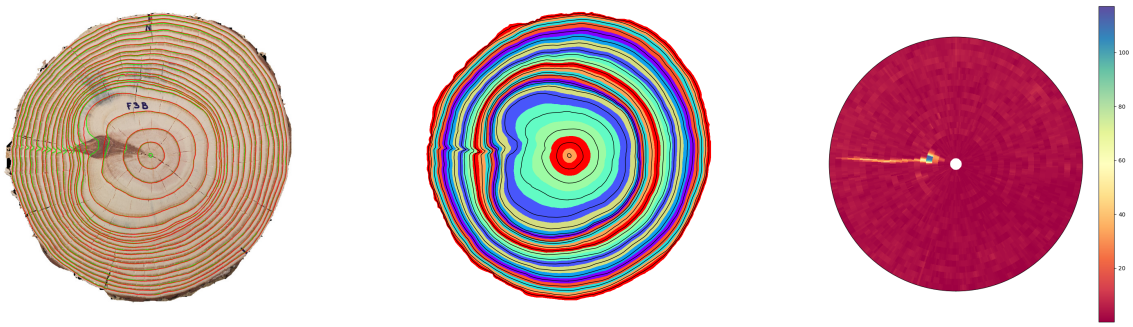
Algorithm 7: PostProcessing

Input: *chains*
Output: A list of post-processed *chains*

```
regions ← get_regions(chains)
current ← get_next_region()
while current do
    ring1, chains_in_region, ring2 ← get_region_elements(current)
    /* Merge chains in region                                          */
    chain ← get_next_chain(chains_in_region)
    while chain do
        candidates ← find_candidates_intersecting_chains(chain, chains_in_region)
        candidates ← split_chains(chain, candidates)
        non_intersecting_chains ← find_non_intersection(chains_in_region, chain)
        Ch_k, E_j ← find_closest(non_intersecting_chains + candidates, chain)
        if connectivity_goodness_condition( chain, Ch_k) then
            chain ← merge_two_chains(chain, Ch_k, E_j, ring1, ring2) // see Algorithm 4
            chains_in_region ← update_chains_list(chains_in_region,chain,Ch_k)
        if chain_nodes(chain) ≥ 0.9 × Nr then
            close_chain(chain, ring1, ring2) // see Algorithm 5
        chain ← get_next_chain(chains_in_region)
    /* Close chains with an angle domain higher than 180 degrees      */
    chains_in_region ← sort_chains(chains_in_region)
    for chain in chains_in_region do
        if chain_nodes(chain) ≥ 0.5 × Nr then
            close_chain(chain, ring1, ring2) // see Algorithm 5
            break
    current ← get_next_region(current)
return chains
```

	stage	Parameter	Description	Default
Basic	Edges detector	–sigma	Gaussian filtering σ	3
	Preprocessing	–height	Rezised image height	None
		–width	Rezised image width	None
	Filtering, sampling connect		Pith Position	Required
Advanced	Edges detector	–th_low	Gradient threshold low	5
		–th_high	Gradient threshold high	15
	Edges filtering	–alpha	Collinearity threshold (α)	30º
	Sampling	–nr	Number of rays (N_r)	360
		–min_chain_lenght	Minimum chain length (m_c)	2

Table 2: Method parameters. The user can modify basic parameters in the demo.



(a) (b) (c)

Figure 11: Measuring the error of automatic detections for image F03d: (a) In green, the GT; in red, the detections produced by the method. (b) Areas of influence of the GT rings. (c) Absolute error, in pixels, between the detections and the GT.

least two experts annotated each image sample using the Labelme tool[5]. We use the ring average between experts' annotations as ground truth. Figure 2 shows some images from the dataset.

The second dataset (**Kennel**) is proposed in [11], which made available a public set of 7 (1280 × 1280 pixels) images of *Abies alba* along with a method for detecting tree rings (the code is not available). We labeled the dataset with the same procedure as the **UruDendro** dataset, as we could not process the annotations given by the authors.

6.2 Metrics

To assess the method, we developed a metric based on the one proposed by Kennel et al. [11]. To determine if a ring is detected, we define a ring influence area as the set of pixels closer to that ring. For each ray, the frontier is the midpoint between the nodes of consecutive ground truth (GT) rings. Figure 11b shows the influence area for rings in disk F03d. Figure 11a shows the detections (in red) and GT marks (in green) for the same image.

The influence region associates a detected curve with a GT ring. In both cases, nodes are associated with the N_r rays. Given a GT ring, we compute the RMSE error (Equation (8)) for each detection and assign the GT ring to the detection (DT) with the lowest RMSE

$$RMSE(GT, DT) = \sqrt{\frac{1}{N_r} \sum_{i=0}^{N_r-1} \left(r_i^d - r_i^{GT}\right)^2}, \tag{8}$$

where r_i^d is the radial distance from the center to node i of the detected (DT) ring, and r_i^{GT} is the same for the GT ring. As a reminder, the radial distance of $node_i$ is defined as the Euclidean distance between $node_i$ and the pith (Equation (2)).

The closest detection may be far from the corresponding GT ring. To match a detected curve with a GT ring, it is essential to ensure that the identified chain is the closest one to the ring and that it is sufficiently close. We utilize the influence area of each GT ring (Figure 11b). Upon detecting a curve, if the proportion of nodes from that chain that falls within the influence area of the nearest ring exceeds a specified threshold parameter ($th\_pre = 60\%$, see Section 6.3.3), we assign the detected curve to the corresponding GT ring. If it falls below the threshold, the detection is not associated with any GT ring. In other words, for a detected curve to be assigned to a GT ring and be considered a true positive, at least 60% of its nodes must be within the influence area of that GT ring.

Dataset	Image Size (pixels)	σ	P	R	F	RMSE	Execution Time (sec.)
UruDendro	1500×1500	3.0	0,93	0,86	0,89	3.89	17.3
Kennel	1500×1500	2.5	0,97	0,97	0,97	2.4	11.1

Table 3: Mean performance and execution time for both datasets at the optimal σ and image resolution.

We define the absolute error $A\epsilon_i$ (in pixels) for the *node* i as the absolute difference in pixels between the GT ring and the detected ring associated with it

$$A\epsilon_i = |r_i^d - r_i^{GT}|, \tag{9}$$

where r_i^d is the radial distance from the center to node i of the detected ring, and r_i^{GT} is the same for the GT ring. Figure 11c shows the absolute error between the GT and the detected rings assigned to them. Red represents a low error, while yellow, green, and blue represent increasing error levels. As can be seen, the error is concentrated around the knot, affecting the precise detection of some rings.

Once all the detected chains are matched with the GT rings, we calculate the following values:

1. True Positive (TP): when the identified closed chain and the GT ring match.
2. False Positive (FP): when the identified closed chain doesn't match any GT ring.
3. False Negative (FN): when a GT ring doesn't match any detected closed chain.

Precision is given by $P = \frac{TP}{TP+FP}$, Recall by $R = \frac{TP}{TP+FN}$ and the F-Score by $F = \frac{2P \times R}{P+R}$.

6.3 Experiments

This section presents some experiments to help us better understand the method and its limitations.

Table 3 presents the results for the optimal values of σ and image resolution on both datasets. For example, CS-TRD fails to detect one ring ($FN = 1$) in sample F03d (see Figure 11a), and the other rings are correctly detected, leading to the displayed values of P, R, and F. The table also compares the mean execution time by image and the RMSE error in pixels (Equation (8)) between the detected and GT rings. All experiments were conducted on an Intel Core i5-10300H workstation with 16GB of RAM. These are very good results, considering the diversity of the data and the presence of perturbations.

6.3.1 Edge Detector Optimization Stage

The algorithm heavily relies on the edge detector stage. In the first experiment, we test different σ values for the Canny Devernay edge detector to get the one that maximizes the F-Score for the UruDendro dataset. This dataset exhibits significant variations in image resolution, enabling us to investigate the overall performance across different input image dimensions. Results are presented in Figures 12a and 12b. We compute the average F-Score for the original image sizes and then scale all images in the dataset to 640×640, 1000×1000, and 1500×1500 pixels. The best result was achieved for size 1500×1500 with $\sigma = 3.0$. Execution time varies with image size. The average execution time for this size is 17 seconds. The execution time decreases as σ increases because fewer edge chains are detected. Results for the same experiment on the Kennel dataset are shown in Figures 12c and 12d. The best F-Score is achieved for 1500×1500 size with $\sigma = 2.5$. The lower optimal σ value can be attributed to the Kennel dataset having images with more rings, averaging 30 rings per disk, while the UruDendro dataset has 19 rings per disk on average. Table 3 summarizes this experiment.

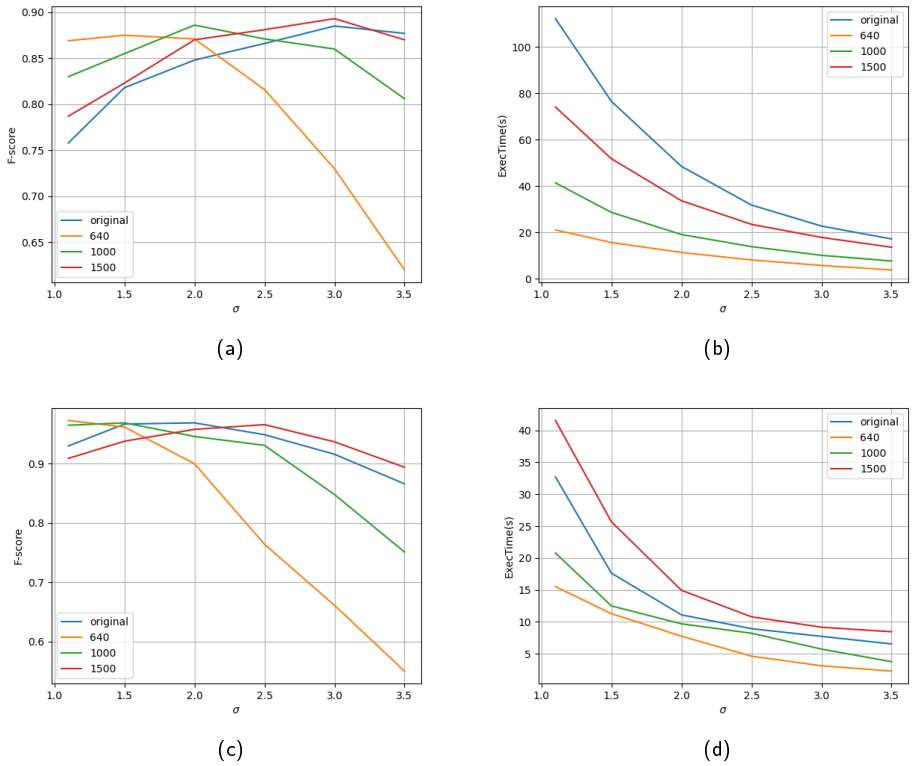


Figure 12: Influence of the image size and edge detector σ parameter experiment. Each curve represents a different image resolution: 640×640, 1000×1000, 1500×1500, and the original resolution (in blue). (a) Average F1 vs. σ for different image sizes of the UruDendro dataset. (b) Average execution time (in seconds) vs σ for different image sizes over the UruDendro dataset. (c) Average F1 vs. σ for different image sizes of the Kennel dataset. (d) Average execution time (in seconds) vs σ for different image sizes over the Kennel dataset.

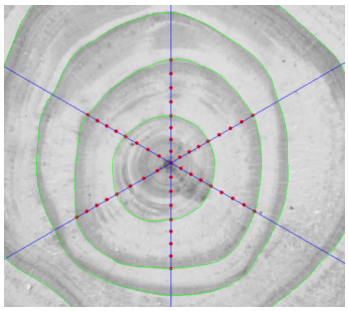
6.3.2 Pith Position Sensibility

The next experiment assesses the method's sensitivity to errors in the pith estimation. Figure 13a shows 48 different pith positions used in this experiment (eight different pith positions across six rays). These radially displaced pith positions are selected as follows:

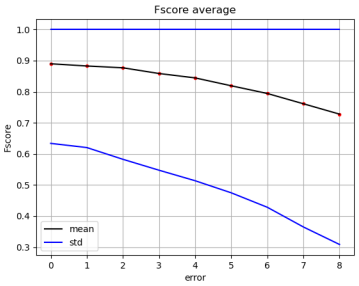
- We define an error step along a ray as 25% of the distance, along each ray, between the pith and the innermost ring.
- Three positions are marked inside ring 1, with errors 25%, 50%, and 75% off the GT center in the ray direction.
- One position is marked on ring 1.
- Three positions are marked between the first and second rings, with a 25% of the distance between both rings along each ray.
- Finally, another position is marked on ring 2.

We executed the algorithm for each disk and pith position of the UruDendro dataset (size of 1500×1500 and $\sigma = 3.0$), resulting in 48 outcomes. We calculated the average RMSE and F-Score measurements for the six-ray directions for each radially displaced pith position. This produced two eight-coordinate vectors, one for RMSE and one for the F-Score. Figures 13b and 13c illustrate the average F-score and RMSE for each error position across the dataset, respectively. The F-Score

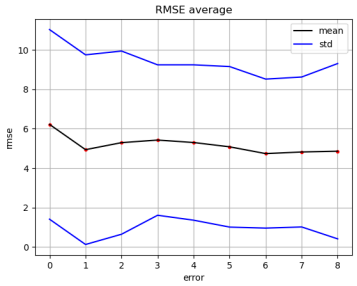
decreases as the error in the pith estimation increases, while the RMSE is less sensitive to pith position errors.



(a) Pith position grid



(b) Average F-Score over the UruDendro dataset.



(c) Average RMSE over the UruDendro dataset.

Figure 13: Pith position experiment. (a) Eight different pith positions are marked on six ray directions. We executed CS-TRD for each pith position. GT rings are in green. (b and c) For each disk of the UruDendro dataset, we run the method using the 48 different pith positions. Results are averaged over the six directions of the rays per error position. The black curve represents the mean, while the blue curve represents the standard deviation.

6.3.3 Detection-to-ground-truth Assignation Threshold

In this experiment, we examine how performance changes with different values of the th_pre parameter, which determines the number of ring nodes within the influence area to be considered in the detection-to-ground-truth assignment step. Figure 14 displays the results for the UruDendro and Kennel datasets. As anticipated, higher precision leads to lower RMSE but lower F-score. Based on these findings, we fixed $th\_pre = 60\%$ as the default value, which appears to be a good compromise.

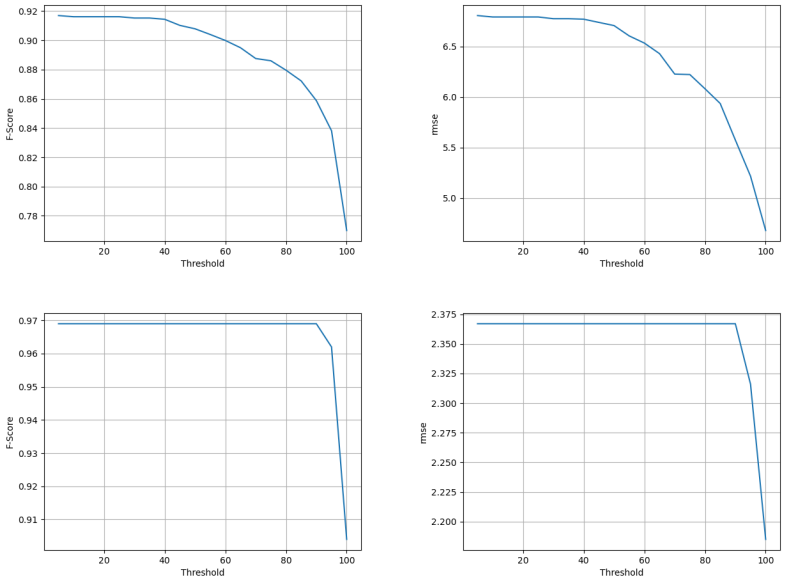


Figure 14: Performance metrics were computed for different values of the th_pre parameter. The first row displays the results for the Urudendro dataset, and the second row shows the results for the Kennel dataset. The left column shows the average F-score, and the right column shows the average RMSE.

6.4 Results

This section provides a quantitative analysis of the CS-TRD method's performance on challenging images in the datasets. Figure 15 illustrates some results of the CS-TRD over the UruDendro dataset. Note the successful performance of CS-TRD on disks with cracks (F02b), knots (L03c), and fungus (L02b). The mean F-Score for the whole dataset is 0.89 (Table 3), indicating the successful detection of rings in complex images containing knots, fungus, and cracks.

(a) F02b (b) F07b (c) L02b

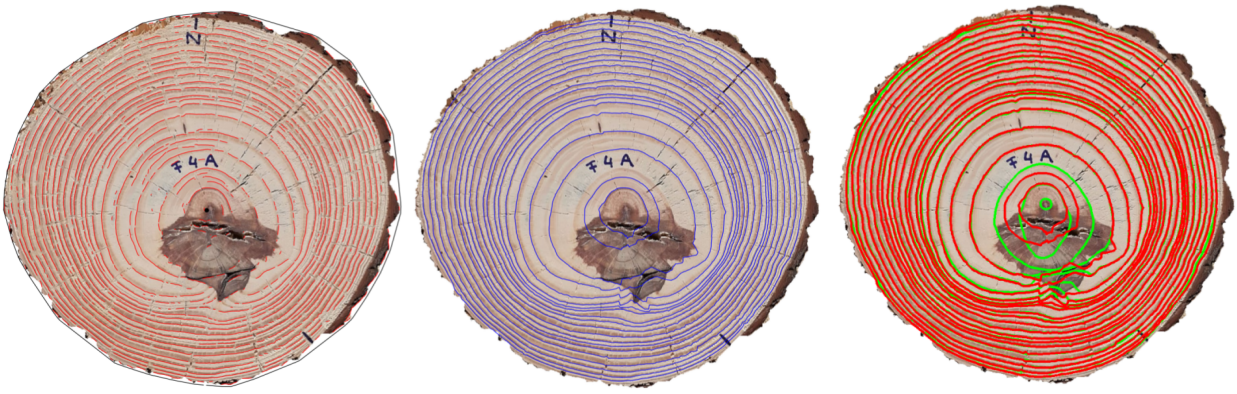
Figure 15: The CS-TRD method predicted detections (shown as blue curves) in some images of the UruDendro dataset.

The upper row of Figure 16 illustrates how the method performs in the presence of knots. For disk F04c the method fails to detect the first and third rings and identifies a false ring over the knot. Despite these errors, the method detected 19 rings, with only one false detection, and missed two rings, resulting in an F1-Score of 93%. The lower row of Figure 16 shows the results for disk L09e. Despite two significant cracks and several fungus stains, the method successfully detects 13 out of 15 rings, with one false detection. As a result, it achieves an F1-Score of 90%. The performance for each sample in the UruDendro dataset can be found in Table 4 of the supplementary material.

Some results for the Kennel dataset are shown in Figure 17. The mean F-Score is 0.97 (Table 3). At most, three rings are not detected per disk. In the worst case, one ring is mistakenly detected per image, usually the last one (sometimes incomplete) or the core. In Figure 18, we illustrate the example of disk *AbiesAlba1*. The edge parameter is set too high ($\sigma = 2.5$), causing the edge detector to fail to detect the pith. Additionally, the red chain in Figure 18c is not closed because its size is smaller than 180 degrees (fixed once and for all). However, the method effectively detects the rings over the knot.

6.5 Comparison between CS-TRD and INBD Methods

The only other method with available code for automatically detecting tree rings in wood cross-section images is the Iterative Next Boundary Detection Instance Segmentation (INBD) proposed by Gillert et al. [8]. Note that this method was designed for Microscopy Images of Shrub Cross Sections, which differ from the ones we are working on in terms of image resolution and species characteristics. The image is segmented into the background, ring boundaries, and pith region using the INBD method. Then, the circular image is transformed into polar coordinates using the pith's center as the origin. Image patches are then extracted iteratively, and rings are segmented individually from the inner to the outer rings. Both stages employ a U-NET network. The ground truth pith location is used as input in the experiments for the CS-TRD. To ensure a fair comparison, the second stage of the INBD method is modified to take the pith boundary as input.



(a) Chains (b) Output (c) Gt and Detections

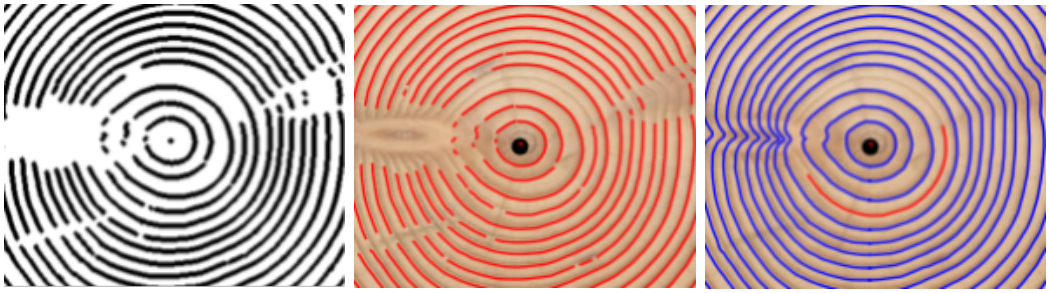
(d) Chains (e) Output (f) Gt and Detections

Figure 16: Upper row: CS-TRD result for disk F04c. Note the impact of the knot on the edge detection step. (a) chains, (b) detected rings, (c) GT rings in green and detected rings in red. Lower row: CS-TRD result for disk L09e. The method successfully detects almost all the rings (FN = 2 and FP = 0) despite the presence of cracks and fungus stains. (d) chains, (e) detected rings, (f) GT rings in green, and detected rings in red.

(a) AbiesAlba1 (b) AbiesAlba2 (c) AbiesAlba3

Figure 17: CS-TRD results for images from the Kennel dataset with 1500×1500 image size and $\sigma = 2.5$.

To train the INBD model with the UruDendro dataset, we randomly divided it into training, validation, and test sets, each containing 40, 12, and 12 images, respectively. Gillert et al. trained their model using the EH dataset, which comprises 82 images with 949 rings. Our UruDendro dataset comprises 64 images and 1123 rings. For training the INBD model, each image is divided into patches determined by successive rings in polar coordinates, ensuring that each patch includes an entire ring. Even though the EH dataset has more images, the UruDendro dataset has more rings per image.



(a) Filter (b) Chains (c) Postprocessing

Figure 18: CS-TRD result for AbiesAlba1 image from the Kennel Dataset (zoom over pith center). a) Filter stage output, b) Chain stage output, c) Postprocessing stage output. In a) and b), we can see that the method fails to detect edges for the pith. The σ threshold may be too high to detect things at this resolution. In c) we can see that the red chain was not closed due to a size smaller than the value 180 (see Algorithm 7).

Considering the total number of tree rings, both datasets are comparable.

The INBD method relies on two crucial hyperparameters: the number of iterations at each epoch (n) and the image size factor (downsample). We seek the best-performing INBD model by exploring a grid with $n \in \{1, 2, 3, 4\}$ and $downsample \in \{0, 2, 4\}$. The model was trained using the UruDendro training and validation sets. The model that exhibited the best performance on the validation set was chosen. For training, we utilized the ClusterUy infrastructure [15], equipped with an Nvidia Tesla P100 GPU with 12GB of RAM. The hyperparameters $n = 3$ and $downsample = 0$ yield the best performance on the validation set.

Table 4 compares the performance of both methods using the test set from the UruDendro dataset. When using the same CPU hardware, we observe that the INBD is faster, with an average of 7.5 seconds and 18 seconds for CS-TRD. In terms of performance, CS-TRD outperforms the INBD model in precision, recall, and F-score. When considering the RMSE metric, CS-TRD performs slightly better than INBD, with a difference of 2.7 pixels. As a reference, the mean difference between human annotators on the UruDendro dataset is around 2.5 pixels.

Method	**P** ↑	**R** ↑	**F** ↑	**RMSE (pixels)** ↓	**Time CPU (seconds)** ↓
INBD	0.75	0.84	0.79	5.7	**7.5**
CS-TRD	**0.96**	**0.90**	**0.93**	**2.09**	18

Table 4: Comparative results between INBD and CS-TRD methods over the test set of images from the UruDendro dataset. The same HW was used to compare the execution times. ↑ (↓) indicates that higher (lower) values are better.

Figure 19 illustrates a qualitative comparison between both methods for two disks of the UruDendro dataset. Subfigures a, c, e, and g superpose the red detected and green ground truth rings. Subfigures b, d, f, and h depict the absolute radial error between the automatic detections and the GT disks. The upper row shows the results for disk F03b. CS-TRD produced 20 True Positives, 1 False Positive, and 3 False Negatives, compared to 19 True Positives, 3 False Positives, and 4 False Negatives for INBD. The INBD method is iterative, and an error produced in a specific ring is propagated to the following rings outward, as seen in subfigure d. This significantly increases the RMSE error (9.2 for the INBD vs. 1.1 for the CS-TRD). The lower row of Figure 19 illustrates the results for a disk with a high amount of fungus stains (L02b), for which INBD produces numerous false detections (20).

6.6 Supplementary Material

In the supplementary material, additional experiments are conducted to justify the decisions made during the algorithm design. We demonstrate how the method performs if the artificial chains are

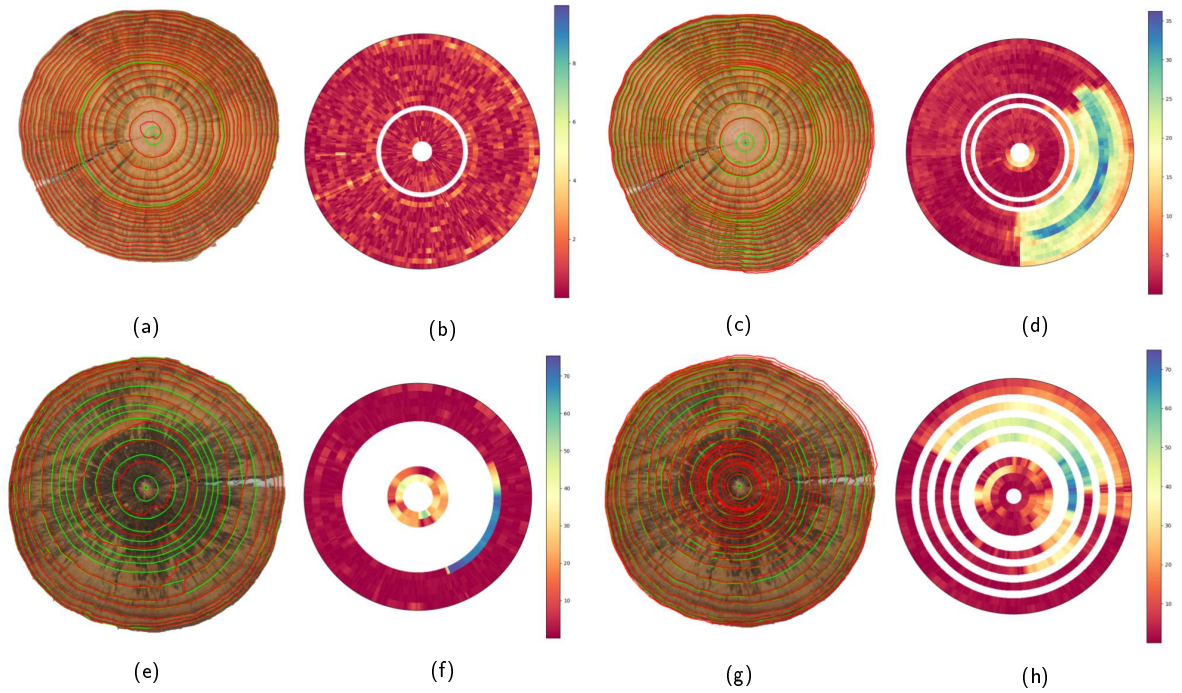


Figure 19: Upper row, results of CS-TRD (left) and INBD (right) methods for disk F03b of UruDendro. In (a) and (c), ring detections are shown in red, and GT rings in green. In (b) and (d), the absolute radial error between ring detections and GT rings is displayed. Red indicates a low error, and blue indicates a high error. A white band indicates that no detection was made for a given ground truth ring. Results of the CS-TRD (left) and INBD (right) methods for disk L02b of UruDendro are shown in the lower row. In (e) and (g), ring detections are shown in red, and GT rings in green. In (f) and (h), the absolute radial error between the ring detections and the GT rings is displayed (red indicates a low error, and blue indicates a high error). A white band indicates that no detection was made for a given ground truth ring.

not included, modifying Equation (5), and at different iterations in Table 1. Additionally, more illustrative examples are provided for the UruDendro and Kennel datasets.

7 Conclusions and Future Work

We presented an automatic method for detecting tree rings in cross-section wood images. It achieves an F-Score of 97% in the Kennel dataset and 91% in the more complex UruDendro dataset. The CS-TRD method outperforms state-of-the-art deep learning methods, such as INBD [8], in the Pinus taeda species. It performs well even in the presence of fungus, cracks, and knots, and in two different species (*Abies alba* and *Pinus taeda*). The method achieves an average execution time of 17 seconds on the UruDendro dataset and 11 seconds on the Kennel dataset, using an Intel Core i5-10300H workstation with 16 GB of RAM (without GPU). Compared to the time each annotator needs to manually delineate every disk, which is 3 hours on average, this is a vast improvement. CS-TRD can be fully implemented in C++ to accelerate the execution time compared to the Python implementation[6]. This will allow using the method in real-time applications. In the future, we plan to include the automatic pith detection, extend the method to other tree species, and explore machine-learning techniques to improve the results.

[6]https://medium.com/agents-and-robots/the-bitter-truth-python-3-11-vs-cython-vs-c-performance-for-simulations-babc85cdfef5

Image Credits

 Images from the UruDendro dataset.

 Images taken from [6]

 Images from the Kennel dataset.

References

[1] M. Cerda, N. Hitschfeld-Kahler, and D. Mery, *Robust Tree-Ring Detection*, in Advances in Image and Video Technology, Second Pacific Rim Symposium, (PSIVT), D. Mery and L. Rueda, eds., vol. 4872 of Lecture Notes in Computer Science, Springer, 2007, pp. 575–585, https://doi.org/10.1007/978-3-540-77129-6_50.

[2] R. Decelle, P. Ngo, I. Debled-Rennesson, F. Mothe, and F. Longuetaud, *Ant Colony Optimization for Estimating Pith Position on Images of Tree Log Ends*, Image Processing On Line, 12 (2022), pp. 558–581, https://doi.org/10.5201/ipol.2022.338.

[3] F. Devernay, *A Non-Maxima Suppression Method for Edge Detection with Sub-Pixel Accuracy*, tech. report, Inria Research Report 2724, Sophia Antipolis, 1995.

[4] P. Duncker, *Detection and Grading of Compression Wood*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 201–224, https://doi.org/10.1007/978-3-642-10814-3_7.

[5] A. Fabijańska and M. Danek, *DeepDendro -- A Tree Rings Detector Based on a Deep Convolutional Neural Network*, Computers and Electronics in Agriculture, 150 (2018), pp. 353–363, https://doi.org/10.1016/j.compag.2018.05.005.

[6] A. Fabijańska, M. Danek, J. Barniak, and A. Piórkowski, *Towards Automatic Tree Rings Detection in Images of Scanned Wood Samples*, Computers and Electronics in Agriculture, 140 (2017), pp. 279–289, https://doi.org/10.1016/j.compag.2017.06.006.

[7] P. Getreuer, *Linear Methods for Image Interpolation*, Image Processing On Line, 1 (2011), pp. 238–259, https://doi.org/10.5201/ipol.2011.g_lmii.

[8] A. Gillert, G. Resente, A. Anadon-Rosell, M. Wilmking, and U. F. Von Lukas, *Iterative Next Boundary Detection for Instance Segmentation of Tree Rings in Microscopy Images of Shrub Cross Sections*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2023, pp. 14540–14548, https://doi.org/10.1109/CVPR52729.2023.01397.

[9] R. Grompone von Gioi and G. Randall, *A Sub-Pixel Edge Detector: an Implementation of the Canny/Devernay Algorithm*, Image Processing On Line, 7 (2017), pp. 347–372, https://doi.org/10.5201/ipol.2017.216.

[10] M. Henke and B. Sloboda, *Semiautomatic Tree Ring Segmentation Using Active Contours and an Optimised Gradient Operator*, Forestry Journal, 60 (2014), pp. 185 – 190, https://doi.org/10.2478/forj-2014-0020.

[11] P. Kennel, P. Borianne, and G. Subsol, *An Automated Method for Tree-Ring Delineation Based on Active Contours Guided by DT-CWT Complex Coefficients in Photographic Images: Application to Abies Alba Wood Slice Images*, Computers and Electronics in Agriculture, 118 (2015), pp. 204–214, https://doi.org/10.1016/j.compag.2015.09.009.

[12] K. Makela, T. Ophelders, M. Quigley, E. Munch, D. Chitwood, and A. Dowtin, *Automatic Tree Ring Detection Using Jacobi Sets*, ArXiv, (2020), https://doi.org/10.48550/arxiv.2010.08691.

[13] H. Marichal, D. Passarella, C. Lucas, L. Profumo, V. Casaravilla, M. N. R. Galli, S. Ambite, and G. Randall, *UruDendro, a Public Dataset of 64 Cross-Section Images and Manual Annual Ring Delineations of Pinus Taeda L.*, Annals of Forest Science, 82 (2025), p. 25, https://doi.org/10.1186/s13595-025-01296-5.

[14] H. Marichal, D. Passarella, and G. Randall, *Automatic Wood Pith Detector: Local Orientation Estimation and Robust Accumulation*, in International Conference on Pattern Recognition (ICPR), 2025, pp. 1–15, https://doi.org/10.1007/978-3-031-78447-7_1.

[15] S. Nesmachnow and S. Iturriaga, *Cluster-UY: Collaborative Scientific High Performance Computing in Uruguay*, in Supercomputing, Cham, 2019, Springer International Publishing, pp. 188–202, https://doi.org/10.1007/978-3-030-38043-4_16.

[16] K. Norell, J. Lindblad, and S. Svensson, *Grey Weighted Polar Distance Transform for Outlining Circular and Approximately Circular Objects*, International Conference on Image Analysis and Processing (ICIAP), (2007), pp. 647–652, https://doi.org/10.1109/ICIAP.2007.4362850.

[17] M. Poláček, A. H. Arizpe, P. Hüther, L. Weidlich, S. Steindl, and K. L. Swarts, *Automation of Tree-Ring Detection and Measurements Using Deep Learning*, Methods in Ecology and Evolution, 14 (2023), pp. 2233–2242, https://doi.org/10.1111/2041-210X.14183.

[18] X. Qin, Z. Zhang, C. Huang, M. Dehghan, O. R. Zaiane, and M. Jagersand, *U2-Net: Going Deeper with Nested U-Structure for Salient Object Detection*, Pattern Recognition, 106 (2020), p. 107404, https://doi.org/10.1016/j.patcog.2020.107404.

[19] H. Zhou, R. Feng, H. hong Huang, E. pei Lin, and J. lin Yu, *Method of Tree-Ring Image Analysis for Dendrochronology*, Optical Engineering, 51 (2012), p. 077202, https://doi.org/10.1117/1.OE.51.7.077202.