

DIGITAL FACHWERK

a KM0-DFAB approach for construction with irregular timber

Fjodor Serov, 5170176
supervisors: Ir.arch. G. (Gilbert) Koskamp, Dr. G. (Gabriele) Mirra
Delft University of Technology
Faculty of Architecture and the Built Environment
Track: Building Technology

DIGITAL FACHWERK

A KM0-DFAB APPROACH FOR CONSTRUCTION WITH IRREGULAR TIMBER

Fjodor Serov (5170176) | 28-05-2026
supervisors: Ir.arch. G. (Gilbert) Koskamp, Dr. G. (Gabriele) Mirra

Delft University of Technology

Faculty of Architecture and the Built Environment

Track: Building Technology

1 ABSTRACT

The contemporary timber industry relies on dimensionally standardised components, whose production generates significant material waste: only 38% of total wood removals is used for building and construction, while geometrically irregular timber, such as urban wood, is often discarded. Beyond material inefficiency, standardisation contributes to increased carbon emissions through processing and transportation, as standardised timber is rarely sourced locally.

By contrast, historical craft practices produced structural systems that naturally integrated heterogeneous, locally sourced materials, resulting in more sustainable forms of construction. Half-timbered framing systems such as English cruck-framing or German fachwerk, both developed during the Middle Ages, feature curved and irregular elements which, besides optimising material use, provided a distinctive architectural expression. Despite their sustainability potential, these practices are today economically unfeasible due to the lack of scalable design and fabrication methods capable of handling material variability.

This paper describes a computational workflow for the design and fabrication of half-timbered frame systems from a heterogeneous stock of wooden components, exploring the potential of pre-harvest structural stock assessment and genetic optimisation for stock-constrained design.

First, a computational tool enables the assessment of standing trees as potential structural stock. Through point-cloud analysis, the local curvature and planarity of branches are determined, assigning structural roles to each element and distinguishing straight components suitable for posts and beams from curved components suited for bracing.

Subsequently, a physical design game is developed to establish a placement logic for curved branches within the frame, which is then encoded into a digital design system. This system is then subjected to optimisation using genetic algorithms. Whilst the genetic algorithm converged to local rather than global optima, it offered promising indications of the approach's potential.

The two algorithms are united in a demonstrative application, enabling the design of structures from irregular timber found in the immediate surroundings. A user can upload tree-scan data, upon which the application generates potential structural designs, bridging the gap between forestry and architectural design.

By combining kilometre-zero construction with digital fabrication, the proposed scan-to-fabrication workflow supports local material sourcing strategies and reduces emissions in timber construction. Beyond providing a scalable method for heterogeneous timber-wall elements, it supports the subsistence of local architectural styles, regional material supply chains, and bio-based building — an approach with potential to extend to other material systems and design domains.

2 FOREWORD

Writing this thesis was a great pleasure. It was one of my favourite projects to date, spanning an unusually broad range of topics: from natural materials and computational design to game development and interface design. I hope it contributes some new knowledge to the field, but if nothing else, the learning curve was steep and the experience profoundly rewarding. I am deeply grateful for the opportunity to dedicate such an extended period to a single project. None of it would have been possible without the support of:

Joan, for endless patience and support right until the very last moment, My parents and Nina, for always being there, Gilbert and Gabriele, for thorough guidance, understanding, and valuable insights, Jelle, for the inspiration and expertise, Lucas, for the field trip to Germany and all the conversations, Rens, for sharing the same path and the enjoyable breaks along the way, Marten, for all the good music throughout.

Contents

1 ABSTRACT 3
2 FOREWORD 4
3 MANIFEST 6
4 INTRODUCTION 10
4.1 STANDARDISATION OF TIMBER . . 11
4.2 IRREGULAR TIMBER: PRE-INDUSTRIAL PRECEDENTS 15
4.3 MASS ADAPTATION 15
4.4 RESEARCH FRAMEWORK 19
4.4.1 PROBLEM STATEMENT . . . 19
4.4.2 RELEVANCE 19
4.4.3 OBJECTIVE & MOTIVATION 19
4.4.4 BOUNDARY CONDITIONS & SCOPE 20
4.4.5 RESEARCH & DESIGN QUESTIONS 22
4.5 RESEARCH AND DESIGN FRAMEWORK 22
5 PART A - PRE-HARVEST YIELD ASSESSMENT 26
5.1 STATUS 26
5.2 METHODOLOGY: DEVELOPMENT OF YIELD-ASSESSMENT ALGORITHM 29
5.2.1 PRE-PROCESSING 30
5.2.2 PLANAR BRANCH SEGMENTATION 33
5.2.3 STRUCTURAL ROLE ALLOCATION 35
5.2.4 ALGORITHM OUTPUTS . . . 36
5.3 CASE STUDIES 36
5.4 RESULTS 37
5.5 DISCUSSION 37
6 PART B - STOCK-CONSTRAINED DESIGN FOR HALF-TIMBERED WALL ELEMENTS WITH IRREGULAR COMPONENTS 42
6.1 STATUS 42
6.2 METHODOLOGY 44
6.2.1 DEVELOPMENT OF DESIGN GAME ENVIRONMENT 46
6.3 GENETIC OPTIMISATION FOR DIGITAL FACHWERK 50
6.4 RESULTS 55
6.4.1 WIND LOAD VARIATIONS . 55
6.4.2 STOCK TYPES 57
6.5 DISCUSSION 62
7 DEMONSTRATOR 66
8 CONCLUSIONS 74
9 FUTURE WORK 76
10 REFLECTIONS 77
Bibliography 78
11 Appendix 85
11.1 Fachwerk system configuration parameters 85
11.2 Source code 86
11.2.1 Appendix — Core Analysis Source Code 86
11.2.2 Stage 1 — Skeleton assessment 87
11.2.3 A.2 Stage 2 — Fachwerk optimisation 109

3 MANIFEST

In current practices of architecture and construction, design is often disconnected from local material and production realities (Stevens, 2015, p vi). Globally ("Buildings, Culture and Environment," 2005), this disconnect is reinforced by an architecture, engineering and construction industry dependent on standardisation for its cost-efficiency, predictability, and design simplicity:

- the notion of mass production suggests that price per unit can be reduced through standardised production (Suffian et al., 2016).
- the notion of design and building codes suggests that standardised materials have a predictable character (Vrouwenvelder et al., 2024).
- the notion of design-build separation suggests that standardised designs require less material-specific expertise (Manaf et al., 2025).

However, this pursuit of standardisation comes at both an architectural and environmental cost.

It is argued that modern architecture has become placeless, driven by the forces of modernisation and standardisation (Frampton, 1993). The standardisation of materials and construction methods has led to a decline in local craftsmanship, an accumulation of centuries of location-specific expertise (Form, 1987). Building techniques that responded to local surroundings, climate, and tectonics have been replaced by globally applied methods that produce generic, exportable forms (Correia et al., 2014), reflecting a "one size fits all" approach. For instance, Figure 1 and Figure 2 show contemporary construction in Nepal and Germany respectively;

Beyond its architectural impact, standardisation also carries significant environmental consequences, such as high embodied carbon in material production (Akbarnezhad et al., 2017; Fennell et al., 2022; Zhong et al., 2021) and emissions from global transportation networks, where transporting high-volume, low-value goods across thousands of miles has become commonplace (Greer & Horvath, 2025; Mohammad et al., 2023; Suffian et al., 2016). A baseline scenario projects these emissions will increase by 130%, particularly in low- and middle-income countries (Zhong et al., 2021).

Furthermore, global supply chains are vulnerable to disruptions from natural disasters, political instability, and market pressures, which proves especially problematic for post-disaster rebuilding (Forbes, 2018; Kijewski-Correa & Taflanidis, 2012).

The construction industry is characterised as one of the least efficient manufacturing sectors (Tuvayanond et al., 2023). The construction industry's fragmented nature, with adversarial project-by-project relationships between clients, contractors, designers and suppliers, prevents effective long-term supply chain management (Donato, 2016). More integrated approaches, such as early contractor involvement (Song et al., 2009) or design for manufacturing and assembly (dfMA) (Gao et al., 2020), could increase project efficiency.

Moreover, the digital transformation associated with Industry 4.0 has become a major focus for sustainability in construction (Wang et al., 2022). Research suggests that digital technologies, including cyber-physical systems, IoT sensing, artificial intelligence, blockchain, and digital fabrication, hold significant potential for optimising design, construction, monitoring, and maintenance in pursuit of greater efficiency and sustainability (De Wolf et al., 2024, p xiv).



Figure 1: Modern housing in Nepal. source: (***House Design in Nepal | Top Construction Company in Nepal | Call Us***, n.d.)



Figure 2: Modern housing in Germany. Source: (***Haus Bohlenstraße - Splietker Bau GmbH & Co. KG***, 2019)

Many developed methodologies focus on mass-customisation of standardised materials such as plastic, concrete, and steel 3D printing (sometimes requiring even greater chemical processing), robotic brick-laying, and engineered wood assembly, providing complex design at near mass-production costs (Claypool et al., 2021; Dakhli & Lafhaj, 2017; Larsen et al., 2019; Mechtcherine et al., 2018; Placzek et al., 2024).

Whilst these efforts are definitely a step in the right direction, providing design freedom that enables optimised geometries, they echo philosophical tension in about how ***form*** (morphē) relates to ***matter*** (hylē).

In the 4th century BCE, Aristotle articulated ***hylomorphism***: Making something means coupling form to matter. A bronze statue only becomes a statue when the material is given its specific form. This concept does not propose a strict hierarchy, it suggests a conceptual priority of form over matter (Ainsworth, 2024).

Today, digital design and fabrication push this logic to an extreme. Whilst computational algorithms explore vast design spaces, robotic systems realise complex geometries, increasingly forcing materials to match the design through moulding, milling, or printing. This thesis frames this condition as "***hyper-hylomorphism***": form is seen as unlimited in the digital realm, whilst matter is muted - processed, engineered, and transported globally to fit the design, often ignoring local material realities entirely.

This top-down paradigm constantly asks "What material can we shape into this geometry?" rather than bottom-up: "We have this material, what can we make with it?". Can such a system ever become truly sustainable?

Aristotle lived in a world where building was largely "KM0": materials were sourced nearby, and design had to respond to the qualities and limits of what was available. The Parthenon, for example, was built from Pentelic marble quarried roughly 17 km from Athens (Dinsmoor, 1973).

Their use of local materials demonstrates not only environmental benefits trough minimal transportation, less energy-intensive production, and designs responding to local bioclimates, but also socio-economic and socio-cultural advantages such as supply chain transparency, communal production, and self-sufficiency (Correia et al., 2015).

However, in that context, a cracked stone could only be replaced by a similar stone, meaning the craftsperson needed to adjust the detailing or assembly accordingly. Material heterogeneity was not a problem of standardisation but rather a means of design.

This thesis explores a potential hybrid framework: Kilometre-Zero Digital Fabrication (KM0DFAB), a sourcing, production, and design system where material extraction, analysis, processing, fabrication, and assembly occur within the smallest radius of the building site as possible. By adapting global knowledge, digital technologies, and mobile tooling to local circumstances and resources, this framework synthesizeso contemporary efficiency and vernacular sustainability, shifting from mass-production and mass-customisation toward mass-adaptation.



Figure 3: The use of curved elements in fachwerk in Tübingen, Germany ***Source: own***

4 INTRODUCTION

This thesis investigates how globally available digital technologies can enable efficient design with locally sourced, non-standardised timber. As for thousands of years, builders worked with timber as they found it: unprocessed and irregular.

Whilst timber is recognised as a circular, renewable, and low-emission construction material (Herzog et al., 2012, p. 47), and one of the most geographically widespread (Herzog et al., 2012, p. 24), the construction industry's strong preference for uniform, homogeneous components generates significant waste streams, excessive processing, and increased transport emissions (Kampelmann, 2021; Lindholm & Berg, 2005; Ramage et al., 2017).

In contrast, historical half-timbered framing systems such as English cruck-framing and German fachwerk (Figure 3), both developed during the Middle Ages, featured curved and irregular elements (Harris, 1979, p. 18).

Informed by the European tradition of half-timbered framing and the Kilometre-Zero Digital Fabrication (KM0DFAB) paradigm, a framework synthesising the efficiency of digital construction with vernacular sustainability, this thesis proposes a new construction system: Digital Fachwerk, a contemporary reinterpretation of half-timbered construction using locally sourced timber.

Drawing on traditional timber construction, which employed local material selection, geometry-driven design strategies and handcrafted joinery guided by material knowledge and community expertise (Mortier, 2025), this thesis digitises two practices that have been largely lost.

A) pre-harvest timber stock assessment

Traditional craftsmen matched tree geometries to structural roles, leveraging natural curves for ship-building, cruck-framing, and bracing (Engelmann, 2017). Current industrial practice only accepts timber within strict tolerances, disregarding the potential of irregular logs (Larsen et al., 2022a; Vestartas, 2021, p. 9). This thesis proposes a computational approach that analyses pre-harvest tree scans to evaluate structural yield potential for both straight and curved logs.

B) stock-constrained design with curved elements

After assessing available resources, craftsmen shaped their designs around what the timber offered. Cruck curvatures were dictated by log geometry, boat construction leveraged specific fork angles, and facade compositions reflected available timber characteristics (Engelmann, 2017; Harris, 1978). This represents a material-responsive design logic, in contrast to current industrial practice where irregular timber is forced to conform to predetermined designs. Building on this logic, this thesis proposes a generative design algorithm that incorporates straight and curved logs from the pre-harvest assessment to produce timber framing wall systems that strategically utilise curved elements for bracing.

Two contextual foundations inform Digital Fachwerk:

- the drivers and consequences of standardisation in timber construction
- the historical methods that enabled material-responsive design

4.1 STANDARDISATION OF TIMBER

In recent years, attention towards timber construction has grown, given its potential to achieve net-negative life cycle impacts (Göswein et al., 2022). The timber industry combines advanced technology with optimised production processes, producing standardised products such as consistent board sizes, certified timber, and engineered wood products (EWPs) (Buri & Weinand, 2011; Larsen et al., 2022a).

Such optimisation has resulted in highly mechanised workflows: in Nordic countries, for instance, nearly 100% of logging is mechanised, integrating cut-to-length systems (Ramage et al., 2017). However, these systems can only handle a limited range of geometric inputs (Gattas et al., 2023; Larsen et al., 2022a), and timber's inherent irregularities, such as curvature, cross-sectional variation, and the absence of reference planes, hinder accurate measurement, simulation, and integration into standardised production workflows (Chai et al., 2024). Standardised machines rely on fixed jigs and rule-based routines, which become a bottleneck when processing irregular logs that require case-specific positioning (Gao et al., 2026). In a Finnish sawmill, for example, only a certain degree of skewness is permitted (Larsen et al., 2022a). Whilst this is prevalent with deciduous urban timber (Kampelmann, 2021), even plantation-grown trees exhibit high variation in shape, composition, and quality (Larsen et al., 2022a). For instance, Vestartas notes that the commercial unavailability of such timber even constrains research purposes. (Vestartas, 2021, p 244)

Beyond mechanical bottlenecks, grading standards further exclude irregular geometries: DIN EN 844-8 (1997) characterises curvature as a serious defect, whilst DIN 4074-1 (2002) limits reaction wood to one-third of the log diameter (Engelmann, 2017).

These bottlenecks have direct consequences on material waste. Globally, only 38% of harvested timber is used in construction (Ramage et al., 2017) (Figure 4), with the remainder processed into pulp or bio-energy. In Finland, only a third of felled trunks are used as sawlogs (Ovando et al., 2025; Torvinen et al., 2025).

Whilst waste wood combustion can be considered carbon-neutral (if adhesive-free), it remains a relatively carbon-intensive process; natural gas emissions amount to only 55% of those released by wood fuel. In addition, industrial roundwood holds on average five times the value of fuelwood, which is inherently single-use. Timber is therefore better utilised as a long-term building product, with energy production left to carbon-free alternatives (Ramage et al., 2017).

As waste is primarily comprised of irregularly shaped logs and branches, it mostly derives from deciduous species (Larsen et al., 2022a; Lok et al., 2021). In focusing on straight, standardised elements, the structural potential of curved timber and deciduous hardwood alike is largely overlooked.(Gao et al., 2026; Torvinen et al., 2025).

Beyond waste streams, standardised timber production carries additional environmental costs. Kiln drying, primarily required for dimensional stability, can account for up to 92% of manufacturing energy (Glass & Zelinka, 2021), whilst adhesives used in engineered wood products (EWPs) can represent up to 52% of total environmental impact (Laurent et al., 2013; Ramage et al., 2017). In general, reducing the degree of processing therefore directly improves timber's overall environmental performance (Vestartas, 2021).

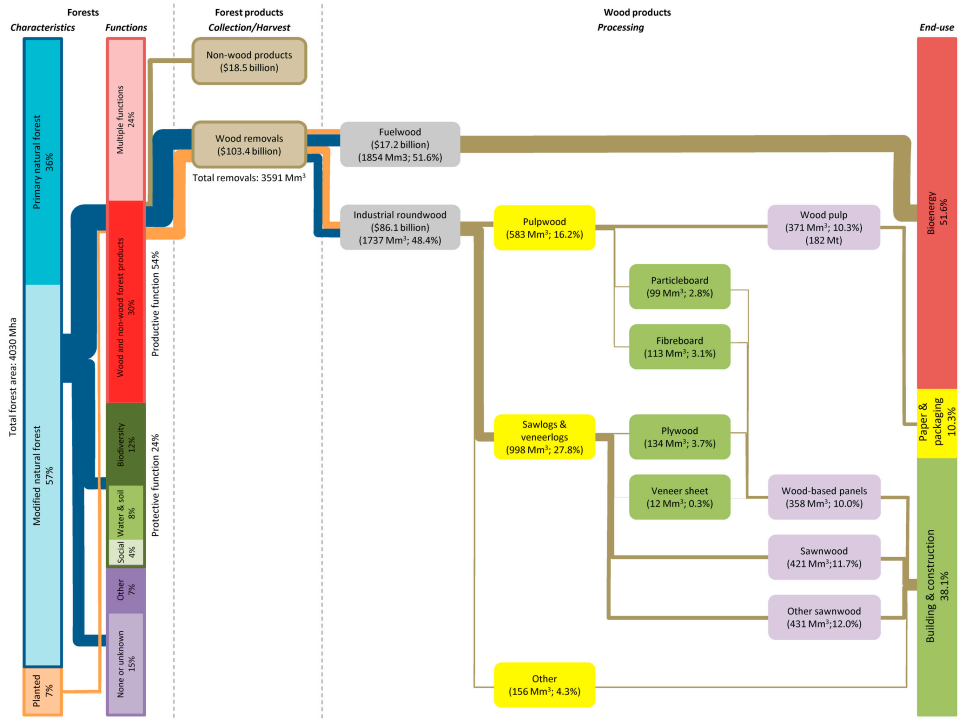


Figure 4: Global wood removal data for 2013. ***Source:*** (Ramage et al., 2017)

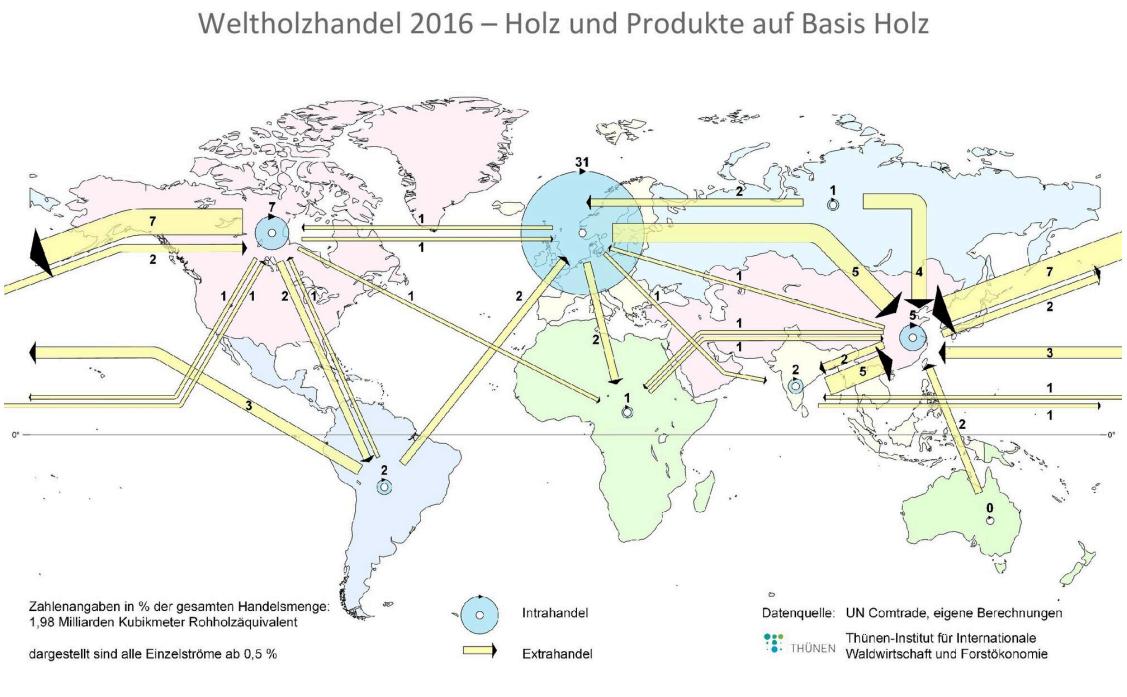


Figure 5: Global timber trade 2016. ***Source: (Thünen, 2019)***

In addition to processing, timber must be transported from harvest to end use, with road transport contributing significantly to its environmental impact (Chesneau et al., 2012). This is especially concerning given that most of Europe's forests are concentrated in northern latitudes or mountainous peripheries, where energy-intensive road transport is largely unavoidable, negating timber's environmental advantages in some cases (Greer & Horvath, 2025). Approximately 25% of industrial roundwood enters the international market (Ramage et al., 2017); In 2016, two billion cubic metres of wood-based products were traded internationally, of which 400 million cubic metres comprised roundwood and sawnwood (Thünen, 2019) Figure 5. The Swiss Alps offer an illustrative case: 15% of the annual harvest is exported raw for cheaper processing abroad, only to be reimported as finished lumber or panels, significantly increasing transport emissions.(Vestartas, 2021).

Beyond environmental impact trough waste and emissions, timber standardisation contributes to ecological degradation. Optimised plantation forestry, whilst enabling timber to compete economically with less sustainable materials, has concentrated production on fast-growing coniferous species, making more durable alternatives costly to procure (Ramage et al., 2017). Tight planting spacing generates an overabundance of small-diameter, low-value trees that are expensive to thin, leaving forests increasingly vulnerable to wildfires, diseases, and insect attacks (Vestartas, 2021). More broadly, these monocultures actively degrade ecosystems at a time when the world faces not only a climate crisis but an accelerating biodiversity crisis (Larsen et al., 2022a).

Finally, converting timber into standardised engineered products strips the material of its naturally optimised structural properties, such as fibre orientation, load paths, and inherent form, only to artificially reinstate them through adhesives, at the cost of recyclability and sustainability (Larsen et al., 2022a; Mollica, 2016a; Ramage et al., 2017; Vestartas, 2021). The fabrication of these EWP's, despite enabling greater architectural complexity, largely perpetuates the "hyper-hylomorphic" industrial logic rather than exploiting timber's inherent properties.

Beyond structural and ecological consequences, timber standardisation has contributed to an architectural homogenisation; where vernacular timber construction produced location-specific and culturally responsive forms shaped by local species, and craft (Schittich, 2019), contemporary timber architecture increasingly reflects a globalised aesthetic divorced from local material reality (Frampton, 1993).

Collectively, these consequences reveal that the economic rationality of timber standardisation comes at significant environmental, structural, and cultural cost Figure 6.

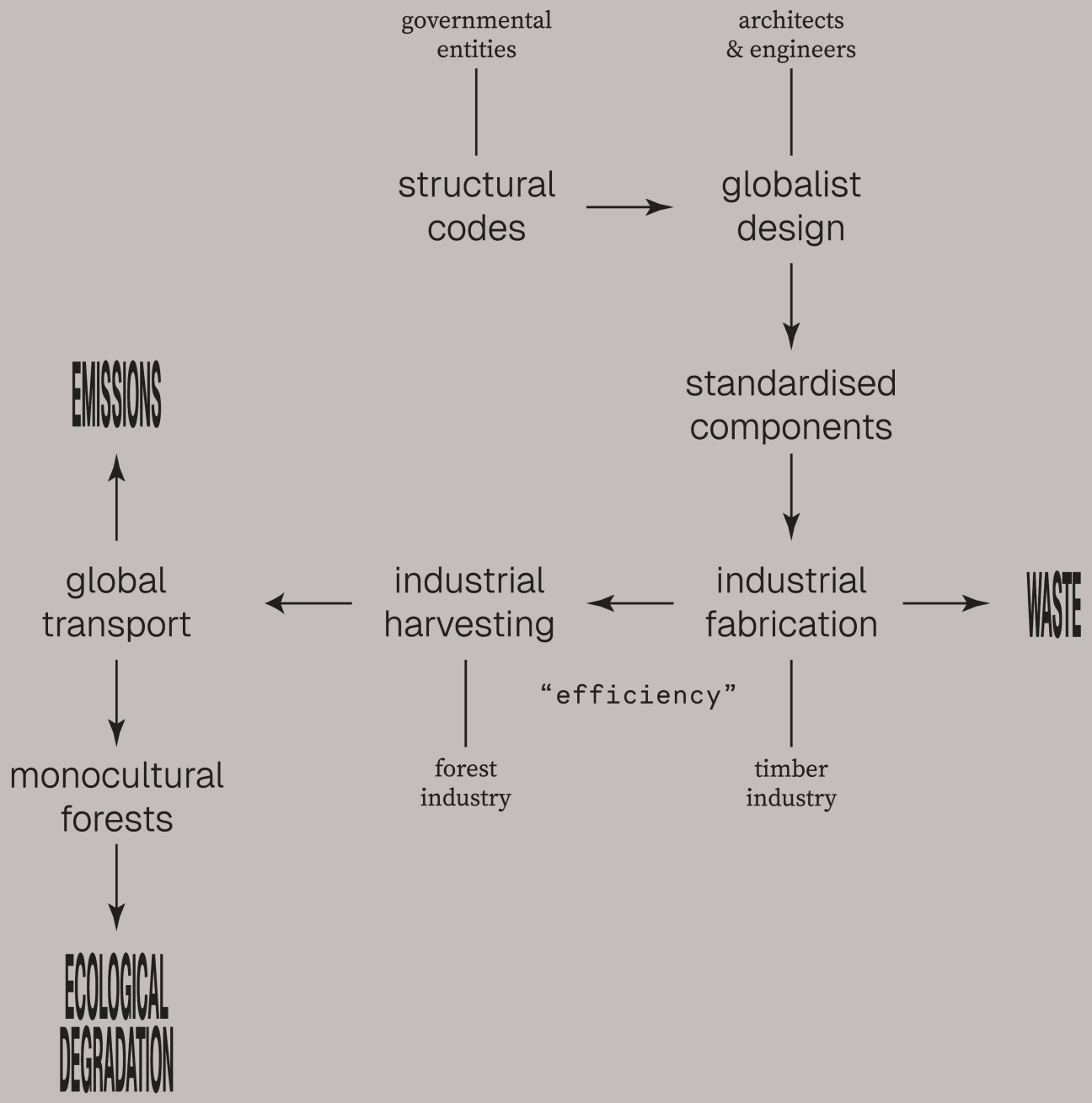


Figure 6: Schematisation of timber supply-chain and it's impact ***Source: own***

4.2 IRREGULAR TIMBER: PRE-INDUSTRIAL PRECEDENTS

In contrast to today, the traditional timber supply chain was local and adaptive, with builders possessing deep knowledge of nearby resources and handcrafting joints to suit each individual piece (Karakul Türk, 2015; Mortier, 2025). This place-based knowledge manifested in regionally distinct architectural solutions: Alpine regions utilised timber for solid log construction, while Central European plains employed available oak for timber-frame systems with infill panels of locally sourced straw, cob, or clay (Herzog et al., 2012, p. 28).

Furthermore, wood quality is inherently context-dependent, defined by intended use rather than absolute standards (Nocetti & Brunetti, 2024). Trees rarely grow straight, and traditional carpenters constantly faced the challenge of maximising the use of irregular timber (Harris, 1978). Equipped with hand tools, carpenters developed adaptive joinery techniques that accommodated wood's natural variability (Mollica, 2016a). This adaptability extended beyond joinery to structural form itself.

Curved timber proved advantageous in cruck truss structures, where natural curvature enabled larger spans and greater headroom than simple rafter roofs; the curved geometry redirected horizontal forces into vertical loads within the curve itself, reducing reliance on tension beams (Engelmann, 2017). Regional species selection differed accordingly: in England, poplar was preferred for cruck trusses whilst oak served posts and beams (Meeson, 2021), whereas German construction primarily employed crooked oak branches for vaults and roof structures (Engelmann, 2017).

In half-timbered construction, which is the main focus of this thesis, curved timber served dual purposes. Functionally, curved pieces acted as wall, head, or foot braces analogous to straight-grown timbers (Figure 7) (Engelmann, 2017). Aesthetically, curved elements appeared prominently in facades, arranged symmetrically by pairing similarly curved timbers or splitting pieces lengthwise. Whether this represented intentional design or pragmatic adaptation to limited straight stock — making a virtue of necessity — remains debated (Engelmann, 2017). Unlike contemporary material grading, the quality of timber in vernacular practice was thus defined by fitness for purpose, a principle that shaped both structural and aesthetic outcomes.

The whole supply chain was local: consisting of informed harvesting, adaptive designs and adapative fabrication, whith each stage informing each other Figure 8. From craftsman-dependent tree-assessment using stencils Figure 12 (Engelmann, 2017; Nocetti & Brunetti, 2024) , to design adaption to available logs (Benson & Gruber, 1981), to joinery based on available beam dimensions (Meeson, 2021). Assembly was described as a communal effort, without heavy machinery, bringing together local builders and residents around the timber frame (Benson & Gruber, 1981).

Currently, these practices are economically unviable: industrialisation prioritised efficiency (Mortier, 2025), while irregularity slows standardised workflows (Chai et al., 2024), and timber frame costs exceed those of prefabricated alternatives such as concrete (Wrzesiński et al., 2023).

4.3 MASS ADAPTATION

Timber building has progressed through three distinct technological phases: the wooden age (hand-tool technology), the industrial age (machine-tool technology), and the digital age (information-tool technology) (Buri & Weinand, 2011). The digital age fundamentally differs from its predecessor:



Figure 7: Use of curved elements as footer and as bracing. ***Source:*** (Engelmann, 2017)

Where industrial machinery requires manual settings to dictate form, computer-controlled design and fabrication allow material data to be integrated directly into the production process, enabling an efficient response to material variation rather than suppressing it. Applied

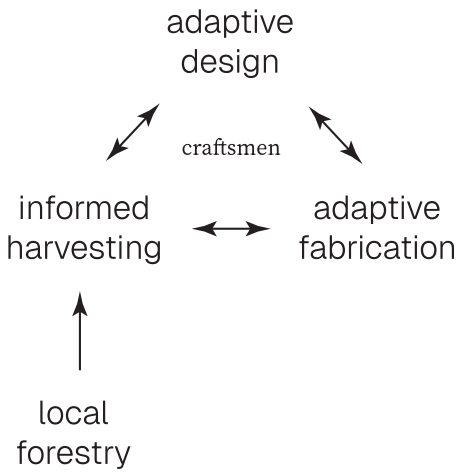


Figure 8: Simple supply chain of traditional timber industry. ***Source:*** own

to irregular timber, these capabilities have produced workflows integrating component scanning, adaptive design, and robotic fabrication (Chai et al., 2024; Larsen et al., 2022a; Mollica, 2016a; Vestartas, 2021).

Rapid digital tools enable a renewed supply chain analogous to vernacular practice but with reduced craft dependency @. Such a workflow may be described as a scan-to-fabrication pipeline:

1. Pre-harvest, LiDAR-based terrestrial laser scanning and photogrammetry enable point cloud generation of standing trees (Krok et al., 2020; Nocetti & Brunetti, 2024), and open-source software already implements geometric primitive fitting to model tree geometry (Du et al., 2019; Stovall et al., 2018). However, no automated method currently exists to segment standing tree geometry into straight and curved planar elements suitable for timber-framing, which could inform both harvesting and conservation strategies.
2. Once felled, mobile sawmills supported by machine vision, CT scanning, and laser imaging can capture internal defects, knot characteristics, and external log geometry (Gata et al., 2024; Nasir et al., 2024), providing the material data needed for informed design and further fabrication.
3. With scanned log geometry, stock-constrained computational design can assign irregular components to various structural typologies(Sun et al., 2026). For instance, optimisation with genetic algorithms methods have successfully incorporated irregular timber into trusses and gridshells (Mollica, 2016a; Rahbek et al., 2023), though application to timber-framing wall systems with geometrically irregular elements remains unexplored.
4. Fabrication can be automated through sawing pattern simulation for accurate log positioning (Nasir et al., 2024) and freeform robotic milling enabling precise custom cuts (Larsen et al., 2022a; Mollica, 2016a; Vestartas, 2021).
5. Automated joinery generation allows otherwise specialist-requiring connections such as mortise-tenon and lap joints to be computed and robotically milled (Vestartas, 2021).
6. Assembly may ultimately be performed robotically (Claypool et al., 2021), though for larger elements manual communal assembly remains viable.

Therefore, this research targets two gaps, both addressed through a dedicated computational workflow enabling rapid adaptive design:

A.: Algorithmic geometric assessment of potential structural yield consisting of both straight and curved timber in the pre-harvest phase.

B.: Algorithmic stock-constrained design for half-timber framing walls using irregular timber, specifically through optimisation with genetic algorithms.

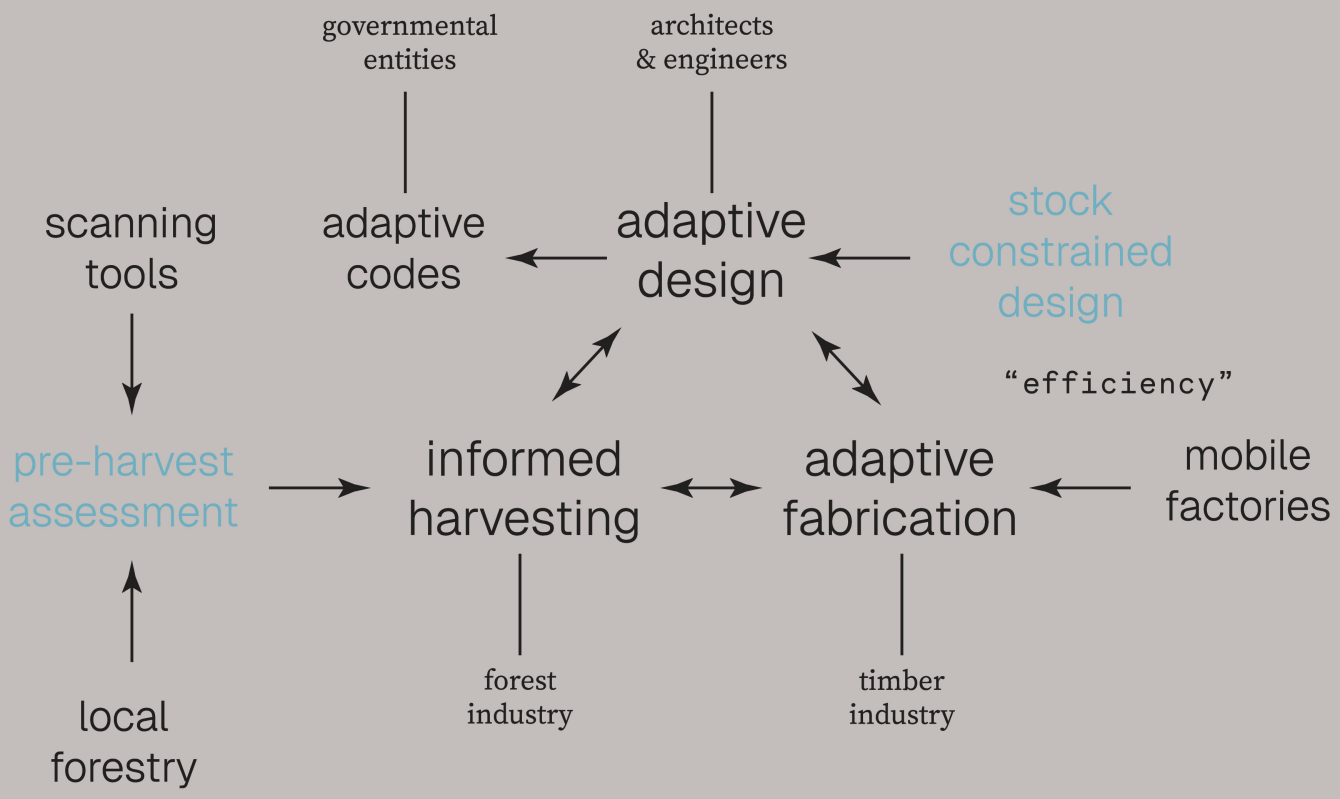


Figure 9: Vernacular construction may be understood as the thesis and mass production as the antithesis; the mass-adaptation methodology proposed here represents their synthesis, which could be seen as a KM0DFAB approach. The scope of this graduation project is highlighted in blue. ***Source: own***

4.4 RESEARCH FRAMEWORK

This section outlines the problem statement, relevance, objectives, and scope of this thesis.

4.4.1 PROBLEM STATEMENT

Given the environmental costs of standardisation, the demonstrated value of irregular timber, and the capabilities of digital fabrication, this thesis addresses the following problem:

While industrial standardisation rejects irregular timber, generating substantial waste and necessitating energy-intensive processing, digital fabrication has demonstrated the capability to handle geometric irregularity, yet no scalable pipeline exists for utilising locally-sourced irregular timber in conventional timber-framing construction.

Drawing from vernacular construction workflows, this thesis addresses two interdependent gaps:

A. No computational method exists to assess the structural yield potential of standing trees across both straight and irregular log geometries for timber-framing.

B. No generative design method accommodates the dual constraints of timber-framing structural logic and irregular stock geometry.

4.4.2 RELEVANCE

This thesis is an effort to contribute to a healthier built environment through the development of a KM0-DFAB approach specifically for timber-framing wall elements.

The projected innovation consists of two parts:

A). A computational algorithm that processes LiDAR point cloud scans of standing trees to automatically identify and quantify yield potential as both straight and curved planar structural components for timber-framing applications. The innovation lies in algorithmic geometry-based analysis of local branch curvature to subdivide tree geometry into extractable logs, enabling material-informed design decisions before harvest based on user-defined parameters such as minimum diameter, maximum curvature, and target lengths.

B). A Genetic Optimisation framework that generates structurally viable timber-framing wall configurations from fixed inventories of geometrically irregular components, strategically placing curved elements, defined by part A, as bracing within conventional half-timbered construction logic. The novelty sits in the development and training of the reinforcement learning agent and the optimised design of half-timbered walls consisting of curved bracing.

To demonstrate the innovation, both algorithms are integrated into a user interface, showcasing an accessible and scalable application.

The projected impact lies in the potential reduction of environmental impact through minimised waste streams, transport, and processing, whilst supporting local economies and re-enabling connections with local architectural language and culture.

4.4.3 OBJECTIVE & MOTIVATION

This paragraph outlines the motivation, primary objective, and specific research objectives of this thesis.

Motivation

Contemporary timber construction's dependence on industrial standardisation generates substantial environmental impact through material waste, energy-intensive processing, and long-distance transport, whilst weakening local forest economies and displacing local building traditions. This thesis is motivated by the hypothesis that computational tools, in particular LiDAR-based tree assessment and stock-constrained design,

can reinstate the material intelligence of pre-industrial construction practices, enabling efficient design and building with locally-sourced heterogeneous timber. In doing so, Digital Fachwerk can contribute to sustainable architecture, engineering, and construction.

Primary Objective

The primary objective of this thesis is to develop, validate, and deploy an integrated KM0-DFAB pipeline for half-timbered construction using irregular local timber. Through a dedicated application accessible to designers, engineers and harvesters, it demonstrates that computational workflows can transform currently rejected material into structurally viable building components whilst reducing environmental impact.

Research Objectives

Objective A (Pre-harvest structural yield assessment):

To develop a computational algorithm that processes terrestrial LiDAR point clouds of standing trees to automatically identify, segment, and quantify structural yield potential as straight and curved planar components for timber-framing applications.

Objective B (Stock-Constrained Design):

To develop and implement a evolutionary optimisation algorithm that generates structurally viable half-timbered wall configurations from constrained inventories of geometrically irregular timber components, strategically placing curved elements as diagonal bracing within rectangular frames.

4.4.4 BOUNDARY CONDITIONS & SCOPE

This thesis addresses kilometre-zero digital fabrication for half-timbered construction. Half-timbered framing is suited to this approach as it features curved elements as bracing members, its application could be applied to a larger scale, and its rich European vernacular tradition provides a well-documented precedent for material-responsive design.

Whilst the complete workflow would span from pre-harvest assessment through building assembly, and many individual components already exist in isolation, an integrated and accessible interface connecting them remains absent. This research therefore focuses on two core algorithmic components, with the following boundary conditions, design criteria, and scope exclusions:

Part A of this thesis, which explores pre-harvest assessment, includes

– geometric analyis of standing trees using LoD3 (elaborated in chapter 5) point clouds of trees (trunk and bracn geometry) retrieved throug terrestrial LiDAR scanning,
– the automated segmentation of the tree into straight and curved planar components, defined by user-input such as minimum diameter and structural role lengths. It targetherefore contributes to the quantification of yield potential of local trees specifically for timber-framing.

The design criteria for Part A require that foresters and designers with available point-cloud tree scans can assess the structural yield potential of their timber stock through an accessible interface, informing harvesting and design decisions. The algorithm should therefore be tree-agnostic and adapt to user-defined parameters, including minimum diameter, maximum curvature, and target lengths, distinguishing between straight components suitable for primary structural elements and curved components suitable for bracing. Extracted component geometries should be exportable to standard CAD formats for use in downstream design workflows, including the algorithm developed in Part B. The tool should support bulk processing

of multiple trees with near-instantaneous feedback, enabling comparative assessment and prioritisation across a harvest selection.

Excluded from Part A are non-geometrical and physical assessment strategies, including internal wood quality such as knot distribution, decay, and density variations, stem damage, strength-class determination, post-harvest grading, mechanical property prediction, and sawing optimisation, as the algorithm strictly focuses on geometrical pre-harvest assessment. Their relevance and potential integration into the broader KM0-DFAB workflow are however briefly discussed. Beyond minimum diameter, no cross-sectional information is taken into account. Whilst bifurcations have been successfully employed in specialised applications such as the Tree Fork Truss (Mollica, 2016a), their geometry exceeds the scope of this framework, which focuses on strictly on linear components characteristic of conventional timber-framing.

Part B of this thesis includes the application of a genetic algorithm for placing irregular bracing elements within rectangular wall frames, structural validation of generated configurations, exploration of design space under stock constraints and predefined vertical and horizontal loads.

The design criteria for Part B require that the algorithm accepts the geometric output and component classification (straight, curved) from Part A. Users should be able to define wall dimensions, load conditions, and stock constraints. Generated configurations must satisfy structural equilibrium under defined loads, with bracing elements placed intelligently to stiffen the frame whilst minimising material waste, indirectly maximising structural utilisation. The algorithm should generate multiple design alternatives near-instantaneously, enabling iterative design selection. Resulting geometries should be exportable for further design stages, and the workflow should be accessible to non-computational designers through a dedicated user interface.

However, in this initial development, architectural integration (windows, doors, building services), infill panel systems (wattle-and-daub, brick, insulation), facade detailing, multi-story coordination, and complete building design workflows remain excluded. Geometric irregularity in this research is limited to curvature variation along branch length; cross-sectional dimensions, and strength classes are assumed constant for each component. The focus remains on core structural framing logic.

Evaluation of the final designs is in terms of structural utilisation efficiency, structural performance, and waste reduction. This thesis develops the core algorithmic pipeline encompassing assessment, design, and fabrication specifically for irregular timber in conventional construction typologies. Integrating this workflow within broader industrial or mobile systems represents critical future work, but extends beyond the scope of this thesis.

4.4.5 RESEARCH & DESIGN QUESTIONS

Based on the defined problem statement and identified research gap, this thesis formulates an answer to:

> *How can computational algorithms for pre-harvest assessment and stock-constrained design enable architecture with locally sourced irregular timber?*

sub-qestions

Both part A and Part B have answer the two questions below respectively:

Part A: To what extent can terrestrial laser scans of standing trees be computationally processed to assess and quantify the potential yield of straight and curved structural timber components prior to harvest?

Part B: To what extent can optimisation with genetic algorithms generate structurally viable half-timbered wall configurations from a constrained inventory of geometrically irregular timber components?

4.5 RESEARCH AND DESIGN FRAMEWORK

This thesis employs a *operations research* methodology, developing algorithmic tools to address material-responsive construction challenges for half-timbered wall elements. The approach combines algorithm development and proof-of-concept implementation:

The research consists of two main parts:

Part A (chapter 5) focuses on pre-harvest structural component yield assessment, encompassing a literature review of tree assessment in traditional timber framing and contemporary methods for geometrical tree analysis, establishing the state of the art. An algorithm is then developed that processes point clouds and outputs the potential yield of structural components, both straight and curved.

Three case studies validate the algorithm by analysing specimens from established databases (PyTreeDB, BioDiv-3DTrees), producing material passports documenting tree geometry, curvature distribution, and length distribution. Cross-specimen comparison reveals selection criteria for harvesting operations whilst identifying directions for algorithm improvement.

Part B (chapter 6) focuses on stock-constrained design of timber-framing wall systems with irregular timber. A literature review examines traditional adapted timber framing practices and stock-constrained design methodologies, establishing the state of the art. An algorithm is then developed that employs multi-objective evolutionary optimisation methods to design half-timbered wall elements from irregular stock, minimising both deflection and material use, with structural validation performed through FEA solvers.

An ablation study across various wind loads and stock types validates the genetic algorithm, with cross-condition comparison providing insight into algorithmic performance.

Chapter 7 presents an integrated algorithmic pipeline demonstrating the complete workflow through an application where users input point cloud data and receive a structural wall design as output. Discussion evaluates the demonstrator against the design criteria established in the introduction, accompanied by a preliminary mock-up design.

Based on the findings, recommendations and conclusions are formulated, followed by a reflection on the research process (chapter 8 and 9).



Figure 10: step-by-step organisation of conducted research

An overview of the process, the used tools and timeline are displayed in Figure 10, Table 1 & Figure 11 respectively.

Category	Tool	Purpose
General	VSCode	Code editor
	Claude	Coding and writing assistance
	Rhino 8	3D modelling environment
	Grasshopper	Parametric modelling GUI
	.obj	Software-agnostic data exchange
Part A	AdTree	Skeletal extraction from point clouds
	PyTreeDB	Database of scanned trees
	BioDiv-3DTrees	Database of scanned trees
	NetworkX	Graph operations
Part B	GHPython	Python scripting in Grasshopper
	Karamba3D	FEA in Grasshopper
	Octopus	Multi-objective optimisation in Grasshopper
	OpenSeesPy	FEA in Python
	PyMoo	Multi-objective optimisation in Python
Demonstrator	Dash	Interactive Python visualisation

Table 1: The most relevant tools and software used throughout the research



Figure 11: Approximate timeline of research project

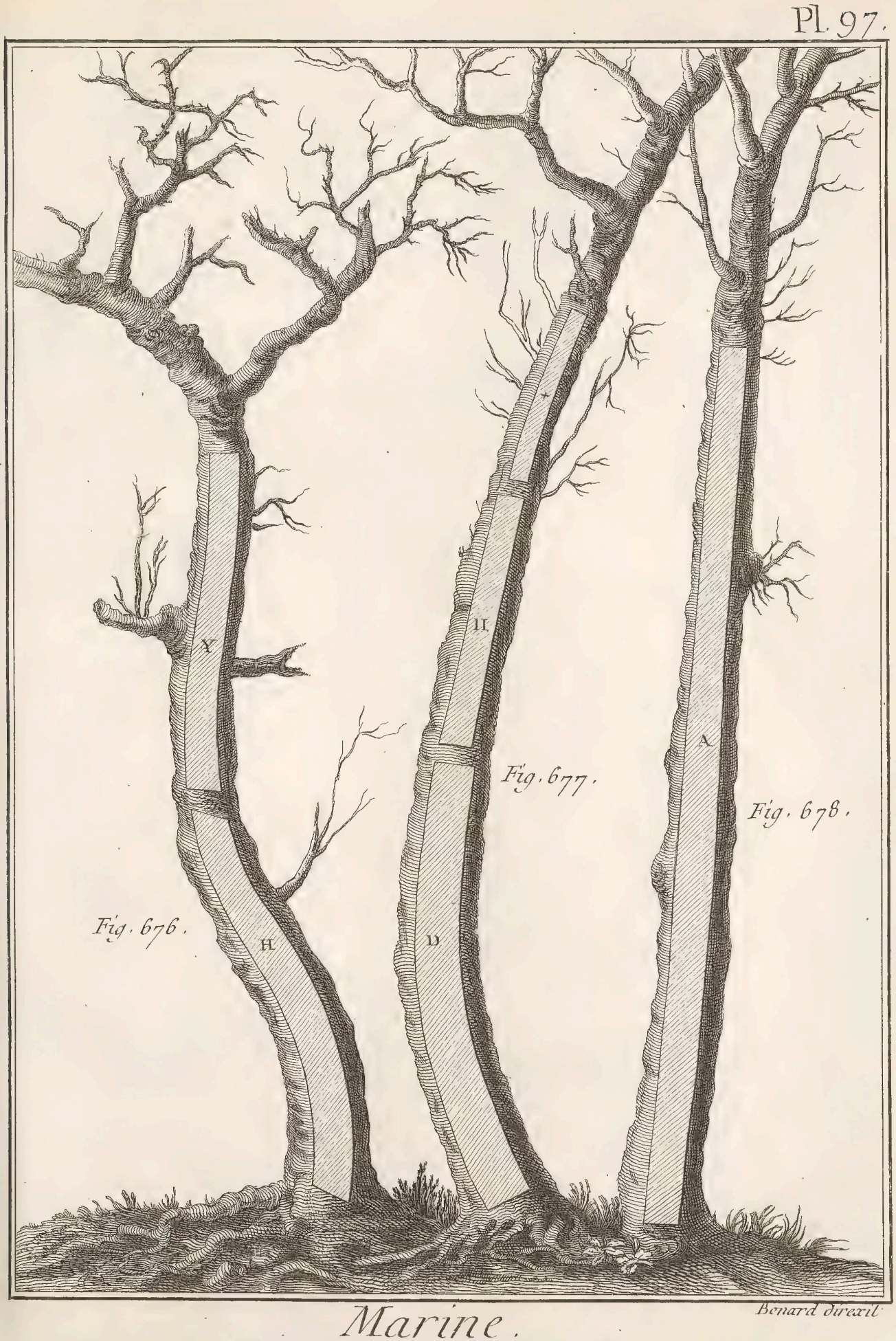


Figure 12: A scan of a drawing from 1787 displaying how stencils were used to assess the geometry trees. ***Source:*** (Blondeau, 1787)

5 PART A - PRE-HARVEST YIELD ASSESSMENT

Part A addresses the foundation of kilometre-zero digital fabrication for Digital Fachwerk: understanding locally available material. This part focuses on pre-harvest structural component yield assessment, encompassing a literature review of tree assessment in traditional timber framing and contemporary methods for geometrical tree analysis. In the methodology chapter, an algorithm is then developed that processes point clouds and outputs the potential yield of structural components, both straight and curved. Validation is done trough trough analysis of three tree specimens.

5.1 STATUS

Timber's bulk and weight historically constrained extraction to local forests—transport infrastructure and manual processing tools limited viable harvest radii. For example, timber often relied on river floating for distribution, restricting markets to navigable watersheds (Clifford & Lučić, 2026).

Within these local forests, builders practised selective harvesting by identifying suitable trees before felling, matching material to structure and adapting structure to available material. Contemporary practice can adopt this pre-harvest assessment logic, now motivated by sustainability rather than transport constraints.

For example, the shipbuilding industry demonstrated pre-harvest assessment of trees, specifically looking for curved elements: builders surveyed standing trees using form templates, seeking specific curved geometries for hull components, as is seen in Figure 12.

While these practices enabled craftsmen to match timber geometry to structural applications, manual assessment remained qualitatively limited. Available technology restricted evaluation to basic measurement (stem-diameter and straightness), or subjective operator judgement. Tasks such as identifying branches at significant heights were often impractical (Nocetti & Brunetti, 2024).

In contrast, contemporary digital technologies enable systematic, quantitative pre-harvest evaluation with reduced labour and improved precision (Krok et al., 2020).

For sustainable forest management, non-destructive tree assessment can provide insight into how local climate and cultivation treatments affect not only growth rates but also available geometry and harvestable volumes, informing both extraction and conservation decisions.

Pre-harvest quality assessment benefits forest owners and manufacturers alike, enabling effective production planning while minimizing reliance on imported standardised timber (Krok et al., 2020; Nocetti & Brunetti, 2024).

The Tree Fork Truss project at Hooke Park in Dorset, England, demonstrates pre-harvest evaluation in practice. Trough photography, the team assessed bifurcations in 204 trees within the park, one of them being visualised in Figure 13. By applying selection criteria based on minimum diameter and angle opening, they identified 40 forks as suitable candidates for their project. Following a detailed inspection to identify any defects that may have been overlooked during the initial survey, 25 forks were harvested (Mollica, 2016a).

Pre-harvest assessment focuses on visually detectable characteristics that can be categorised into four groups: tree dimensions (diameter, height, volume), stem form (straightness, ovality, taper, lean), branch characteristics (number, size, distribution), and stem damage from abiotic or mechanical sources (Nocetti & Brunetti, 2024). This

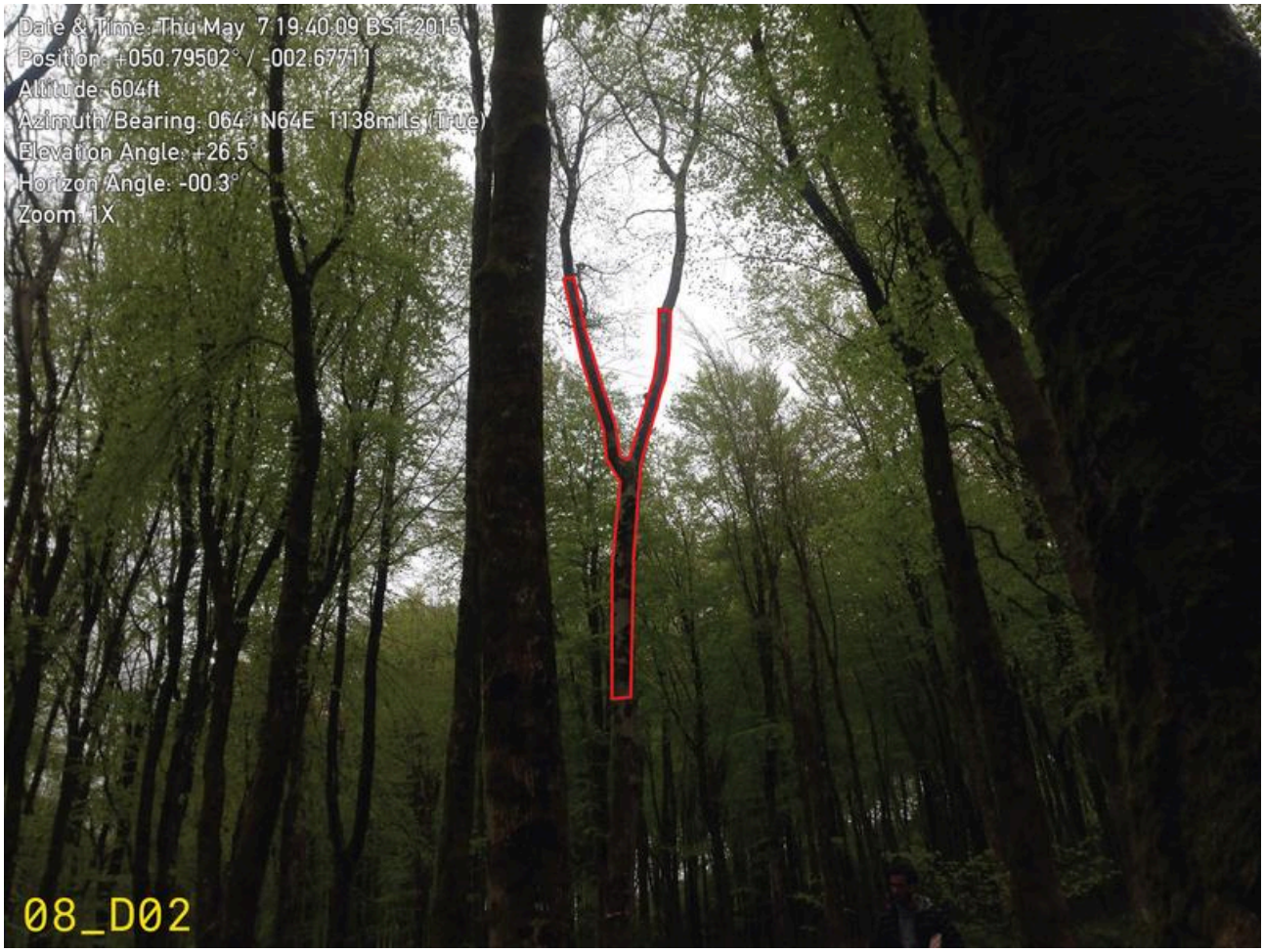


Figure 13: Pre-harvest assessment of bifurcations in at the Hooke Park in Dorset, England ***Source:*** (Mollica, 2016a)

research addresses the first three categories, with damage assessment beyond the scope.

These characteristics are for instancequantifiable trough LiDAR-based terrestrial laser scanning (TLS) and photogrammetry, which generate detailed 3D point clouds that reduce dependence on subjective human evaluation. (Krok et al., 2020; Nocetti & Brunetti, 2024).

Point cloud outputs are classified into Levels of Detail (LoD) (Figure 14): LoD1 captures tree height and diameter at breast height (DBH); LoD2 adds stem diameter at multiple points along the trunk; LoD3 incorporates branches, enabling biomass quantification and geometric analysis (Krok et al., 2020).

Leafless tree scanning enables woody component estimation by fitting geometric primitives to trunk and branch point cloud segments, reconstructing complete tree geometry. Open-source implementations (AdTree, SimpleForest, 3DForest) (Du et al., 2019), (Stovall et al., 2018), (Trochta et al., 2017) combined with consumer smartphone scanning capabilities (Syafuan et al., 2025) demonstrate a democratisation trend in tree assessment, offsetting the high cost of specialised TLS systems (Krok et al., 2020).

Whilst geometric tree assessment offers value across the forestry supply chain (Nocetti & Brunetti, 2024), no automated analysis currently exists for segmenting standing trees into straight or curved planar structural elements

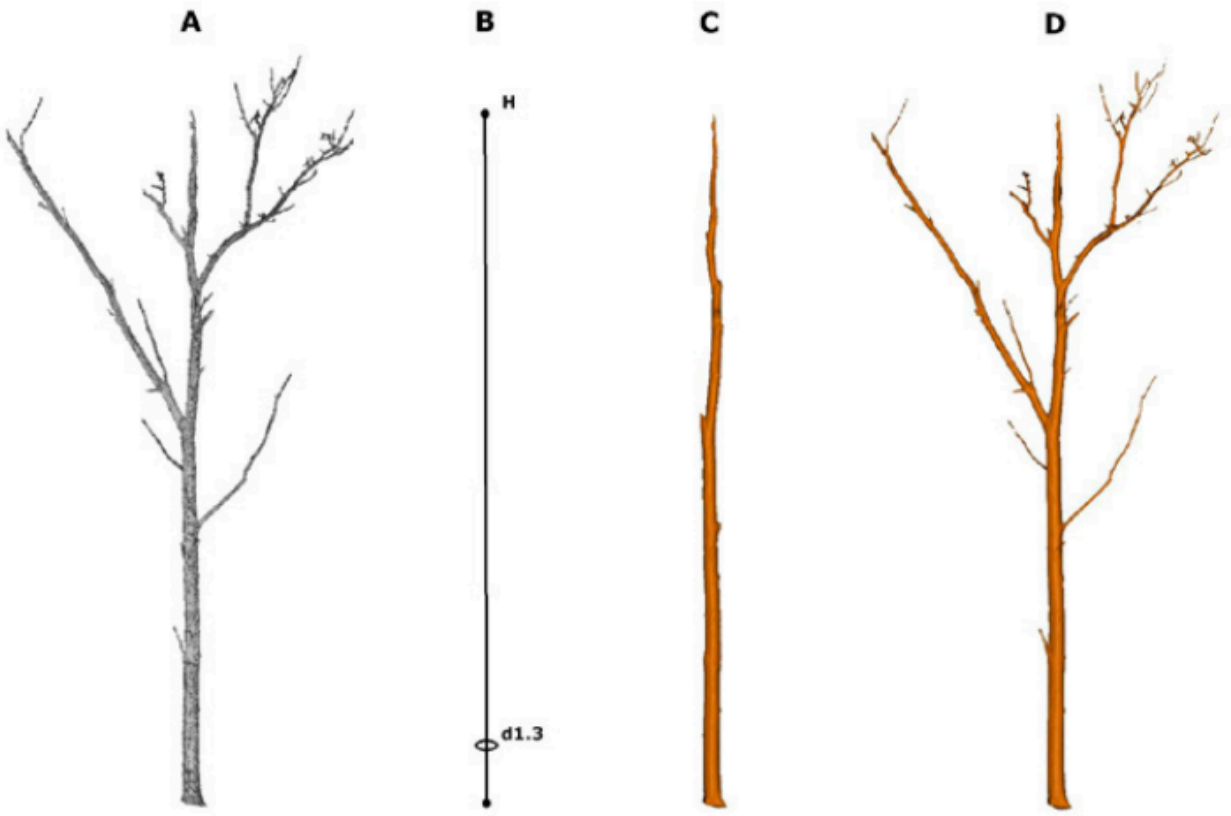


Figure 14: A. TLS point cloud, B. LOD 1 C. LoD 2 D. LoD 3 ***Source:*** (Krok et al., 2020)

suitable for timber framing applications. Part A addresses this gap by developing an algorithm that processes tree point clouds and identifies potential yield as geometric planar components.

Since wood quality is context-dependent rather than absolute (Nocetti & Brunetti, 2024), establishing structural applications for curved elements yields dual benefits: increasing economic value in material that conventional grading systems would downgrade, while enabling its use supports more localised supply chains, including urban forestry contexts (Kampelmann, 2021). Given this focus on curved element extraction, the research targets deciduous species such as oak and maple.

Comprehensive quality assessment requires internal evaluation of knot distribution, decay, structural defects, measurable through acoustic analysis, CT scanning, or microdrilling (Llana et al., 2020; Nasir et al., 2024; Pagella et al., 2025). These techniques enable sawing optimisation and timber grading (Nasir et al., 2024), (Sandvik et al., 2024), whilst automated multi-sensor grading systems employ machine learning to predict mechanical properties from wave propagation data (Nasir et al., 2024). AI-driven image analysis further enables surface defect detection and automated grade assignment (Shi et al., 2025).

Whilst these methods could enhance later KM0-DFAB stages through mechanical property-based optimisation, this research focuses exclusively on pre-harvest geometric assessment to inform design alternatives.

5.2 METHODOLOGY: DEVELOPMENT OF YIELD-ASSESSMENT ALGORITHM

This chapter presents a computational methodology for pre-harvest assessment of available timber stock through standing tree analysis. The algorithm processes LiDAR-derived point clouds to identify and quantify both straight and curved planar structural components suitable for timber framing applications.

Whilst conventional industrial workflows reject curved timber, this assessment explicitly identifies curved elements alongside straight components, enabling material inventory compilation before harvest. The algorithm outputs geometric catalogues and component metadata that directly inform the stock-constrained design framework presented in Part B, enabling material-responsive design decisions prior to extraction.

This pre-harvest assessment allows forest managers and designers to determine which tree sections can serve structural roles in timber framing, distinguishing load-bearing components from material destined for downgrading or alternative applications.

This methodology evaluates external geometric properties only. Internal quality assessment (knot distribution, strength grading) could enhance later workflow stages but remains outside scope.

Standing trees are scanned using terrestrial laser scanners (e.g., Faro Focus 120) (Stovall et al., 2018). The resulting point cloud is analysed to extract tree properties including dimensions, stem form, and branch characteristics (Figure 15).

Trees are well-suited to graph representation due to their hierarchical stem-branch structure. Points in close proximity within the point cloud likely belong to the same branch, enabling automatic skeleton extraction (Du et al., 2019). Whilst AdTree was originally developed for biomass and volume estimation, this research applies it to extract geometric properties for structural timber assessment.

The AdTree algorithm generates both geometric outputs and a structured `.ply` file representing the tree as a graph. In this representation, nodes correspond to points along the tree's centreline skeleton, with edges connecting nodes to form the complete branching structure. Each node contains spatial coordinates `(x, y, z)`, branch



Figure 15: from left to right: pointloud of tree, skeleton of tree, skeleton with local diamaters. Analysed by Adtree ***Source:*** (Du et al., 2019)

diameter at that location, and connectivity information specifying adjacent nodes. This graph-based centreline skeleton provides the foundation for subsequent segmentation into extractable structural components. The point clouds require translation to local coordinates (origin-centred) before processing to avoid numerical precision issues with large coordinate values.

Algorithm accuracy depends on point cloud quality, with typical geometric deviations around 10 cm (Du et al., 2019). This precision suits early-stage design assessment but may require refinement for detailed fabrication planning in later workflow stages, for example trough robotic log scanning (Vestartas & Weinand, 2020).

A Python-based algorithm systematically searches this graph representation for extractable structural components, both straight and curved.

The workflow proceeds through four main stages:

1. pre-processing
2. planar branch segmentation
3. structural role allocation
4. post-processing

To validate the algorithm and assess individual tree stock potential, a series of case studies on various tree specimens is presented.

5.2.1 PRE-PROCESSING

This part describes the pre-processing steps of the .ply file outputted by the AdTree algorithm.

diameter based filtering

Structural timber components require minimum cross-sectional dimensions to resist anticipated loads. Whilst internal wood properties (density, knot distribution, grain orientation, defects) determine strength classification and fall outside this research scope, external geometric dimensions provide an initial structural filter. A user-defined minimum diameter threshold filters the tree graph (Figure 16). Nodes below this threshold are removed, as insufficient cross-sections cannot serve as load-bearing components regardless of internal wood quality. Diameter thresholds should account for sapwood thickness to improve stock estimation accuracy.

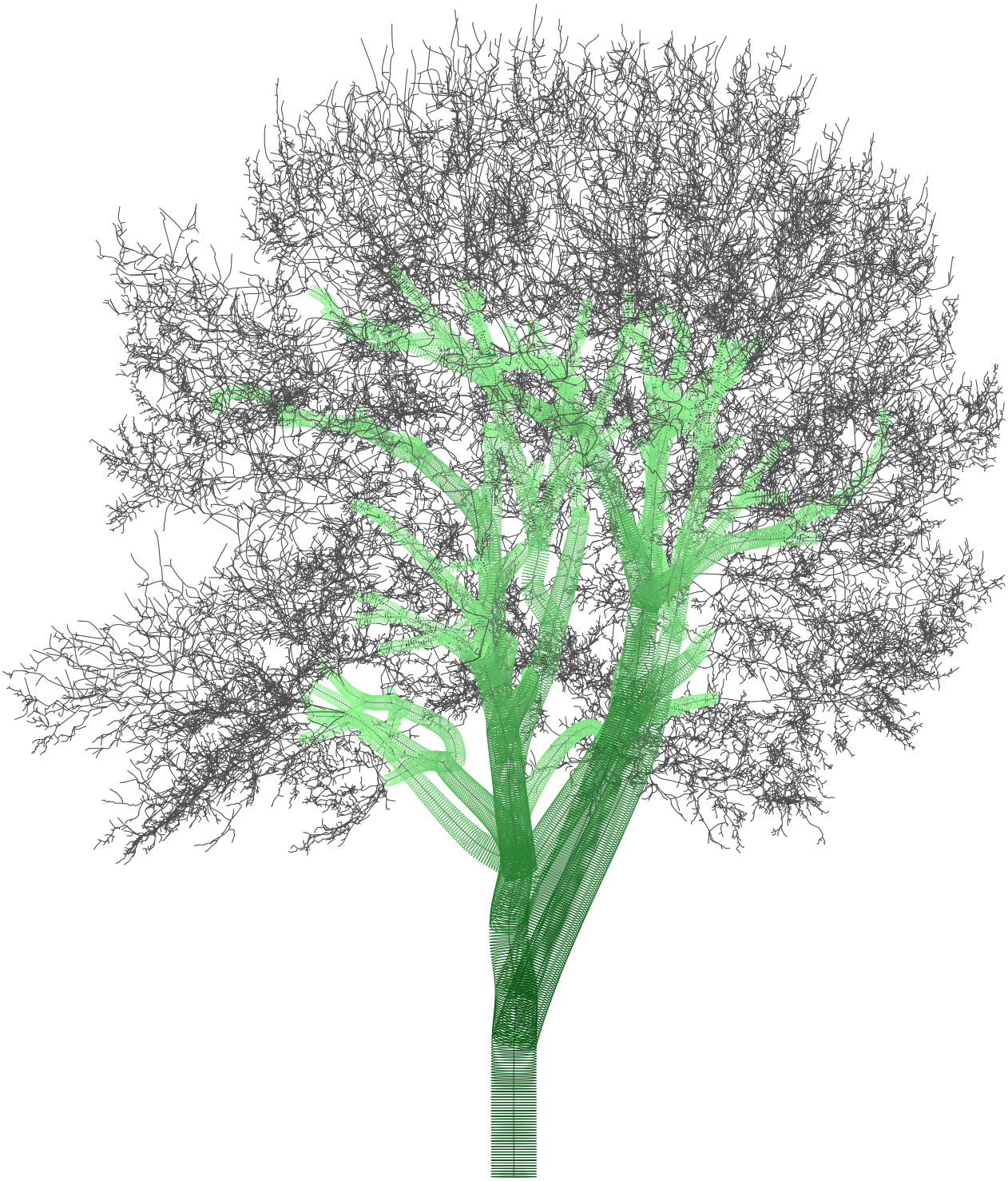


Figure 16: Tree skeleton filtered by diameter. ***Source:*** own

splitting at bifurcation

The tree graph is split at bifurcation points, producing linear branch segments. Whilst forks represent structurally valuable elements due to their inherent strength, they are not extracted as components in this workflow.This is is essential for deciduous species where multiple branches diverge from parent stems.

AdTree's graph output exhibited a limitation: bifurcations (nodes with degree >2) appeared as disconnected segments with spatially coincident endpoints rather than explicit connectivity. A preprocessing step merges spatially proximate endpoint nodes into single bifurcation nodes with proper connectivity (Figure 17).

This reconstruction enables accurate segmentation; without it, continuous branches would remain unsplit at fork points. NetworkX provides optimised graph algorithms for structure representation and traversal (Hagberg et al., 2008).

root-end trimming: The root end of each branch (where it joins a parent branch at a bifurcation) is trimmed proportionally to local diameter. The default factor (2× d_fork) requires empirical validation in practice. Terminal tips remain intact. This removes two geometric artefacts: inadequate centreline representation at junction diameter transitions, and overlapping centrelines from adjacent branches at the same fork (Figure 18).

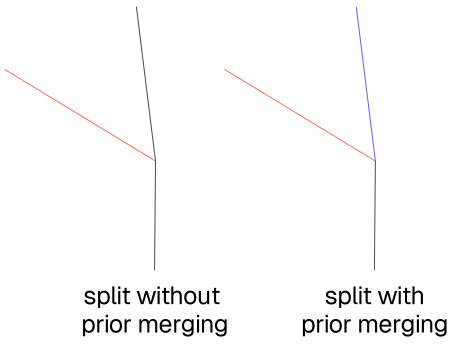


Figure 17: merging is necessary for recognising bifurcation in a graph. ***Source:*** own

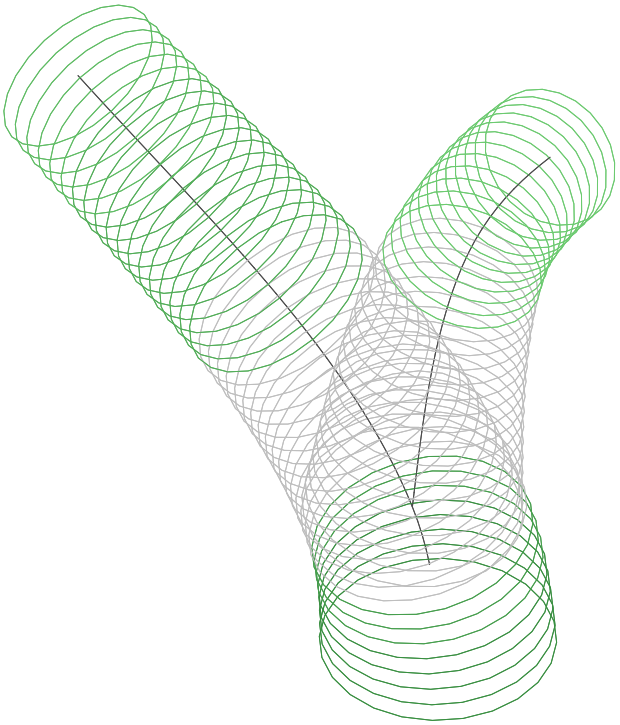


Figure 18: Overlapping volume at bifurcations is trimmed away. ***Source:*** own

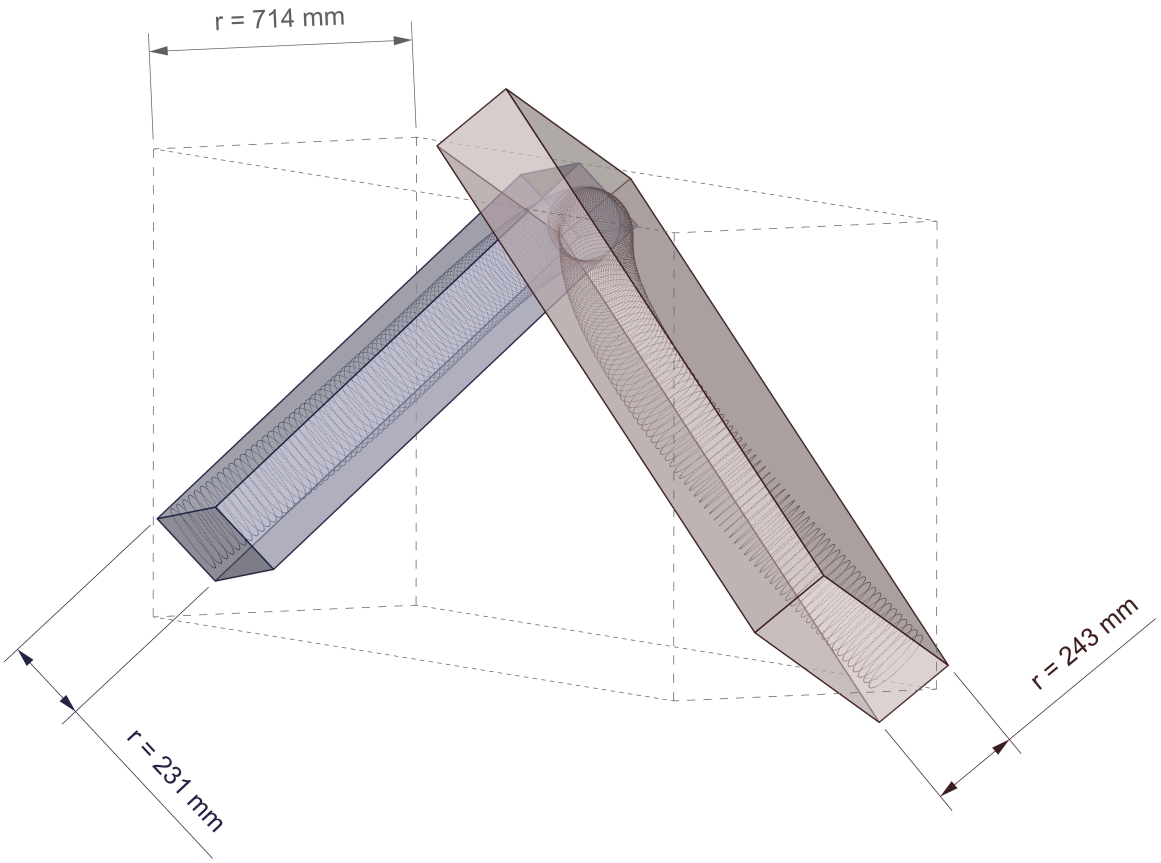


Figure 19: The algorithm segments branches into planar sub-components, maximising segment length whilst minimising planarity deviation. ***Source:*** own

5.2.2 PLANAR BRANCH SEGMENTATION

The reconstructed graph comprises unbranched and trimmed segments between bifurcation points (from preprocessing). Geometric analysis now identifies which segments are extractable as planar components, straight or curved.

planarity detection via SVD:

A curved branch in 3D space may twist through multiple planes. These doubly-curved branches must be subdivided into planar sub-segments, as structural timber for half-timbered walls must lie approximately in a single plane.

Singular Value Decomposition (SVD) finds the best-fit plane for any set of points, in this case

the nodes of a candidate sub-segment of a branch (Akritas & Malaschonok, 2004).

For a centred point matrix $\boldsymbol{X} \in \mathbb{R}^{n\times 3}$, where each row is a skeleton point with the segment centroid $\boldsymbol{c} = \frac{1}{n}\sum_i \boldsymbol{p}_i$ subtracted, SVD factorises:

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$$

where $\boldsymbol{V}^T \in \mathbb{R}^{3\times 3}$. The last row of $\boldsymbol{V}^T$, corresponding to the direction of minimum variance, yields the plane normal vector $\boldsymbol{n}$. Points lying in a plane exhibit near-zero variance perpendicular to that plane.

By construction, the best-fit plane passes through the centroid $\boldsymbol{c}$; each skeleton point $\boldsymbol{p}_i$ is therefore projected along the normal as $d_i = |(\boldsymbol{p}_i - \boldsymbol{c}) \cdot \boldsymbol{n}|$, giving its perpendicular distance to the fitted plane. The planarity residual is defined as $r = \max_i d_i$, the maximum perpendicular distance from any skeleton point to the fitted plane. A window is admissible when $r < 0.05$ m. In Figure 19, is visualised how a non-planar branch can be divided in planar elements.

Optimal segmentation via Dijkstra's algorithm:

Rather than arbitrarily splitting doubly-curved branches at fixed intervals into planar sub-segments, the segmentation is cast as a shortest-path problem and solved with Dijkstra's algorithm (Dijkstra, 1959) via the NetworkX package (Hagberg et al., 2008).

Each branch, represented as an ordered node sequence $[n_0, n_1, n_2, ..., n_k]$, generates a directed graph $G = (V, E)$ where:

- **Vertices** $V = \{n_0, n_1, ..., n_k\}$ represent skeleton nodes sequentially along the branch.
- **Edges** E represent candidate planar sub-segment. An edge connects non-adjacent nodes n_i and n_j (where $j \geq i + 2$) only if the nodes between them form a planar span—that is, the SVD residual $r_{i,j} < 0.05$ m for all points from n_i to n_j

Each edge (n_i, n_j) carries weight:

$$w_{i,j} = \frac{1}{(j-i)^2}$$

The weight is strictly subadditive and thus decreases as spans get longer. For any span of length $L = L_1 + L_2$, splitting it increases cost since $\frac{1}{L^2} < \frac{1}{L_1^2} + \frac{1}{L_2^2}$.

The algorithm searches for the minimum-cost path from n_0 to n_k through this graph. For example, a branch might be segmented as $n_0 \to n_2 \to n_7$ (two sub-segments) or $n_0 \to n_3 \to n_5 \to n_7$ (three sub-segments). The former is preferred because it has lower total cost: $w_{0,2} + w_{2,7} = \frac{1}{2^2} + \frac{1}{5^2} = 0.25 + 0.04 = 0.29$, whilst the three-edge path costs $w_{0,3} + w_{3,5} + w_{5,7} = \frac{1}{3^2} + \frac{1}{2^2} + \frac{1}{2^2} = 0.11 + 0.25 + 0.25 = 0.61$. Despite similar total node coverage, fewer longer spans cost less than many shorter spans.

The algorithm first identifies all possible planar windows within each branch. It tests every potential span between non-adjacent nodes to determine which satisfy the planarity threshold ($r < 0.05$ m). An early-exit heuristic stops extending a span once planarity is violated, as adding more points rarely restores it.

Dijkstra's algorithm then finds the minimum-cost path through these admissible windows, automatically producing the decomposition with the fewest, longest possible planar segments.

The output is that each branch is optimally decomposed into planar sub-segments at shared nodes, representing potentially extractable timber components.

Finally, all segments shorter than a user-defined threshold (e.g., 0.5 m) are removed, as short pieces rarely provide structural value.

5.2.3 STRUCTURAL ROLE ALLOCATION

With planar sub-segments identified, the algorithm categorised the components by structural role:

- **Primary structural elements**: Long, straight pieces suitable for load-bearing frame members
- **Curved bracing**: Curved planar elements for diagonal bracing
- **Straight bracing**: Shorter straight elements for infill or secondary bracing

Centerline smoothing:

Before curvature assessment, each planar sub-segment produced by the Dijkstra decomposition has its centreline spatially smoothed by a 200 mm arc-length moving average, following the methodology of (Gorges et al., 2021). This removes digitisation noise from the skeleton-extraction from the pointcloud.

Role definition: Primary structural roles are defined in a configuration file specifying role name, required length, and target quantity. For example:

- 2 beams of 4.0 m
- 4 posts of 3.0 m

Roles are sorted by descending length, prioritising the longest (most constrained) requirements first, as these are typically most difficult to satisfy from available stock.

Sliding window greedy allocation assessment:

For each role, a sliding window matching the required length slides along each smoothed centreline in increments of 5 cm. At each position, bow-height curvature is computed following (Gorges et al., 2021). Windows exceeding the maximum allowable curvature threshold are rejected; remaining candidates receive a curvature score where 0 represents perfectly straight material.

An optional penalty discourages extracting components from branch centres. Cuts near branch ends minimise waste by leaving single long offcuts suitable for future extraction, whilst central cuts produce two shorter fragments of limited utility. The penalty scales linearly from 1.0 at branch ends to a user-defined value at the midpoint:

$$\text{penalty} = 1.0 + (\text{penalty}_{\text{factor}} - 1.0) \cdot \text{rel}_{\text{pos}}$$

$$\text{local_score} = \text{curvature} \cdot \text{penalty}$$

where $\text{rel}_{\text{pos}} \in [0, 1]$ represents relative position from end to centre. Setting the $\text{penalty}_{\text{factor}}$ to 1.0 disables this feature; values such as 1.1 penalise central cuts by 10%.

All candidate windows across all sub-segments are collected. Within each sub-segment, the best window is selected by score (curvature × penalty), but global ranking across sub-segments uses raw bow-height curvature alone. The globally straightest window is allocated to the current role; the corresponding sub-segment is split at window boundaries, producing up to two remnants that re-enter the available material pool. This process repeats until the role's target quantity is satisfied or no valid material remains .

After all straight roles are filled, remaining sub-segments longer than a minimum threshold are classified by bow-height curvature: those below the maximum curvature threshold are retained as straight leftovers (potentially usable for shorter straight roles), whilst the rest are exported as curved structural components for the Part B design framework.

Each allocated component carries metadata identifying its parent branch and source tree, enabling multi-tree analysis. Components from

all analysed trees are merged into a unified stock inventory for subsequent processing.

5.2.4 ALGORITHM OUTPUTS

The algorithm generates standardised material passports for each analysed tree, documenting:

- Structured data: Geometry files (OBJ format) with component catalogues, and metadata tables (CSV) documenting arc length, diameters, taper rates, curvature values, planarity residuals, role assignments, and parent branch/tree identifiers
- Quantitative metrics: Total yield (cumulative stock length), yield rate (ratio usable stock relative to straight stock potential), structural role fulfilment, curvature and length distributions

These outputs enable stock assessment and integrate directly into the Part B design workflow.

5.3 CASE STUDIES

To validate the algorithm and assess individual tree stock potential, three specimens were analysed: one maple and two oak trees. This selection evaluates species-agnostic functionality whilst demonstrating variation between individuals of the same species.

analysis parameters

The following settings were applied:

- Minimum diameter threshold: 0.2 m (both segment endpoints)
- Maximum planarity residual: 0.05 m (maximum out-of-plane deviation)
- Structural role requirements (4 m × 3 m wall): 2× 4.0 m beams, 2× 3.0 m posts

dataset sources Tree specimens were sourced from:

- **Acer pseudoplatanus** and **Quercus petraea** from PyTreeDB (Weiser et al., 2022)
- **Quercus robur** from BioDiv3D (Griese et al., 2025)

data pre-processing

Point cloud files (.laz) were converted to .xyz format and translated to the origin to avoid numerical precision issues with large real-world coordinates. AdTree extracted skeletal structures from all specimens (Du et al., 2019). Visual overlay of extracted skeletons onto source point clouds revealed some discrepancies, though all specimens yielded usable skeletal representations for algorithm testing.

material passport as output

For each tree, the algorithm generates a standardised material passport documenting:

Quantitative Metrics:

- Total yield: cumulative linear length of extractable stock
- Yield rate: potential usable stock relative to potential straight stock
- Structural role fulfilment: count of allocated predefined elements
- Curvature distribution: histogram of element bow-height values
- Length distribution: histogram of extractable component lengths

Visual Documentation:

- Point cloud overlays with identified elements
- Element catalogue: side-view projections with cross-sectional profiles (tilted ellipses indicate local non-planarity)
- Curvature visualisations via circle fitting on curved centrelines

Reported yields represent theoretical upper bounds. Actual usable stock would be reduced by structural defects (rot, splits, damage), knots at branch junctions, and dimensional variability exceeding tolerance thresholds. Values should be interpreted as indicative potential rather than guaranteed outputs.

5.4 RESULTS

The material passports for each specimen are shown below.

5.5 DISCUSSION

Cross-specimen comparison reveals substantial variation in stock potential (Figure 20). Among the analysed specimens, Quercus petraea demonstrated superior performance across all metrics: highest total yield, highest yield rate, complete fulfilment of all structural role requirements, and reduced element curvature compared to other specimens (Figure 21).

These comparative results demonstrate the algorithm's ability to inform tree selection in harvesting operations, enabling selection criteria based on specific design requirements. For instance, specimens with minimal curvature can be prioritised for designs requiring straight elements.

Including curved elements increased yields by 19.1% to 186.3% compared to straight-only harvesting, demonstrating substantial material efficiency gains from curved structural element incorporation.

Per sample, tree analysis completed in approximately 15 seconds, demonstrating practical computational efficiency. The identified stock from each analysed tree feeds directly into the design framework presented in Part B, of which a complete workflow demonstration is presented in chapter 7.

Several limitations should be acknowledged. The sample of three deciduous specimens is insufficient for broad generalisation, and results were only evaluated against a single wall configuration (4.0 m × 3.0 m). Skeletal extraction is dependent on both AdTree's accuracy, which is an approximation, and input point cloud quality (Du et al., 2019). Diameter thresholds were applied to outer dimensions without accounting for sapwood thickness (Larsen et al., 2022b), overestimating usable stock. The planarity residual remains an approximation, and real-world yields would be further reduced by structural defects, knots, and splits not captured in current scan data. Finally, the algorithm does not account for the structural properties of elements, including material strength and reaction wood behaviour in curved members (Nocetti & Brunetti, 2024), which would require internal assessment data and structural verification before deployment.

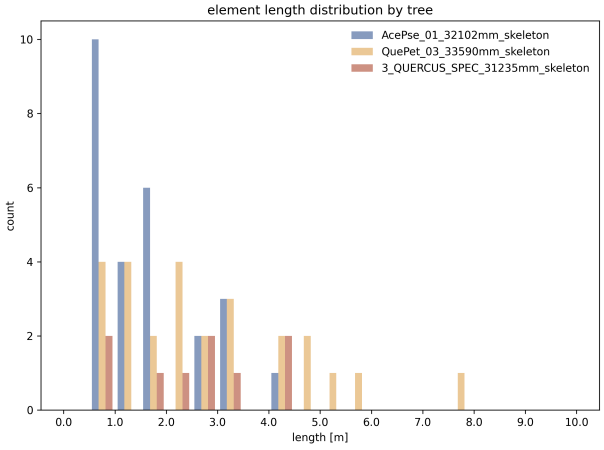


Figure 20: Curvature distribution by specimen. *Source:* own

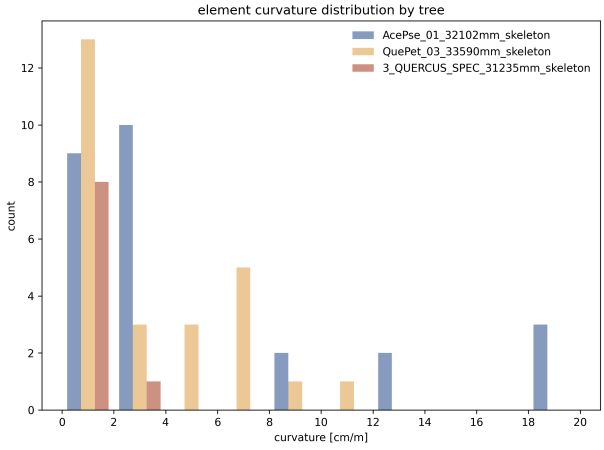


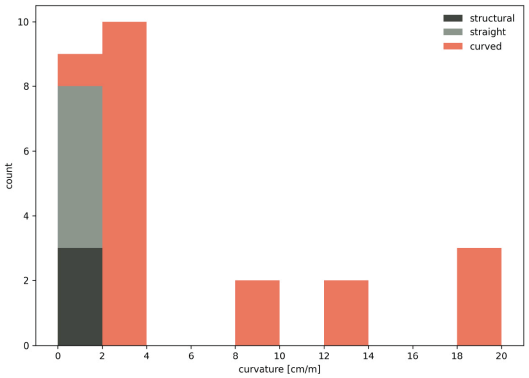
Figure 21: Curvature distribution by specimen. *Source:* own

Acer pseudoplatanus (Sycamore maple)

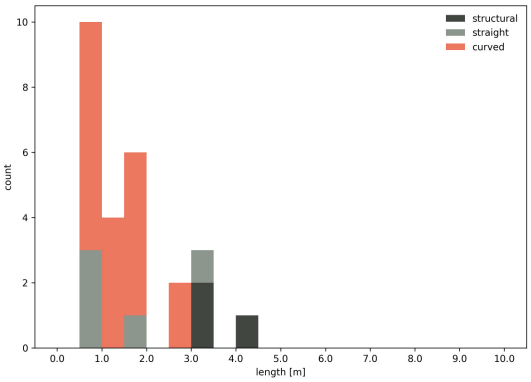
N_{usable}: **31**
total yield: L = **43.3 m**
yield increase rate: **2.447 (+144.7 %)**

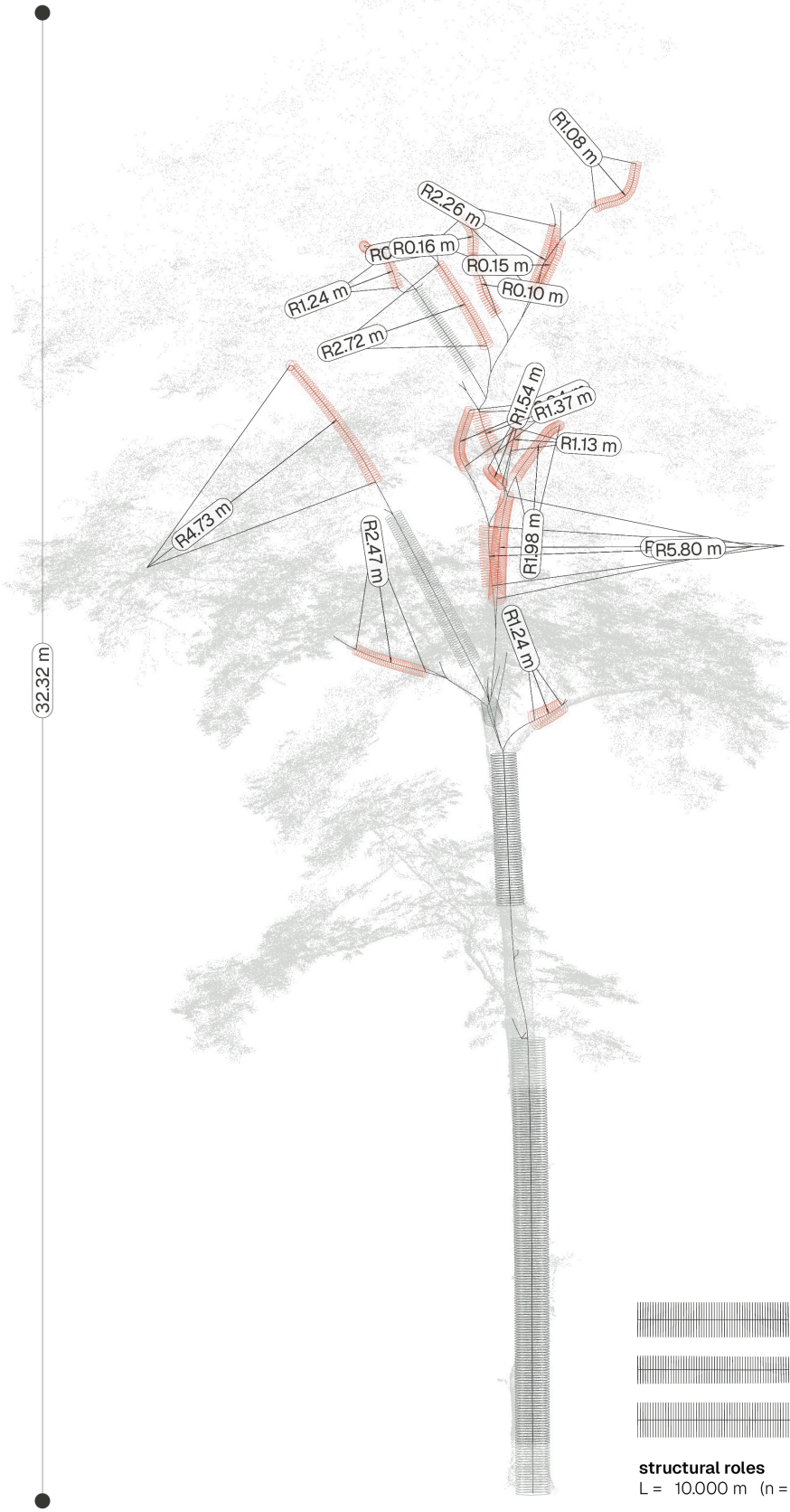
structural role targets:
2x beam 4.0 m: **1/2 found**
2x post 3.0 m: **2/2 found**

element curvature distribtion



element length distribtion







structural roles
L = 10.000 m (n = 3)

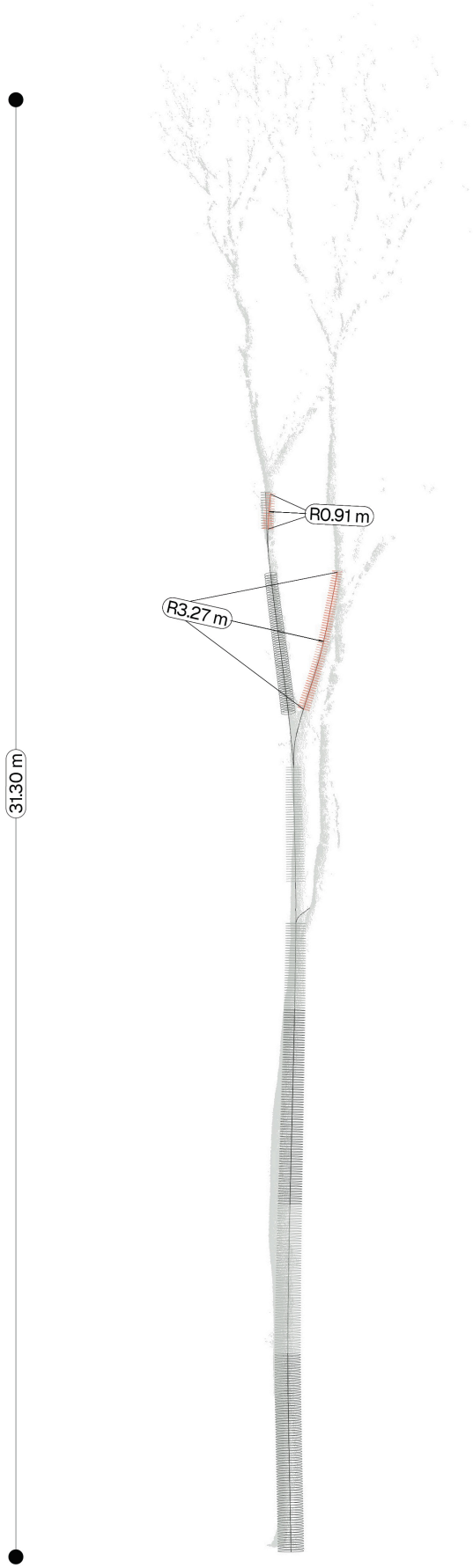
straight elements
L = 7.7 m (n = 5)

curved elements
L = 25.6 m (n = 18)

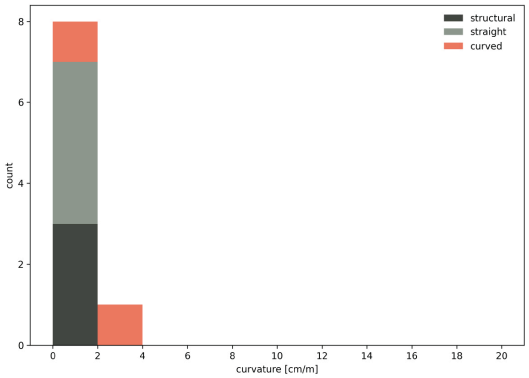
Quercus robur (English oak)

N_{usable} : **6**
total yield: L = **22.3 m**
yield increase rate: **1.191 (+19.1 %)**

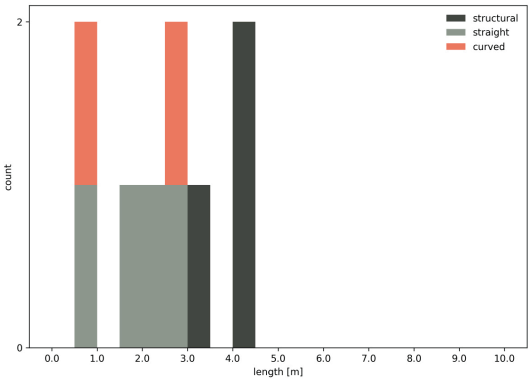
structural role targets:
2x beam 4.0 m: **2/2 found**
2x post 3.0 m: **1/2 found**

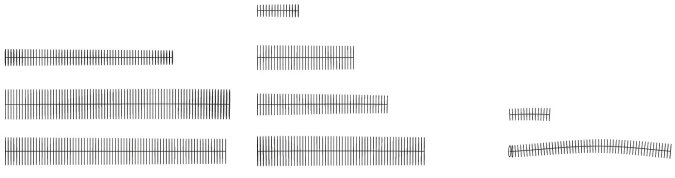


element curvature distribtion



element length distribtion





structural roles
L = 11.0 m (n = 3)

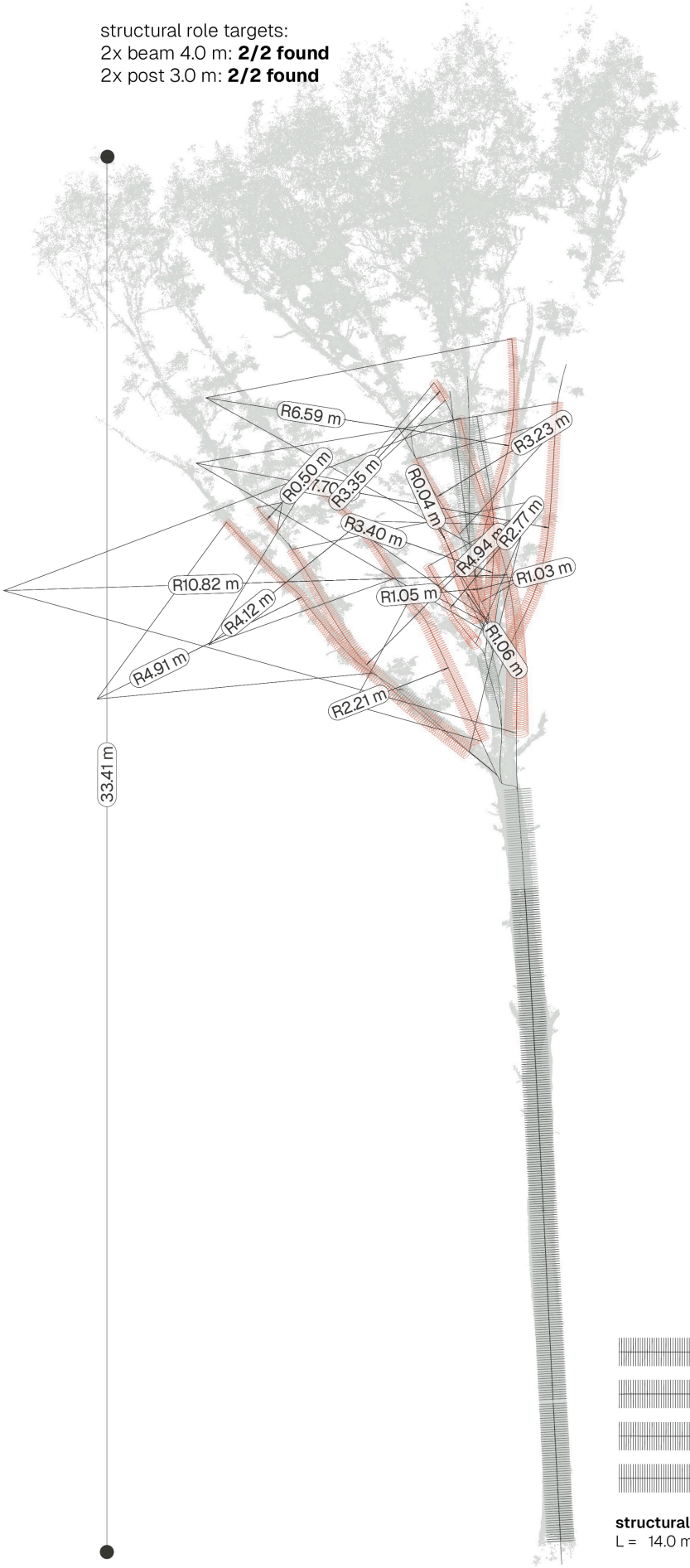
straight elements
L = 7.7 m (n = 2)

curved elements
L = 3.6 m (n = 2)

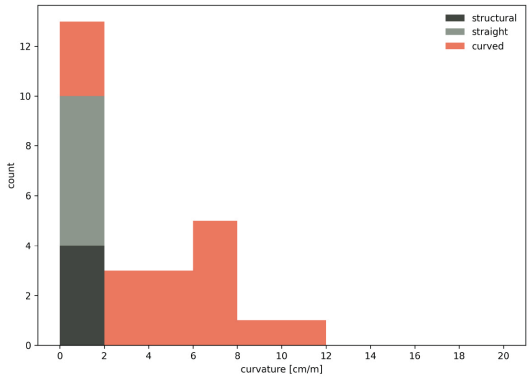
Quercus petraea (Sessile oak)

N_{usable}: **30**
total yield: L = **72.61 m**
yield increase rate: **2.863 (+186.3 %)**

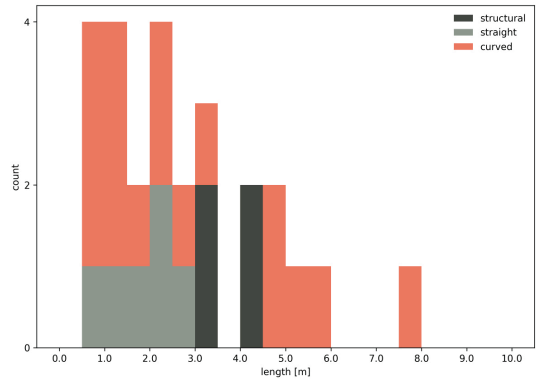
structural role targets:
2x beam 4.0 m: **2/2 found**
2x post 3.0 m: **2/2 found**

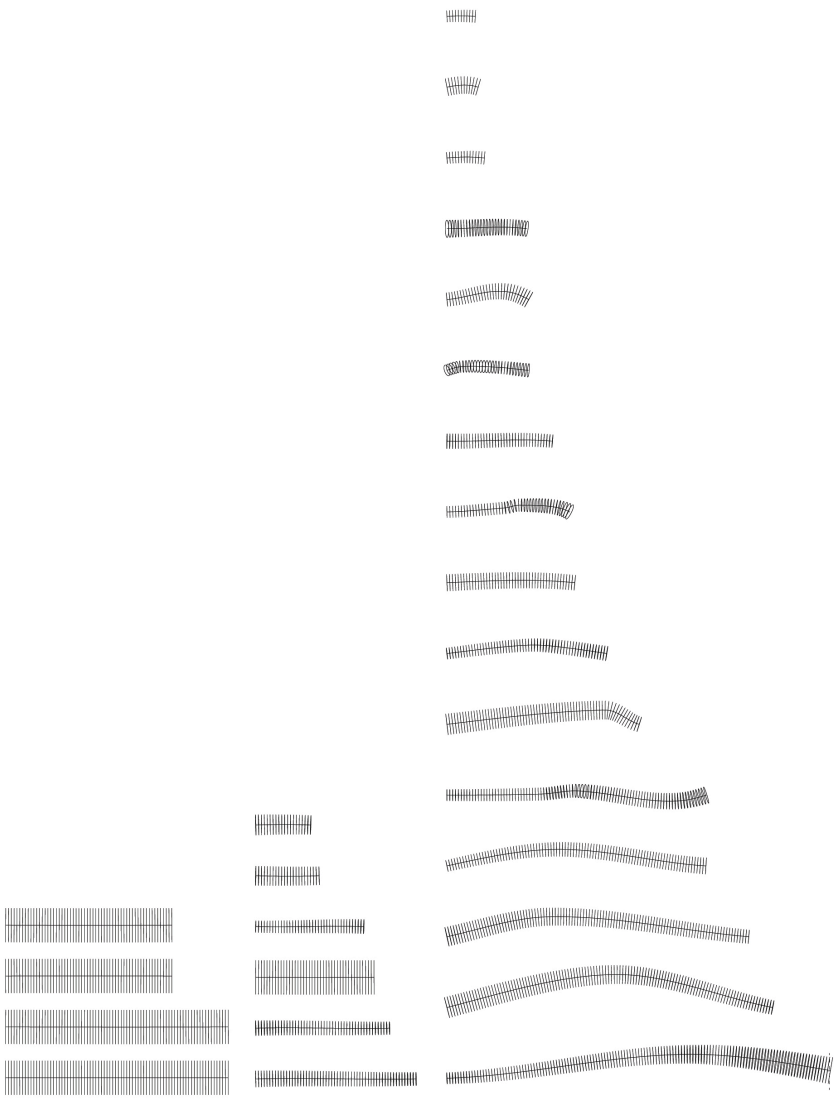


element curvature distribtion



element length distribtion





structural roles
L = 14.0 m (n = 4)

straight elements
L = 11.3 m (n = 6)

curved elements
L = 47.2 m (n = 16)



Figure 22: Irregular components used in the construction of the Sülchenkirche, Rottenburg am Neckar . ***Source:*** Own

6 PART B - STOCK-CONSTRAINED DESIGN FOR HALF-TIMBERED WALL ELEMENTS WITH IRREGULAR COMPONENTS

6.1 STATUS

Having established methods for geometry based pre-harvest material inventory assessment, this chapter addresses the challenge of designing with heterogeneous timber stocks in structural applications.

Historically, reliance on locally sourced timber necessitated design strategies accommodating material heterogeneity by adapting structural configurations to available stock (Engelmann, 2017), (Meeson, 2021). Whilst industrialisation displaced these manual “stock-constrained” practices in favour of dimensional lumber, contemporary computational methods enable efficient stock-constrained design at computational speeds through varying approaches, including geometric representations such as curves, meshes, or discrete elements, and computational methods ranging from heuristics to machine learning,(Sun et al., 2026).

For instance, The Tree Fork Truss project at Hooke Park demonstrated computational stock-constrained design using strictly locally-sourced naturally forked timber elements. After digitalisation of the material inventory, a Vierendeel-truss was designend using genetic algorithms. The algorithm matched centerlines of bifurcated logs to predefined truss chords. The objectives of the optimisation were to align contact points and collision avoidance. Early optimisation approaches using `Galapagos` could only process forks sequentially, preventing global reconsideration of previously placed elements. To address this limitation, a final script was developed that simultaneously optimised all 25 forks across the entire truss. This computational process required a week to generate seventeen viable arrangements. The search space is shown in Figure 23. A physical demonstrator provied the concept (Mollica, 2016a).

Similarly, Rahbek et al. applied genetic algorithms for the design of a gridshell using irregular logs. Employing `SPEA2` with permutation encoding to assign curved logs to predefined curves representing structural members, as shown in Figure 24. Their Team-Based Repair algorithm addressed feasibility constraints by dynamically swapping unfeasible placements with feasible logs from idle pools, increasing solution validity from near-zero to 75%. The framework was validated on a 186-member arch gridshell with robotic fabrication verification (Rahbek et al., 2023).

In contrast, this research explores genetic algorithms for generative “bottom-up” design, where neither member count nor configuration is predetermined. In this sequential placement problem, the algorithm must simultaneously determine how many branches to place and where to place them, constrained only by frame dimensions and structural performance requirements.

Such approaches, where design is defined solely by performance objectives rather than predefined topologies, have been successfully addressed using reinforcement learning (Mirra & Mueller, 2025). However, RL implementation requires substantial training infrastructure and domain-specific reward engineering. This thesis therefore explores whether multi-objective genetic algorithms can meaningfully guide generative design processes, or whether sequential placement problems fundamentally require state-based optimisation methods.

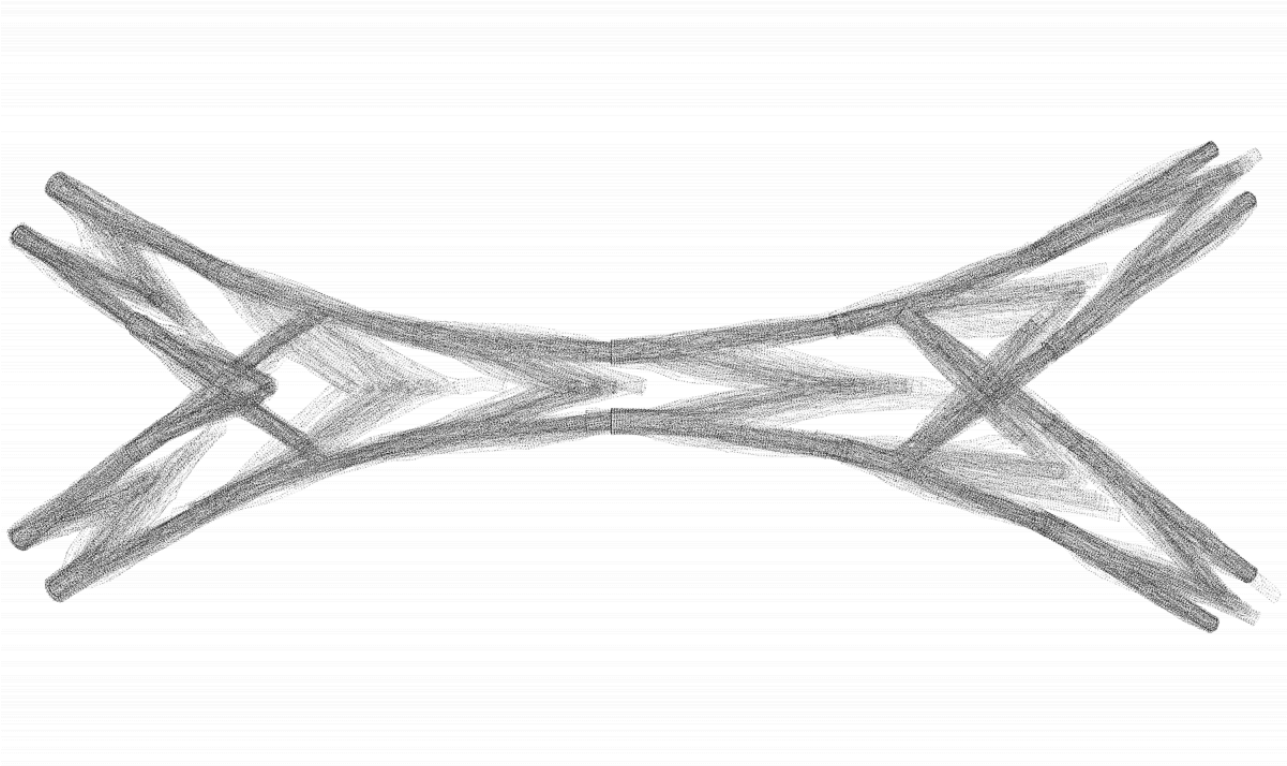


Figure 23: Iterations of Tree Fork Truss optimisation showing how predefined topology has defined the search space.. ***Source:*** (Mollica, 2016b)

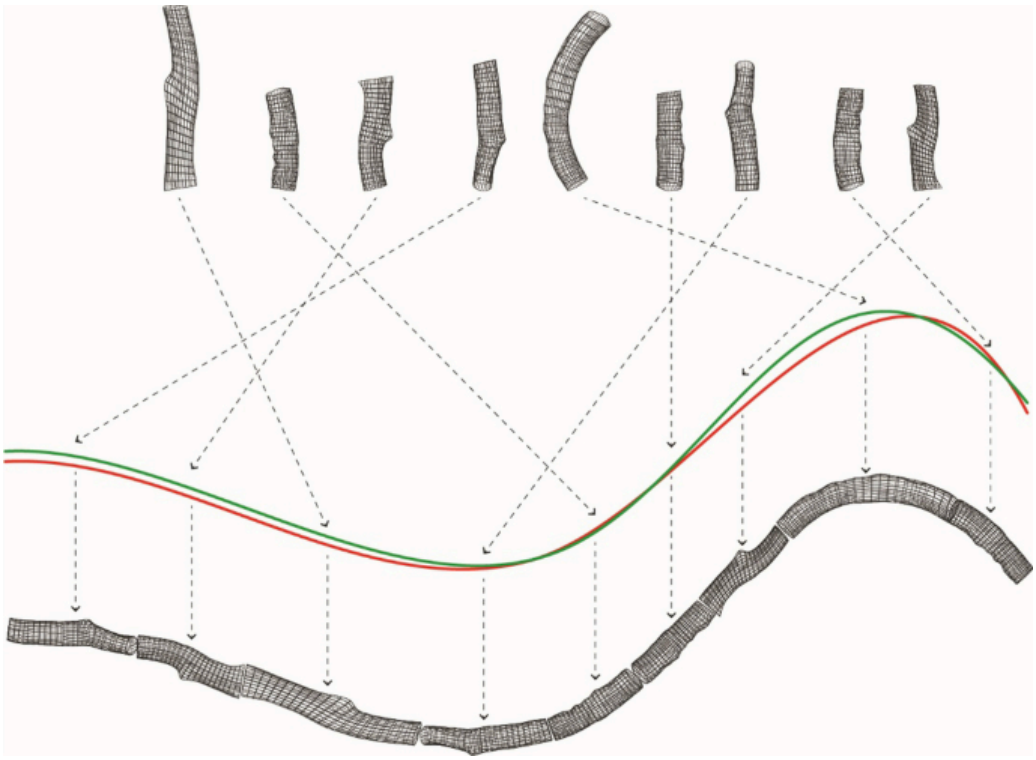


Figure 24: Matching curved logs with predefined curves in the structure. ***Source:*** (Rahbek et al., 2023)

6.2 METHODOLOGY

The previous paragraph examined the application of genetic algorithms for design trusses and gridshells using timber stocks. This thesis addresses a related but distinct challenge: generative design of half-timbered wall frames stiffened by a minimised amount of irregular components sourced from the inventory established in Part A.

The proposed computational framework comprises two components: (1) a rule-based design game environment, and (2) a genetic algorithm agent that plays the game by evolving placement strategies.

The design game logic is the core of the Digital Fachwerk methodolgy, inverting the conventional design workflow where material conforms to predetermined structural layouts. The design game can defined as followed:

The design game begins with a frame as a single undivided cell.

At each game step, the designer selects a cell and the longest element from the inventory. The element is positioned to intersect at least two edges of the target cell, subdividing it into two smaller cells. The placed element is trimmed to the cell boundaries, with offcuts exceeding the minimum length returned to the inventory. The deflections (under design loads) of the configuration are evaluated through finite element analysis. This process repeats until stock is depleted or structural performance requirements are satisfied.

The design objective is to place elements strategically such that structural deflection is minimised using the least possible material.

Important to note is that With that placement of one branch, infleunces the posibilities of the placement of the consequent placements. This means that the design space changes with every succesful placement.

Critically, each placement alters the design space for subsequent decisions. By subdividing cells and consuming stock, every action constrains future possibilities, creating path-dependent design trajectories where placement order determines achievable outcomes.

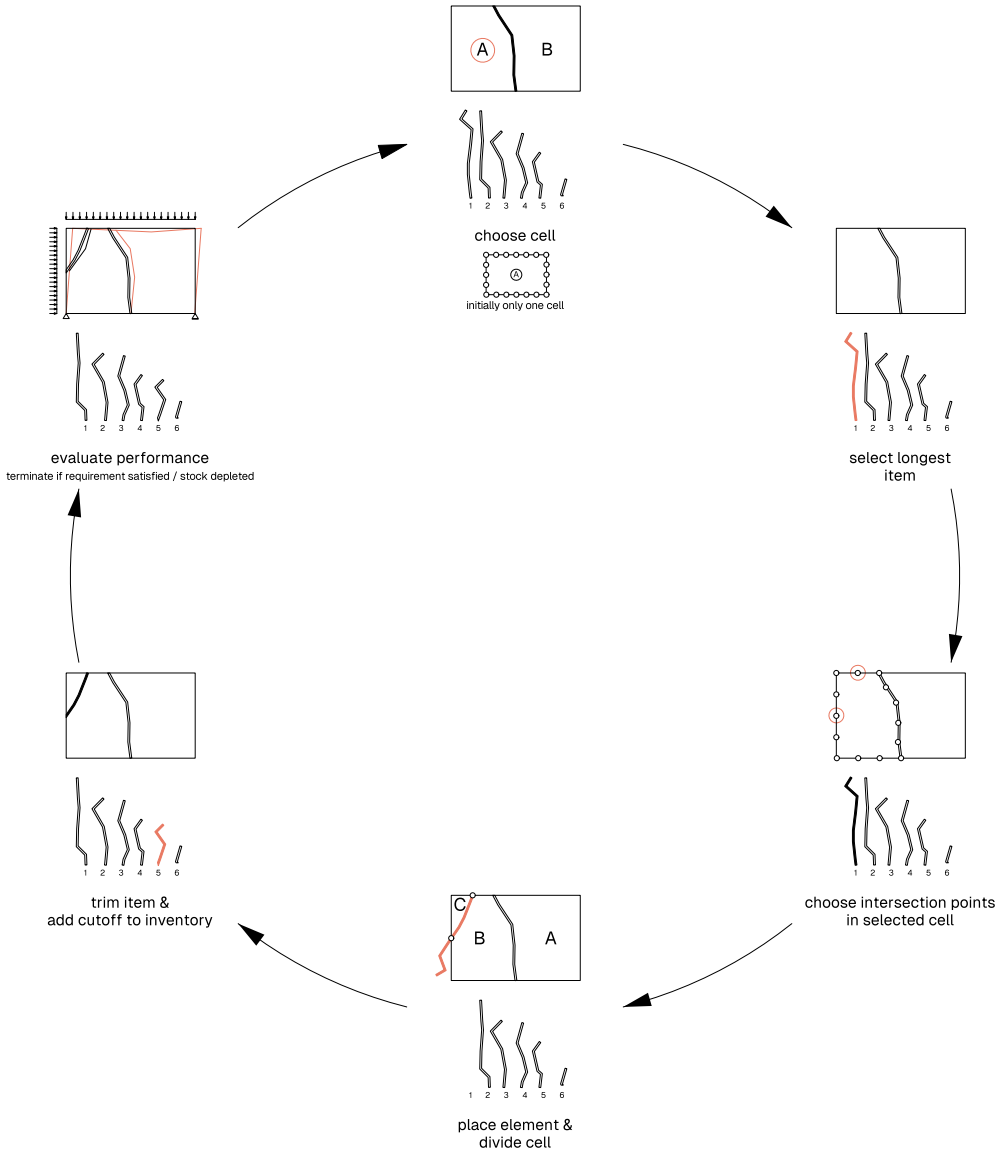


Figure 25: initiation: one cell sized with chosen dimensions

1. choose a target cell within the current panel subdivision
2. the longest available element is selected from inventory
3. choose two points on the boundary of the target cell
4. if placement is valid, the target cell is divided by element
5. placed element is trimmed to cell boundaries; offcuts exceeding minimum length return to inventory
6. assesment of structural performance (deflection under design loads) via finite element analysis

termination: if structural requirements are satisfied / stock is depleted

Source: own

6.2.1 DEVELOPMENT OF DESIGN GAME ENVIRONMENT

This section describes the development of the design game environment, proceeding from the physical design game, through its digital encoding, to the setup of an optimisation algorithm for searching optimal designs.

physical fachwerk design game

whilst the primary design objectives, minimizing material use and minimising deflections, are clearly defined, the specific placement strategies and constraints remain open to interpretation. Numerous valid rule sets could govern the placement process.

To develop placement rules informed by design intuition, a physical modeling study was conducted using processed sticks representing the design proces of half-timber framed wall elements with irregular timber stock. The model is shown in Figure 26.

Next to the placement logic, this analog exploration revealed fundamental constraints:

- element orientation produces distinct outcomes even when boundary points are identical (flipped vs. unflipped positioning);
- boundary point pairs must be distinct (selecting the same point twice produces no span);
- and the distance between chosen points must not exceed the selected element's length.
- elements parallel to an earlier placed element are considered illegal.

The physical design game enabled rapid evaluation of placement sequences, quickly revealing which strategies produced effective configurations.

Whilst the scaled elements cannot accurately represent structural behavior due to scale effects and simplified geometry, the models provided insight into visual design qualities. Physical exploration revealed that highly asymmetric configurations, whilst potentially material-efficient, produced visually unbalanced compositions. This observation informed a design decision: the genetic algorithm should incorporate a symmetry bias to improve visual coherence. This bias is elaborated in the sub-paragraph about the FEA for the optimisation with genetic algorithms.

Since the physical model does it is not allow for accurate verifiifcation of the performance of a design a digital environment was developed. This digital model is elaborated on in the next sub-paragraph.



Figure 26: Physical design game. ***Source:*** own

digital fachwerk

To explore stock-constrained design strategies with irregular timber, an interactive placement tool was developed in `GHPython` within `Grasshopper/Rhino`. This tool enables manual assembly of half-timbered wall configurations, for instance using the geometric inventory identified in Part A.

The digital implementation employs the same recursive cell-subdivision logic as the physical model, but requires encoding placement decisions as discrete computational parameters. To accommodate genetic algorithm optimisation (and potential future reinforcement learning approaches), this encoding must provide fixed dimensionality across all game states, meaning that each placement step should offer the same number of selectable parameters, even as actual geometric possibilities vary.

Beginning with a rectangular frame boundary representing the outer wall dimensions, the designer selects target cells via a 7x7 grid overlay. At each step, one of the 49 grid points is selected. The selected point acts like a “cursor”, since the cell containing that point becomes the target cell for placement. This grid-based selection maintains constant dimensionality regardless of how many cells currently exist. The grid is visible in Figure 27.

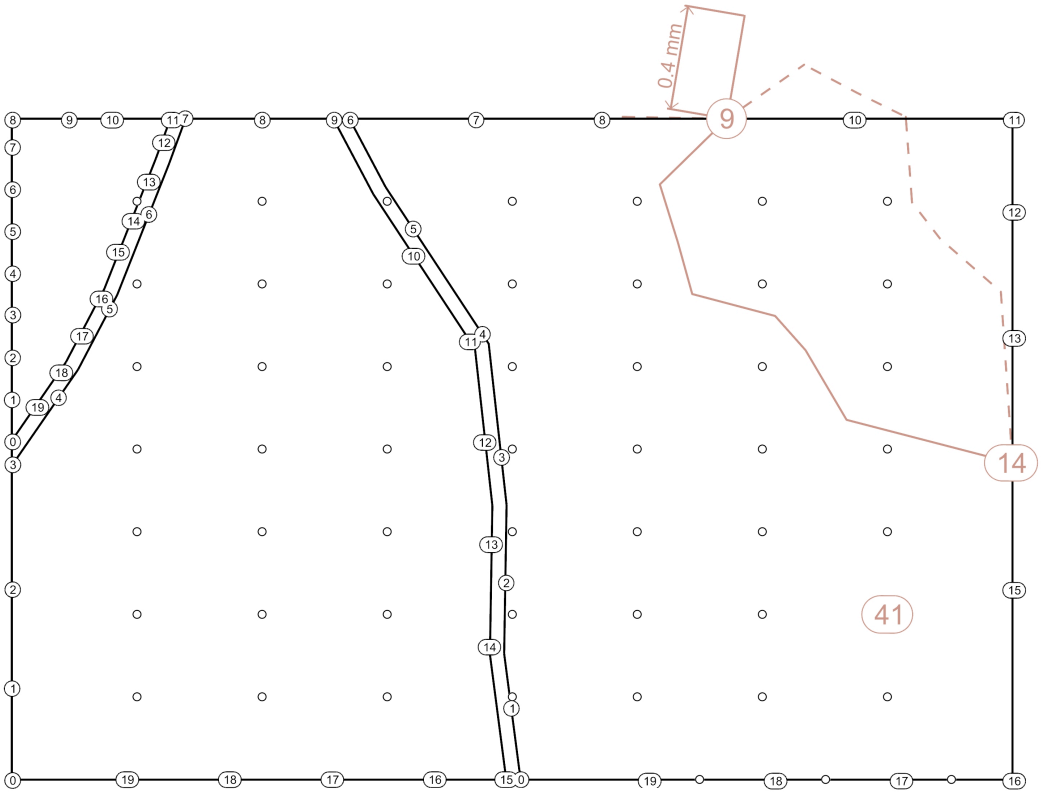
To maintain fixed decision dimensionality, each cell boundary is resampled into 20 equally spaced points, preserving the location of intersection points. Indexing begins at the bottom-left corner and proceeds clockwise, providing a degree of spatial consistency across cell geometries. The algorithm thus selects from exactly 20 boundary points at every placement step, regardless of cell complexity. The boundary points are shown in Figure 27. After point selection, an endpoint of the longest available branch is translated to the first point, rotated to align its chord toward the second point, and optionally mirrored. After placement, the branch is trimmed to the cell boundary as the cell splits into two sub-polygons along the placed element, creating new placement opportunities. Segments exceeding 0.5 m re-enter the available stock sorted by descending length, enabling usage of material leftovers.

Structural validation is performed via Karamba3D finite element analysis after each placement, evaluating the evolving frame’s load-bearing capacity under user-defined loading conditions. This immediate feedback loop enables designers to iteratively refine configurations, assessing the structural implications of each branch position before proceeding. The tool thus functions as both a design interface and an experimental platform for understanding how geometric constraints of irregular stock influence structural topology in timber-framing applications.

Placements are validated against the earlier defined geometric constraints:

- the branch chord must at least be longer then the distance the two chosen points.
- Both resulting sub-cells must exceed a minimum area threshold ($> 0.5\mathrm{m}^2$)
- the trimmed branch must only contain connectivity to the cell exterior at both endpoints. Mid-branch intersection are illegal.

All tunable parameters are listed in the appendix in Table 4, covering frame geometry and grid resolution, placement heuristics, cross-section dimensions, applied loads, and timber material properties. Structural values follow relatively low-strength hardwood as a conservative assumption for heterogeneous natural oak.



current selection
grid: **41**
point A: **14**
point B: **19**
flip: **false**
offcut > 0.5 m: **false**

Figure 27: Fixed dimensionality across all game states. ***Source:*** own

6.3 GENETIC OPTIMISATION FOR DIGITAL FACHWERK

Whilst the interactive interface enables manual exploratorion of various designs, systematic evaluation of the design space requires automated optimisation. To this end, the placement logic was adapted into a stateless replay component compatible with multi-objective genetic algorithm optimisation, similarly to approaches employed by (Mollica, 2016a) and (Rahbek et al., 2023).

Initially, this optimisation algorithm was set-up with multi-objective optimisation plugin `Octopus` for `Grasshopper` in `Rhino` with FEA performed with `Karamba`. Eventually the algorithm migrated to fully python using `PyMoo` package for the optimisation and `OpenSeesPy` for the FEA. This not only allowed for faster compute & batch-runs for testing various settings, but also allows for custom visualisation UI's through `Shapely`, `PyPlot` and `Dash`. On top of that this makes the optimisation 100% open-source, making it scalable.

FEA for the evolutionary optimisation methods

The geometric layout is converted into a beam-element structural model with nodes at frame corners, branch endpoints (both pinned), and intermediate polyline vertices (rigid, for curved element representation). Coincident points within 1 mm tolerance are merged for connectivity. Pinned connections provide a conservative simplification of real timber joints, which possess partial moment capacity. A permanent top beam midspan node captures deflection regardless of branch arrangement. The model is pinned at both lower frame corners, representing a simply supported base condition.

Frame and branch elements are subdivided at intersection points where branch endpoints coincide with frame edges or other branch centrelines, ensuring continuous load transfer paths throughout the structural model. The element cross-sectional and material properties are according to the table in Table 4.

Gravity loading (25 kN uniformly = 6.25 kN/m for 4 meter) is applied as line loading to the top beam.

Wind pressure is simulated through horizontal line loads applied to the windward column, with corresponding suction on the leeward column. Wind analysis is conducted bi-directionally, retaining the worst-case displacement and utilisation across both loading directions for fitness evaluation. This ensures adequate performance regardless of wind direction whilst implicitly biasing optimisation toward geometric symmetry.

Using the OpenSeesPy finite element solver in 2D space, displacements and utilisation is calculated.

For each element, axial force N and governing end moment M are extracted from the element force vector. The utilisation ratio is computed as:

– Compression + bending:

$$\left(\frac{\sigma_n}{f_{c,0,k}}\right)^2 + \frac{\sigma_m}{f_{m,k}} \leq 1.0$$

– Tension + bending:

$$\frac{\sigma_n}{f_{t,0,k}} + \frac{\sigma_m}{f_{m,k}} \leq 1.0$$

where

$$\sigma_n = \frac{|N|}{A} \quad \text{and} \quad \sigma_m = \frac{M}{W_z} = \frac{M}{\frac{w \cdot d^2}{6}}$$

The maximum utilisation across all elements in both wind directions is the governing value. Solutions with $\max_{\text{utilisation}} > 1.0$ are structurally infeasible and penalised accordingly by the optimiser.

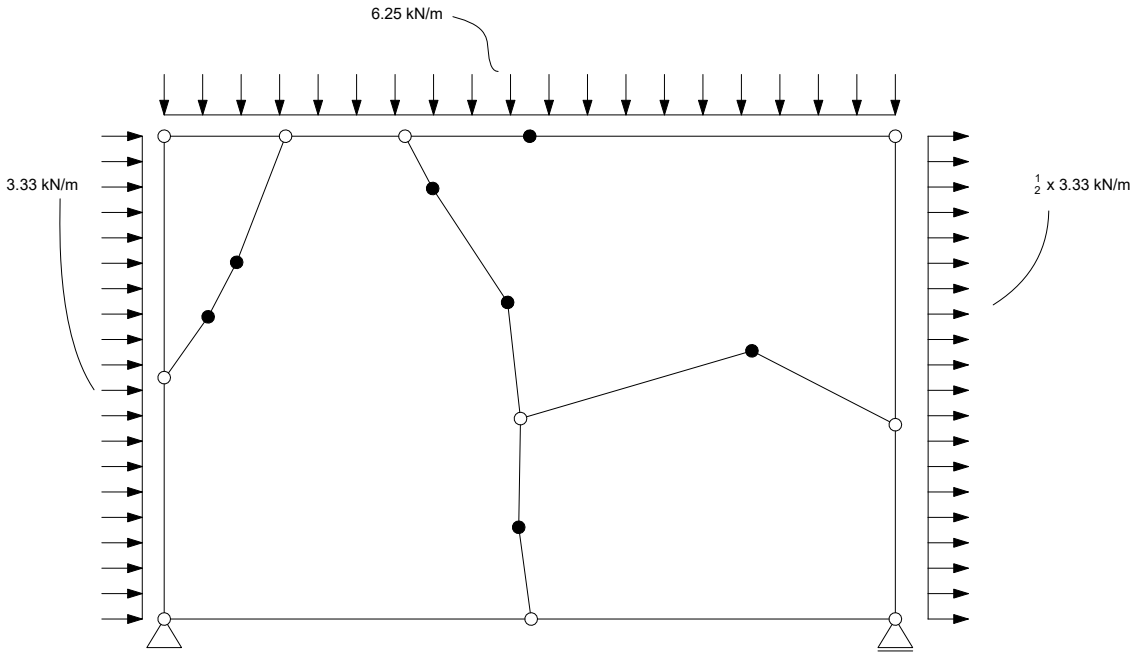


Figure 28: The strucutal represenation of a game with three placement. Open circles are hinged, filled circles are rigid connections. Current sel *Source:* own

Buckling and partial safety factors are neglected.

Optimisation Framework

The structural layout of the frames is determined through multi-objective evolutionary optimisation using the `NSGA-II` (Non-dominated Sorting Genetic Algorithm II) framework. This approach simultaneously minimises two competing objectives:

1. Structural performance - Maximum nodal displacement under governing load combinations
2. Material efficiency - Total cumulative length of positioned branches

$$\left[\underbrace{\mathrm{gp}_0, \mathrm{pa}_0, \mathrm{pb}_0, \mathrm{fl}_0}_{\text{step } 0},\ \underbrace{\mathrm{gp}_1, \mathrm{pa}_1, \mathrm{pb}_1, \mathrm{fl}_1}_{\text{step } 1},\ \ldots,\ \underbrace{\mathrm{gp}_{n-1}, \mathrm{pa}_{n-1}, \mathrm{pb}_{n-1}, \mathrm{fl}_{n-1}}_{\text{step } n-1}\right] \tag{1}$$

$$\left[\underbrace{24, 7, 13, 0}_{\text{step } 0},\ \underbrace{31, 15, 4, 1}_{\text{step } 1},\ \underbrace{17, 2, 19, 0}_{\text{step } 2},\ \underbrace{42, 11, 8, 1}_{\text{step } 3},\ \underbrace{5, 18, 0, 0}_{\text{step } 4},\right.$$
$$\left.\underbrace{36, 9, 14, 1}_{\text{step } 5},\ \underbrace{20, 3, 17, 0}_{\text{step } 6},\ \underbrace{48, 12, 6, 1}_{\text{step } 7},\ \underbrace{10, 16, 1, 0}_{\text{step } 8},\ \underbrace{28, 5, 11, 1}_{\text{step } 9}\right] \tag{2}$$

$$n_{\text{grid}}^{n_{\text{step}}} \times n_{\text{ptA}}^{n_{\text{step}}} \times n_{\text{ptB}}^{n_{\text{step}}} \times n_{\text{flip}}^{n_{\text{step}}} = 49^{10} \times 20^{10} \times 20^{10} \times 2^{10} \approx 8.56 \times 10^{45} \tag{3}$$

These objectives are inherently conflicting: increasing branch density improves structural stiffness but consumes additional material. The optimisation yields a Pareto-optimal front representing non-dominated trade-off solutions between structural performance and material usage.

Configurations with utilisation exceeding 1.0 are classified as infeasible. However, the optimiser retains gradient information from these solutions through constraint-domination ranking, which enables infeasible solutions closer to the feasibility boundary (e.g., utilisation = 1.05) to be ranked higher than severely infeasible ones (e.g., utilisation = 2.5), guiding evolution toward feasible regions.

Utilisation as a third objective was explored preliminarily but not evaluated conclusively.

Each candidate solution is encoded as an integer vector of length $4 \times n$ (Equation 1), where n represents the maximum number of sequential placement steps.

The genome comprises four gene types:

1. grid location genes (integer range 0–48, mapping to the 7×7 spatial grid),
2. boundary point A genes (0 to n_cell−1, where n_cell defines the number of discrete sample points along each cell's perimeter)
3. boundary point B genes (0 to n_cell−1), and
4. flip orientation genes (binary 0 or 1, controlling branch curvature mirroring).

Initial empirical testing established the genome length. Well-performing solutions typically employed seven branches, motivating the selection of 10 placement steps.

With $n = 10$ steps, $n_{\text{grid}} = 49$ grid cells, and $n_{\text{cell}} = 20$ boundary sample points, a valid genome could be the one shown in Equation 2.

Here, step 0 places a branch in grid cell 24 between boundary points 7 and 13 without flipping, step 1 places a branch in cell 31 between points 15 and 4 with flipped curvature, and so on. The interpretation of each step depends on the cell states left by all previous placements.

The sequential placement dependency creates a vast search space, quantified in Equation 3.

However, a large part of this search space consists of invalid/illegal placements. These placements are not evaluated by the FEA.

Placement validity cannot be determined at genome generation, only during execution. The retry mechanism handles geometrically infeasible gene configurations, adapting the team-based repair logic of Rahbek et al. (2023). Failed placements are retried with different genes up to a predefined maximum attempt limit before the element is discarded and the

next stock element selected. Low retry limits increase the likelihood of failed placements, enabling solutions with fewer branches than the maximum step count. This mechanism is essential for discovering material-efficient configurations, as solutions are not forced to exhaust all available placement opportunities. Across 17500 steps a 30.8% success rate of placements was established.

The optimisation with genetic algorithms is dependent on a set of paramters:

1. Crossover recombines genetic material from two high-performing parent solutions, creating offspring that inherit characteristics from both. Two-point crossover selects two random positions in the genome and exchanges the segment between them, potentially transferring beneficial placement sequences from one solution context to another Figure 29.
2. Mutation introduces variation by perturbing individual gene values, enabling exploration beyond the existing population's genetic material.

As optimal parameter values cannot be determined analytically, grid-based hyperparameter tuning explored combinations of population size, crossover probability, mutation rate, and mutation eta. Each configuration was evaluated across three

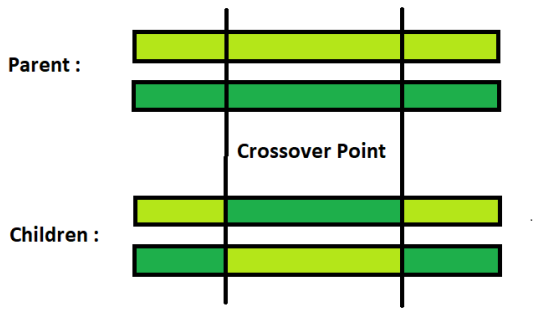


Figure 29: all tuning parameters result in similar performance. ***Source:*** own

ID	Configuration	pop_size	indpb_mult	mut_eta	cx_prob
a	Baseline (run_overnight)	500	4.0	20	0.9
b	More mutation	500	6.0	20	0.9
c	Less mutation	500	2.0	20	0.9
d	Coarser mutation jumps	500	4.0	3	0.9
e	Lower crossover probability	500	4.0	20	0.6
f	Crossover prob. 0.5 (3-obj match)	500	4.0	20	0.5

Table 2: Hyperparameter configurations evaluated in the TwoPointCrossover grid-based search.

random seeds to account for stochastic variation (Table 2)

Tuning revealed negligible performance differences across tested configurations. All parameter combinations produced statistically equivalent Pareto fronts. This parameter insensitivity suggests optimisation performance is constrained by fundamental problem characteristics rather than operator calibration. Baseline parameters from Table 2 were therefore adopted for most subsequent experiments, unless stated otherwise.

To address the research questions, ablation studies evaluated sensitivity to wind loading magnitude and stock geometric characteristics.

Wind loading variations (1 N, 2,500 N, 5,000 N, 7,500 N, 10,000 N) were evaluated across five random seeds using mixed-stock

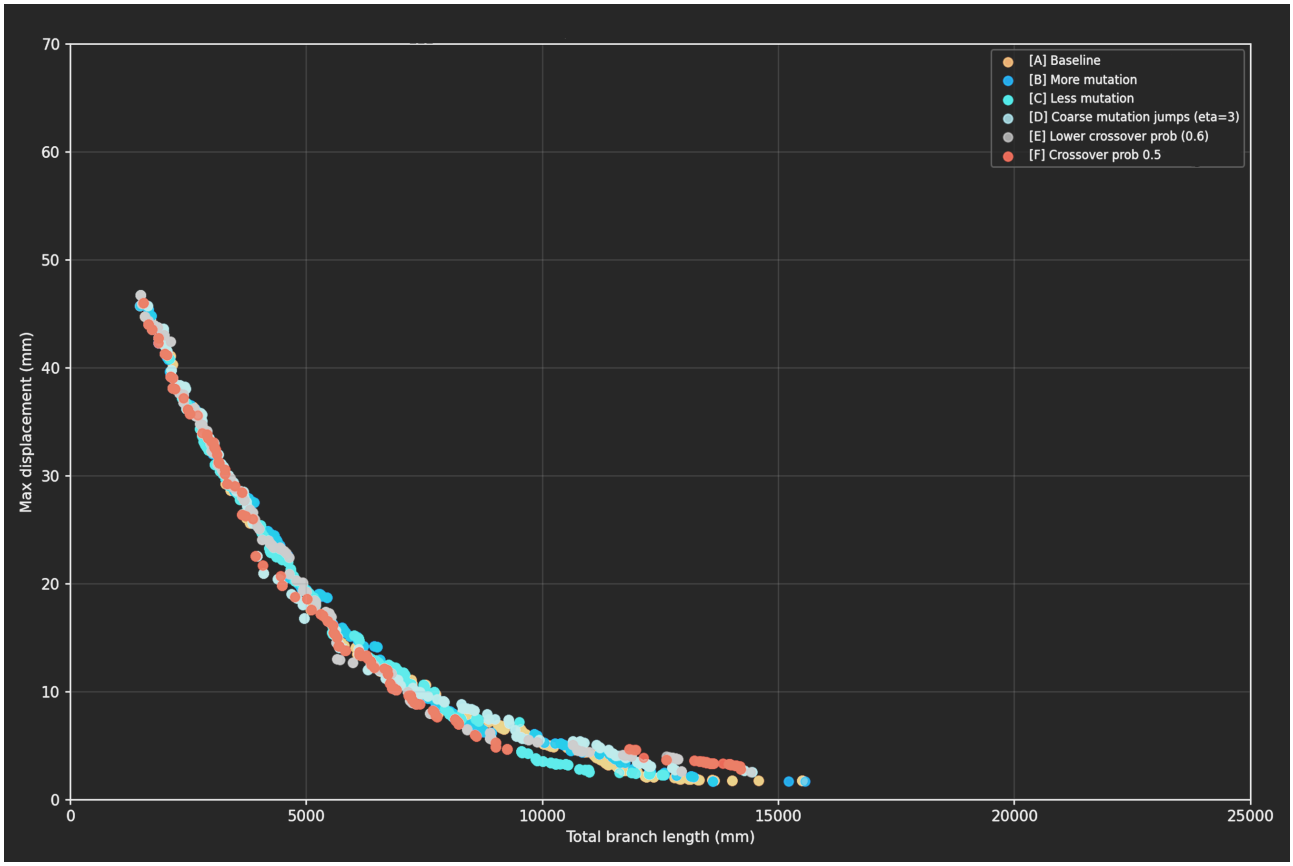


Figure 30: all tuning parameters result in similar performance. ***Source:*** own

configuration (from Table 3), producing 25 runs total. All analyses employed bi-directional wind evaluation, with the worst-case displacement and utilisation across both loading directions retained for fitness assessment.

Five stock geometry categories were systematically generated using the computational pipeline established in Part A. Seventeen trees were processed producing a pool of branch segments. These segments were classified as curved or straight through greedy allocation, with 20 elements per category selected according to specific geometric criteria:

Each stock configuration was optimised across five random seeds, yielding 25 runs per category.

Results examining genetic algorithm sensitivity to boundary conditions and stock characteristics are presented in the following section.

Group	**Sort key**	**Length filter**
curved	20 most curved first	≥ 1.5 m
straight	20 least curved first)	≥ 1.5 m
mixed	10 most-curved + 10 straightest	≥ 1.5 m each
long	20 longest	none
short	20 shortest	≥ 1.0 m

Table 3: Stock configurations derived from the Part A inventory, with corresponding selection criteria.

6.4 RESULTS

Having established the design game framework and genetic algorithm implementation, this section evaluates optimisation performance through wind loading and stock geometry ablation studies. The findings reveal critical limitations in applying genetic algorithms to sequential placement problems, where the principle of causality—the dependency of each decision on prior state—is absent from the genome-based encoding.

Wind loading sensitivity is presented first, followed by stock geometry effects.

6.4.1 WIND LOAD VARIATIONS

Multi-seed analysis and wind load variation testing revealed two primary findings.

Multiple optimisation runs with various seeds converge to equivalent Pareto fronts performance, yet show geometrically distinct branch configurations, as is seen in Figure 31. This indicates the presence of multiple locally optimal topologies with comparable structural efficiency.

When the results of across various lateral wind loads (1 N to 10,000 N) reveals a critical deficiency: the optimiser does not adapt material consumption to load magnitude. Whilst displacement scales predictably with increasing load, material usage distributions remain nearly identical across all cases. Configurations under 1 N wind consume the same 2,000–20,000 mm material range as those under 10,000 N wind, as shown in Figure 32.

Since the genome permits sparse solutions through failed placement attempts, a truly load-responsive optimisation should shift the Pareto front both downward (reduced displacement) and leftward (reduced material) for lighter loads.

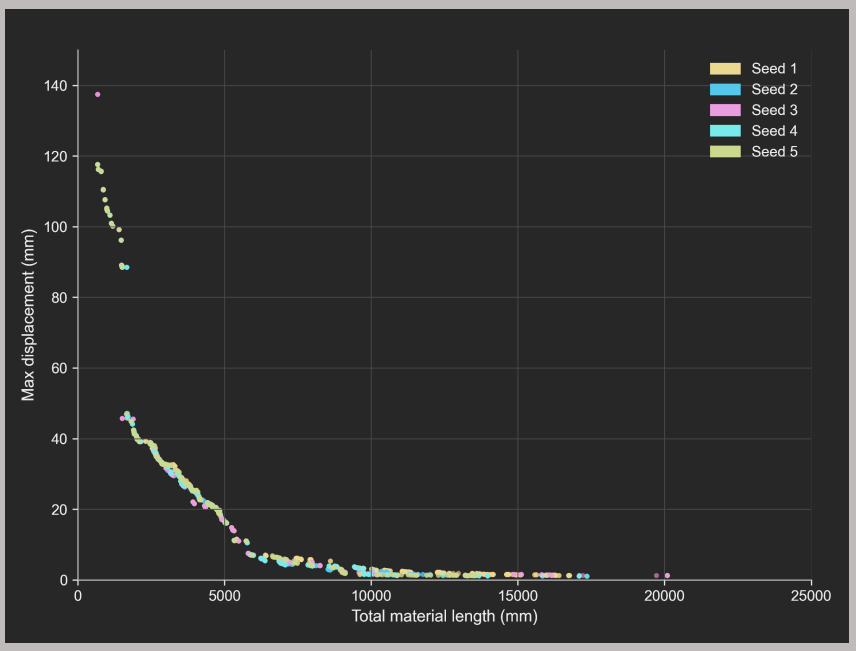


Figure 31: Pareto-fronts for various seeds for wind load of 10kN. ***Source:*** own

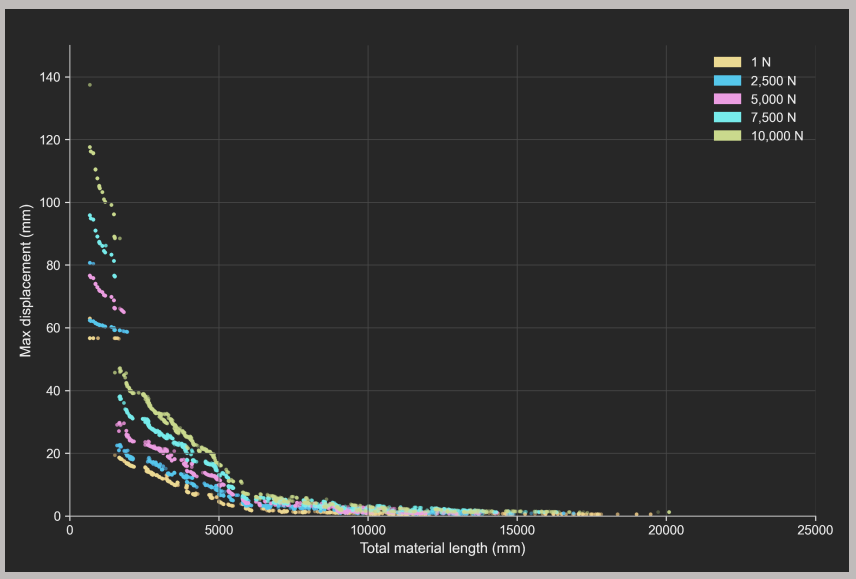


Figure 32: Pareto-fronts for various wind loads. ***Source:*** own

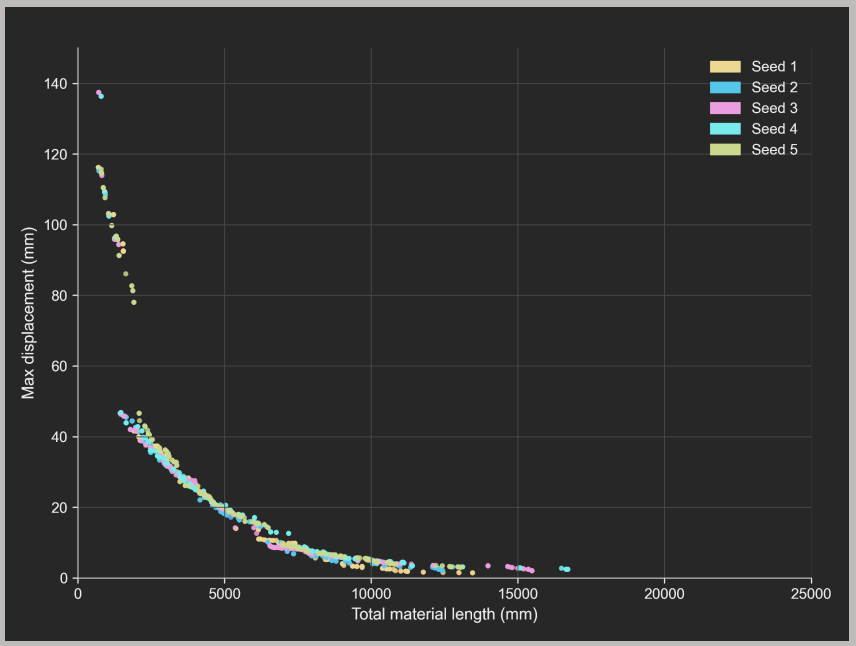


Figure 33: Pareto-fronts for various seeds for curved stock. ***Source:*** own

6.4.2 STOCK TYPES

Next to the windload, an ablation study has been performed on various stock types:

- short-stocks < 1 m
- long-stocks > 1.5 m
- mixed-stocks
- straight stocks
- curved stocks

Short-stock configurations (< 1 m) produced predominantly infeasible solutions with sparse or empty Pareto fronts. This failure stems from a representational mismatch: achieving adequate bracing with sub-meter segments requires far more placement operations than the fixed genome length (n = 10) permits. Additionally, most gene configurations produce geometric failures—short members cannot span distant boundary points—resulting in extremely low rates of valid placements. The optimizer exhausts its evaluation budget on infeasibility rather than performance optimisation.

Long-stock and mixed-stock configurations converged to equivalent Pareto fronts, confirming that the greedy longest-first selection effectively prioritizes high-value placements.

Straight-stock optimisation across multiple seeds produced similar population patterns, with the majority of solutions converging to similar topogies regardless of seed (Figure 34 and Figure 35). This pattern homogeneity reflects the reduced design space when member interchangeability (straight beams are rotationally equivalent) creates clear structural optima—diagonal bracing from bottom corners to top-center consistently outperforms alternative arrangements with this encoding.

In contrast, curved-stock optimisation exhibited greater diversity, with equivalent-performing solutions with various amounts of members in varied configurations as is seen in Figure 33, Figure 36 and Figure 37. Unique branch geometries prevent direct substitutability, expanding design freedom relative to straight stock.

Pareto front evolution across 700 generations reveals a two-phase trajectory: initial generations (1–100) rapidly establish constraint feasibility, whilst subsequent iterations (100–500) yield negligible performance improvement. This stagnation persists across all stock configurations despite hyperparameter adjustments (Figure 38 & Figure 39).

Additionally, examination of Pareto front solutions reveals that member utilisation ratios remain consistently low across the solution set, indicating that the optimiser has not discovered configurations that efficiently exploit material capacity. This suggests the presence of superior, more material-efficient solutions that have remained unexplored.

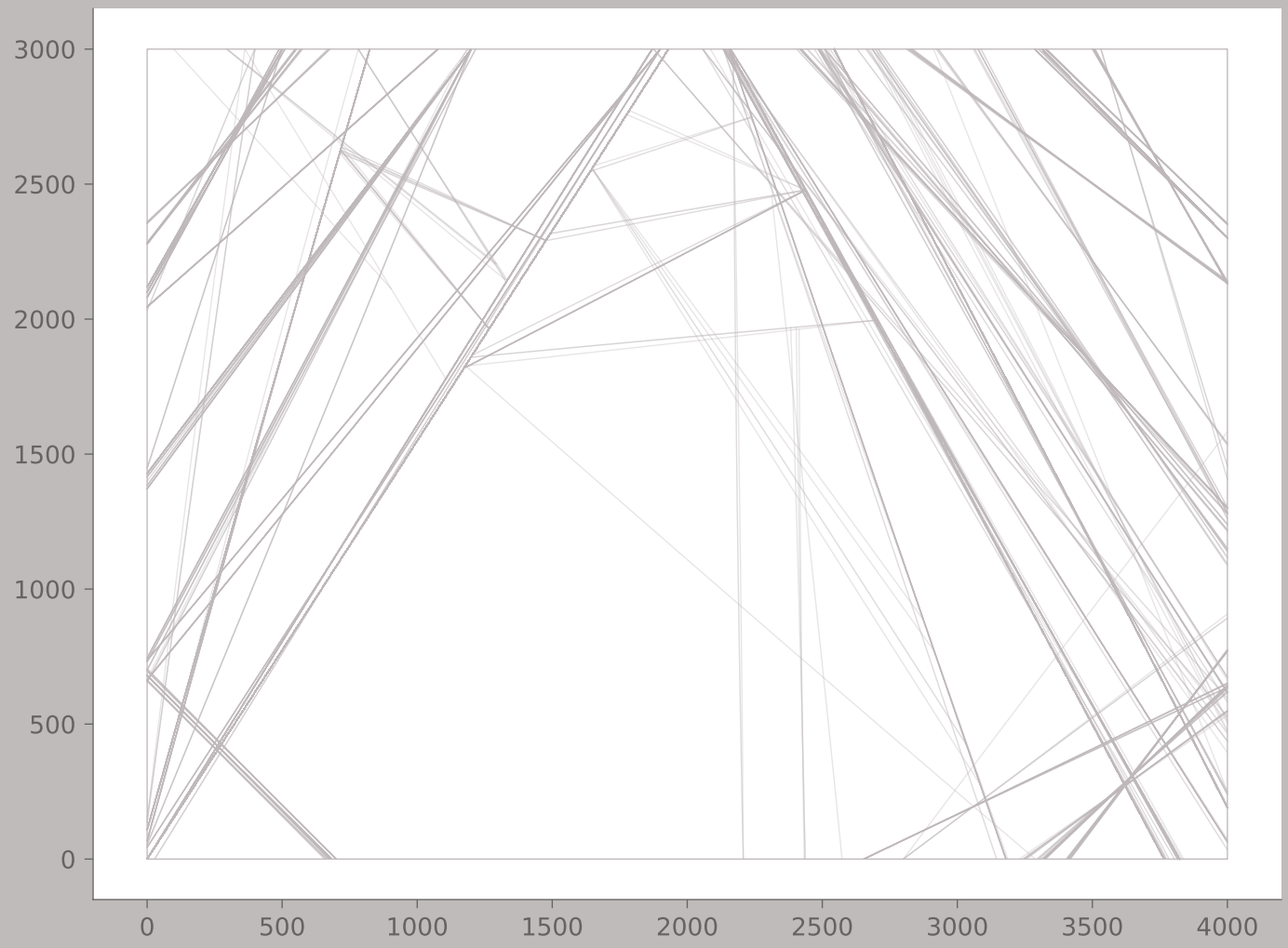


Figure 34: Pareto front overlay for straight stock - seed 1. While not globally optimal, these are structurally viable solutions. ***Source:*** own

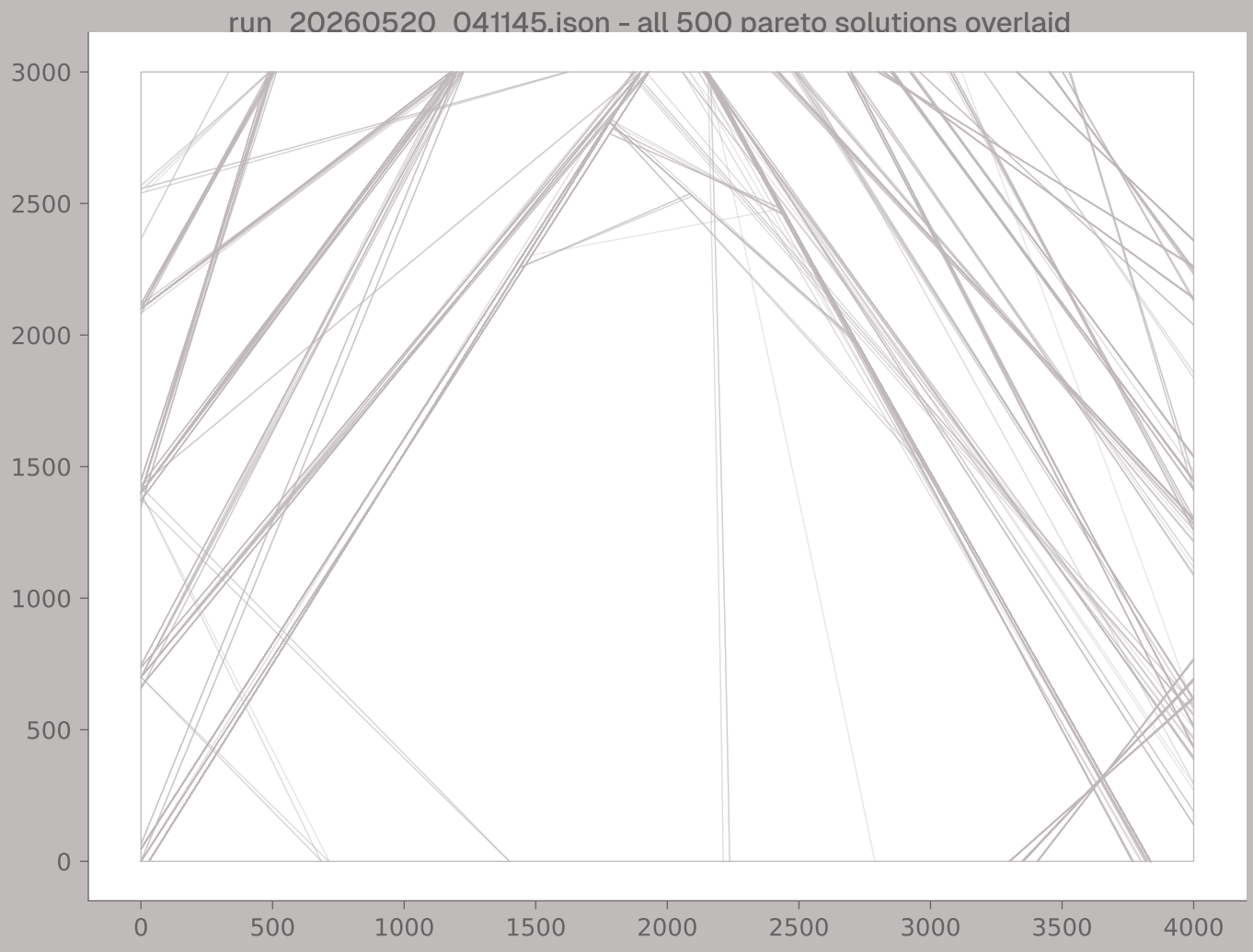


Figure 35: Pareto front overlay for straight stock - seed 2.While not globally optimal, these are structurally viable solutions. ***Source:*** own

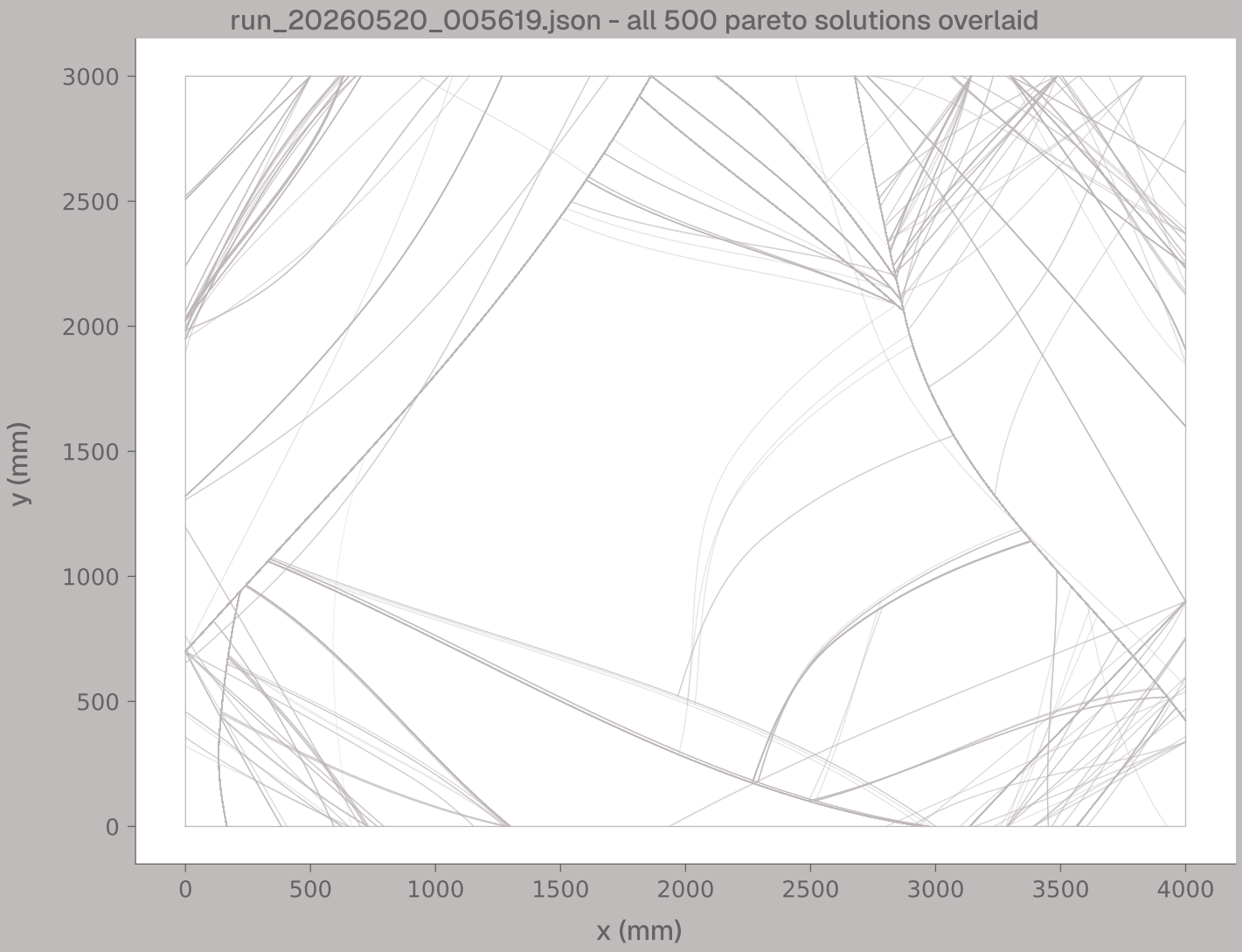


Figure 36: Pareto front overlay for curved stock - seed 1. While not globally optimal, these are structurally viable solutions.***Source:*** own

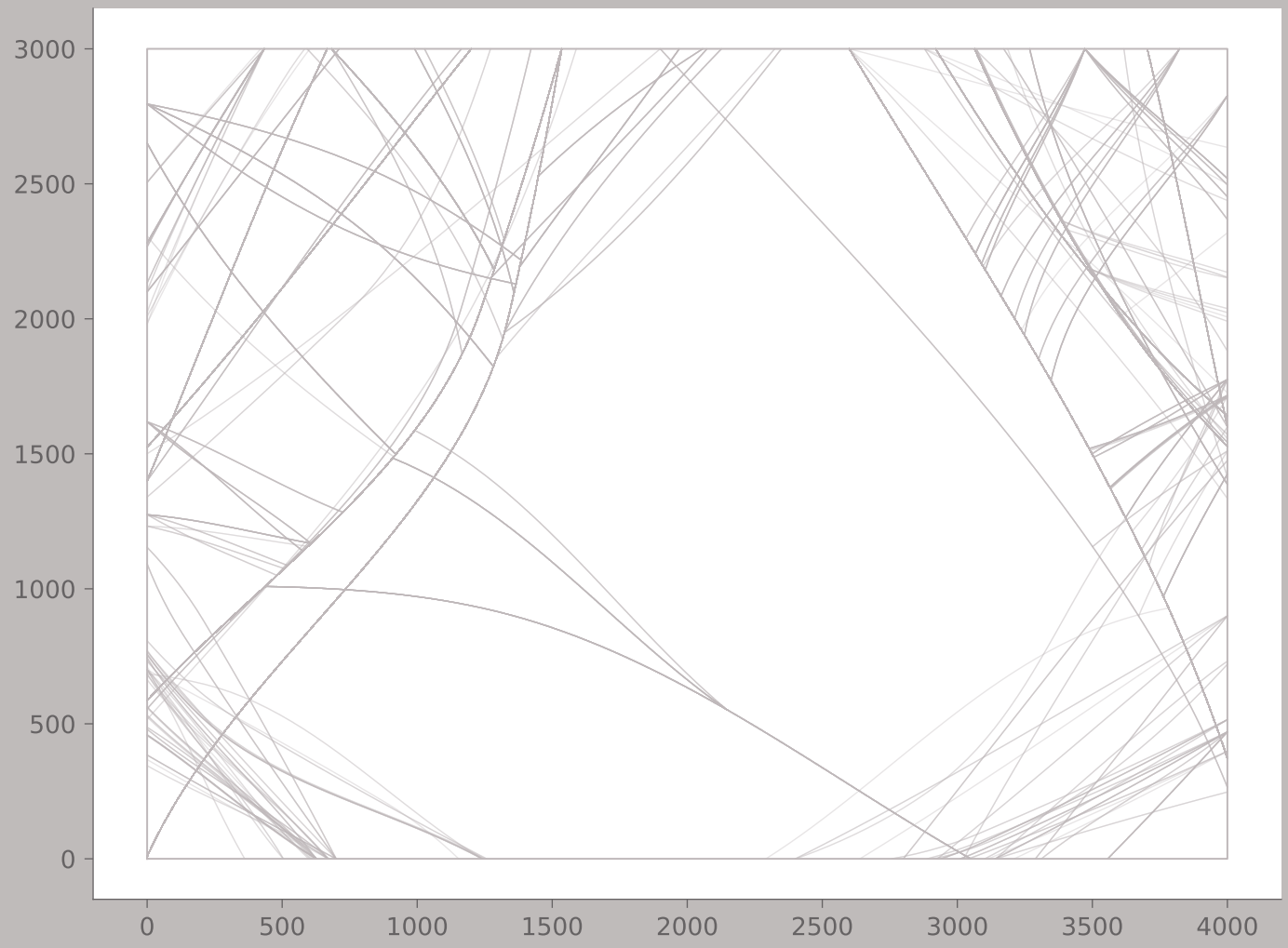


Figure 37: Pareto front overlay for curved stock - seed 2.While not globally optimal, these are structurally viable solutions. ***Source:*** own

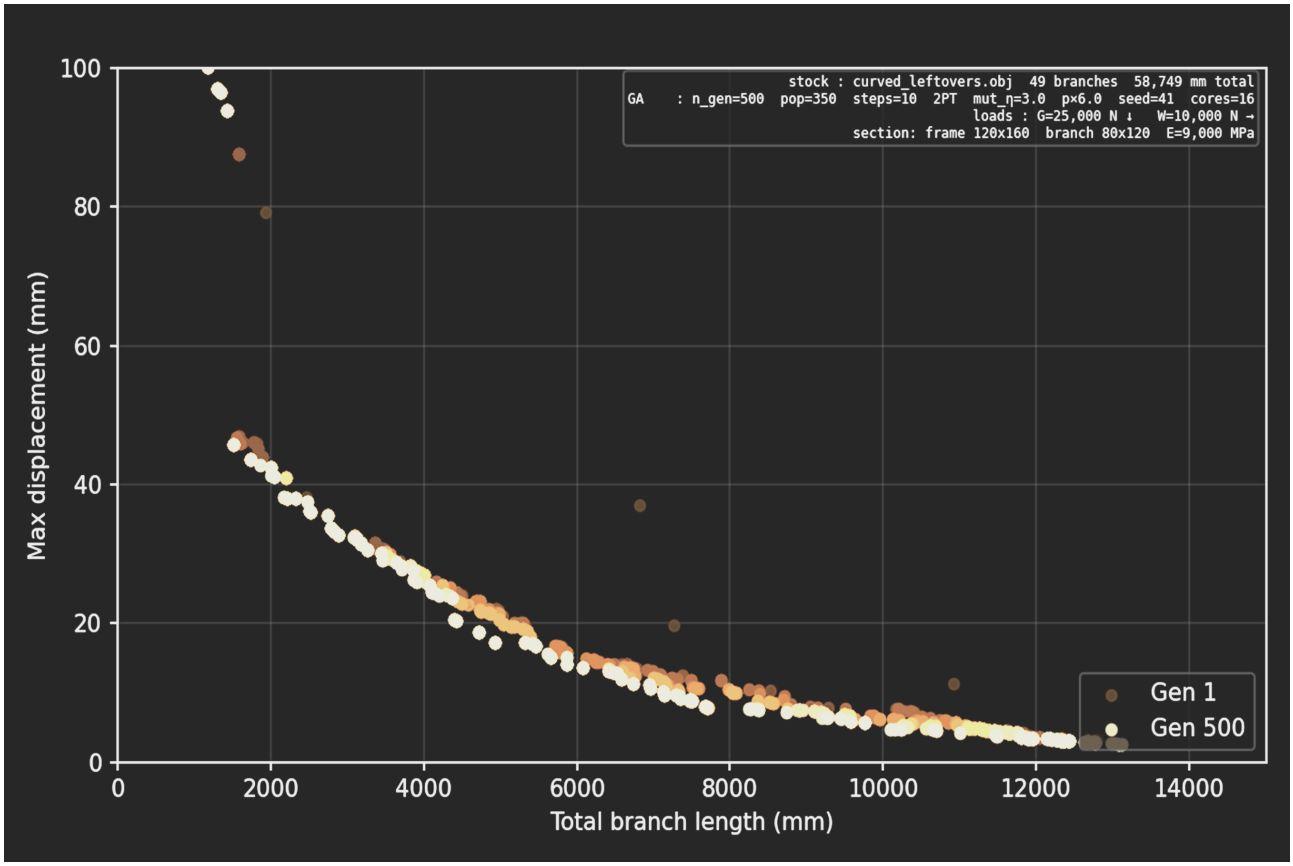


Figure 38: stagnation of pareto front optimisation. ***Source:*** own

6.5 DISCUSSION

Analysis of the results indicates that crossover operations frequently produce geometrically invalid offspring due to sequential genome encoding: placement step N depends on cell states from steps 1 through N-1, rendering recombined gene segments contextually incompatible. This contrasts with both reviewed examples, which represent design approaches with predefined topologies (member count, positions and connections) Figure 23 Figure 24. This ensures that each gene position has consistent structural meaning across the population. Gene N always governs member position N and thereby the placement of one element does not alter the structural meaning of subsequent genes. This enables effective crossover for future generations as recombined gene segments retain their structural roles in offspring genomes.

Additionally, whilst boundary point indexing maintains consistent dimensionality (20 points per cell, clockwise from bottom-left), the spatial consistency is limiteldly. For instance, the spatial meaning of “point 5” varies across cells with different geometries.

The retry mechanism further complicates genome interpretation: failed placements trigger repositioning attempts without updating the genome, meaning identical genomes can produce different outcomes through retry-driven exploration. Tests disabling retry functionality confirmed no performance improvement, indicating repair mechanisms cannot overcome the fundamental encoding limitation.

Optimisation progress thus relies primarily on mutation-based local search, limiting escape from locally optimal configurations.

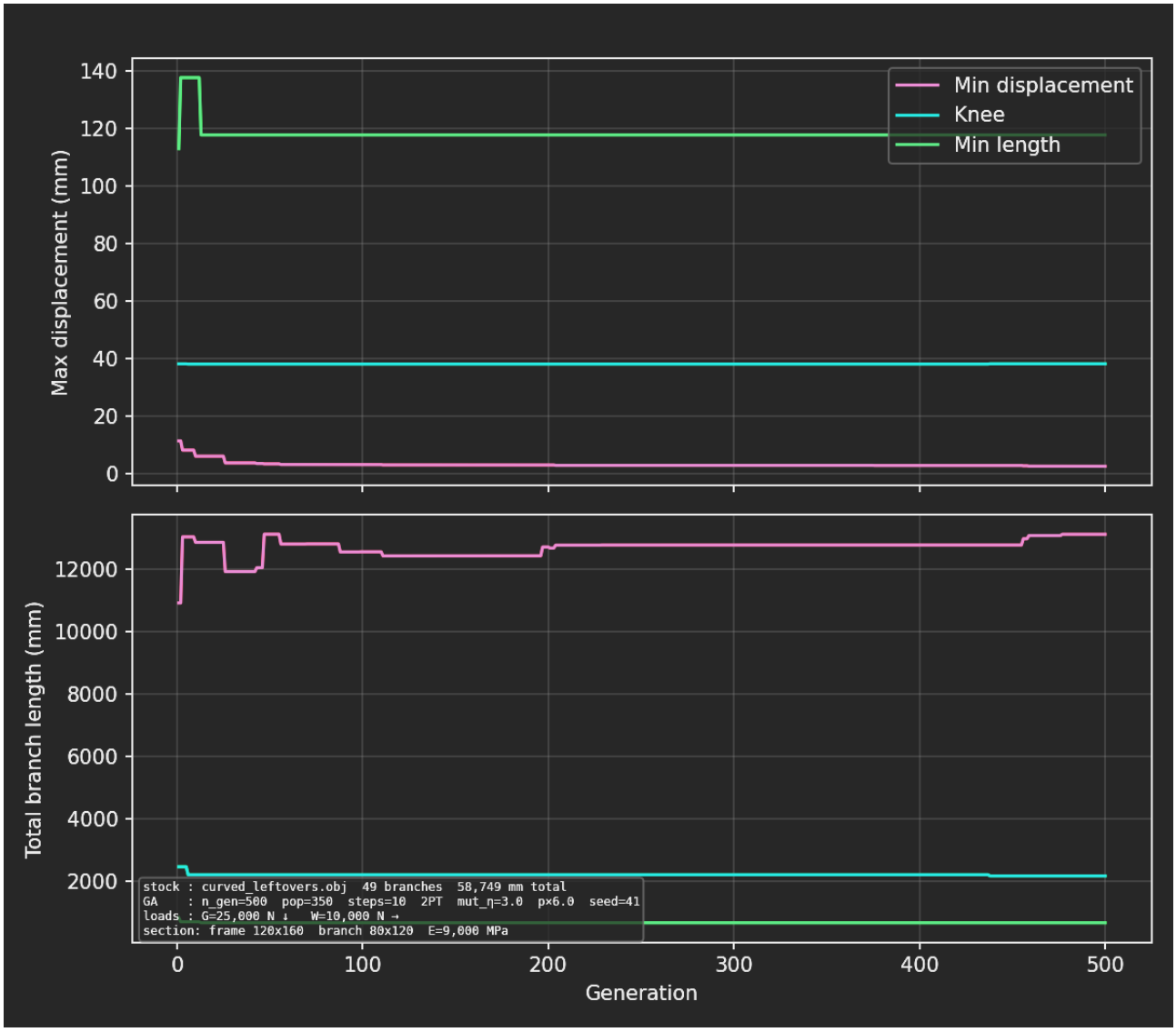


Figure 39: trajectory of knee point and anchor points across generations. ***Source:*** own

To verify this hypothesis, comparison between two grid-based search was performed: one with two-point crossover (2PT) and simulated binary crossover (SBX). Despite their distinct recombination strategies, both operators produced similar Pareto fronts, confirming that the crossover mechanism does not influence optimisation quality. This equivalence supports the conclusion that sequential placement constraints neutralise genetic recombination effectiveness regardless of operator type. Uniform crossover and multiple point crossover have also been tested, but did not show any improvement.

The optimisation framework demonstrates practical utility despite its evolutionary limitations. Computationally, 700-generation runs execute in approximately 4 minutes on multi-core systems, enabling rapid iteration during preliminary design phases. The resulting Pareto fronts contain structurally feasible solutions suitable for design exploration, though the algorithm's constraint-satisfaction focus and premature convergence behavior exclude guarantees of global optimality.

Additionaly, the Pareto approach provides can provide value for architectural applications by showing the material-performance trade-off

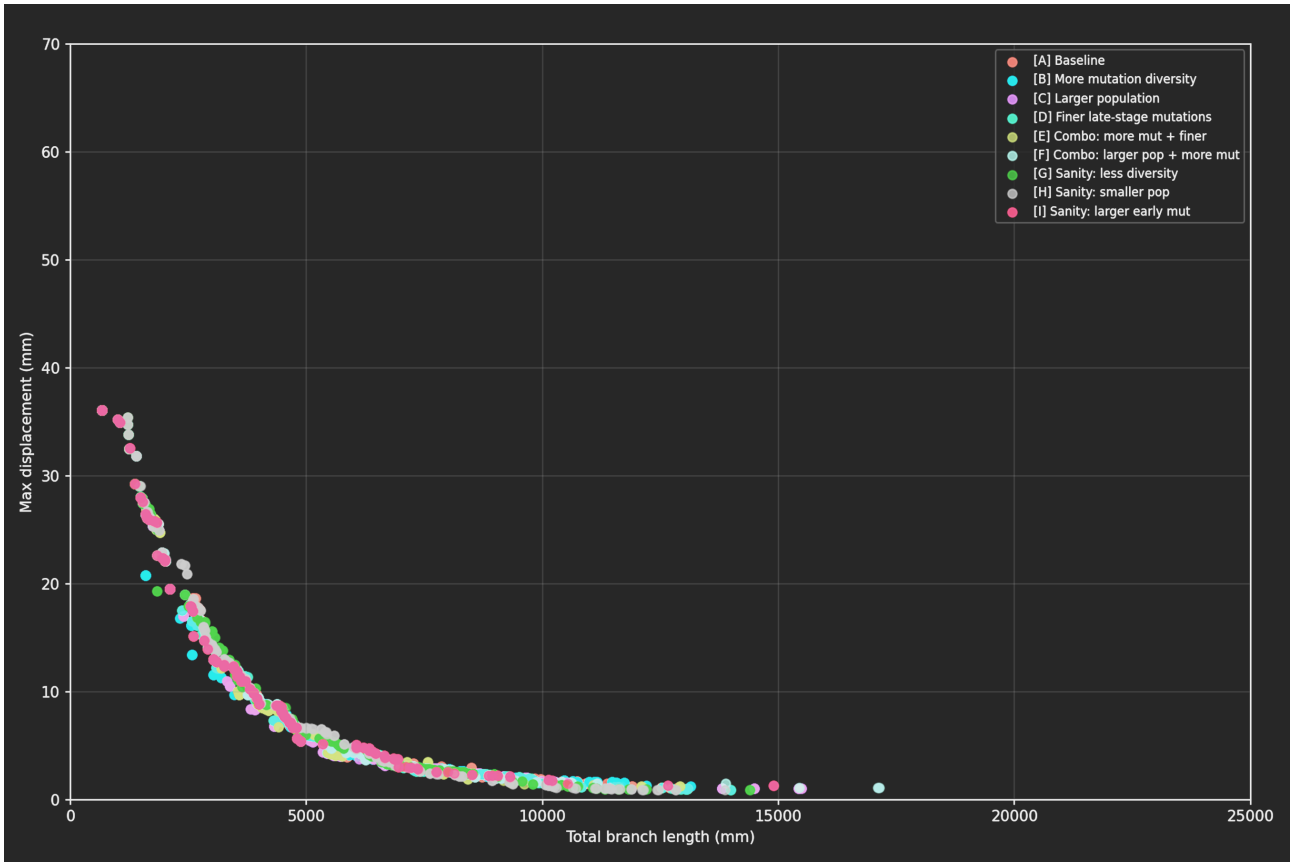


Figure 40: straigt pareto history. ***Source:*** own

spectrum. Designers can select configurations from the soltuion set based on project-specific priorities, such material economy, displacement tolerance, or aesthetic considerations,

The framework exhibits generalisability across panel geometries and stock characteristics. The algorithm could adapt to arbitrary frame configurations such as rectangular wall panels, triangular roof elements, or non-orthogonal geometries. Since it requires only that stock dimensions remain geometrically compatible with frame proportions (e.g., member lengths sufficient to form diagonal bracing), it is excpected that it will output some valid solutions.

Structural verification through finite element analysis, incorporating per-member material data, would strengthen the framework's applicability beyond preliminary design.

The observed ineffectiveness of crossover in the designed sequential placement encoding suggests that alternative state-based optimisation methods could prove more effective, such as reinforcement learning agents for stock constrained design (Mirra & Mueller, 2025). The sequential placement problem inherently embodies the ***principle of causality*** where each placement action determines the state space for subsequent decisions. Methods that explicitly model these state transitions and causal dependencies, for instance through implementation of a discount factor, may enable more effective exploration of the solution space.



Figure 41: A pile of logs waiting to be processed in Esslingen, Germany. ***Source:*** Own

7 DEMONSTRATOR

The demonstrator brings Part A and Part B together in a single browser application. It is organised as a two-stage workflow: Stage 1 (steps 1–2) turns a scanned tree skeleton into a stock inventory, and Stage 2 (steps 3–6) draws on that accumulated stock to optimise, choose, and commit a frame. The two stages loop: after committing a frame the user can add more trees or re-run the optimiser to design the next frame from the updated stock.

Stage 1 — From tree to stock

1. Load item & set parameters (Figure 42)

The user selects a `.ply` tree skeleton and adjusts the segmentation settings as seen in Figure 42. Clicking ***process & add tree*** runs the skeleton assessment and appends the resulting stock pieces to the session.

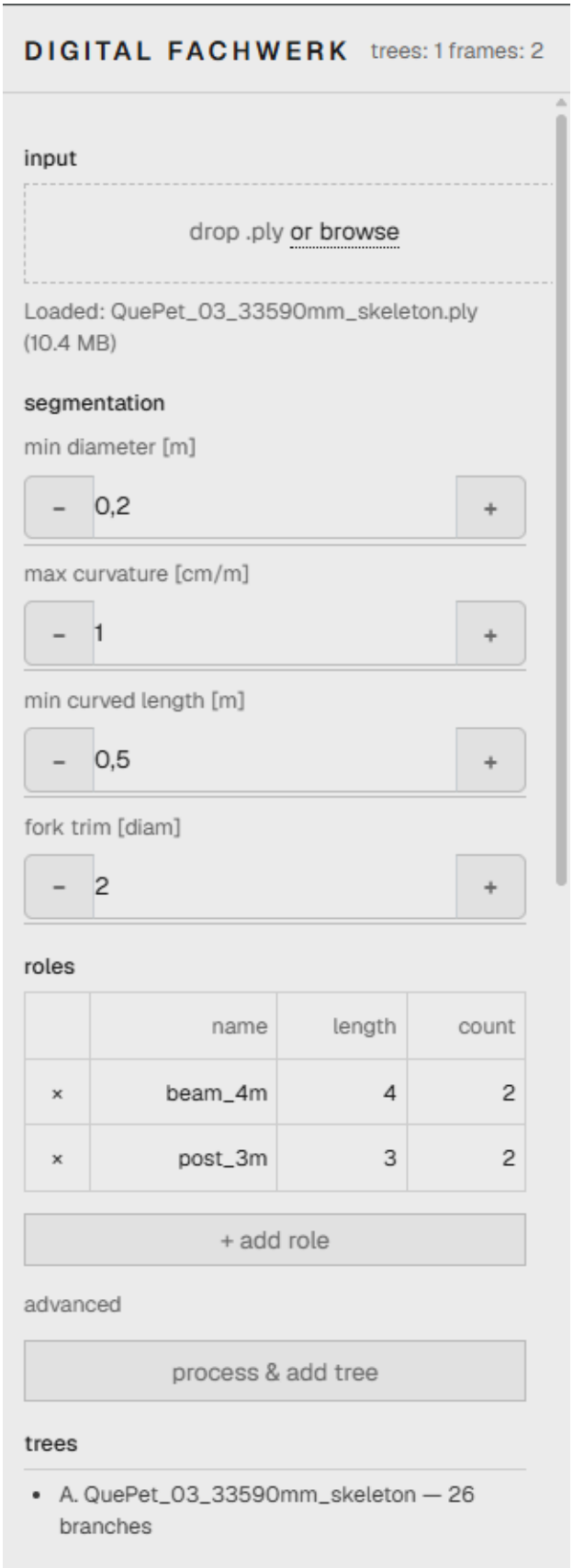


Figure 42: Step 1: uploading a tree skeleton and configuring the segmentation parameters before processing. ***Source:*** own

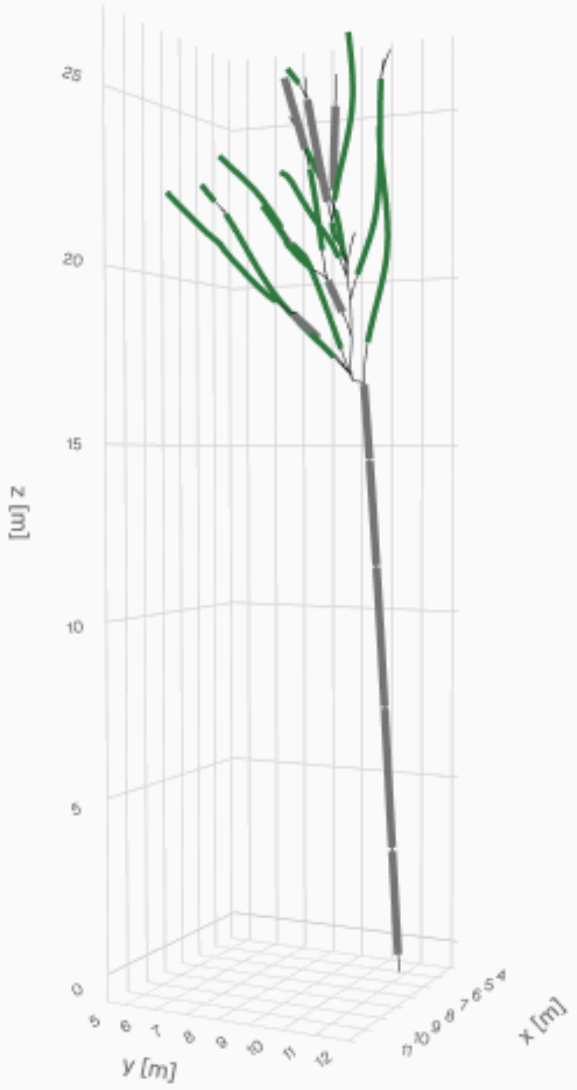


Figure 43: Step 2: the segmented skeleton, branches coloured by their assigned outcome. ***Source:*** own

2. Visualise the tree (Figure 43)

The processed skeleton is rendered with branches coloured by assigned outcome: structural roles, curved, straight, and leftovers.

Stage 2 — From stock to frame

3. Set optimiser settings (Figure 44)

The user selects stock kinds, frame dimensions, wind load, and NSGA-II parameters (population size, number of generations, and CPU cores) *Run optimiser* launches the multi-objective search against the accumulated session stock.

Repeated runs with identical settings yield different designs, a consequence of both genetic stochasticity and the sequential placement encoding's limited convergence behaviour.

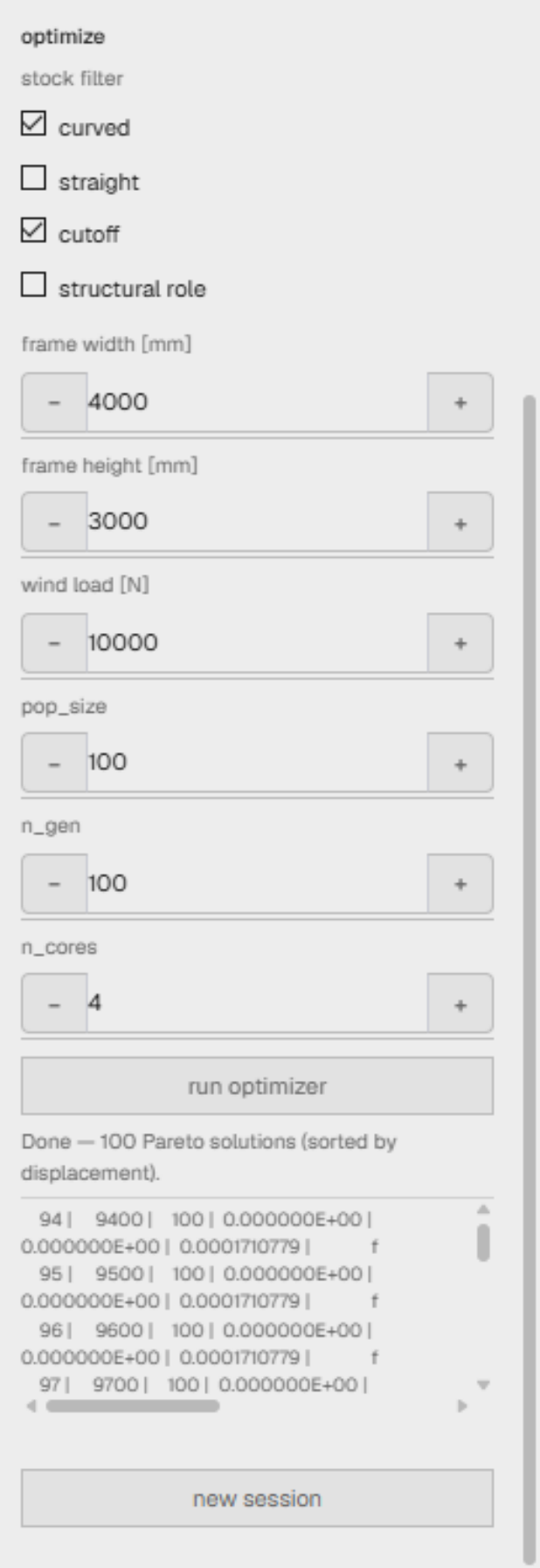


Figure 44: Step 3: selecting the stock kinds, frame dimensions, wind load, and NSGA-II settings for the run. ***Source:*** own

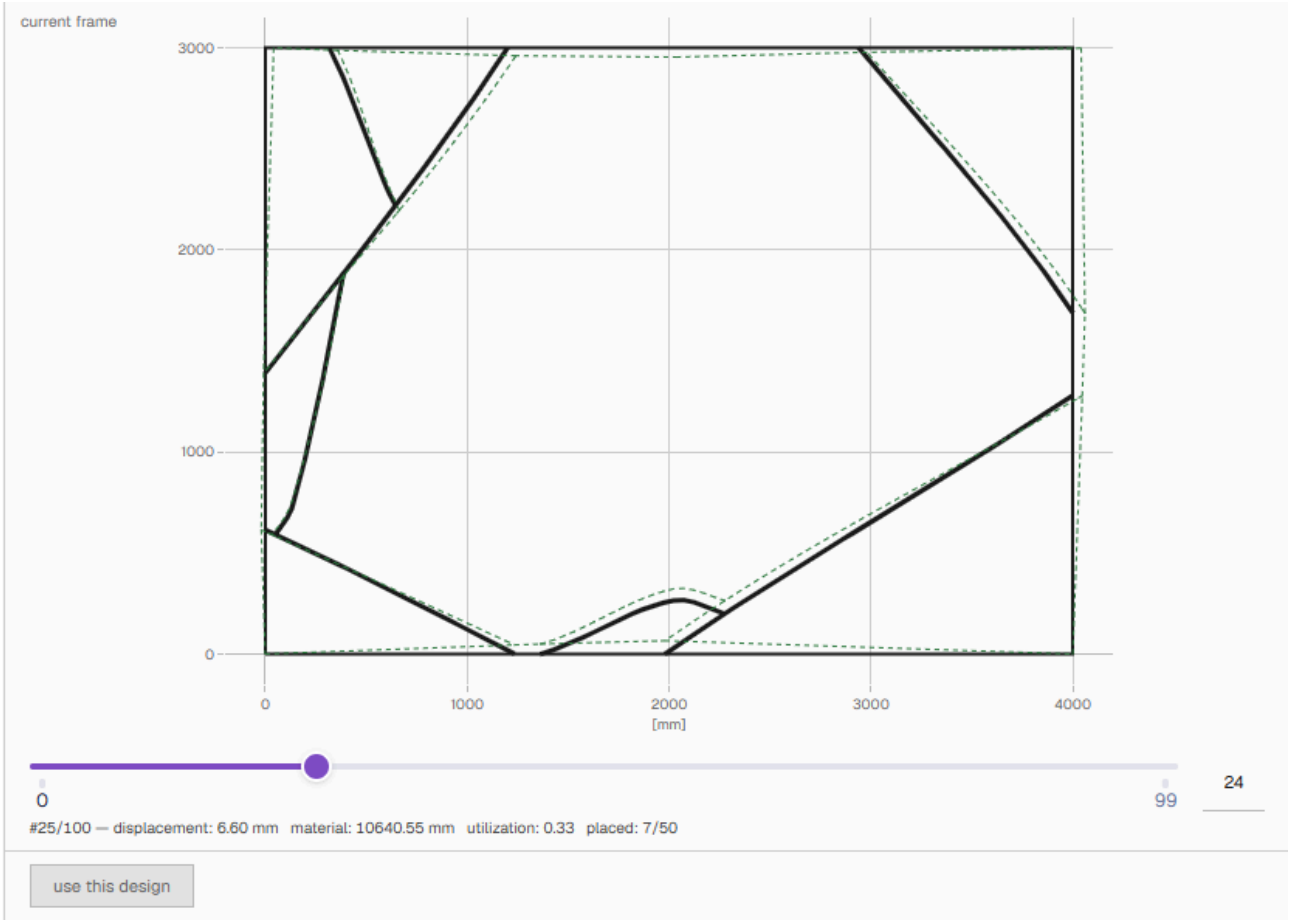


Figure 45: Step 4: sweeping the Pareto front to trade stiffness against material economy. ***Source:*** own

.

4. Choose a design (Figure 45)

After optimisation, the current-frame panel displays the resulting truss layout with displacements. A slider browses the Pareto front (sorted by displacement), with each candidate reporting displacement, material length, and median utilisation, allowing the user to trade stiffness against material economy and select a preferred design.

5. See used stock (Figure 46)

The stock inventory highlights which pieces the selected design consumes and what remains available.

6. Save & export designs (Figure 47)

Use this design commits the solution, returning offcuts to stock. Frames collect in the saved-designs grid and can be exported as OBJ files. Whilst the design is performed in two dimensionsial space, elements in reality carry a slight out-of-plane offset owing to the planarity tolerance (Figure 48).

Using the trees identified in the case studies and the application, a set of preliminary designs was generated using the default settings (Figure 49, Figure 50, Figure 51, Figure 52, Figure 53).

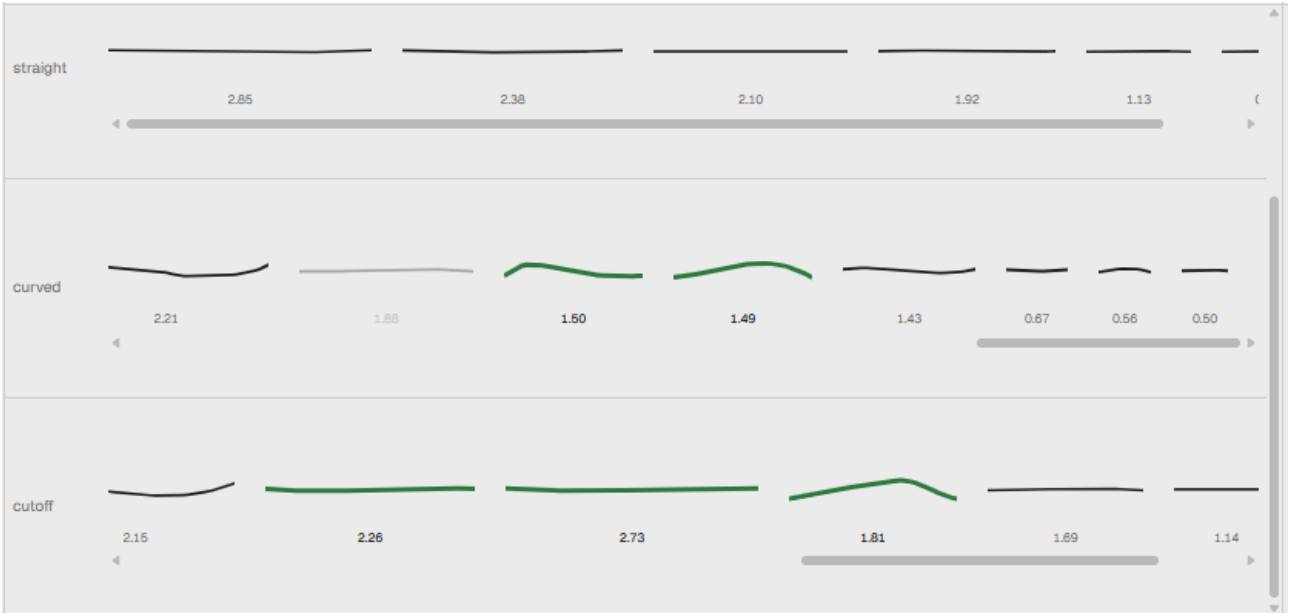


Figure 46: Step 5: the stock inventory, with the pieces consumed by the selected design highlighted. ***Source:*** own

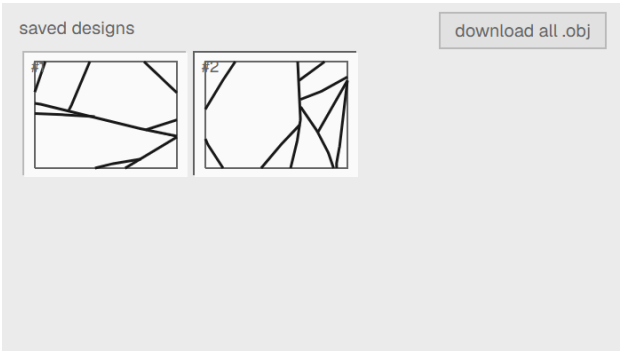


Figure 47: Step 6: committing the design, returning offcuts to stock, and exporting the saved frames. ***Source:*** own

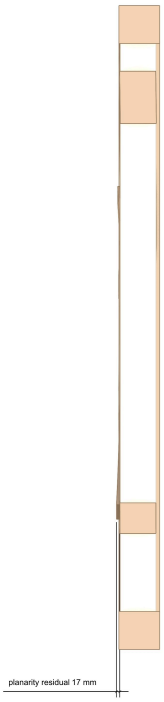


Figure 48: This section shows how the timber beams can be slightly out of plane, as the tree-analysis allows for slight inplanarities. ***Source:*** own

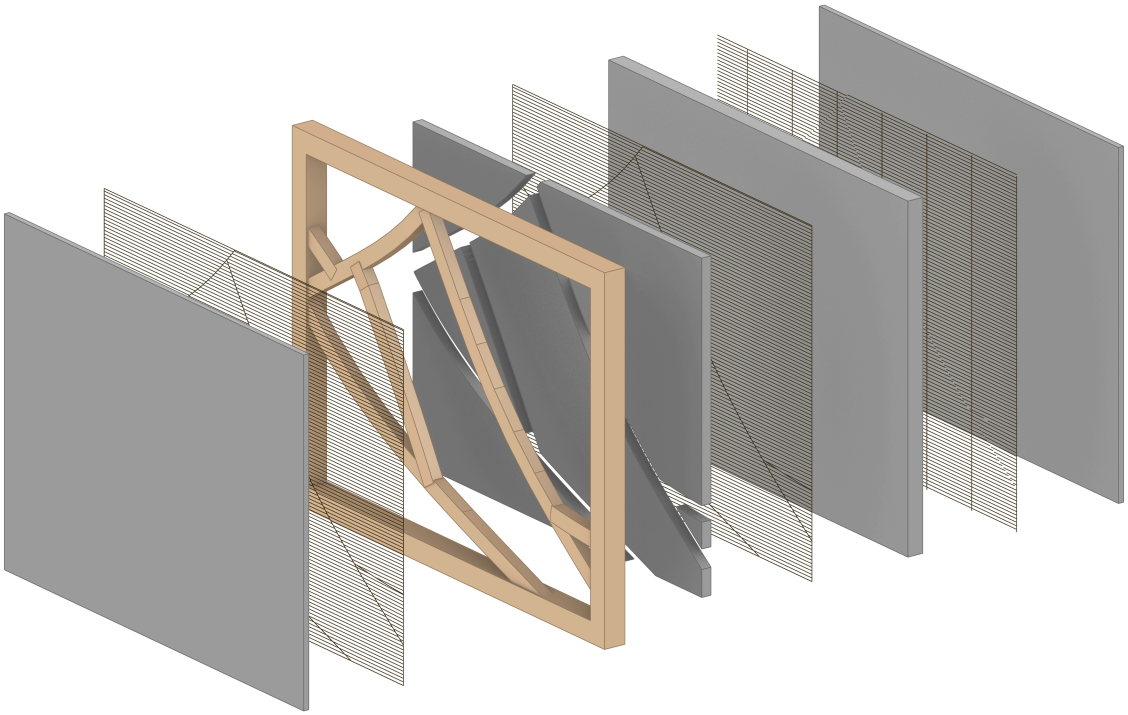


Figure 49: Example of traditional infill for fachwerk using locally sourced materials. From left to right (exterior to interior): stabilised clay plaster, wattle, structural frame, fiber-reinforced-clay-insulation, wattle, straw-bale insulation, hiss-reed mesh, clay plaster. ***Source:*** own

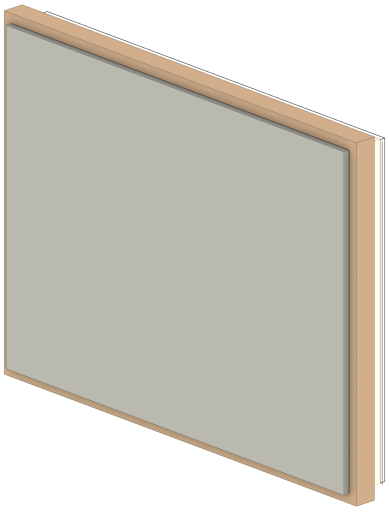


Figure 50: The expressive character can be concealed behind plaster. ***Source:*** own

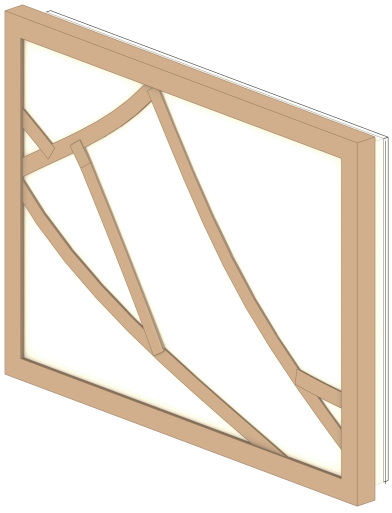


Figure 51: The expressive character of the frame can be embraced. ***Source:*** own

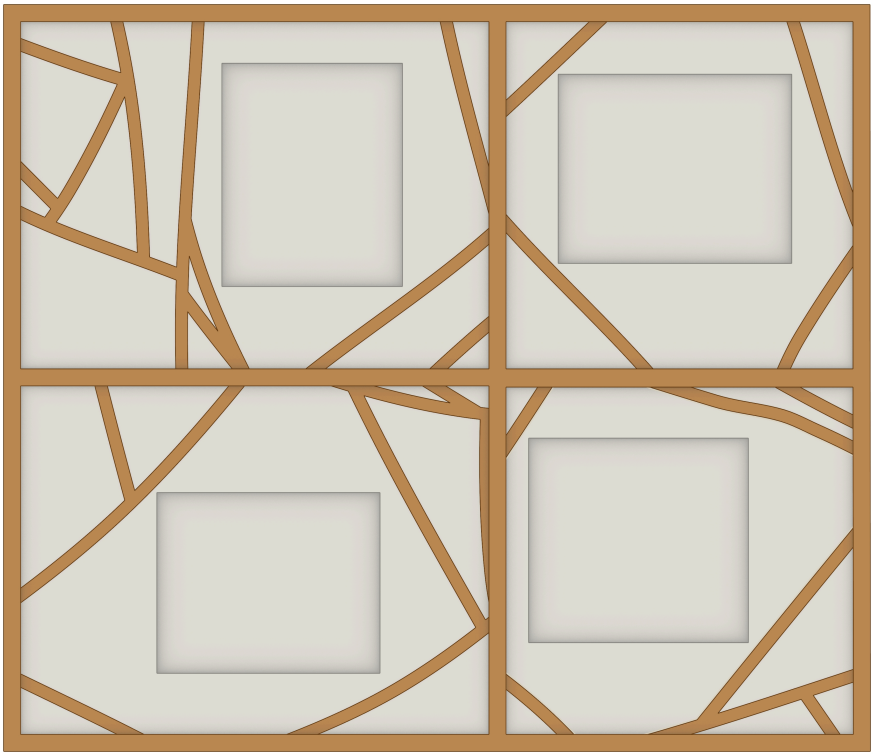


Figure 52: Generating multiple frames of varying sizes and combining them allows designers to construct all exterior wall elements of a building. ***Source:*** own



Figure 53: AI-generated visualisation of contemporary half-timbered framing using the Digital Fachwerk framework. ***Source:*** own



Figure 54: A potential use case would be the design of top-up housing in urban contexts using urban forestry. ***Source:*** own



Figure 55: A potential use case would be the design of top-up housing in urban contexts using urban forestry. ***Source:*** own

8 CONCLUSIONS

Whilst the construction sector's dependence on standardised materials carries significant environmental and architectural costs, the emergence of Industry 4.0 technologies presents an opportunity to move beyond mass-production and mass-customisation towards mass-adaptation, where the digital and the natural are no longer opposing but complementary. This thesis operationalises this opportunity through Digital Fachwerk and the KM0DFAB framework.

Working with local timber requires two conditions: A) a thorough understanding of available material stock, and B) a design approach where form follows material rather than the reverse.

Working with heterogeneous physical materials through computational tools inherently demands a cyber-physical approach: the physical world must be accurately digitised to inform the design process. The quality and completeness of this digitisation is critical: the parameters captured define the dimensionality of the design space, whilst those not captured must be assumed. More thoroughly digitised materials yield more faithful computational design spaces, and consequently, higher quality designs.

Part A shows *to what extent terrestrial laser scans of standing trees can be computationally processed to assess and quantify the potential yield of straight and curved structural timber components prior to harvest.*

Terrestrial laser scanning captures only the external geometry of trees, providing no information about internal material properties. For heterogeneous materials such as timber, where both geometry and internal properties vary, this represents a significant limitation: the design space is constrained to geometrical parameters, whilst properties such as density, grain orientation, and defect distribution must be assumed.

Whilst geometry alone is insufficient for full structural design, it establishes a valuable upper bound on material potential. Geometry serves as a proxy for physical existence: the presence of a geometric element guarantees the presence of internal properties, even if they remain unmeasured. This upper bound is therefore meaningful: it enables comparison of tree stock availability, both straight and curved, between specimens and informs selection criteria before any internal assessment is performed.

Furthermore, the analysis of the trees allows one to compare the availability of straight and curved stock within a single specimen, showing the increased yield if curved elements are considered. Crucially, if the industry were to embrace curved elements as structural components, the value of current timber stock would increase significantly. A tree in one's garden, previously considered of little structural value, could suddenly become a viable construction resource. Some specimens in this study more than doubled in yield when curved elements were incorporated, considering only components above a minimum diameter of 200 mm. Realising this potential, however, requires a fundamental paradigm shift in how timber is assessed, designed, and valued.

Part B shows *to what extent optimisation with genetic algorithms can generate structurally viable half-timbered wall configurations from a constrained inventory of geometrically irregular timber components.*

The manual design game demonstrated that designing with heterogeneous materials and geometries fundamentally differs

from designing with standardised straight components. Where two straight elements of identical cross-section placed in the same position yield effectively the same design, two geometrically distinct curved elements produce meaningfully different structural configurations. Whilst the irregularity of stock expands rather than constrains the design space, offering greater design freedom, it simultaneously makes identifying optimal solutions less intuitive, underscoring the need for computational optimisation.

Whilst the algorithm produced structurally viable configurations, identifying truly optimal designs for a given stock remains an open question. The sequential placement encoding proved inadequate for optimisation with genetic algorithms: each placement step alters the state space for subsequent decisions, reducing crossover effectiveness significantly. From this, the hypothesis arises that optimisation methods that explicitly model causal dependencies between placement decisions, such as reinforcement learning agents, would prove more effective for stock-constrained timber framing design.

Nevertheless, the current encoding demonstrated practical computational efficiency, with 700-generation runs completing in approximately 4 minutes, which is encouraging for integration into an interactive design tool.

Ultimately, this thesis demonstrates ***how computational algorithms for pre-harvest assessment and stock-constrained design can enable architecture with locally sourced irregular timber***.

The findings demonstrate that algorithms can enable pipelines in which different stakeholders, such as foresters, constructors, and designers, can inform one another through a cyber-physical framework. Such a system, realised here as an application for Digital Fachwerk, shows that it is feasible to generate structural designs directly from a scanned timber stock, thereby lowering the barrier to building with locally sourced material. In this way, a design can guide harvesters on which tree to fell first, or conversely, the available forest stock can inform which designs are potentially feasible.

Whilst addressing current shortcomings, such as incorporating internal log data and finding optimal design solutions, would strengthen the pipeline, a full scan-to-fabrication workflow must be developed before construction with local timber sources becomes practically viable and truly impactful.

Nevertheless, even at this preliminary stage, the results suggest that a user-friendly interface and near-instantaneous calculations are key characteristics of a successful implementation; qualities that, combined with its open-source nature, could make the tool accessible to non-computational designers at scale. In doing so, it points towards a model in which global computational methods reconnect the building industry with local material realities.

9 FUTURE WORK

Several directions for future development are identified.

Part A: Pre-Harvest Assessment

Integration of internal assessment data, such as defect mapping, could further refine yield estimates. Additionally, the structural properties of all elements, including material strength and reaction wood behaviour in curved members, could be incorporated to enable structural verification in later stages.

Part B: Design Framework

In the first instance, the sequential placement encoding that limits crossover effectiveness suggests that state-based optimisation may prove more effective. In particular, employing reinforcement learning agents that explicitly model causal dependencies between placement decisions (Mirra & Mueller, 2025).

Beyond this, the framework could be tested across a broader range of panel geometries, including triangular roof elements and non-orthogonal configurations. Similar design approaches could be applied to other typologies such as roofs and floors, or extended to reclaimed timber as an alternative stock source. Structural verification through finite element analysis, incorporating per-member material data, would strengthen the framework's applicability beyond preliminary design.

KM0DFAB Framework

Beyond the two tools developed here, future work could integrate Part A and Part B into a seamless end-to-end workflow, from standing tree to fabricated wall system. Extension of this workflow to include automated joinery design (Vestartas, 2021). Robotic manufacturing could enable mobile, on-site production, realising the full KM0DFAB paradigm. Beyond irregular timber from trees, extension to other locally available materials such as waste wood, stone, earth, or natural fibres would broaden the framework's applicability, potentially establishing KM0DFAB as a viable construction method across diverse geographic contexts.

10 REFLECTIONS

As outlined in the main body of this report, the Digital Fachwerk tool represents a subset of the broader KM0-DFAB workflow proposed in the introduction. Initially, the ambition was to implement the full pipeline; however, time constraints necessitated a narrower scope, and robotic fabrication was excluded entirely.

This narrowing had consequences beyond simply removing parts from the research scope. The research process itself exhibited the same sequential causality that ultimately limited the genetic algorithm: early decisions shaped later outcomes in ways that could not easily be revisited. Choices made in the beginning defined the path of the research, and as is common in computational design research, tasks consistently took longer than anticipated. This is a particular characteristic of algorithmic development: whether a given approach is viable can often only be determined once implementation is complete, making it genuinely difficult to plan and scope such work in advance, particularly when attempting it for the first time.

That said, had it been clear from the outset that the genetic algorithm would be the endpoint rather than a preliminary step, the encoding might have been approached differently, potentially yielding globally optimised solutions. Reflecting on this, however, it seems plausible that the sequential causality inherent to the design problem renders global optimisation unachievable through any non-state-based method. The literature, combined with the empirical findings of this thesis, now provides a well-founded argument that genetic algorithms are not the appropriate method for this problem type.

But in this case, the optimisation with genetic algorithms was conceived as a stepping stone towards a reinforcement learning implementation, and in that sense it may be considered a qualified success. In that sense, the negative result is informative: it confirms that reinforcement learning is not merely preferable but likely necessary.

Moreover, the encoding developed for the 4444 algorithm was deliberately structured to facilitate a future RL integration, which represents a tangible contribution in itself.

Through the use of FEA solvers and custom algorithms, it is evident that this research operated at the intersection of structural design and computational engineering, and the challenges inherent to that field were felt throughout.Nevertheless, the broader problem this research seeks to address is one that fundamentally calls for digital tools, and in that respect the overall methodology remains a credible and promising path forward.

Bibliography

Ainsworth, T. (2024). Form vs. Matter. *The Stanford Encyclopedia of Philosophy*.

Akbarnezhad, A., Xiao, J., Akbarnezhad, A., & Xiao, J. (2017). Estimation and Minimization of Embodied Carbon of Buildings: A Review. *Buildings*, *7*(1). https://doi.org/10.3390/buildings 7010005

Akritas, A. G., & Malaschonok, G. I. (2004). Applications of Singular-Value Decomposition (SVD). *Mathematics and Computers in Simulation*, *67*(1), 15–31. https://doi.org/10.1016/j.matcom.2004. 05.005

Benson, T., & Gruber, J. (1981). *Building the Timber Frame House: The Revival of a Forgotten Craft*. Simon & Schuster.

Blondeau, E.-N. (1787). *[Illustrations of ships and maritime subjects, taken from the Encyclopédie méthodique]*. [Paris : Panckoucke ; Liège : Plomteux].

Buildings, Culture and Environment: Informing Local and Global Practices. (2005). *International Journal of Sustainability in Higher Education*, *6*(1). https://doi.org/10.1108/ijshe.2005.24906aae. 006

Buri, H. U., & Weinand, Y. (2011). The Tectonics of Timber Architecture in the Digital Age. *Building with Timber Paths into the Future*, 56–63.

Chai, H., Zhou, X., Gao, X., Yang, Q., Zhou, Y., & Yuan, P. F. (2024). Integrated Workflow for Cooperative Robotic Fabrication of Natural Tree Fork Structures. *Automation in Construction*, *165*, 105524. https://doi.org/10.1016/j.autcon.2024.105524

Chesneau, J.-B., Le Net, E., & Berg, S. (2012). A Transport Tool to Evaluate Sustainability Impacts of Transport Processes within the Forest Wood Chain. *European Journal of Forest Research*, *131*(1), 73–80. https://doi.org/10.1007/s10342-011-0530-4

Claypool, M., Retsin, G., Jimenez Garcia, M., Jaschke, C., & Saey, K. (2021,). *Automated Production: At the Intersection of Digital Labour and Building Practice* [Proceedings Paper]. Association of Collegiate Schools of Architecture.

Clifford, J., & Lučić, I. (2026). Thinking through Forests: Colonialism, Capitalism, and the Global Timber Trade. *Journal of Historical Geography*. https://doi.org/10.1016/j.jhg.2026.04.005

Correia, M., Dipasquale, L., & Mecca, S. (Eds.). (2014). *VERSUS: Heritage for Tomorrow: Vernacular Knowledge for Sustainable Architecture*. Firenze University Press. https://doi.org/10.36253/978-88-6655-742-5

Correia, M., Dipasquale, L., & Mecca, S. (Eds.). (2015). *VERSUS: Heritage for Tomorrow: Vernacular Knowledge for Sustainable Architecture* (1st ed.). Firenze University Press. https://doi.org/10. 36253/978-88-6655-742-5

Dakhli, Z., & Lafhaj, Z. (2017). Robotic Mechanical Design for Brick-Laying Automation. *Cogent Engineering*, *4*(1), 1361600. https://doi.org/10.1080/23311916.2017.1361600

De Wolf, C., Çetin, S., & Bocken, N. M. P. (Eds.). (2024). ***A Circular Built Environment in the Digital Age***. Springer International Publishing. https://doi.org/10.1007/978-3-031-39675-5

Dijkstra, E. W. (1959). A Note on Two Problems in Connexion with Graphs. ***Numerische Mathematik***, ***1***(1), 269–271. https://doi.org/10.1007/BF01386390

Dinsmoor, W. B. (1973). ***The Architecture of Ancient Greece: An Account of Its Historic Development***. Biblo & Tannen Publishers.

Donato, M. (2016). ***The Influence of Resource Dependency on Collaboration in the Construction Supply Chain*** [Doctoral dissertation].

Du, S., Lindenbergh, R., Ledoux, H., Stoter, J., & Nan, L. (2019). AdTree: Accurate, Detailed, and Automatic Modelling of Laser-Scanned Trees. ***Remote Sensing***, ***11***(18), 2074. https://doi.org/10.3390/rs11182074

Engelmann, I. (2017,). ***Intentionale Verwendung von Eichen-Krummholz in Glockenstühlen Des 15. Und 16. Jahrhunderts***. https://doi.org/DOI 10.25536/20170105

Fennell, P., Driver, J., Bataille, C., & Davis, S. J. (2022). Cement and Steel — Nine Steps to Net Zero. ***Nature***, ***603***(7902), 574–577. https://doi.org/10.1038/d41586-022-00758-4

Forbes, C. (2018). Rebuilding Nepal: Traditional and Modern Approaches, Building or Diminishing Resilience?. ***International Journal of Disaster Resilience in the Built Environment***, ***9***(3), 218–229. https://doi.org/10.1108/IJDRBE-01-2017-0001

Form, W. (1987). On the Degradation of Skills. ***Annual Review of Sociology***, ***13***(Volume13, 1987), 29–47. https://doi.org/10.1146/annurev.so.13.080187.000333

Frampton, K. (1993). Toward a Critical Regionalism: Six Points for an Architecture of Resistance. In ***Postmodernism: Postmodernism***. Routledge.

Gao, S., Jin, R., & Lu, W. (2020). Design for Manufacture and Assembly in Construction: A Review. ***Building Research & Information***, ***48***(5), 538–550. https://doi.org/10.1080/09613218.2019.1660608

Gao, T., Sun, Y., Zhu, Y., Li, J., Yuan, M., & Yuan, P. F. (2026). Enabling Circular Timber Construction Through Vision-Based Robotic Processing of Irregular Reclaimed Wood. ***Cleaner Engineering and Technology***, 101205. https://doi.org/10.1016/j.clet.2026.101205

Gata, K. M., Spahn, F., Klinkel, S., & Trautz, M. (2024). Geometrical Analysis of Naturally Grown Timber for the Design of Load-Bearing Structures. ***Research Directions: Biotechnology Design***, ***2***, e19. https://doi.org/10.1017/btd.2024.12

Gattas, J., Burry, J., Chen, C., & Koch, C. (2023). ***Inventory-Constrained Design of a Variable Small Diameter Round Timber Structure***.

Glass, S. van, & Zelinka, S. L. (2021). Chapter 4: Moisture Relations and Physical Properties of Wood. ***Chapter 4 in FPL-GTR-282***, 4–22.

Gorges, J., Huber, M., Sauter, U. H., & Dormann, C. F. (2021). Curvature of Logs—Development of and Comparison between Different Calculation Approaches. ***Forests***, ***12***(7). https://doi.org/10.3390/f12070857

Greer, F., & Horvath, A. (2025). Exploring the Significance of Transportation Emissions in Upfront Embodied Carbon in Buildings. ***Building and Environment***, ***269***, 112457. https://doi.org/10.1016/j.buildenv.2024.112457

Griese, N., Ritzert, M., & Nölke, N. (2025, November). ***BioDiv-3DTrees: A Large Labelled Single Tree Point Cloud, QSM and Tree Graph Dataset***. GRO.data. https://doi.org/10.25625/8PB1IF

Göswein, V., Arehart, J., Pittau, F., Pomponi, F., Lamb, S., Zea Escamilla, E., Freire, F., Silvestre, J. D., & Habert, G. (2022). Wood in Buildings: The Right Answer to the Wrong Question. ***IOP Conference Series: Earth and Environmental Science***, ***1078***(1), 12067. https://doi.org/10.1088/1755-1315/1078/1/012067

Hagberg, A. A., Schult, D. A., & Swart, P. J. (2008). Exploring Network Structure, Dynamics, and Function Using NetworkX. ***Python in Science Conference***. https://doi.org/10.25080/TCWV9851

Harris, R. (1978). ***Discovering Timber-framed Buildings***.

Harris, R. (1979). ***Discovering Timber-Framed Buildings***. Aylesbury : Shire Publications.

Haus Bohlenstraße - Splietker Bau GmbH & Co. KG. (2019, October).

Herzog, T., Natterer, J., Schweitzer, R., Volz, M., & Winter, W. (2012). Timber Construction Manual. In ***Timber Construction Manual: Timber Construction Manual***. Birkhäuser. https://doi.org/10.11129/detail.9783034614634

House Design in Nepal | Top Construction Company in Nepal | Call Us. (n.d.).

Kampelmann, S. (2021). Knock on Wood: Business Models for Urban Wood Could Overcome Financing and Governance Challenges Faced by Nature-Based Solutions. ***Urban Forestry & Urban Greening***, ***62***, 127108. https://doi.org/10.1016/j.ufug.2021.127108

Karakul Türk, Ö. (2015). An Integrated Methodology for the Conservation of Traditional Craftsmanship in Historic Buildings. ***INTERNATIONAL JOURNAL OF INTANGIBLE HERITAGE***, ***10***.

Kijewski-Correa, T., & Taflanidis, A. A. (2012). The Haitian Housing Dilemma: Can Sustainability and Hazard-Resilience Be Achieved?. ***Bulletin of Earthquake Engineering***, ***10***(3), 765–771. https://doi.org/10.1007/s10518-011-9330-y

Krok, G., Kraszewski, B., & Stereńczak, K. (2020). ***Application of Terrestrial Laser Scanning in Forest Inventory – an Overview of Selected Issues***.

Larsen, M. S. S., Lindhard, S. M., Brunoe, T. D., Nielsen, K., & Larsen, J. K. (2019). Mass Customization in the House Building Industry: Literature Review and Research Directions. ***Frontiers in Built Environment***, ***5***. https://doi.org/10.3389/fbuil.2019.00115

Larsen, N. M., Aagaard, A. K., Hudert, M., & Rahbek, L. W. (2022a). Timber Structures Made of Naturally Curved Oak Wood: Prototypes and Processes. ***Architecture, Structures and Construction***, ***2***(4), 493–507. https://doi.org/10.1007/s44150-022-00046-9

Larsen, N. M., Aagaard, A. K., Hudert, M., & Rahbek, L. W. (2022b). Timber Structures Made of Naturally Curved Oak Wood: Prototypes and Processes. ***Architecture, Structures and Construction***, ***2***(4), 493–507. https://doi.org/10.1007/s44150-022-00046-9

Laurent, A.-B., Gaboury, S., Wells, J.-R., Bonfils, S., Boucher, J.-F., Sylvie, B., D'Amours, S., & Villeneuve, C. (2013). Cradle-to-Gate Life-Cycle Assessment of a Glued-Laminated Wood Product from Quebec's Boreal Forest. ***Forest Products Journal***, ***63***(5–6), 190–198. https://doi.org/10.13073/FPJ-D-13-00048

Lindholm, E.-L., & Berg, S. (2005). Energy Requirement and Environmental Impact in Timber Transport. ***Scandinavian Journal of Forest Research***, ***20***(2), 184–191. https://doi.org/10.1080/02827580510008329

Llana, D. F., Short, I., & Harte, A. M. (2020). Use of Non-Destructive Test Methods on Irish Hardwood Standing Trees and Small-Diameter Round Timber for Prediction of Mechanical Properties. ***Annals of Forest Science***, ***77***(3), 62. https://doi.org/10.1007/s13595-020-00957-x

Lok, L., Samaniego, A., & Spencer, L. (2021). Timber De-Standardized. ***ACADIA 2021: Realignments: Toward Critical Computation***, 222–231. https://doi.org/10.52842/conf.acadia.2021.222

Manaf, A. A., Latip, N. S. A., Ismail, N. H., Ismail, W. N. W., Rashid, N. S. A., & Nizar, W. `. W. A. (2025). Challenges in Adapting Vernacular Architecture to The Contemporary: A Systematic Literature Review. ***Semarak International Journal of Design, Built Environment and Sustainability***, ***3***(1), 17–29. https://doi.org/10.37934/sijdbes.3.1.1729

Mechtcherine, V., Grafe, J., Nerella, V. N., Spaniol, E., Hertel, M., & Füssel, U. (2018). 3D-printed Steel Reinforcement for Digital Concrete Construction – Manufacture, Mechanical Properties and Bond Behaviour. ***Construction and Building Materials***, ***179***, 125–137. https://doi.org/10.1016/j.conbuildmat.2018.05.202

Meeson, B. (2021). Richard Hayman, Timber-Framed Buildings: (Oxford: Shire Publications, Bloomsbury Publishing, 2021). 80 Pp. (Incl. 73 Col. Illus. and 4\,Line Drawings). ISBN 978-1-78442-428-2. £8.99 (Pb). ***Vernacular Architecture***, ***52***(1), 129–130. https://doi.org/10.1080/03055477.2021.1992715

Mirra, G., & Mueller, L.-M. (2025). ***A Reinforcement Learning Agent for Enhancing the Reuse of Structural Components in New Buildings***.

Mohammad, W. A., Nazih Diab, Y., Elomri, A., & Triki, C. (2023). Innovative Solutions in Last Mile Delivery: Concepts, Practices, Challenges, and Future Directions. ***Supply Chain Forum: An International Journal***, ***24***(2), 151–169. https://doi.org/10.1080/16258312.2023.2173488

Mollica, Z. (2016a). ***Tree Fork Truss: An Architecture of Inherent Forms***.

Mollica, Z. (2016b,). ***Woodchip-Barn***.

Mortier, D. (2025). Rediscovering the Half-Timber Tradition. ***Journal of Traditional Building, Architecture and Urbanism***, ***6***, 218–233.

Nasir, V., Rahimi, S., Mohammadpanah, A., Hansen, E., & Sassani, F. (2024). Intelligent Lumber Production (Sawmill 4.0): Opportunities, Challenges, and Pathways to Adoption. In M.-R. Alam & M. Fathi (Eds.), ***Integrated Systems: Data Driven Engineering: Integrated Systems: Data Driven Engineering*** (pp. 213–231). Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-53652-6_13

Nocetti, M., & Brunetti, M. (2024). Advancements in Wood Quality Assessment: Standing Tree Visual Evaluation—A Review. ***Forests***, ***15***(6), 943. https://doi.org/10.3390/f15060943

Ovando, P., Olivera, S. M., Rodea-Chavez, A., García-Franco, L., Heard, C. L., & Rozas-Vásquez, D. (2025). From Forest to Building: Enhancing the Use of Lower Grade Wood for Climate-Responsive Construction. ***Trees, Forests and People***, ***22***, 101022. https://doi.org/10.1016/j.tfp.2025.101022

Pagella, G., Urso, T., Mirra, M., Naldini, S., & Van De Kuilen, J.-W. (2025). Traditional Wooden Foundation Piles in Amsterdam and Venice: Techniques for the Assessment of Their State of Conservation. ***Wood Material Science & Engineering***, 1–16. https://doi.org/10.1080/17480272.2025.2466104

Placzek, G., Schwerdtner, P., Placzek, G., & Schwerdtner, P. (2024). A Global Snapshot of 3D-Printed Buildings: Uncovering Robotic-Oriented Fabrication Strategies. ***Buildings***, ***14***(11). https://doi.org/10.3390/buildings14113410

Rahbek, L., Terp, C., Alibrandi, U., & Kirkegaard, P. (2023). Stock Optimization of Naturally Curved Wood Logs on Freeform Truss Structures. ***Journal of the International Association for Shell and Spatial Structures***, ***64***(4), 257–265. https://doi.org/10.20898/j.iass.2023.029

Ramage, M. H., Burridge, H., Busse-Wicher, M., Fereday, G., Reynolds, T., Shah, D. U., Wu, G., Yu, L., Fleming, P., Densley-Tingley, D., Allwood, J., Dupree, P., Linden, P. F., & Scherman, O. (2017). Harvest. ***Renewable and Sustainable Energy Reviews***, ***68***, 333–359. https://doi.org/10.1016/j.rser.2016.09.107

Sandvik, Y. J., Futsæther, C. M., Liland, K. H., Tomic, O., Sandvik, Y. J., Futsæther, C. M., Liland, K. H., & Tomic, O. (2024). A Comparative Literature Review of Machine Learning and Image Processing Techniques Used for Scaling and Grading of Wood Logs. ***Forests***, ***15***(7). https://doi.org/10.3390/f15071243

Schittich, C. (2019). ***Vernacular Architecture: Atlas for Living throughout the World***. Birkhäuser Verlag.

Shi, A., Bergsagel, D., Owen, J., & Heisel, F. (2025). Scalable AI-driven Automation for Visual Lumber Grading. ***Journal of Cleaner Production***, ***533***, 146986. https://doi.org/10.1016/j.jclepro.2025.146986

Song, L., Mohamed, Y., & AbouRizk, S. M. (2009). Early Contractor Involvement in Design and Its Impact on Construction Schedule Performance. ***Journal of Management in Engineering***, ***25***(1), 12–20. https://doi.org/10.1061/(ASCE)0742-597X(2009)25:1(12)

Stevens, J. C. (2015). ***Digital Vernacular: Architectural Principles, Tools, and Processes***. Routledge. https://doi.org/10.4324/9781315730271

Stovall, A. E. L., Anderson-Teixeira, K. J., & Shugart, H. H. (2018). Assessing Terrestrial Laser Scanning for Developing Non-Destructive Biomass Allometry. ***Forest Ecology and Management***, ***427***, 217–229. https://doi.org/10.1016/j.foreco.2018.06.004

Suffian, S., Dzombak, R., & Mehta, K. (2016). Future Directions for Nonconventional and Vernacular Material Research and Applications. In ***Nonconventional and Vernacular Construction Materials:***

Nonconventional and Vernacular Construction Materials (pp. 63–80). Elsevier. https://doi.org/10.1016/B978-0-08-100038-0.00003-2

Sun, Q., Reisach, D., Langenberg, S., & Dillenburger, B. (2026). Computational Design Methods for Geometry-Driven Upcycling of Found Objects in Construction. *Automation in Construction*, *183*, 106803. https://doi.org/10.1016/j.autcon.2026.106803

Syafuan, W. M., Ab Moin, N. A., & Satar, A. A. (2025). Evaluating the Accuracy of Smartphone Light Detection and Ranging Technology for Three-Dimensional Building Mapping. *IOP Conference Series: Earth and Environmental Science*, *1548*(1), 12011. https://doi.org/10.1088/1755-1315/1548/1/012011

Thünen. (2019,). *World timber trade in 2016*. Thünen Institut für Internationale Waldwirtschaft und Forstökonomie.

Torvinen, J., Sieder-Semlitsch, J., Pedersen, J., Aagaard, A. K., Larsen, N. M., & Kuittinen, M. (2025). Digital Age Crafting from Misfit Wood: A Bundled Pillar. In *Structures and Architecture: Structures and Architecture*. CRC Press.

Trochta, J., Krůček, M., Vrška, T., & Král, K. (2017). 3D Forest: An Application for Descriptions of Three-Dimensional Forest Structures Using Terrestrial LiDAR. *PLOS ONE*, *12*(5), e176871. https://doi.org/10.1371/journal.pone.0176871

Tuvayanond, W., Prasittisopin, L., Tuvayanond, W., & Prasittisopin, L. (2023). Design for Manufacture and Assembly of Digital Fabrication and Additive Manufacturing in Construction: A Review. *Buildings*, *13*(2). https://doi.org/10.3390/buildings13020429

Vestartas, P. (2021). *Design-to-Fabrication Workflow for Raw-Sawn-Timber Using Joinery Solver* [Doctoral dissertation]. https://doi.org/10.5075/epfl-thesis-8928

Vestartas, P., & Weinand, Y. (2020). Laser Scanning with Industrial Robot Arm for Raw-wood Fabrication. *International Symposium on Automation and Robotics in Construction (ISARC) Proceedings*, 773–780. https://doi.org/10.22260/ISARC2020/0107

Vrouwenvelder, T., Dimova, S., Sousa, M. L., Marková, J., Mancini, G., Kuhlmann, U., Taras, A., Jockwer, R., Jäger, W., Schweckendiek, T., Franchin, P., Skejić, D., Sørensen, J. D., Spehl, P., Stacy, M., Feldmann, M., Croce, P., Zhao, B., Tanner, P., … Spross, J. (2024). *Reliability Background of the Eurocodes. Support to the Implementation, Harmonization and Further Development of the Eurocodes*.

Wang, X., Kumar, V., Kumari, A., & Kuzmin, E. (2022). Impact of Digital Technology on Supply Chain Efficiency in Manufacturing Industry. In V. Kumar, J. Leng, V. Akberdina, & E. Kuzmin (Eds.), *Digital Transformation in Industry: Digital Transformation in Industry*. https://doi.org/10.1007/978-3-030-94617-3_25

Weiser, H., Schäfer, J., Winiwarter, L., Krašovec, N., Fassnacht, F. E., & Höfle, B. (2022). Individual Tree Point Clouds and Tree Measurements from Multi-Platform Laser Scanning in German Forests. *Earth System Science Data*, *14*(7), 2989–3012. https://doi.org/10.5194/essd-14-2989-2022

Wrzesiński, G., Pawluk, K., Lendo-Siwicka, M., & Kowalski, J. (2023). Analysis of Technology, Time and Costs of Three Methods of Building a Single-Family House: Traditional Brick, Reinforced Concrete Prefabrication, Timber Frame. ***Archives of Civil Engineering***, ***nr2***. https://doi.org/10.24425/ace.2023.145250

Zhong, X., Hu, M., Deetman, S., Steubing, B., Lin, H. X., Hernandez, G. A., Harpprecht, C., Zhang, C., Tukker, A., & Behrens, P. (2021). Global Greenhouse Gas Emissions from Residential and Commercial Building Materials and Mitigation Strategies to 2060. ***Nature Communications***, ***12***(1), 6126. https://doi.org/10.1038/s41467-021-26212-z

11 Appendix

11.1 Fachwerk system configuration parameters

All tunable parameters of the design system are centralised in `fachwerk/config.py`, covering frame geometry and grid resolution, placement heuristics, cross-section dimensions, applied loads, and timber material properties. Structural values follow EN 338 D30 hardwood characteristics as a conservative assumption for heterogeneous natural oak, with no partial safety factors applied since the study compares relative — not code-compliant — performance.

Parameter	Default	Meaning
Geometry and scale		
`FRAME_WIDTH_MM`	4000	Wall panel width [mm]
`FRAME_HEIGHT_MM`	3000	Wall panel height [mm]
`GRID_NX`, `GRID_NY`	7, 7	Grid tiles in x and y (49 points total)
`N_CELL` / `MAX_BOUNDARY_PTS`	20	Boundary sample points per cell
`OBJ_UNIT_SCALE`	1000	Convert OBJ metres to mm
Placement		
`STOCK_DESCENDING`	True	Sort stock longest-first before placement
`USE_CUTOFFS`	True	Re-use off-cut segments as additional stock
`MIN_CUTOFF_LEN`	500 mm	Minimum off-cut length to keep
`MIN_CELL_AREA_MM2`	100 000 mm^2	Skip placement if sub-cell smaller
`MAX_RETRIES`	3	Failed attempts before discarding branch
Structural — material and section		
`TIMBER_E`	9 000 N/mm^2	Young's modulus (reduced for natural oak)
`TIMBER_DENSITY_KG_M3`	700 kg/m^3	Oak density for self-weight
`FRAME_SECTION_WIDTH_MM`	120 mm	Frame column/beam width
`FRAME_SECTION_DEPTH_MM`	160 mm	Frame column/beam depth
`BRANCH_SECTION_WIDTH_MM`	80 mm	Branch infill member width
`BRANCH_SECTION_DEPTH_MM`	120 mm	Branch infill member depth
Loads		
`FEA_GRAVITY_LOAD_N`	−25 000 N	UDL on top beam (≈ 6.25 kN/m over 4 m)
`FEA_WIND_LOAD_N`	10 000 N	Windward pressure (≈ 3.33 kN/m over 3 m)
`FEA_WIND_SUCTION_RATIO`	0.5	Leeward suction fraction (EN 1991-1-4)
Timber strength — EN 338 D30		
`TIMBER_FMK`	30 N/mm^2	Characteristic bending strength
`TIMBER_FT0K`	18 N/mm^2	Tension strength parallel to grain
`TIMBER_FC0K`	23 N/mm^2	Compression strength parallel to grain

Table 4: Tunable parameters in `fachwerk/config.py`.

11.2 Source code

This section contains the complete source code of the KM0DFAB computational pipeline. The pipeline processes AdTree skeleton point clouds to extract and allocate timber elements for Fachwerk facade construction.

Files are listed in logical execution order: configuration, main orchestrator, geometry processing, curvature scoring and allocation, input/output, and visualisation.

11.2.1 Appendix — Core Analysis Source Code

Generated 2026-05-28 by `make_appendix.py`.

Contents

Part A — Skeleton assessment (`skeleton_assessment/`)

- **A.1.1** `skeleton_assessment/main.py` — Stage-1 driver: per-tree load → filter → segment → allocate, then merge trees and export. The canonical pipeline entry point.
- **A.1.2** `skeleton_assessment/configuration.py` — Stage-1 parameters: PLY inputs, diameter filter, planarity/curvature thresholds, structural roles.
- **A.1.3** `skeleton_assessment/io_F.py` — PLY loading (vertices+radius, edges) and Rhino-style OBJ/CSV export of the allocated/curved/straight subsets.
- **A.1.4** `skeleton_assessment/geometry.py` — Graph construction, branch extraction, fork trimming, Dijkstra planar sub-segmentation, curvature stats, canonical-frame transform.
- **A.1.5** `skeleton_assessment/gorges.py` — Gorges et al. greedy bow-height allocation: picks the straightest window of the required length for each structural role.

Part B — Fachwerk optimisation (`digital_fachwerk/`)

- **A.2.1** `digital_fachwerk/fachwerk/config.py` — Single source of truth for Stage-2 parameters: frame size, 7×7 grid, FEA loads, oak (D30) material.
- **A.2.2** `digital_fachwerk/fachwerk/geometry.py` — Pure geometry helpers: OBJ branch loader, normalisation, curvature profile, boundary sampling, distance-to-chord.
- **A.2.3** `digital_fachwerk/fachwerk/genome.py` — decode_genome(genome, n_steps, layout) and detect_genome_layout — decode either the interleaved or block genome layout from a result JSON.
- **A.2.4** `digital_fachwerk/fachwerk/placement.py` — replay() — the core placement simulation: places branches one by one and splits Shapely cells. Called thousands of times per run.
- **A.2.5** `digital_fachwerk/fachwerk/fea.py` — build_structural_model + analyze — OpenSeesPy linear-static solve; returns max displacement, per-element utilisation.
- **A.2.6** `digital_fachwerk/fachwerk/optimizer.py` — FachwerkProblem (pymoo ElementwiseProblem) and the NSGA-II run loop: 2 objectives, 1 constraint, interleaved genome, worst-case wind symmetry.

A.0 Pipeline overview

The two stages are strictly sequential and communicate through OBJ centreline files:

```
.ply (TLS skeleton)
      │  io_F.load_ply
      ▼
Stage 1  main.py: io_F → geometry → gorges
(skeleton_assessment/)
      │  export_rhino_obj /
export_curved_obj
      ▼
stock OBJ (curved / straight centrelines)
      │
fachwerk.geometry.load_branches_from_obj
      ▼
Stage 2  placement.replay → fea.analyze
(digital_fachwerk/)
      │  NSGA-II
(optimizer.run_optimization)
```

▼
run_<ts>.json (Pareto front of frame designs)

11.2.2 Stage 1 — Skeleton assessment

A.1.1 `skeleton_assessment/main.py`

Stage-1 driver: per-tree load → filter → segment → allocate, then merge trees and export. The canonical pipeline entry point.

242 lines

```
import os
import numpy as np
from collections import Counter

from configuration import (PLY_FILES,
TREE_SPACING_M,
                           MIN_DIAMETER_M,
MAX_PLANARITY_RESIDUAL_M,
                           ROLES,
MIN_SEGMENT_LENGTH_M, SMOOTH_WINDOW_M,
CURVATURE_STEP_M,
FORK_TRIM_DIAMETERS, MIDDLE_CUT_PENALTY,
MIN_CURVED_LENGTH_M,
MAX_CURVATURE_CM_PER_M,
STACK_TREES_VERTICALLY, TREE_STACK_GAP_M,
SHOW_DEBUG_WINDOWS)

from geometry import
(filter_edges_by_diameter,
merge_coincident_tips,
                     build_adjacency,
extract_branches, trim_branches_at_forks,
segment_branch_dijkstra,
compute_node_curvatures,
compute_branch_stats)
from gorges import greedy_allocate
from io_F import (load_ply,
export_branches_csv, export_elements_ply,
export_allocations_csv,
export_straight_elements_ply,
                     export_rhino_obj,
export_curved_obj, export_stage1_obj)

#
============================================================
# PHASE 1 — Per-tree processing
#
============================================================
tree_results = []
z_stack_offset = 0.0  # cumulative Z offset
for vertical stacking

for tree_idx, ply_file in
enumerate(PLY_FILES):
    tree_name = os.path.splitext(ply_file)
[0]  # e.g. "almere-001_skeleton"
    print(f"\n{'='*60}")
    print(f"Processing tree {tree_idx+1}/
{len(PLY_FILES)}: {tree_name}")
    print(f"{'='*60}")

    # --- Load ---
    vertices, edges = load_ply(ply_file)

    # --- Spatial offset: centre XY, then
stack vertically (Z) or side-by-side (X)
---
    if len(PLY_FILES) > 1:
        centroid_xy =
vertices[:, :2].mean(axis=0)
        vertices[:, 0] -= centroid_xy[0]
        vertices[:, 1] -= centroid_xy[1]
        if STACK_TREES_VERTICALLY:
            z_min = vertices[:, 2].min()
            vertices[:, 2] -= z_min
# sit tree on Z=0 locally
            vertices[:, 2] +=
z_stack_offset  # lift to stacked position
        else:
            vertices[:, 0] += tree_idx *
TREE_SPACING_M

    # --- Build graph ---
    edges_filtered =
filter_edges_by_diameter(vertices, edges,
MIN_DIAMETER_M)
    edges_filtered =
merge_coincident_tips(vertices,
edges_filtered, edges)
    adjacency      =
build_adjacency(edges_filtered)

    # --- Debug: graph topology ---
    degree_counts =
Counter(dict(adjacency.degree()).values())
    print(f"Graph:
{adjacency.number_of_nodes()} nodes,
{adjacency.number_of_edges()} edges")
    print(f"Degree distribution: { {k:
degree_counts[k] for k in
sorted(degree_counts)} }")
    print(f"  Tips  (deg 1):
```

```
{degree_counts[1]}")
    print(f"  Forks (deg>2): {sum(v for k,v
in degree_counts.items() if k > 2)}")

    # --- Extract branches ---
    branches     =
extract_branches(edges_filtered, adjacency,
vertices)
    branches     =
trim_branches_at_forks(branches, vertices,
adjacency, FORK_TRIM_DIAMETERS)
    print(f"Branches extracted:
{len(branches)},  lengths: {sorted(len(b)
for b in branches)[:20]}...")
    node_curvatures =
compute_node_curvatures(vertices,
adjacency)
    branch_stats    =
compute_branch_stats(vertices, branches,
node_curvatures)

    # Tag each stat with tree source
    for s in branch_stats:
        s['tree_source'] = tree_name

    # --- Segment by planarity ---
    sub_segs   = []
    branch_ids = []
    for i, b in enumerate(branches):
        segs, _ =
segment_branch_dijkstra(b, vertices,
MAX_PLANARITY_RESIDUAL_M)
        sub_segs.extend(segs)
        branch_ids.extend([i] * len(segs))

    tree_results.append({
        'tree_name':    tree_name,
        'vertices':     vertices,
        'edges':        edges,
        'edges_filtered': edges_filtered,
        'branches':     branches,
        'branch_stats': branch_stats,
        'sub_segs':     sub_segs,
        'branch_ids':   branch_ids,
        'n_vertices':   len(vertices),
        'n_branches':   len(branches),
    })

    if STACK_TREES_VERTICALLY and
len(PLY_FILES) > 1:
        z_extent = vertices[:, 2].max() -
vertices[:, 2].min()
        z_stack_offset += z_extent +
TREE_STACK_GAP_M

#
============================================================
# PHASE 2 — Merge all trees into combined
arrays
#
============================================================
print(f"\n{'='*60}")
print(f"Merging {len(tree_results)} trees")
print(f"{'='*60}")

vertex_offset  = 0
branch_offset  = 0

combined_vertices_list      = []
combined_edges_list         = []
combined_edges_filtered_list = []
combined_branches           = []
combined_sub_segs           = []
combined_branch_ids         = []
combined_tree_sources       = []   # one
label per sub-segment
combined_branch_stats       = []

for tr in tree_results:
    # Vertices — already spatially offset
combined_vertices_list.append(tr['vertices'])

    # Edges — offset indices
    e_all = np.asarray(tr['edges'],
dtype=int) + vertex_offset
    combined_edges_list.append(e_all)

    ef = [(u + vertex_offset, v +
vertex_offset) for u, v in
tr['edges_filtered']]
    combined_edges_filtered_list.extend(ef)

    # Branches — offset node indices
    for branch in tr['branches']:
        combined_branches.append([n +
vertex_offset for n in branch])

    # Sub-segments — offset node indices
    for seg in tr['sub_segs']:
        combined_sub_segs.append([n +
vertex_offset for n in seg])

    # Branch IDs — offset by cumulative
branch count
    combined_branch_ids.extend([bid +
branch_offset for bid in tr['branch_ids']])

    # Tree sources — one label per sub-
```

```
segment

combined_tree_sources.extend([tr['tree_name']]
* len(tr['sub_segs']))

    # Branch stats

combined_branch_stats.extend(tr['branch_stats'])

    vertex_offset += tr['n_vertices']
    branch_offset += tr['n_branches']

combined_vertices       =
np.vstack(combined_vertices_list)
combined_edges          =
np.vstack(combined_edges_list)
combined_edges_filtered =
combined_edges_filtered_list  # list of
tuples
combined_adjacency      =
build_adjacency(combined_edges_filtered)

print(f"Combined: {len(combined_vertices)}
vertices, "
      f"{len(combined_branches)} branches,
{len(combined_sub_segs)} sub-segments")

#
============================================================
# PHASE 3 — Combined allocation
#
============================================================
allocations, curved, straight_leftovers =
greedy_allocate(combined_vertices,
combined_sub_segs,

combined_branch_ids, ROLES,

MIN_SEGMENT_LENGTH_M, SMOOTH_WINDOW_M,
CURVATURE_STEP_M,

tree_sources=combined_tree_sources,

middle_cut_penalty=MIDDLE_CUT_PENALTY,

min_curved_length_m=MIN_CURVED_LENGTH_M,

max_curvature_cm_per_m=MAX_CURVATURE_CM_PER_M)

#
============================================================
# Summary statistics
#
============================================================
tree_names = [tr['tree_name'] for tr in
tree_results]

print(f"\n{'='*80}")
print("MULTI-TREE SUMMARY")
print(f"{'='*80}")
header = f"{'Tree':<30} {'Branches':>8}
{'Total L(m)':>10} {'Straight':>8} {'Curved
LO':>9} {'Straight LO':>11} {'Yield %':>8}"
print(header)
print("-" * 80)

for name in tree_names:
    t_stats   = [s for s in
combined_branch_stats if
s.get('tree_source') == name]
    t_allocs  = [a for a in allocations if
a.get('tree_source') == name]
    t_curved  = [c for c in curved if
c.get('tree_source') == name]
    t_str_lo  = [s for s in
straight_leftovers if s.get('tree_source')
== name]

    total_branch_len = sum(s['length_m']
for s in t_stats)
    total_straight   =
sum(a['role_length_m'] for a in t_allocs)
    yield_pct = (total_straight /
total_branch_len * 100) if total_branch_len
> 0 else 0
    print(f"{name:<30} {len(t_stats):>8}
{total_branch_len:>10.2f} "
          f"{len(t_allocs):>8}
{len(t_curved):>9} {len(t_str_lo):>11}
{yield_pct:>7.1f}%")

total_len = sum(s['length_m'] for s in
combined_branch_stats)
total_str = sum(a['role_length_m'] for a in
allocations)
total_yield = (total_str / total_len * 100)
if total_len > 0 else 0
print("-" * 80)
print(f"{'TOTAL':<30}
{len(combined_branch_stats):>8}
{total_len:>10.2f} "
      f"{len(allocations):>8}
{len(curved):>9}
{len(straight_leftovers):>11}
{total_yield:>7.1f}%")
print("=" * 80)

#
============================================================
# Export
#
```

```
# ============================================================
output_dir = "./output"
os.makedirs(output_dir, exist_ok=True)

export_branches_csv(combined_branch_stats,
f"{output_dir}/branches.csv")
export_elements_ply(combined_vertices,
combined_branches,
f"{output_dir}/branches_catalog.ply")

# Straight roles
export_allocations_csv(allocations,
f"{output_dir}/allocations.csv")
export_straight_elements_ply(combined_vertices,
allocations,
f"{output_dir}/straight.ply")

# Curved leftovers (fachwerk optimizer
input)
export_allocations_csv(curved,
f"{output_dir}/curved_leftovers.csv")
export_straight_elements_ply(combined_vertices,
curved,
f"{output_dir}/curved_leftovers.ply")
export_curved_obj(combined_vertices,
curved,
f"{output_dir}/curved_leftovers.obj")

# Straight leftovers (other use)
export_allocations_csv(straight_leftovers,
f"{output_dir}/straight_leftovers.csv")
export_straight_elements_ply(combined_vertices,
straight_leftovers,
f"{output_dir}/straight_leftovers.ply")

# Combined Rhino file (all groups)
export_rhino_obj(combined_vertices,
allocations, curved, straight_leftovers,
f"{output_dir}/rhino_export.obj")

# Stage-1 raw skeleton + radius-gradient
rings (for Rhino visualisation)
export_stage1_obj(combined_vertices,
combined_edges, combined_edges_filtered,
                  combined_adjacency,
f"{output_dir}/stage1_raw_skeleton.obj")

#
============================================================
# Visualise
#
============================================================
from visualisation import
(visualise_stage1, visualise_stage1b,
visualise_stage2, visualise_stage2b,
visualise_stage3,
visualise_combined, visualise_grid)

if SHOW_DEBUG_WINDOWS:
    visualise_stage1(combined_vertices,
combined_edges, combined_edges_filtered,
combined_adjacency)
    #visualise_stage1b(combined_vertices,
combined_edges, combined_edges_filtered,
combined_adjacency)
    #visualise_stage2(combined_vertices,
combined_edges_filtered, combined_branches)
    #visualise_stage2b(combined_vertices,
combined_edges_filtered, combined_sub_segs)
    #visualise_stage3(combined_vertices,
combined_edges_filtered, allocations,
curved)
    visualise_combined(combined_vertices,
combined_edges_filtered, allocations,
curved)

visualise_grid(combined_vertices,
allocations, curved)
```

A.1.2 skeleton_assessment/configuration.py

Stage-1 parameters: PLY inputs, diameter filter, planarity/curvature thresholds, structural roles.

36 lines

```
## USER INPUTS
from pathlib import Path
_HERE = Path(__file__).parent

PLY_FILES = [
    str(_HERE / "tree-plys" /
"almere-001_skeleton.ply"),
]

TREE_SPACING_M = 10.0  # metres between
tree centres along X-axis (used when
STACK_TREES_VERTICALLY = False)
STACK_TREES_VERTICALLY = True   # stack
multiple trees in Z instead of side-by-side
in X
TREE_STACK_GAP_M = 2.0          # vertical
gap (m) between adjacent tree bounding
boxes
SHOW_DEBUG_WINDOWS = True     # True to re-
enable all 5 diagnostic stage windows

# Only keep edges where BOTH endpoints have
diameter > MIN_DIAMETER_M
```

```
MIN_DIAMETER_M = 0.2

# Segmentation thresholds
MAX_PLANARITY_RESIDUAL_M = 0.05   # max
distance out of plane (meter)
# MAX_LOCAL_CURVATURE_DEG  = 5.0   # max
local deflection angle at a node

# Requiered segments
ROLES = [
    ("beam_4m", 4.0, 2),
    ("post_3m", 3.0, 2)#,
]

SMOOTH_WINDOW_M = 0.2          # [m]
smoothing moving-average window
CURVATURE_STEP_M = 0.05        # [m]
sliding-window step for curvature
calculation
FORK_TRIM_DIAMETERS = 2.0      # [-]   trim
distance at fork end = N × fork-end
diameter
MIDDLE_CUT_PENALTY = 1.1       # [-]   max
penalty at piece centre (1.0 = disabled)
MIN_CURVED_LENGTH_M = 0.5      # [m]
minimum arc length for curved (non-role)
leftovers
MAX_CURVATURE_CM_PER_M = 1     # [cm/m] max
bow-height curvature for straight role
elements

## CONFIGS
MIN_SEGMENT_LENGTH_M = min(length for _,
length, _ in ROLES)
```

A.1.3 skeleton_assessment/io_F.py

PLY loading (vertices+radius, edges) and Rhino-style OBJ/CSV export of the allocated/curved/straight subsets.

581 lines

```
# READ PLY FILE
import os
from plyfile import PlyData
import numpy as np
import struct  # needed for struct.pack in
PLY writing
from visualisation import
transform_branch_to_canonical


def load_ply(filepath: str):
    return parse_adtree_ply(filepath)


def parse_adtree_ply(filepath: str):
    """
    Read an AdTree PLY skeleton:
    - vertex element must have x,y,z,radius
    - edge element must have vertex_indices
(list of 2 ints per edge)

    Returns:
        vertices: (N,4) float array
[x,y,z,radius]
        edges:    (M,2) int array   [u,v]
    """
    plydata = PlyData.read(filepath)

    v = plydata["vertex"]
    vertices = np.column_stack([v["x"],
v["y"], v["z"], v["radius"]]).astype(float)

    e = plydata["edge"]
    edges = np.zeros((len(e), 2),
dtype=int)
    for i in range(len(e)):
        idx = e["vertex_indices"][i]
        edges[i, 0] = idx[0]
        edges[i, 1] = idx[1]

    return vertices, edges


## OUTPUTS

import csv

def export_allocations_csv(allocations,
output_path):
    fieldnames = [
        'role_name', 'role_length_m',
'curvature_cm_per_m',
        'bow_height_m', 'window_start_m',
'window_end_m',
        'branch_id', 'tree_source'
    ]
    with open(output_path, 'w', newline='')
as f:
        writer = csv.DictWriter(f,
fieldnames=fieldnames,
extrasaction='ignore')
        writer.writeheader()
        writer.writerows(allocations)

def export_branches_csv(branch_stats,
output_path):
```

```
    if not branch_stats:
        return

    rows = sorted(branch_stats, key=lambda
s: s["length_m"], reverse=True)

    has_tree_source = any('tree_source' in
s for s in rows)

    fieldnames = [
        "branch_id", "length_m",
"avg_diameter_m", "start_diameter_m",
        "end_diameter_m", "taper_m",
"mean_curvature_deg",
        "max_curvature_deg", "planarity_m",
"n_nodes"
    ]
    if has_tree_source:
        fieldnames.insert(1, "tree_source")

    with open(output_path, 'w', newline='')
as f:
        writer = csv.DictWriter(f,
fieldnames=fieldnames)
        writer.writeheader()
        for i, s in enumerate(rows,
start=1):
            row = {
                "branch_id":          i,
                "length_m":
round(s["length_m"], 3),
                "avg_diameter_m":
round((s["start_diameter_m"] +
s["end_diameter_m"]) / 2, 3),
                "start_diameter_m":
round(s["start_diameter_m"], 3),
                "end_diameter_m":
round(s["end_diameter_m"], 3),
                "taper_m":
round(abs(s["start_diameter_m"] -
s["end_diameter_m"]), 3),
                "mean_curvature_deg":
round(s["mean_curvature_deg"], 2),
                "max_curvature_deg":
round(s["max_curvature_deg"], 2),
                "planarity_m":
round(s["planarity_m"], 3),
                "n_nodes":
s["n_nodes"],
            }
            if has_tree_source:
                row["tree_source"] =
s.get("tree_source", "")
            writer.writerow(row)

    print(f"Exported {len(rows)} branches
to: {output_path}")


def export_elements_ply(vertices, branches,
output_path, y_spacing=2.0):
    from visualisation import
transform_branch_to_canonical
    branch_lengths = []
    for branch in branches:
        length = 0.0
        for i in range(len(branch) - 1):
            a = vertices[branch[i], :3]
            b = vertices[branch[i + 1], :3]
            length += np.linalg.norm(b - a)
        branch_lengths.append(length)

    sorted_indices =
sorted(range(len(branches)), key=lambda i:
branch_lengths[i], reverse=True)
    sorted_branches = [branches[i] for i in
sorted_indices]

    all_vertices = []
    all_edges = []
    vertex_offset = 0

    for branch_idx, branch in
enumerate(sorted_branches):
        branch_id = branch_idx + 1
        y_offset = branch_idx * y_spacing
        coords, radii, _ =
transform_branch_to_canonical(vertices,
branch, y_offset)
        for i in range(len(coords)):
            all_vertices.append({'x':
coords[i, 0], 'y': coords[i, 1], 'z':
coords[i, 2],
                                 'radius':
radii[i], 'branch_id': branch_id})
        for i in range(len(coords) - 1):
            all_edges.append({'vertex1':
vertex_offset + i, 'vertex2': vertex_offset
+ i + 1,
                              'branch_id':
branch_id})
        vertex_offset += len(coords)

    with open(output_path, 'wb') as f:
        header = (f"ply\nformat
binary_little_endian 1.0\n"
                  f"element vertex
{len(all_vertices)}\n"
```

```
                f"property float
x\nproperty float y\nproperty float z\n"
                f"property float
radius\nproperty int branch_id\n"
                f"element edge
{len(all_edges)}\n"
                f"property int
vertex1\nproperty int vertex2\nproperty int
branch_id\n"
                f"end_header\n")
        f.write(header.encode('ascii'))
        for v in all_vertices:
            f.write(struct.pack('ffffi',
v['x'], v['y'], v['z'], v['radius'],
v['branch_id']))
        for e in all_edges:
            f.write(struct.pack('iii',
e['vertex1'], e['vertex2'],
e['branch_id']))

    print(f"Exported {len(sorted_branches)}
branches to: {output_path}")


def export_straight_elements_ply(vertices,
allocations, output_path, y_spacing=2.0):
    from visualisation import
transform_branch_to_canonical
    all_vertices = []
    all_edges = []
    element_metadata = []
    vertex_offset = 0
    element_idx = 0

    for alloc in allocations:
        window_nodes =
alloc['global_node_indices']
        if len(window_nodes) < 2:
            continue
        y_offset = element_idx * y_spacing
        branch_id = element_idx + 1
        element_metadata.append({
            'branch_id': branch_id,
            'role': alloc['role_name'],
            'role_length_m':
alloc['role_length_m'],
            'curvature_cm_per_m':
f"{alloc['curvature_cm_per_m']:.4f}",
            'tree_source':
alloc.get('tree_source', ''),
            'parent_branch_id':
alloc.get('branch_id', ''),
        })
        coords, radii, _ =
transform_branch_to_canonical(vertices,
window_nodes, y_offset)
        for i in range(len(coords)):
            all_vertices.append({'x':
coords[i, 0], 'y': coords[i, 1], 'z':
coords[i, 2],
                                 'radius':
radii[i], 'branch_id': branch_id})
        for i in range(len(coords) - 1):
            all_edges.append({'vertex1':
vertex_offset + i, 'vertex2': vertex_offset
+ i + 1,
                              'branch_id':
branch_id})
        vertex_offset += len(coords)
        element_idx += 1

    label =
os.path.splitext(os.path.basename(output_path))
[0]
    if not all_vertices:
        print(f"No {label} elements to
export.")
        return output_path

    with open(output_path, 'wb') as f:
        header = (f"ply\nformat
binary_little_endian 1.0\n"
                  f"element vertex
{len(all_vertices)}\n"
                  f"property float
x\nproperty float y\nproperty float z\n"
                  f"property float
radius\nproperty int branch_id\n"
                  f"element edge
{len(all_edges)}\n"
                  f"property int
vertex1\nproperty int vertex2\nproperty int
branch_id\n"
                  f"end_header\n")
        f.write(header.encode('ascii'))
        for v in all_vertices:
            f.write(struct.pack('ffffi',
v['x'], v['y'], v['z'], v['radius'],
v['branch_id']))
        for e in all_edges:
            f.write(struct.pack('iii',
e['vertex1'], e['vertex2'],
e['branch_id']))

    csv_path = output_path.replace('.ply',
'.csv')
    with open(csv_path, 'w', newline='') as
csvfile:
```

```
        fieldnames = ['branch_id', 'role',
'role_length_m', 'curvature_cm_per_m',
                      'tree_source',
'parent_branch_id']
        writer = csv.DictWriter(csvfile,
fieldnames=fieldnames)
        writer.writeheader()
        writer.writerows(element_metadata)

    print(f"Exported {element_idx} {label}
elements to: {output_path}")
    return output_path


#
============================================================
# Rhino OBJ export — centrelines +
variable-radius tube meshes
#
============================================================

def _ring_points(center, normal, radius,
K):
    """Return (K, 3) array of ring vertices
for a tube cross-section."""
    n = np.asarray(normal, dtype=float)
    nn = np.linalg.norm(n)
    n = n / nn if nn > 1e-12 else
np.array([0., 0., 1.])
    helper = np.array([1., 0., 0.]) if
abs(n[0]) < 0.9 else np.array([0., 1., 0.])
    u = np.cross(n, helper)
    ul = np.linalg.norm(u)
    if ul < 1e-12:
        u = np.cross(n, np.array([0., 0.,
1.]))
        ul = np.linalg.norm(u)
    u /= ul
    v = np.cross(n, u)
    angles = np.linspace(0, 2 * np.pi, K,
endpoint=False)
    return np.asarray(center, dtype=float)
+ radius * (np.cos(angles)[:, None] * u +
np.sin(angles)[:, None] * v)


def export_rhino_obj(vertices, allocations,
curved, straight_leftovers, output_path,
ring_segments=12):
    """
    Export centrelines + variable-radius
tube meshes to OBJ for Rhino.

    OBJ groups:
        centrelines_straight          —
polyline l-statements for role-allocated
elements
        centrelines_curved            —
polyline l-statements for curved leftovers
        centrelines_straight_leftover —
polyline l-statements for straight
leftovers
        tubes_straight                —
triangle mesh for allocated elements
        tubes_curved                  —
triangle mesh for curved leftovers
        tubes_straight_leftover       —
triangle mesh for straight leftovers

    Args:
        vertices:      (N, 4) float array
[x, y, z, radius]
        allocations:   list of allocation
dicts
        curved:        list of curved
element dicts
        output_path:   path to .obj output
file
        ring_segments: cross-section
polygon vertex count (default 12)
    """
    K = ring_segments
    v_idx = 1  # OBJ vertex indices are 1-
based

    def node_normal(coords, i):
        N = len(coords)
        if i == 0:
            d = coords[1] - coords[0]
        elif i == N - 1:
            d = coords[N - 1] - coords[N -
2]
        else:
            d = coords[i + 1] - coords[i -
1]
        norm = np.linalg.norm(d)
        return d / norm if norm > 1e-12
else np.array([0., 0., 1.])

    with open(output_path, 'w') as fh:
        fh.write("# KM0DFAB Rhino
export\n")
        fh.write(f"# ring_segments:
{K}\n\n")

        def write_centreline(elem):
            nonlocal v_idx
            nodes =
```

```
elem['global_node_indices']
            if len(nodes) < 2:
                return
            start = v_idx
            for n in nodes:
                xyz = vertices[n, :3]
                fh.write(f"v {xyz[0]:.6f}
{xyz[1]:.6f} {xyz[2]:.6f}\n")
                v_idx += 1
            indices = " ".join(str(start +
i) for i in range(len(nodes)))
            fh.write(f"l {indices}\n")

        def write_tube(elem):
            nonlocal v_idx
            nodes =
elem['global_node_indices']
            if len(nodes) < 2:
                return
            coords = vertices[nodes, :3]
            radii_arr = vertices[nodes, 3]
            N_nodes = len(nodes)

            # Write ring vertices, record
start index of each ring
            ring_starts = []
            for i in range(N_nodes):
                r =
max(float(radii_arr[i]), 1e-6)
                ring_pts =
_ring_points(coords[i], node_normal(coords,
i), r, K)
                ring_starts.append(v_idx)
                for pt in ring_pts:
                    fh.write(f"v
{pt[0]:.6f} {pt[1]:.6f} {pt[2]:.6f}\n")
                    v_idx += 1

            # End-cap centre vertices
            start_cap = v_idx
            fh.write(f"v {coords[0, 0]:.6f}
{coords[0, 1]:.6f} {coords[0, 2]:.6f}\n")
            v_idx += 1
            end_cap = v_idx
            fh.write(f"v {coords[-1,
0]:.6f} {coords[-1, 1]:.6f} {coords[-1,
2]:.6f}\n")
            v_idx += 1

            # Tube side faces (quads split
into 2 triangles)
            for i in range(N_nodes - 1):
                rs = ring_starts[i]
                rn = ring_starts[i + 1]
                for k in range(K):
                    k1 = (k + 1) % K
                    fh.write(f"f {rs+k}
{rn+k} {rn+k1}\n")
                    fh.write(f"f {rs+k}
{rn+k1} {rs+k1}\n")

            # Start end-cap fan
            rs0 = ring_starts[0]
            for k in range(K):
                k1 = (k + 1) % K
                fh.write(f"f {start_cap}
{rs0+k1} {rs0+k}\n")

            # End end-cap fan
            rsN = ring_starts[-1]
            for k in range(K):
                k1 = (k + 1) % K
                fh.write(f"f {end_cap}
{rsN+k} {rsN+k1}\n")

        # Group 1 — centrelines for
allocated (straight) elements
        fh.write("g
centrelines_straight\n")
        for elem in allocations:
            write_centreline(elem)
        fh.write("\n")

        # Group 2 — centrelines for curved
leftovers
        fh.write("g centrelines_curved\n")
        for elem in curved:
            write_centreline(elem)
        fh.write("\n")

        # Group 3 — tube meshes for
allocated (straight) elements
        fh.write("g tubes_straight\n")
        for elem in allocations:
            write_tube(elem)
        fh.write("\n")

        # Group 4 — tube meshes for curved
leftovers
        fh.write("g tubes_curved\n")
        for elem in curved:
            write_tube(elem)
        fh.write("\n")

        # Group 5 — centrelines for
straight leftovers
        fh.write("g
centrelines_straight_leftover\n")
```

```
        for elem in (straight_leftovers or
[]):
            write_centreline(elem)
        fh.write("\n")

        # Group 6 — tube meshes for
straight leftovers
        fh.write("g
tubes_straight_leftover\n")
        for elem in (straight_leftovers or
[]):
            write_tube(elem)
        fh.write("\n")

    n_sl = len(straight_leftovers) if
straight_leftovers else 0
    print(f"Exported Rhino OBJ:
{len(allocations)} straight + {len(curved)}
curved + {n_sl} straight_leftover →
{output_path}")


def export_stage1_obj(vertices, edges,
edges_filtered, adjacency, output_path,
                      ring_segments=24,
n_color_bins=12):
    """
    Export the Stage-1 raw-skeleton
visualisation to OBJ for Rhino.

    Mirrors visualise_stage1():
      - Raw skeleton edges as polylines,
group 'skeleton_raw', material 'skel_gray'.
      - One closed-loop ring per filtered
node, coloured light→dark green by radius,
        binned into n_color_bins materials
so Rhino imports each bin as one object.

    A sidecar .mtl is written next to
the .obj.

    Args:
        vertices:        (N, 4) [x, y, z,
radius]
        edges:           (M, 2) raw edges
(pre-filter)
        edges_filtered:  iterable of (u, v)
after diameter filter + merge
        adjacency:       NetworkX graph
from build_adjacency(edges_filtered)
        output_path:     path to .obj
output
        ring_segments:   polygon vertex
count per ring (default 24)
        n_color_bins:    discretisation of
the green gradient (default 12)
    """
    edges = np.asarray(edges, dtype=int)
    light = np.array([0.6, 1.0, 0.6])
    dark  = np.array([0.0, 0.30, 0.05])
    gray  = (0.3, 0.3, 0.3)

    unique_nodes =
np.unique(np.asarray(list(edges_filtered),
dtype=int))
    radii = vertices[unique_nodes, 3]
    valid = radii > 1e-6
    unique_nodes = unique_nodes[valid]
    radii = radii[valid]
    rmin, rmax = (radii.min(), radii.max())
if len(radii) else (0.0, 1.0)

    def _bin_color(r):
        t = (r - rmin) / (rmax - rmin +
1e-10)
        b = int(min(t * n_color_bins,
n_color_bins - 1))
        t_mid = (b + 0.5) / n_color_bins
        return b, (light * (1 - t_mid) +
dark * t_mid)

    def _node_normal(node):
        nbrs =
list(adjacency.neighbors(node)) if node in
adjacency else []
        if nbrs:
            d = vertices[nbrs[0], :3] -
vertices[node, :3]
            if np.linalg.norm(d) > 1e-12:
                return d
        return np.array([0.0, 0.0, 1.0])

    mtl_path =
os.path.splitext(output_path)[0] + ".mtl"
    mtl_name = os.path.basename(mtl_path)

    with open(mtl_path, 'w') as mh:
        mh.write("# KM0DFAB Stage-1
materials\n")
        mh.write(f"newmtl skel_gray\nKd
{gray[0]:.4f} {gray[1]:.4f}
{gray[2]:.4f}\nKa 0 0 0\nKs 0 0 0\nillum
1\n\n")
        for b in range(n_color_bins):
            t_mid = (b + 0.5) /
n_color_bins
            c = light * (1 - t_mid) + dark
* t_mid
```

```
            mh.write(f"newmtl
ring_bin_{b:02d}\nKd {c[0]:.4f} {c[1]:.4f}
{c[2]:.4f}\nKa 0 0 0\nKs 0 0 0\nillum
1\n\n")

    v_idx = 1
    with open(output_path, 'w') as fh:
        fh.write("# KM0DFAB Stage-1 raw
skeleton + radius-gradient rings\n")
        fh.write(f"mtllib {mtl_name}\n\n")

        # Write all vertices once (1-based
indexing in OBJ)
        for xyz in vertices[:, :3]:
            fh.write(f"v {xyz[0]:.6f}
{xyz[1]:.6f} {xyz[2]:.6f}\n")
        v_idx += len(vertices)

        # Group 1 — raw skeleton edges
        fh.write("\ng skeleton_raw\n")
        fh.write("usemtl skel_gray\n")
        for u, v in edges:
            fh.write(f"l {int(u)+1}
{int(v)+1}\n")

        # Group 2 — radius-gradient rings,
one sub-group per colour bin
        # Pre-bin nodes so we can emit all
l-statements of a bin under one usemtl
        bins = [[] for _ in
range(n_color_bins)]
        for node, r in zip(unique_nodes,
radii):
            b, _ = _bin_color(r)
            bins[b].append((int(node),
float(r)))

        fh.write("\ng rings_filtered\n")
        for b, members in enumerate(bins):
            if not members:
                continue
            fh.write(f"usemtl
ring_bin_{b:02d}\n")
            for node, r in members:
                ring_pts =
_ring_points(vertices[node, :3],
_node_normal(node), r, ring_segments)
                start = v_idx
                for pt in ring_pts:
                    fh.write(f"v
{pt[0]:.6f} {pt[1]:.6f} {pt[2]:.6f}\n")
                    v_idx += 1
                loop = " ".join(str(start +
k) for k in range(ring_segments))
                fh.write(f"l {loop}
{start}\n")

    print(f"Exported Stage-1 OBJ:
{len(edges)} edges + {len(unique_nodes)}
rings → {output_path}")


def export_curved_obj(vertices, curved,
output_path):
    """Write centreline-only OBJ for the
fachwerk optimizer (curved leftovers
only)."""
    with open(output_path, 'w') as fh:
        fh.write("# KM0DFAB curved
leftovers — fachwerk input\n")
        fh.write("g curved_leftovers\n")
        v_idx = 1
        for elem in curved:
            nodes =
elem['global_node_indices']
            if len(nodes) < 2:
                continue
            start = v_idx
            for n in nodes:
                xyz = vertices[n, :3]
                fh.write(f"v {xyz[0]:.6f}
{xyz[1]:.6f} {xyz[2]:.6f}\n")
                v_idx += 1
            indices = " ".join(str(start +
i) for i in range(len(nodes)))
            fh.write(f"l {indices}\n")
    print(f"Exported curved OBJ:
{len(curved)} branches → {output_path}")


#
============================================================
# Grid catalog OBJ — chord-aligned,
centerlines + ring curves
#
============================================================

def export_grid_obj(vertices, allocations,
curved, output_path, ring_segments=24,
                    grid_cols=5,
spacing_y=2.0, spacing_z=5.0):
    """
    Export a chord-aligned grid catalog to
OBJ: centerline polylines + ring curves.
    Each element is rotated so its chord
(first→last node) aligns with +X,
    then placed in a grid (grid_cols per
row, rows stacked in Z).
```

```
    Rings are closed polyline curves (l-
statements), not a solid mesh.

    OBJ groups:
        straight  — allocated elements
(centerline + rings per element)
        curved    — curved leftovers
(centerline + rings per element)

    Args:
        vertices:      (N, 4) float array
[x, y, z, radius]
        allocations:   list of allocation
dicts
        curved:        list of curved
element dicts
        output_path:   path to .obj output
file
        ring_segments: vertices per ring
circle (default 24 for smooth curves)
        grid_cols:     elements per row
(default 5)
        spacing_y:     Y gap between
elements in a row (default 2.0 m)
        spacing_z:     Z gap between rows
(default 5.0 m)
    """
    from geometry import
rotation_to_align_with_x

    K = ring_segments
    v_idx = 1  # OBJ vertex indices are 1-
based
    curved = curved if curved else []

    def _node_normal(coords, i):
        N = len(coords)
        if i == 0:
            d = coords[1] - coords[0]
        elif i == N - 1:
            d = coords[N - 1] - coords[N -
2]
        else:
            d = coords[i + 1] - coords[i -
1]
        norm = np.linalg.norm(d)
        return d / norm if norm > 1e-12
else np.array([1., 0., 0.])

    def _grid_coords(nodes, elem_idx):
        """Return transformed (M, 3)
coords: chord → +X, placed in grid."""
        coords = vertices[nodes, :3].copy()
        chord = coords[-1] - coords[0]
        chord_len = np.linalg.norm(chord)
        if chord_len < 1e-10:
            return None
        R =
rotation_to_align_with_x(chord / chord_len)
        coords_rot = (coords - coords[0]) @
R.T
        col = elem_idx % grid_cols
        row = elem_idx // grid_cols
        return coords_rot + np.array([0.0,
col * spacing_y, row * spacing_z])

    with open(output_path, 'w') as fh:
        fh.write("# KM0DFAB grid catalog
export\n")
        fh.write(f"# ring_segments: {K},
grid_cols: {grid_cols}, "
                 f"spacing_y: {spacing_y}m,
spacing_z: {spacing_z}m\n\n")

        def write_element(elem, elem_idx):
            nonlocal v_idx
            nodes =
elem['global_node_indices']
            if len(nodes) < 2:
                return
            coords = _grid_coords(nodes,
elem_idx)
            if coords is None:
                return
            radii_arr = vertices[nodes, 3]

            # Centerline polyline
            cl_start = v_idx
            for pt in coords:
                fh.write(f"v {pt[0]:.6f}
{pt[1]:.6f} {pt[2]:.6f}\n")
                v_idx += 1
            cl_indices = "
".join(str(cl_start + i) for i in
range(len(nodes)))
            fh.write(f"l {cl_indices}\n")

            # Ring circles at each node
(closed polyline)
            for i in range(len(nodes)):
                r =
max(float(radii_arr[i]), 1e-6)
                ring_pts =
_ring_points(coords[i],
_node_normal(coords, i), r, K)
                ring_start = v_idx
                for pt in ring_pts:
                    fh.write(f"v
```

```
{pt[0]:.6f} {pt[1]:.6f} {pt[2]:.6f}\n")
                    v_idx += 1
                # Close the ring by
repeating the first index at the end
                ring_indices = "
".join(str(ring_start + k) for k in
range(K))
                fh.write(f"l {ring_indices}
{ring_start}\n")

        fh.write("g straight\n")
        for i, elem in
enumerate(allocations):
            write_element(elem, i)
        fh.write("\n")

        fh.write("g curved\n")
        for i, elem in enumerate(curved):
            write_element(elem,
len(allocations) + i)
        fh.write("\n")

    n_curved = len(curved)
    print(f"Exported grid OBJ:
{len(allocations)} straight + {n_curved}
curved → {output_path}")
```

A.1.4 skeleton_assessment/geometry.py

Graph construction, branch extraction, fork trimming, Dijkstra planar sub-segmentation, curvature stats, canonical-frame transform.

514 lines

```
import numpy as np
import open3d as o3d
import networkx as nx

def filter_edges_by_diameter(vertices,
edges, min_diameter_m):
    """
    Keep edges where both endpoints have
diameter >= min_diameter_m.
    """
    r = vertices[:, 3]
    out = []
    for u, v in edges:
        if 2 * r[u] >= min_diameter_m and 2
* r[v] >= min_diameter_m:
            out.append((int(u), int(v)))
    return out


def make_disc(center, normal, radius,
thickness=0.001, segments=32):
    """
    creates discs with radius based on node
diameter, to visualise nodes as circles
instead of points.

    Args:
        center:  (3,) xyz
        normal:  (3,) direction
perpendicular to the ring plane
        radius:  circle radius
        segments: number of line segments
around the ring

    Returns:
        open3d.geometry.LineSet
    """
    center = np.asarray(center,
dtype=float)
    normal = np.asarray(normal,
dtype=float)

    # Normalize normal safely
    n = np.linalg.norm(normal)
    if n < 1e-12:
        normal = np.array([0.0, 0.0, 1.0])
    else:
        normal = normal / n

    # Build an orthonormal basis (u, v)
spanning the ring plane
    helper = np.array([1.0, 0.0, 0.0]) if
abs(normal[0]) < 0.9 else np.array([0.0,
1.0, 0.0])
    u = np.cross(normal, helper)
    u_norm = np.linalg.norm(u)
    if u_norm < 1e-12:
        # Fallback helper if we
accidentally picked a parallel vector
        helper = np.array([0.0, 0.0, 1.0])
        u = np.cross(normal, helper)
        u_norm = np.linalg.norm(u)

    u /= u_norm
    v = np.cross(normal, u)

    # Sample points around the circle
    angles = np.linspace(0, 2 * np.pi,
segments, endpoint=False)
    pts = center + radius * (np.cos(angles)
[:, None] * u + np.sin(angles)[:, None] *
v)

    # Connect points into a closed loop
```

```
    lines =
np.column_stack([np.arange(segments),
(np.arange(segments) + 1) % segments])

    ring = o3d.geometry.LineSet()
    ring.points =
o3d.utility.Vector3dVector(pts)
    ring.lines =
o3d.utility.Vector2iVector(lines)
    return ring




def merge_coincident_tips(vertices,
edges_thick, edges_all, snap_dist=0.1):
    """
    Connect filtered-graph tips that only
became tips due to filtering.

    A tip that was degree-2+ in the full
graph lost edges during diameter
    filtering and became degree-1.  If it
is within snap_dist of another
    thick-graph node, add an edge to create
a degree-3 junction.

    Tips that were already degree-1 in the
full graph (true branch ends)
    are left untouched.

    Args:
        vertices:    (N, 4) array [x, y, z,
radius]
        edges_thick: diameter-filtered
edges
        edges_all:   all edges from the PLY
        snap_dist:   max distance to snap
(metres)

    Returns:
        edges_thick with new edges added at
fork points
    """
    from scipy.spatial import cKDTree

    G_full = nx.Graph()
    G_full.add_edges_from(edges_all)

    G_thick = nx.Graph()
    G_thick.add_edges_from(edges_thick)

    # Tips that were already degree-1 in
the full graph (true branch ends)
    filtered_tips = [n for n in
G_thick.nodes() if G_thick.degree(n) == 1]
    candidates = [t for t in filtered_tips
if G_full.degree(t) == 1]

    if not candidates:
        print("merge: no filtering-induced
tips found")
        return edges_thick

    # Non-tip nodes to snap to (exclude the
candidates themselves)
    candidate_set = set(candidates)
    snap_targets = [n for n in
G_thick.nodes() if n not in candidate_set]

    if not snap_targets:
        print("merge: no snap targets")
        return edges_thick

    target_coords =
np.array([vertices[n, :3] for n in
snap_targets])
    tree = cKDTree(target_coords)

    result = list(edges_thick)
    n_merged = 0

    for tip in candidates:
        tip_pos = vertices[tip, :3]
        dist, idx = tree.query(tip_pos)
        if dist <= snap_dist:
            target_node = snap_targets[idx]
            result.append((tip,
target_node))
            n_merged += 1

    print(f"merge: connected {n_merged}
filtering-induced tip(s)
(snap_dist={snap_dist}m)")
    return result


def build_adjacency(edges_uv):
    G = nx.Graph()
    G.add_edges_from(edges_uv)
    return G

def extract_branches(edges_filtered,
adjacency, vertices):
    visited_edges = set()
    branches = []

    tip_nodes  = [n for n in
```

```
adjacency.nodes() if adjacency.degree(n) ==
1]
    fork_nodes = [n for n in
adjacency.nodes() if adjacency.degree(n) >
2]
    split_nodes = set(fork_nodes)

    queue = []
    for tip in tip_nodes:
        queue.append((tip,
list(adjacency.neighbors(tip))[0]))
    for node in fork_nodes:
        for nb in
adjacency.neighbors(node):
            queue.append((node, nb))

    while queue:
        start, next_node = queue.pop(0)
        e = tuple(sorted((start,
next_node)))
        if e in visited_edges:
            continue

        chain = [start, next_node]
        visited_edges.add(e)
        current = next_node

        while current not in split_nodes:
            unvisited = [n for n in
adjacency.neighbors(current)
                         if
tuple(sorted((current, n))) not in
visited_edges]
            if len(unvisited) != 1:
                break
            nxt = unvisited[0]
visited_edges.add(tuple(sorted((current,
nxt))))
            chain.append(nxt)
            current = nxt

        if len(chain) >= 2:
            branches.append(chain)

    return branches

def trim_branch_fork_end(branch, vertices,
fork_nodes_set, trim_m, trim_start):
    """
    Remove nodes within trim_m of the root-
side fork endpoint.
    trim_start=True means branch[0] is the
root end; False means branch[-1] is.
    Only trims if that end is a fork node.
    """
    if len(branch) < 2 or trim_m <= 0:
        return branch

    pts = vertices[branch, :3]
    diffs = np.linalg.norm(np.diff(pts,
axis=0), axis=1)
    arc = np.concatenate([[0.0],
np.cumsum(diffs)])
    total = arc[-1]

    start_trim = 0
    end_trim = len(branch)

    if trim_start:
        if branch[0] in fork_nodes_set:
            for i in range(len(branch)):
                if arc[i] >= trim_m:
                    start_trim = i
                    break
            else:
                return []
    else:
        if branch[-1] in fork_nodes_set:
            for i in range(len(branch) - 1,
-1, -1):
                if (total - arc[i]) >=
trim_m:
                    end_trim = i + 1
                    break
            else:
                return []

    trimmed = branch[start_trim:end_trim]
    return trimmed if len(trimmed) >= 2
else []


def trim_branches_at_forks(branches,
vertices, adjacency, trim_diameters):
    """Trim each branch only at the root-
side fork end (root = lowest Z node).

    trim_diameters: trim distance =
trim_diameters × diameter at the fork
endpoint.
    """
    import collections
    fork_nodes_set = set(n for n in
adjacency.nodes() if adjacency.degree(n) >
2)

    root = min(adjacency.nodes(),
key=lambda n: vertices[n, 2])
```

```
    dist = {root: 0}
    queue = collections.deque([root])
    while queue:
        node = queue.popleft()
        for nb in
adjacency.neighbors(node):
            if nb not in dist:
                dist[nb] = dist[node] + 1
                queue.append(nb)

    result = []
    for branch in branches:
        trim_start = dist.get(branch[0], 0)
<= dist.get(branch[-1], 0)
        fork_node  = branch[0] if
trim_start else branch[-1]
        trim_m     = trim_diameters * 2.0 *
float(vertices[fork_node, 3])
        trimmed =
trim_branch_fork_end(branch, vertices,
fork_nodes_set, trim_m, trim_start)
        if len(trimmed) >= 2:
            result.append(trimmed)
    return result


def compute_node_curvatures(vertices,
adjacency):
    """
    For degree-2 nodes, compute deflection
angle (in degrees) between incoming/
outgoing edges.

    Returns:
        dict[node] = angle_deg
    """
    curv = {}
    for node in adjacency.nodes():
        if adjacency.degree(node) != 2:
            continue
        a, b =
list(adjacency.neighbors(node))
        p = vertices[node, :3]
        pa = vertices[a, :3]
        pb = vertices[b, :3]

        vin = p - pa
        vout = pb - p
        ni = np.linalg.norm(vin)
        no = np.linalg.norm(vout)
        if ni < 1e-10 or no < 1e-10:
            continue

        vin /= ni
        vout /= no

        dot = np.clip(np.dot(vin, vout),
-1.0, 1.0)
        curv[node] =
float(np.degrees(np.arccos(dot)))
    return curv

def fit_plane_residual(points: np.ndarray)
-> float:
    """
    Fit a best-fit plane using SVD and
return max absolute distance to that plane.
    """
    if len(points) < 4:
        return 0.0

    centroid = np.mean(points, axis=0)
    X = points - centroid
    _, _, Vt = np.linalg.svd(X,
full_matrices=False)
    normal = Vt[-1]
    residuals = np.abs(X @ normal)
    return float(residuals.max())

def segment_branch(branch, vertices,
max_planarity_m):
    """
    Split a branch into planar sub-
segments.
    Cuts at the node contributing most to
non-planarity.

    Returns:
        sub_segments: list[list[int]]
        split_nodes:  list[int]
    """
    if len(branch) < 4:
        return [branch], []

    sub_segments = []
    split_nodes = []
    window_start = 0

    for i in range(1, len(branch)):
        pts =
vertices[branch[window_start:i + 1], :3]
        if fit_plane_residual(pts) >
max_planarity_m:
            if i - window_start >= 4:
                centroid = np.mean(pts,
axis=0)
                X = pts - centroid
                _, _, Vt = np.linalg.svd(X,
```

```
full_matrices=False)
                normal = Vt[-1]
                worst_local =
int(np.argmax(np.abs(X @ normal)))
                worst_node = window_start +
worst_local

sub_segments.append(branch[window_start:worst_node])
split_nodes.append(branch[worst_node])
                window_start = worst_node

    rem = branch[window_start:]
    if len(rem) >= 4:
        sub_segments.append(rem)
    elif sub_segments:
        sub_segments[-1].extend(rem)
    else:
        sub_segments = [branch]

    return sub_segments, split_nodes

def segment_branch_dijkstra(branch,
vertices, max_planarity_m):
    """
    Split a branch into the MINIMUM NUMBER
of planar sub-segments using
    a shortest-path graph algorithm
(Dijkstra).

    Returns:
        sub_segments: list[list[int]]
(Lists of vertex indices)
        split_nodes:  list[int]       (The
nodes where cuts occurred)
    """
    if len(branch) < 4:
        return [branch], []

    G = nx.DiGraph()
    n = len(branch)

    #
----------------------------------------------------
    # STEP 1: Build the Graph of all
possible "flat" flights
    #
----------------------------------------------------
    for i in range(n):
        G.add_node(i)

        # Check every possible endpoint 'j'
for a segment starting at 'i'
        # A segment needs at least 3 points
to define a plane, so start at i + 2
        for j in range(i + 2, n):
            pts = vertices[branch[i : j +
1], :3]

            # Calculate planarity using SVD
            centroid = np.mean(pts, axis=0)
            X = pts - centroid
            _, _, Vt = np.linalg.svd(X,
full_matrices=False)
            normal = Vt[-1]

            max_deviation = np.max(np.abs(X
@ normal))

            if max_deviation <=
max_planarity_m:
                # The segment is flat!
Weights longer edges higher then shorter
edges!
                G.add_edge(i, j, weight =
1.0 / (j - i)**2)
            else:
                # OPTIMIZATION HEURISTIC:
                # If adding point 'j' makes
the segment warp out of tolerance,
                # adding points j+1, j+2,
etc., will almost certainly not make
                # it flat again. We break
here to save massive compute time.
                break

    #
----------------------------------------------------
    # STEP 2: Find the Shortest Path
(Fewest Layovers)
    #
----------------------------------------------------
    try:
        # NetworkX uses Dijkstra
automatically when weights are assigned
        path = nx.shortest_path(G,
source=0, target=n - 1, weight='weight')
    except nx.NetworkXNoPath:
        # Fallback: If the tolerance is so
tiny that no valid path can be found.
        print("Warning: Tolerance is too
strict. No continuous planar path found.")
        return [], []

    #
----------------------------------------------------
    # STEP 3: Format the output to match
your original structure
```

```
    #
------------------------------------------------------------
    sub_segments = []
    split_nodes = []

    for k in range(len(path) - 1):
        start_idx = path[k]
        end_idx = path[k + 1]

        # Extract the actual node indices
from the branch list
        # We overlap by 1 node so segments
share a physical joint
        segment = branch[start_idx :
end_idx + 1]
        sub_segments.append(list(segment))

        # Don't add a split node for the
very last endpoint
        if k < len(path) - 2:
split_nodes.append(branch[end_idx])

    return sub_segments, split_nodes

def compute_branch_stats(vertices,
branches, node_curvatures):
    stats = []
    for branch in branches:
        pts = vertices[branch, :3]
        length =
sum(np.linalg.norm(pts[i+1] - pts[i])
                    for i in
range(len(pts)-1))
        curv_vals = [node_curvatures[n] for
n in branch if n in node_curvatures]

        stats.append({
            "n_nodes":
len(branch),
            "length_m":
float(length),
            "start_diameter_m":
float(vertices[branch[0], 3] * 2),
            "end_diameter_m":
float(vertices[branch[-1], 3] * 2),
            "mean_curvature_deg":
float(np.mean(curv_vals)) if curv_vals else
0.0,
            "max_curvature_deg":
float(np.max(curv_vals))  if curv_vals else
0.0,
            "planarity_m":
float(fit_plane_residual(pts)),
        })
    return stats


def compute_best_fit_line(points:
np.ndarray):
    """
    Compute best-fit line through 3D points
using PCA/SVD.

    Args:
        points: (N, 3) array of 3D
coordinates

    Returns:
        centroid: (3,) point on the line
        direction: (3,) unit vector along
line
    """
    centroid = np.mean(points, axis=0)
    X = points - centroid
    _, _, Vt = np.linalg.svd(X,
full_matrices=False)
    direction = Vt[0]  # First principal
component
    return centroid, direction

def rotation_to_align_with_x(direction:
np.ndarray) -> np.ndarray:
    """
    Compute rotation matrix to align
direction vector with +X axis.

    Returns:
        R: (3, 3) rotation matrix
    """
    direction = direction /
np.linalg.norm(direction)
    x_axis = np.array([1.0, 0.0, 0.0])

    # If already aligned, return identity
    if np.allclose(direction, x_axis):
        return np.eye(3)

    # If opposite direction, rotate 180°
around Y
    if np.allclose(direction, -x_axis):
        return np.array([[-1, 0, 0], [0, 1,
0], [0, 0, -1]])

    # Rodrigues rotation formula
    v = np.cross(direction, x_axis)
    s = np.linalg.norm(v)
    c = np.dot(direction, x_axis)
```

```
    vx = np.array([[0, -v[2], v[1]],
                   [v[2], 0, -v[0]],
                   [-v[1], v[0], 0]])

    R = np.eye(3) + vx + vx @ vx * ((1 -
c) / (s ** 2))
    return R


def transform_branch_to_canonical(
    vertices: np.ndarray,
    branch_nodes: list,
    y_offset: float
):
    """
    Transform branch to canonical
orientation with Y-offset.

    Args:
        vertices: (N, 4) array [x, y, z,
radius]
        branch_nodes: List of node indices
for this branch
        y_offset: Y-axis offset for catalog
spacing

    Returns:
        transformed_coords: (M, 3)
transformed xyz coordinates
        radii: (M,) radius at each node
        thick_end_idx: Index of thick end
(becomes 0 after transformation)
    """
    branch_verts = vertices[branch_nodes]
    coords = branch_verts[:, :3]
    radii = branch_verts[:, 3]

    diameters = radii * 2
    thick_end_idx =
int(np.argmax(diameters))

    centroid, direction =
compute_best_fit_line(coords)

    thick_pos = coords[thick_end_idx]
    thin_end_idx = 0 if thick_end_idx ==
len(coords) - 1 else len(coords) - 1
    thin_pos = coords[thin_end_idx]
    expected_dir = thin_pos - thick_pos
    if np.dot(direction, expected_dir) < 0:
        direction = -direction

    R = rotation_to_align_with_x(direction)
    coords_centered = coords - centroid
    coords_rotated = coords_centered @ R.T
    thick_end_pos =
coords_rotated[thick_end_idx]
    coords_final = coords_rotated -
thick_end_pos

    coords_final[:, 1] += y_offset

    return coords_final, radii,
thick_end_idx
```

A.1.5 skeleton_assessment/gorges.py

Gorges et al. greedy bow-height allocation: picks the straightest window of the required length for each structural role.

262 lines

```
import numpy as np

# ----------------------------
# Gorges et al. bow-height curvature
scoring
# ----------------------------

def arc_length_cumulative(nodes:
np.ndarray) -> np.ndarray:
    diffs = np.linalg.norm(np.diff(nodes,
axis=0), axis=1)
    return np.concatenate([[0.0],
np.cumsum(diffs)])


def smooth_centerline(nodes: np.ndarray, s:
np.ndarray,
                      window_m: float =
0.2) -> np.ndarray:
    """
    Gorges et al. 200mm spatial moving
average.

    For each node i, the smoothed position
is the mean of all nodes j
    where |s[j] - s[i]| <= window_m / 2.

    Args:
        nodes:    (K, 3) xyz positions
        s:        (K,) cumulative arc
length (monotonically increasing)
        window_m: smoothing bandwidth in
metres (default 0.2m = 200mm)

    Returns:
```

```
        smoothed: (K, 3) smoothed positions
    """
    half = window_m / 2.0
    smoothed = np.empty_like(nodes)
    for i in range(len(nodes)):
        lo = np.searchsorted(s, s[i] -
half, side='left')
        hi = np.searchsorted(s, s[i] +
half, side='right')
        smoothed[i] = np.mean(nodes[lo:hi],
axis=0)
    return smoothed


def bow_height_curvature(window_nodes:
np.ndarray):
    """
    Gorges et al. centerline bow height
curvature for a sequence of 3D nodes.

    Method:
      1. Chord = straight line from first
to last node
      2. bow_height = max perpendicular
distance from intermediate nodes to chord
      3. curvature = (bow_height * 100) /
chord_length  [cm/m]

    Args:
        window_nodes: (W, 3) array of
smoothed centerline positions

    Returns:
        (curvature_cm_per_m, bow_height_m):
both float
    """
    A, B = window_nodes[0],
window_nodes[-1]
    AB = B - A
    chord_length = np.linalg.norm(AB)
    if chord_length < 1e-10:
        return 0.0, 0.0

    intermediate = window_nodes[1:-1]
    if len(intermediate) == 0:
        return 0.0, 0.0

    AP = intermediate - A
    t = (AP @ AB) / (AB @ AB)
    distances = np.linalg.norm(intermediate
- (A + t[:, None] * AB), axis=1)
    bow_height = float(distances.max())
    return (bow_height * 100.0) /
chord_length, bow_height


def
find_best_position_in_piece(nodes_smooth:
np.ndarray, s: np.ndarray,
                            role_length_m:
float, step_m: float = 0.05,
middle_cut_penalty: float = 1.0):
    if s[-1] < role_length_m:
        return None

    piece_length = s[-1]
    max_possible_dist = (piece_length -
role_length_m) / 2.0

    best = None
    best_score = float('inf')
    start_arc = 0.0
    while start_arc + role_length_m <=
s[-1] + 1e-9:
        lo = np.searchsorted(s, start_arc,
side='left')
        hi = np.searchsorted(s, start_arc +
role_length_m, side='right')
        if hi - lo >= 3:
            curv, bh =
bow_height_curvature(nodes_smooth[lo:hi])

            # Linear middle-cut penalty:
1.0 at edges, middle_cut_penalty at center
            w_start = float(s[lo])
            w_end = float(s[min(hi-1,
len(s)-1)])
            dist_to_nearest_end =
min(w_start, piece_length - w_end)
            if max_possible_dist > 1e-9:
                rel_pos =
min(dist_to_nearest_end /
max_possible_dist, 1.0)
            else:
                rel_pos = 0.0
            penalty = 1.0 +
(middle_cut_penalty - 1.0) * rel_pos
            score = curv * penalty

            if score < best_score:
                best_score = score
                best = {
                    'curvature_cm_per_m':
curv,
                    'bow_height_m':
bh,
                    'window_start_m':
w_start,
```

```
                    'window_end_m':
w_end,
                    'start_node_idx':
int(lo),
                    'end_node_idx':
int(min(hi-1, len(s)-1)),
                }
        start_arc += step_m
    return best


# ----------------------------
# Greedy non-overlapping allocation
# ----------------------------

def make_piece(nodes_smooth, s,
sub_seg_nodes, vertices, branch_id,
tree_source=""):
    if len(sub_seg_nodes) < 4:
        return None
    s_rezeroed = s - s[0]
    radii = vertices[sub_seg_nodes, 3]
    return {
        'nodes_smooth':  nodes_smooth,
        's':             s_rezeroed,
        'sub_seg_nodes': sub_seg_nodes,
        'arc_length_m':
float(s_rezeroed[-1]),
        'branch_id':     branch_id,
        'tree_source':   tree_source,
        'mean_radius_m':
float(np.mean(radii)),
        'min_radius_m':
float(np.min(radii)),
    }


def precompute_pieces(vertices,
sub_segments, branch_ids, min_length_m,
                      smooth_window_m=0.2,
tree_sources=None):
    pieces = []
    for seg_idx, (sub_seg, branch_id) in
enumerate(zip(sub_segments, branch_ids)):
        nodes = vertices[sub_seg, :3]
        s = arc_length_cumulative(nodes)
        if s[-1] < min_length_m:
            continue
        nodes_smooth =
smooth_centerline(nodes, s,
smooth_window_m)
        piece = make_piece(nodes_smooth, s,
sub_seg, vertices, branch_id,
tree_sources[seg_idx] if tree_sources else
"")
        if piece:
            pieces.append(piece)
    return pieces


def find_best_windows_for_role(pieces,
role_length_m, step_m=0.05,
middle_cut_penalty=1.0,
max_curvature_cm_per_m=None):
    candidates = []
    for piece_idx, piece in
enumerate(pieces):
        if piece['arc_length_m'] <
role_length_m:
            continue
        window =
find_best_position_in_piece(piece['nodes_smooth'],
piece['s'],
role_length_m, step_m, middle_cut_penalty)
        if window is not None:
            if max_curvature_cm_per_m is
not None and window['curvature_cm_per_m'] >
max_curvature_cm_per_m:
                continue
            window['piece_idx'] = piece_idx
            window['global_node_indices'] =
piece['sub_seg_nodes'][
window['start_node_idx']:window['end_node_idx']
+ 1
            ]
            candidates.append((piece_idx,
window))
    candidates.sort(key=lambda x: x[1]
['curvature_cm_per_m'])
    return candidates


def split_piece_around_window(piece,
window, vertices):
    lo, hi = window['start_node_idx'],
window['end_node_idx']
    leftovers = []

    if lo > 0:
        left =
make_piece(piece['nodes_smooth'][:lo],
piece['s'][:lo],
piece['sub_seg_nodes'][:lo], vertices,
piece['branch_id'], piece['tree_source'])
        if left:
```

```
            leftovers.append(left)

    if hi + 1 <
len(piece['sub_seg_nodes']):
        right =
make_piece(piece['nodes_smooth'][hi+1:],
piece['s'][hi+1:],
piece['sub_seg_nodes'][hi+1:], vertices,
piece['branch_id'], piece['tree_source'])
        if right:
            leftovers.append(right)

    return leftovers

def greedy_allocate(vertices, sub_segments,
branch_ids, roles,
                    min_length_m,
smooth_window_m=0.2, step_m=0.05,
                    tree_sources=None,
middle_cut_penalty=1.0,
min_curved_length_m=0.0,
max_curvature_cm_per_m=None):
    sorted_roles = sorted(roles, key=lambda
r: r[1], reverse=True)
    available_pieces =
precompute_pieces(vertices, sub_segments,
branch_ids,
min(min_length_m, min_curved_length_m),
smooth_window_m, tree_sources)
    allocations = []

    for role_name, role_length, role_count
in sorted_roles:
        allocated = 0
        while allocated < role_count:
            candidates =
find_best_windows_for_role(available_pieces,
role_length, step_m,
middle_cut_penalty,
max_curvature_cm_per_m)
            if not candidates:
                print(f"  {role_name}:
{allocated}/{role_count} (no more
material)")
                break

            piece_idx, window =
candidates[0]
            piece =
available_pieces[piece_idx]

            allocations.append({
                'role_name':
role_name,
                'role_length_m':
role_length,
                'curvature_cm_per_m':
window['curvature_cm_per_m'],
                'bow_height_m':
window['bow_height_m'],
                'window_start_m':
window['window_start_m'],
                'window_end_m':
window['window_end_m'],
                'global_node_indices':
list(window['global_node_indices']),
                'branch_id':
piece['branch_id'],
                'tree_source':
piece['tree_source'],
            })
            allocated += 1

            leftovers =
split_piece_around_window(piece, window,
vertices)
            available_pieces.pop(piece_idx)
            for i, leftover in
enumerate(leftovers):
available_pieces.insert(piece_idx + i,
leftover)

        if allocated >= role_count:
            print(f"  {role_name}:
{allocated}/{role_count} OK")

    print(f"\n  Total allocated:
{len(allocations)} elements")

    # Collect leftovers — split by
curvature threshold
    curved_elements    = []
    straight_leftovers = []
    for piece in available_pieces:
        if piece['arc_length_m'] <
min_curved_length_m:
            continue
        curv, bh =
bow_height_curvature(piece['nodes_smooth'])
        entry = {
            'role_length_m':
piece['arc_length_m'],
            'curvature_cm_per_m': curv,
            'bow_height_m':       bh,
            'window_start_m':     0.0,
```

```
            'window_end_m':
piece['arc_length_m'],
            'global_node_indices':
list(piece['sub_seg_nodes']),
            'branch_id':
piece['branch_id'],
            'tree_source':
piece['tree_source'],
        }
        if max_curvature_cm_per_m is not
None and curv <= max_curvature_cm_per_m:
straight_leftovers.append({**entry,
'role_name': 'straight_leftover'})
        else:
curved_elements.append({**entry,
'role_name': 'curved'})
    curved_elements.sort(key=lambda x:
x['role_length_m'], reverse=True)
    straight_leftovers.sort(key=lambda x:
x['role_length_m'], reverse=True)
    print(f"  Curved leftovers:
{len(curved_elements)} pieces")
    print(f"  Straight leftovers:
{len(straight_leftovers)} pieces")

    return allocations, curved_elements,
straight_leftovers
```

11.2.3 A.2 Stage 2 — Fachwerk optimisation

A.2.1 `digital_fachwerk/fachwerk/config.py`

Single source of truth for Stage-2 parameters: frame size, 7×7 grid, FEA loads, oak (D30) material.

58 lines

```
# --- Paths ---
from pathlib import Path
_REPO_ROOT   = Path(__file__).parent.parent
TREE_OBJ_PATH = _REPO_ROOT.parent /
'skeleton_assessment' / 'output' /
'curved_leftovers.obj'

# --- Units ---
OBJ_UNIT_SCALE = 1000.0   # OBJ file is in
metres; multiply to get mm

# --- Frame ---
FRAME_WIDTH_MM  = 4000
FRAME_HEIGHT_MM = 3000

# --- Boundary sampling ---
N_CELL          = 20    # fixed number of
boundary sample points per cell
MAX_BOUNDARY_PTS = N_CELL

# --- Grid ---
GRID_NX = 7
GRID_NY = 7

# --- Stock management ---
STOCK_DESCENDING = True   # True = longest
branch first

# --- Branch export ---
MAX_BRANCH_PTS = 30   # upper bound on
vertices in a trimmed placed branch

# --- Placement ---
USE_CUTOFFS    = True
MIN_CUTOFF_LEN = 500      # mm — discard
offcuts shorter than this
MIN_CELL_AREA_MM2 = 100_000.0  # mm² (= 0.1
m²) — skip placement if a resulting cell
would be smaller
MAX_RETRIES    = 3        # retries per
branch after first failure before discard

# --- Material (Oak timber) ---
TIMBER_E             = 9_000.0   # N/mm² —
Young's modulus - reduced value since we
work with heterogeneous timber (oak)
TIMBER_DENSITY_KG_M3 = 700.0     # kg/m³

# Frame members (columns + beams)
FRAME_SECTION_WIDTH_MM = 120   # mm
FRAME_SECTION_DEPTH_MM = 160   # mm

# Branch members (infill braces)
BRANCH_SECTION_WIDTH_MM = 80  # mm
BRANCH_SECTION_DEPTH_MM = 120   # mm

# --- FEA ---
FEA_GRAVITY_LOAD_N    = -25_000.0  # N —
UDL on top beam (~6.25 kN/m over 4 m,
approx. floor dead+live load)
FEA_WIND_LOAD_N       =  10_000.0  # N —
windward pressure magnitude (~3.33 kN/m
over 3 m)
FEA_WIND_SUCTION_RATIO =      0.5  #
leeward suction as fraction of windward
pressure (EN 1991-1-4 ~0.5)
FEA_NODE_TOLERANCE_MM = 1.0        # mm —
max distance to merge coincident nodes

# Timber strength — EN 338 D30 hardwood
```

```
characteristic values [N/mm²]
# Oak falls in the D30–D40 range. Using D30
characteristic values (no partial factor)
# is intentionally slightly conservative
for heterogeneous natural branches.
TIMBER_FMK  = 30.0   # bending strength
TIMBER_FT0K = 18.0   # tension parallel to
grain
TIMBER_FC0K = 23.0   # compression parallel
to grain
FEA_DEFORM_SCALE      = 20         #
deformed shape display amplification factor
```

A.2.2 digital_fachwerk/fachwerk/geometry.py

Pure geometry helpers: OBJ branch loader, normalisation, curvature profile, boundary sampling, distance-to-chord.

251 lines

```
import math
import numpy as np
from shapely.geometry import LineString,
Polygon

from fachwerk import config



#
------------------------------------------------------------
# OBJ loading
#
------------------------------------------------------------

def load_branches_from_obj(filepath: str) -
> list:
    vertices = []
    raw_branches = []

    with open(filepath, 'r',
encoding='latin-1') as f:
        for line in f:
            line = line.strip()
            if line.startswith('v '):
                parts = line.split()
                vertices.append((
                    float(parts[1]) *
config.OBJ_UNIT_SCALE,
                    float(parts[2]) *
config.OBJ_UNIT_SCALE,
                    float(parts[3]) *
config.OBJ_UNIT_SCALE,
                ))
            elif line.startswith('l '):
                parts = line.split()
                indices = [int(p) - 1 for p
in parts[1:]]  # OBJ is 1-based
raw_branches.append([vertices[i] for i in
indices])

    branches = []
    for pts in raw_branches:
        geometry = normalize_branch(pts)
        profile =
compute_curvature_profile(geometry)
        branches.append({'geometry':
geometry, **profile})

    return branches



#
------------------------------------------------------------
# Branch normalization
#
------------------------------------------------------------

def normalize_branch(points_3d) ->
LineString:
    pts = np.array(points_3d, dtype=float)
    centroid = pts.mean(axis=0)
    centered = pts - centroid
    _, _, Vt = np.linalg.svd(centered,
full_matrices=False)
    coords_2d = centered @ Vt[:2].T
    coords_2d -= coords_2d[0]

    end = coords_2d[-1]
    angle = np.arctan2(end[1], end[0])
    cos_a, sin_a = float(np.cos(angle)),
float(np.sin(angle))
    rotated = np.empty_like(coords_2d)
    rotated[:, 0] = cos_a * coords_2d[:, 0]
+ sin_a * coords_2d[:, 1]
    rotated[:, 1] = -sin_a * coords_2d[:,
0] + cos_a * coords_2d[:, 1]

    return LineString(rotated)


def
compute_curvature_profile(normalized_branch:
LineString) -> dict:
    coords =
np.array(normalized_branch.coords)
    y = coords[:, 1]
```

```
    chord_length = float(coords[-1, 0])
    max_offset = float(np.max(np.abs(y)))
    avg_offset = float(np.mean(np.abs(y)))

    nonzero_y = y[y != 0]
    if len(nonzero_y) > 1:
        signs = np.sign(nonzero_y)
        n_inflections =
int(np.sum(np.diff(signs) != 0))
    else:
        n_inflections = 0

    direction = 'up' if float(np.mean(y)) >
0 else 'down'

    return {
        'chord_length':  chord_length,
        'max_offset':    max_offset,
        'avg_offset':    avg_offset,
        'n_inflections': n_inflections,
        'profile':       y.tolist(),
        'direction':     direction,
    }


#
------------------------------------------------------------
# Frame creation
#
------------------------------------------------------------

def create_frame(width_mm: float,
height_mm: float) -> Polygon:
    return Polygon([(0, 0), (width_mm, 0),
(width_mm, height_mm), (0, height_mm)])


#
------------------------------------------------------------
# Shapely utilities
#
------------------------------------------------------------

def insert_sorted(stock: list, branch_dict:
dict, descending: bool = True) -> None:
    length = branch_dict['chord_length']
    for i, item in enumerate(stock):
        if descending and length >=
item['chord_length']:
            stock.insert(i, branch_dict)
            return
        if not descending and length <=
item['chord_length']:
            stock.insert(i, branch_dict)
            return
    stock.append(branch_dict)


def cell_area_centroid(polygon: Polygon) ->
tuple:
    return polygon.area,
(polygon.centroid.x, polygon.centroid.y)


def _anchor_indices(coords, anchor_pts):
    n_pts = len(coords)
    if anchor_pts is None:
        return list(range(n_pts))
    indices = [
        i for i, (x, y) in
enumerate(coords)
        if (round(x, 2), round(y, 2)) in
anchor_pts
    ]
    return indices if indices else
list(range(n_pts))


def _build_segments(coords,
anchor_indices):
    n_pts = len(coords)
    n_anchors = len(anchor_indices)
    segments = []
    for k in range(n_anchors):
        start = anchor_indices[k]
        end   = anchor_indices[(k + 1) %
n_anchors]
        seg   = [coords[start]]
        j = (start + 1) % n_pts
        while j != end:
            seg.append(coords[j])
            j = (j + 1) % n_pts
        seg.append(coords[end])
        arc = sum(math.dist(seg[m], seg[m +
1]) for m in range(len(seg) - 1))
        segments.append((seg, arc))
    return segments


def _interpolate_targets(seg, seg_lens,
targets):
    pts = []
    cum, ei = 0.0, 0
    for t in targets:
        while ei < len(seg_lens) - 1 and
cum + seg_lens[ei] < t:
            cum += seg_lens[ei]
            ei  += 1
```

```
        local_t = (t - cum) / seg_lens[ei]
if seg_lens[ei] > 0 else 0.0
        x0, y0 = seg[ei]
        x1, y1 = seg[ei + 1]
        pts.append((x0 + local_t * (x1 -
x0), y0 + local_t * (y1 - y0)))
    return pts


def _dist_to_chord(px, py, ax, ay, bx, by)
-> float:
    dx, dy = bx - ax, by - ay
    length = math.hypot(dx, dy)
    if length < 1e-9:
        return 0.0
    return ((px - ax) * dy - (py - ay) *
dx) / length


def sample_boundary_n(polygon: Polygon,
                      n: int =
config.N_CELL,
                      anchor_pts: set =
None) -> list:
    coords = list(polygon.exterior.coords)
[:-1]

    # Rotate so point 0 is the bottom-left
vertex (min y, then min x)
    start = min(range(len(coords)),
key=lambda i: (round(coords[i][1], 2),
round(coords[i][0], 2)))
    coords = coords[start:] +
coords[:start]

    n_verts = len(coords)
    seg_lens = [math.dist(coords[i],
coords[(i + 1) % n_verts]) for i in
range(n_verts)]
    total_len = sum(seg_lens)

    result = []
    for k in range(n):
        target = total_len * k / n
        cum = 0.0
        for i, sl in enumerate(seg_lens):
            if cum + sl >= target - 1e-9 or
i == n_verts - 1:
                local_t = (target - cum) /
sl if sl > 1e-9 else 0.0
                x0, y0 = coords[i]
                x1, y1 = coords[(i + 1) %
n_verts]
                px = x0 + local_t * (x1 -
x0)
                py = y0 + local_t * (y1 -
y0)
                is_anchor = (anchor_pts is
not None and
                             (round(px, 2),
round(py, 2)) in anchor_pts)
                result.append(((px, py),
is_anchor))
                break
            cum += sl

    return result


def encode_boundary_obs(polygon: Polygon,
                        delta_mm: float =
224.0,
                        anchor_pts: set =
None) -> list:
    """Return per-segment boundary
observation for the RL environment.

    Each segment runs from one anchor to
the next. Interior points are spaced
    delta_mm apart; unused slots are padded
with float('nan') up to
    config.MAX_SEGMENT_PTS. Each point is
(x, y, dist_chord) where dist_chord
    is the signed perpendicular distance
from the anchor-to-anchor chord line.

    Returns a list of segments, each a
list:
        [(ax, ay, 0.0), pt_or_nan, ...,
(bx, by, 0.0)]
    of length MAX_SEGMENT_PTS + 2.
    """
    coords = list(polygon.exterior.coords)
[:-1]
    indices = _anchor_indices(coords,
anchor_pts)
    segments = _build_segments(coords,
indices)

    obs = []
    nan = float('nan')
    for seg, arc in segments:
        ax, ay = seg[0]
        bx, by = seg[-1]
        targets = [ki * delta_mm for ki in
range(1, int(arc / delta_mm) + 1)
                   if ki * delta_mm < arc -
1e-9]
```

```
        interior = []
        if targets:
            seg_lens = [math.dist(seg[m],
seg[m + 1]) for m in range(len(seg) - 1)]
            for px, py in
_interpolate_targets(seg, seg_lens,
targets):
                interior.append((px, py,
_dist_to_chord(px, py, ax, ay, bx, by)))
        max_seg_pts =
int((config.FRAME_WIDTH_MM +
config.FRAME_HEIGHT_MM) / delta_mm)
        while len(interior) < max_seg_pts:
            interior.append(nan)
        obs.append([(ax, ay, 0.0)] +
interior + [(bx, by, 0.0)])

    return obs


def angle_between_vectors(v1, v2) -> float:
    return math.atan2(
        v1[0] * v2[1] - v1[1] * v2[0],
        v1[0] * v2[0] + v1[1] * v2[1],
    )



# [demo / visualisation harness (if
__name__ == '__main__') omitted — not part
of the analysis method]
```

A.2.3 digital_fachwerk/fachwerk/genome.py

decode_genome(genome, n_steps, layout) and detect_genome_layout — decode either the interleaved or block genome layout from a result JSON.

41 lines

```
"""Gene-encoding helpers shared by the
optimisers and the result-JSON viewers.

A genome is a flat integer array of length
``4 * n_steps`` carrying four
per-step values: grid point, boundary point
A, boundary point B, and a flip
flag. Two physical layouts exist in saved
result JSONs:

  'block'        [gp_0, ..., gp_{n-1},
pa_0, ..., pa_{n-1},
                  pb_0, ..., pb_{n-1},
fl_0, ..., fl_{n-1}]
  'interleaved'  [gp_0, pa_0, pb_0, fl_0,
gp_1, pa_1, pb_1, fl_1,  ...]

Newer runs save ``'genome_layout'`` in the
result-JSON metadata, so the layout
can always be detected. For older JSONs the
presence of ``'n_cores'`` is a
reliable signal that the file came from
optimizer_parallel.py (already
interleaved before the marker was
introduced).
"""


def decode_genome(genome, n_steps,
layout='block'):
    """Split a flat integer genome into
(gp, pa, pb, fl) lists."""
    if layout == 'interleaved':
        return (list(genome[0::4]),
list(genome[1::4]),
                list(genome[2::4]),
list(genome[3::4]))
    n = n_steps
    return (list(genome[0:n]),
list(genome[n:2 * n]),
            list(genome[2 * n:3 * n]),
list(genome[3 * n:4 * n]))


def detect_genome_layout(run_meta):
    """Return 'interleaved' or 'block' for
a result-JSON metadata dict.

    Precedence:
      1. explicit ``genome_layout`` key
(written by current optimisers)
      2. ``n_cores`` present  → legacy
optimizer_parallel.py JSON (interleaved)
      3. fallback             →
'block' (legacy single-run and 3-obj JSONs)
    """
    if 'genome_layout' in run_meta:
        return run_meta['genome_layout']
    if 'n_cores' in run_meta:
        return 'interleaved'
    return 'block'
```

A.2.4 digital_fachwerk/fachwerk/placement.py

replay() — the core placement simulation: places branches one by one and splits Shapely cells. Called thousands of times per run.

527 lines

```
"""placement.py

Run as a script to visually test placement:

    python -m fachwerk.placement

Edit the GENES section at the bottom to try different sequences.
Each gene list must have the same length (= number of placement steps).

  grid_points  0 … GRID_NX*GRID_NY-1  which tile to target
  point_a      0 … MAX_BOUNDARY_PTS-1  first boundary point index
  point_b      0 … MAX_BOUNDARY_PTS-1  second boundary point index
  flip         0 or 1                  mirror branch about chord
"""

import math
import numpy as np
from shapely.geometry import (
    GeometryCollection, LineString, MultiLineString, Point, Polygon,
)
from shapely.ops import split

from fachwerk import config
from fachwerk.geometry import (
    _dist_to_chord,
    compute_curvature_profile,
    insert_sorted,
    normalize_branch,
    sample_boundary_n,
)


#--------------------------------------------------------------------
# Module-private helpers
#--------------------------------------------------------------------


def _as_linestrings(geom) -> list:
    """Flatten any geometry into a list of LineStrings."""
    if isinstance(geom, LineString):
        return [geom]
    if isinstance(geom, (MultiLineString, GeometryCollection)):
        return [g for g in geom.geoms if isinstance(g, LineString)]
    return []


def _extend_line(line: LineString, distance: float) -> LineString:
    """Extend a LineString by 'distance' at both ends so it crosses boundaries."""
    coords = np.array(line.coords)

    d_start = coords[0] - coords[1]
    d_start = d_start / np.linalg.norm(d_start) * distance

    d_end = coords[-1] - coords[-2]
    d_end = d_end / np.linalg.norm(d_end) * distance

    return LineString(np.vstack([coords[0] + d_start, coords, coords[-1] + d_end]))


#--------------------------------------------------------------------
# Public API
#--------------------------------------------------------------------

def build_grid(frame_polygon: Polygon) -> list:
    """Return GRID_NX * GRID_NY tile-centre points in row-major order."""
    minx, miny, maxx, maxy = frame_polygon.bounds
    dx = (maxx - minx) / config.GRID_NX
    dy = (maxy - miny) / config.GRID_NY
    return [
        (minx + (gx + 0.5) * dx, miny + (gy + 0.5) * dy)
        for gy in range(config.GRID_NY)
        for gx in range(config.GRID_NX)
    ]


def position_branch(branch_dict: dict, pt_A: tuple, pt_B: tuple, flip: int) -> LineString:
    """Translate + rotate a normalised branch so it runs from pt_A toward pt_B.

    Flip is applied in normalised space (before rotation) so it always mirrors
    about the chord regardless of chord
```

```
direction.
    """
    coords =
np.array(branch_dict['geometry'].coords,
dtype=float)

    if flip:
        coords[:, 1] *= -1

    angle = math.atan2(pt_B[1] - pt_A[1],
pt_B[0] - pt_A[0])
    cos_a, sin_a = math.cos(angle),
math.sin(angle)

    rotated = np.empty_like(coords)
    rotated[:, 0] = cos_a * coords[:, 0] -
sin_a * coords[:, 1] + pt_A[0]
    rotated[:, 1] = sin_a * coords[:, 0] +
cos_a * coords[:, 1] + pt_A[1]

    return LineString(rotated)


def trim_branch_to_cell(branch: LineString,
cell: Polygon):
    """Return the portion of branch inside
cell, or None if invalid/too short.

    Both endpoints of the trimmed result
must lie on the cell exterior —
    i.e. the branch must enter on one side
and exit on another.
    """
    result = branch.intersection(cell)

    if result.is_empty:
        return None
    if isinstance(result, LineString):
        trimmed = result
    elif isinstance(result,
MultiLineString):
        trimmed = max(result.geoms,
key=lambda g: g.length)
    else:
        return None

    if trimmed.length <
config.MIN_CUTOFF_LEN:
        return None

    # Reject if either endpoint is not on
the cell boundary
    exterior = cell.exterior
    tol = 1e-3
    if
(exterior.distance(Point(trimmed.coords[0]))
> tol or
exterior.distance(Point(trimmed.coords[-1]))
> tol):
        return None

    return trimmed


def split_cell(cell: Polygon,
trimmed_branch: LineString, _debug: bool =
False):
    """Split cell along trimmed_branch.
Returns list of 2 Polygons or None.

    trim_branch_to_cell already guarantees
both endpoints lie on the cell
    exterior, so we only need to confirm
they are distinct before splitting.
    """
    start = Point(trimmed_branch.coords[0])
    end   =
Point(trimmed_branch.coords[-1])
    if start.distance(end) < 1e-3:
        if _debug:
            print("    [split_cell] FAIL:
start and end are the same point")
        return None

    try:
        extended =
_extend_line(trimmed_branch, 1.0)
        result   = split(cell, extended)
        polys    = [g for g in result.geoms
if isinstance(g, Polygon)]
        if len(polys) != 2:
            if _debug:
                print(f"    [split_cell]
FAIL: split produced {len(polys)}
polygon(s), expected 2")
            return None
        return polys
    except Exception as e:
        if _debug:
            print(f"    [split_cell] FAIL:
exception during split — {e}")
        return None


def replay(
    frame: Polygon,
    branches: list,
    grid_points_gene: list,
```

```
    point_a_gene: list,
    point_b_gene: list,
    flip_gene: list,
    use_cutoffs: bool = config.USE_CUTOFFS,
    reverse_order: bool = False,
    verbose: bool = False,
    out_cutoffs: list | None = None,
) -> tuple:
    """Stateless replay of a full placement
sequence from gene arrays.

    Returns (placed, placement_info, cells,
centroids, n_placed, n_skipped).

    placement_info is a list parallel to
placed; each entry is a dict:
      orig_idx  — index into the original
branches list
      is_cutoff — True if this piece is an
offcut of a previously placed branch
      t_start   — arc-length fraction along
the full branch where the cut begins
      t_end     — arc-length fraction along
the full branch where the cut ends

    If out_cutoffs is provided, each
accepted cutoff (length >= MIN_CUTOFF_LEN)
    is appended as {'source_orig_idx',
'coords_mm', 'length_mm'} so callers
    can persist the cutoff geometry as new
stock pieces.
    """
    descend  = not reverse_order
    cells    = [frame]
    stock    = sorted(
        [{**b, 'orig_idx': i, 'is_cutoff':
False} for i, b in enumerate(branches)],
        key=lambda b: b['chord_length'],
        reverse=descend,
    )
    grid_pts  = build_grid(frame)
    n_grid    = config.GRID_NX *
config.GRID_NY

    placed         = []
    placement_info  = []
    n_skipped       = 0
    retry_count = 0
    anchor_pts = [(round(x, 2), round(y,
2)) for x, y in frame.exterior.coords[:-1]]

    def _fail():
        nonlocal n_skipped, retry_count
        n_skipped += 1
        retry_count += 1
        if retry_count >
config.MAX_RETRIES:
            stock.pop(0)
            retry_count = 0

    for i in range(len(grid_points_gene)):
        if not stock:
            break

        # 1 — Grid -> target cell
        g_idx = max(0,
min(int(grid_points_gene[i]), n_grid - 1))
        gp    = Point(grid_pts[g_idx])
        target_cell = next((c for c in
cells if c.contains(gp)), None)
        if target_cell is None:
            _fail(); continue

        # 2 — Boundary sample
        boundary_pts =
sample_boundary_n(target_cell,
anchor_pts=anchor_pts)
        if len(boundary_pts) < 2:
            _fail(); continue

        # 3 — Validate point indices
        n_pts = len(boundary_pts)
        idx_a = max(0,
min(int(point_a_gene[i]), n_pts - 1))
        idx_b = max(0,
min(int(point_b_gene[i]), n_pts - 1))
        if idx_a == idx_b:
            _fail(); continue

        pt_A = boundary_pts[idx_a][0]
        pt_B = boundary_pts[idx_b][0]

        # 4 — Length check
        dist_ab = math.dist(pt_A, pt_B)
        if dist_ab > stock[0]
['chord_length']:
            _fail(); continue

        # 5 — Position + trim
        flip       = int(flip_gene[i])
        positioned =
position_branch(stock[0], pt_A, pt_B, flip)
        trimmed    =
trim_branch_to_cell(positioned,
target_cell)
        if trimmed is None:
            _fail(); continue
```

```
        # 6 — Try to split cell; skip
placement if split fails or any cell is too
small
        new_cells = split_cell(target_cell,
trimmed, _debug=verbose)
        if new_cells is None:
            if verbose:
                print(f"  step {i:2d} |
grid={g_idx:2d} | split=FAIL -> skipped |
cells={len(cells)}")
            _fail(); continue
        if any(nc.area <
config.MIN_CELL_AREA_MM2 for nc in
new_cells):
            if verbose:
                print(f"  step {i:2d} |
grid={g_idx:2d} | cell too small -> skipped
| cells={len(cells)}")
            _fail(); continue

        # 7 — Record placement
        placed.append(trimmed)
        t_start =
positioned.project(Point(trimmed.coords[0])) /
positioned.length
        t_end   =
positioned.project(Point(trimmed.coords[-1])) /
positioned.length
        placement_info.append({
            'orig_idx':  stock[0]
['orig_idx'],
            'is_cutoff': stock[0]
['is_cutoff'],
            't_start':   min(t_start,
t_end),
            't_end':     max(t_start,
t_end),
        })
        for cx, cy in [trimmed.coords[0],
trimmed.coords[-1]]:
            anchor_pts.add((round(cx, 2),
round(cy, 2)))

        # 8 — Offcuts
        if use_cutoffs:
            offcut_geom =
positioned.difference(target_cell)
            for oc in
_as_linestrings(offcut_geom):
                if oc.length >=
config.MIN_CUTOFF_LEN:
                    pts      = [(x, y, 0.0)
for x, y in oc.coords]
                    norm     =
normalize_branch(pts)
                    new_b    = {
                        'geometry':  norm,
                        'orig_idx':
stock[0]['orig_idx'],
                        'is_cutoff': True,
**compute_curvature_profile(norm),
                    }
                    if out_cutoffs is not
None:
out_cutoffs.append({
'source_orig_idx': int(new_b['orig_idx']),
                            'coords_mm':
[(float(x), float(y), 0.0) for x, y in
oc.coords],
                            'length_mm':
float(oc.length),
                        })
                    insert_sorted(stock,
new_b, descend)

        # 9 — Pop stock + update cells
        stock.pop(0)
        retry_count = 0
        cells.remove(target_cell)
        cells.extend(new_cells)

        if verbose:
            print(f"  step {i:2d} |
grid={g_idx:2d} | placed #{len(placed)-1} |
"
                  f"split=OK |
cells={len(cells)}")

    centroids = [(c.centroid.x,
c.centroid.y) for c in cells]
    return placed, placement_info, cells,
centroids, len(placed), n_skipped


def collect_replay_steps(
    frame: Polygon,
    branches: list,
    grid_points_gene: list,
    point_a_gene: list,
    point_b_gene: list,
    flip_gene: list,
    use_cutoffs: bool = config.USE_CUTOFFS,
    reverse_order: bool = False,
) -> list:
    """Like replay() but captures the state
before each step for debug visualization.
```

```
    Returns a list of dicts, one per gene
step, each containing:
      cells_before, placed_before,
anchor_pts, target_cell,
      boundary_pts, grid_idx, new_branch,
status ('placed'|'retry'|'skipped')
    """
    descend     = not reverse_order
    cells       = [frame]
    stock       = sorted(
        [{**b, 'orig_idx': i, 'is_cutoff':
False} for i, b in enumerate(branches)],
        key=lambda b: b['chord_length'],
        reverse=descend,
    )
    grid_pts    = build_grid(frame)
    n_grid      = config.GRID_NX *
config.GRID_NY
    placed      = []
    retry_count = 0
    anchor_pts  = {(round(x, 2), round(y,
2)) for x, y in frame.exterior.coords[:-1]}
    steps       = []

    def _fail_step(step):
        nonlocal retry_count
        retry_count += 1
        step['status'] = 'skipped' if
retry_count > config.MAX_RETRIES else
'retry'
        steps.append(step)
        if retry_count >
config.MAX_RETRIES:
            stock.pop(0)
            retry_count = 0

    for i in range(len(grid_points_gene)):
        if not stock:
            break

        step = {
            'step':          i,
            'cells_before':  list(cells),
            'placed_before': list(placed),
            'anchor_pts':
set(anchor_pts),
            'branch_id':     stock[0]
['orig_idx'],
            'is_cutoff':     stock[0]
['is_cutoff'],
            'grid_idx':      None,
            'target_cell':   None,
            'boundary_pts':  [],
            'pt_A':          None,
            'pt_B':          None,
            'new_branch':    None,
            'status':        'skipped',
        }

        g_idx = max(0,
min(int(grid_points_gene[i]), n_grid - 1))
        step['grid_idx'] = g_idx
        gp = Point(grid_pts[g_idx])
        target_cell = next((c for c in
cells if c.contains(gp)), None)
        if target_cell is None:
            _fail_step(step); continue

        step['target_cell'] = target_cell
        boundary_pts =
sample_boundary_n(target_cell,
anchor_pts=anchor_pts)
        step['boundary_pts'] = boundary_pts
        if len(boundary_pts) < 2:
            _fail_step(step); continue

        n_pts = len(boundary_pts)
        idx_a = max(0,
min(int(point_a_gene[i]), n_pts - 1))
        idx_b = max(0,
min(int(point_b_gene[i]), n_pts - 1))
        if idx_a == idx_b:
            _fail_step(step); continue

        pt_A = boundary_pts[idx_a][0]
        pt_B = boundary_pts[idx_b][0]
        step['pt_A'] = pt_A
        step['pt_B'] = pt_B

        if math.dist(pt_A, pt_B) > stock[0]
['chord_length']:
            _fail_step(step); continue

        flip       = int(flip_gene[i])
        positioned =
position_branch(stock[0], pt_A, pt_B, flip)
        trimmed    =
trim_branch_to_cell(positioned,
target_cell)
        if trimmed is None:
            _fail_step(step); continue

        new_cells = split_cell(target_cell,
trimmed)
        if new_cells is None:
            _fail_step(step); continue
        if any(nc.area <
config.MIN_CELL_AREA_MM2 for nc in
```

```
new_cells):
            _fail_step(step); continue

        step['status']     = 'placed'
        step['new_branch'] = trimmed
        placed.append(trimmed)
        for cx, cy in [trimmed.coords[0],
trimmed.coords[-1]]:
            anchor_pts.add((round(cx, 2),
round(cy, 2)))
        step['new_cells']       = new_cells
        step['post_anchor_pts'] =
set(anchor_pts)

        cutoff_length = 0.0
        if use_cutoffs:
            offcut_geom =
positioned.difference(target_cell)
            for oc in
_as_linestrings(offcut_geom):
                cutoff_length += oc.length
                if oc.length >=
config.MIN_CUTOFF_LEN:
                    pts  = [(x, y, 0.0) for
x, y in oc.coords]
                    norm =
normalize_branch(pts)
                    new_b = {
                        'geometry':  norm,
                        'orig_idx':
stock[0]['orig_idx'],
                        'is_cutoff': True,
**compute_curvature_profile(norm),
                    }
                    insert_sorted(stock,
new_b, descend)
        step['cutoff_length'] =
cutoff_length

        stock.pop(0)
        retry_count = 0
        cells.remove(target_cell)
        cells.extend(new_cells)
        steps.append(step)

    return steps


def export_replay_csv(steps: list, path:
str) -> None:
    """Write one row per placement step to
a CSV file.

    Metadata columns: step, status,
grid_idx, branch_id, chord_length,
    pt_A_x, pt_A_y, pt_B_x, pt_B_y,
cell_area.
    Boundary columns: bp_0_x, bp_0_y,
bp_0_anchor, bp_1_x, ... up to
MAX_BOUNDARY_PTS.
    NaN fills slots where boundary_pts
entry is None.
    """
    import csv
    import math as _math

    nan = float('nan')
    n_bp = config.MAX_BOUNDARY_PTS
    bp_headers = []
    for j in range(n_bp):
        bp_headers += [f'bp_{j}_x',
f'bp_{j}_y', f'bp_{j}_anchor']

    n_br = config.MAX_BRANCH_PTS
    br_headers = []
    for j in range(n_br):
        br_headers += [f'br_{j}_x',
f'br_{j}_y', f'br_{j}_dist_chord']

    cell0_headers = ['cell0_area']
    cell1_headers = ['cell1_area']
    for j in range(n_bp):
        cell0_headers += [f'cell0_bp_{j}
_x', f'cell0_bp_{j}_y', f'cell0_bp_{j}
_anchor']
        cell1_headers += [f'cell1_bp_{j}
_x', f'cell1_bp_{j}_y', f'cell1_bp_{j}
_anchor']

    headers = [
        'step', 'status', 'grid_idx',
        'branch_id', 'is_cutoff',
'chord_length', 'cutoff_length',
        'pt_A_x', 'pt_A_y', 'pt_B_x',
'pt_B_y',
        'cell_area',
    ] + bp_headers + br_headers +
cell0_headers + cell1_headers

    with open(path, 'w', newline='') as f:
        writer = csv.writer(f)
        writer.writerow(headers)
        for s in steps:
            tc = s['target_cell']
            branch = s.get('new_branch')
            chord =
_math.dist(branch.coords[0],
branch.coords[-1]) if branch else nan
```

```
            row = [
                s['step'],
                s['status'],
                s['grid_idx'],
                s.get('branch_id', nan),
                int(s.get('is_cutoff',
False)),
                chord,
                s.get('cutoff_length',
nan),
                s['pt_A'][0] if s['pt_A']
else nan,
                s['pt_A'][1] if s['pt_A']
else nan,
                s['pt_B'][0] if s['pt_B']
else nan,
                s['pt_B'][1] if s['pt_B']
else nan,
                tc.area if tc else nan,
            ]
            for pt in s['boundary_pts']:
                if pt is None:
                    row += [nan, nan, nan]
                else:
                    (x, y), is_anchor = pt
                    row += [x, y,
int(is_anchor)]
            br_pts = []
            if branch:
                coords =
list(branch.coords)
                ax, ay = coords[0]
                bx, by = coords[-1]
                for px, py in coords:
                    br_pts.append((px, py,
_dist_to_chord(px, py, ax, ay, bx, by)))
            for k in range(n_br):
                if k < len(br_pts):
                    px, py, dc = br_pts[k]
                    row += [px, py, dc]
                else:
                    row += [nan, nan, nan]
            new_cells      =
s.get('new_cells') or []
            post_anchors   =
s.get('post_anchor_pts')
            for nc in (new_cells if
len(new_cells) == 2 else [None, None]):
                if nc is not None:
                    row.append(nc.area)
                    for pt in
sample_boundary_n(nc,
anchor_pts=post_anchors):
                        if pt is None:
                            row += [nan,
nan, nan]
                        else:
                            (x, y),
is_anchor = pt
                            row += [x, y,
int(is_anchor)]
                else:
                    row += [nan] + [nan,
nan, nan] * n_bp
            writer.writerow(row)


#
------------------------------------------------------------

# [demo / visualisation harness (if
__name__ == '__main__') omitted — not part
of the analysis method]
```

A.2.5 `digital_fachwerk/fachwerk/fea.py`

build_structural_model + analyze — OpenSeesPy linear-static solve; returns max displacement, per-element utilisation.

595 lines

```
"""fea.py

Solver-agnostic structural model for
Fachwerk FEA.

build_structural_model(frame_polygon,
placed_branches) converts the
continuous (mm) geometry produced by the
placement engine into a node/element
description ready for any beam-element FEA
backend (e.g. OpenSeesPy).

Key conventions
---------------
- All coordinates in millimetres.
- Node IDs are dense integers starting at
0.
- Frame corners are rigid; branch-to-frame
and branch-to-branch connections
  are pinned (moment release at branch
endpoints only).
- Endpoints of different branches that
coincide within `tolerance` mm share
  one node.  Internal branch vertices are
always unique nodes.
- Supports are applied to the two bottom
frame corners (pinned: ux, uy fixed; rz
```

```
free).

Material defaults (Oak) — set in config.py
-----------------------------------------
  TIMBER_E             = 9 000 N/mm²
  TIMBER_DENSITY_KG_M3 = 700 kg/m³
  SECTION_WIDTH_MM     = 100 mm
  SECTION_DEPTH_MM     = 200 mm
"""

from __future__ import annotations

import math
from dataclasses import dataclass
from typing import Optional

import openseespywin.opensees as ops
from shapely.geometry import Polygon

from fachwerk import config


#
# ------------------------------------------------------
# Data model
#
# ------------------------------------------------------

@dataclass
class Node:
    id: int
    x: float
    y: float


@dataclass
class Element:
    id: int
    node_i: int          # start node id
    node_j: int          # end node id
    member_type: str     # 'frame' |
'branch'
    branch_idx: Optional[int]   # index
into placed_branches; None for frame
members
    release: str         # 'none' | 'pin_i'
| 'pin_j' | 'pin_both'


@dataclass
class Support:
    node_id: int
    ux: bool   # fix x-translation
    uy: bool   # fix y-translation
    rz: bool   # fix z-rotation


@dataclass
class SectionProps:
    width: float   # mm
    depth: float   # mm

    @property
    def A(self) -> float:
        return self.width * self.depth

    @property
    def Iz(self) -> float:
        return self.width * self.depth **
3 / 12


@dataclass
class TimberProps:
    E: float                    # N/mm²
    density_kg_m3: float        # kg/m³
    frame_section: SectionProps
    branch_section: SectionProps

    def section(self, member_type: str) ->
SectionProps:
        return self.frame_section if
member_type == 'frame' else
self.branch_section

    def weight_per_mm(self, member_type:
str) -> float:
        """Self-weight load intensity [N/
mm]."""
        g = 9810.0  # mm/s²
        density_N_per_mm3 =
self.density_kg_m3 * 1e-9 * g
        return density_N_per_mm3 *
self.section(member_type).A


@dataclass
class StructuralModel:
    nodes: dict          # {node_id: Node}
    elements: list       # [Element, ...]
    supports: list       # [Support, ...]
    timber: TimberProps
    frame_node_ids: list        # node ids
in frame-corner order
    branch_node_ids: list       #
[[node_ids per branch], ...] — one list per
branch
```

```
    #                                                       )
# ------------------------------------------------------------------
    # Convenience accessors
    #                                                     #
# ------------------------------------------------------------------------------

    def n_nodes(self) -> int:                           # Builder
        return len(self.nodes)                          #
                                                        # ------------------------------------------------------------

    def n_elements(self) -> int:                        def build_structural_model(
        return len(self.elements)                           frame_polygon: Polygon,
                                                            placed_branches: list,
    def frame_elements(self) -> list:                       E: float = config.TIMBER_E,
        return [e for e in self.elements if             density_kg_m3: float =
e.member_type == 'frame']                           config.TIMBER_DENSITY_KG_M3,
                                                            frame_section_width: float =
    def branch_elements(self, branch_idx:           config.FRAME_SECTION_WIDTH_MM,
int = None) -> list:                                        frame_section_depth: float =
        if branch_idx is None:                      config.FRAME_SECTION_DEPTH_MM,
            return [e for e in                              branch_section_width: float =
self.elements if e.member_type == 'branch']         config.BRANCH_SECTION_WIDTH_MM,
        return [e for e in self.elements                    branch_section_depth: float =
                if e.member_type ==                 config.BRANCH_SECTION_DEPTH_MM,
'branch' and e.branch_idx == branch_idx]                    tolerance: float =
                                                    config.FEA_NODE_TOLERANCE_MM,
    def element_length(self, element:               ) -> StructuralModel:
Element) -> float:                                      """Convert continuous placement
        ni, nj =                                    geometry to a structural beam-element
self.nodes[element.node_i],                         model.
self.nodes[element.node_j]
        return math.dist((ni.x, ni.y),                  Parameters
(nj.x, nj.y))                                           ----------
                                                        frame_polygon:    Shapely Polygon
    def summary(self) -> str:                       defining the wall panel boundary (mm).
        n_frame  =                                      placed_branches:  List of Shapely
len(self.frame_elements())                          LineStrings — output of placement engine.
        n_branch =                                      E:                Young's modulus [N/
len(self.branch_elements())                         mm²].
        n_sup    = len(self.supports)                   density_kg_m3:    Timber density [kg/
        n_br     =                                  m³].
len(self.branch_node_ids)                               section_width:    Cross-section width
        total_L  =                                  [mm].
sum(self.element_length(e) for e in                     section_depth:    Cross-section depth
self.elements)                                      [mm].
        return (                                        tolerance:        Max distance [mm] at
            f"StructuralModel | "                   which two points share a node.
            f"nodes={self.n_nodes()}
elements={self.n_elements()} "                          Returns
            f"(frame={n_frame},                         -------
branch={n_branch})  "                                   StructuralModel
            f"supports={n_sup}                      with .nodes, .elements, .supports, .timber,
branches={n_br}  "                                      .frame_node_ids, .branch_node_ids.
            f"total_length={total_L:.0f}                """
mm"                                                     nodes: dict[int, Node] = {}
```

```
    _next_id = [0]

    def _get_or_create(x: float, y: float)
-> int:
        for nid, nd in nodes.items():
            if math.dist((x, y), (nd.x,
nd.y)) <= tolerance:
                return nid
        nid = _next_id[0]
        nodes[nid] = Node(id=nid, x=x, y=y)
        _next_id[0] += 1
        return nid

    def _new_unique(x: float, y: float) ->
int:
        nid = _next_id[0]
        nodes[nid] = Node(id=nid, x=x, y=y)
        _next_id[0] += 1
        return nid

    def _t_on_segment(px, py, ax, ay, bx,
by):
        """Return t in (0,1) if point lies
strictly on segment interior, else None.

        Also rejects t values that would
produce a sub-element shorter than
        `tolerance` mm, preventing zero-
length element errors in OpenSees.
        """
        dx, dy = bx - ax, by - ay
        L2 = dx * dx + dy * dy
        if L2 < 1e-10:
            return None
        t = ((px - ax) * dx + (py - ay) *
dy) / L2
        if t <= 0.0 or t >= 1.0:
            return None
        dist = math.sqrt((px - ax - t * dx)
** 2 + (py - ay - t * dy) ** 2)
        if dist > tolerance:
            return None
        L = math.sqrt(L2)
        if t * L <= tolerance or (1.0 - t)
* L <= tolerance:
            return None
        return t

    def _seg_release(orig: str, is_first:
bool, is_last: bool) -> str:
        """Release for a sub-segment after
splitting: preserve outer-end pins only."""
        has_pin_i = orig in ('pin_i',
'pin_both')
        has_pin_j = orig in ('pin_j',
'pin_both')
        i_rel = has_pin_i and is_first
        j_rel = has_pin_j and is_last
        if i_rel and j_rel:
            return 'pin_both'
        if i_rel:
            return 'pin_i'
        if j_rel:
            return 'pin_j'
        return 'none'

    elements: list[Element] = []
    ele_id = [0]

    def _add_element(ni, nj, member_type,
branch_idx, release):
        elements.append(Element(
            id=ele_id[0], node_i=ni,
node_j=nj,
            member_type=member_type,
branch_idx=branch_idx, release=release,
        ))
        ele_id[0] += 1

    def _add_split_elements(chain,
member_type, branch_idx, seg_release):
        """Emit elements for a node chain,
preserving outer-end pin releases."""
        n = len(chain)
        for i in range(n - 1):
            rel = _seg_release(seg_release,
is_first=(i == 0), is_last=(i == n - 2))
            _add_element(chain[i], chain[i
+ 1], member_type, branch_idx, rel)

    #
----------------------------------------------------------
    # Phase 1 — Create ALL nodes before any
elements
    #
----------------------------------------------------------

    # Frame corner nodes
    frame_coords =
list(frame_polygon.exterior.coords[:-1])
    frame_node_ids = [_get_or_create(x, y)
for x, y in frame_coords]

    # Supports: pinned at the two lowest
frame corners (translation fixed, rotation
free)
    min_y = min(nodes[nid].y for nid in
frame_node_ids)
```

```
    supports = [
        Support(node_id=nid, ux=True,
uy=True, rz=False)
        for nid in frame_node_ids
        if abs(nodes[nid].y - min_y) <=
tolerance
    ]

    # Branch nodes
    branch_node_ids: list[list[int]] = []
    for branch in placed_branches:
        coords = list(branch.coords)
        n_pts  = len(coords)
        b_nids = []
        for k, (x, y) in enumerate(coords):
            is_ep = (k == 0 or k == n_pts -
1)
            b_nids.append(_get_or_create(x,
y) if is_ep else _new_unique(x, y))
        branch_node_ids.append(b_nids)

    # All branch endpoints (nodes that can
split frame or branch segments)
    frame_corner_set = set(frame_node_ids)
    branch_endpoints: set[int] = set()
    for b_nids in branch_node_ids:
        if b_nids:
            branch_endpoints.add(b_nids[0])
branch_endpoints.add(b_nids[-1])
    splitter_ids = branch_endpoints -
frame_corner_set

    # Permanent midspan node on the top
beam — ensures vertical deflection at
    # beam midspan is always captured in
max_displacement even with no branches
there.
    max_y = max(nodes[nid].y for nid in
frame_node_ids)
    top_nids = [nid for nid in
frame_node_ids if abs(nodes[nid].y - max_y)
<= tolerance]
    if len(top_nids) == 2:
        mx = (nodes[top_nids[0]].x +
nodes[top_nids[1]].x) / 2
        my = (nodes[top_nids[0]].y +
nodes[top_nids[1]].y) / 2
        beam_mid_nid = _get_or_create(mx,
my)
        splitter_ids.add(beam_mid_nid)

    # Top beam-column connections are
pinned: frame is a mechanism without
branches,
    # forcing the optimizer to place
diagonals for lateral stability.
    top_corner_set = set(top_nids)

    #
----------------------------------------------------
    # Phase 2 — Build elements, splitting
at interior connection points
    #
----------------------------------------------------

    # Frame edges: split where branch
endpoints land on their interior
    n_frame = len(frame_node_ids)
    for k in range(n_frame):
        a_id = frame_node_ids[k]
        b_id = frame_node_ids[(k + 1) %
n_frame]
        a_nd, b_nd = nodes[a_id],
nodes[b_id]
        pts = [(t, nid) for nid in
splitter_ids
               for t in
[_t_on_segment(nodes[nid].x, nodes[nid].y,
a_nd.x, a_nd.y, b_nd.x, b_nd.y)]
               if t is not None]
        pts.sort()
        chain = [a_id] + [nid for _, nid in
pts] + [b_id]
        has_pin_i = a_id in top_corner_set
        has_pin_j = b_id in top_corner_set
        orig_rel = ('pin_both' if has_pin_i
and has_pin_j else
                    'pin_i'   if has_pin_i
else
                    'pin_j'   if has_pin_j
else 'none')
        n_chain = len(chain)
        for i in range(n_chain - 1):
            rel = _seg_release(orig_rel,
is_first=(i == 0), is_last=(i == n_chain -
2))
            _add_element(chain[i], chain[i
+ 1], 'frame', None, rel)

    # Branch elements: split where OTHER
branch endpoints land on their segments
    for b_idx, b_nids in
enumerate(branch_node_ids):
        # Endpoints of all OTHER branches
(not this one) that could split our
segments
        other_eps = splitter_ids -
{b_nids[0], b_nids[-1]}
```

```
        n_segs = len(b_nids) - 1
        for seg_k in range(n_segs):
            ni_id, nj_id = b_nids[seg_k],
b_nids[seg_k + 1]
            ni_nd, nj_nd = nodes[ni_id],
nodes[nj_id]

            if n_segs == 1:
                seg_rel = 'pin_both'
            elif seg_k == 0:
                seg_rel = 'pin_i'
            elif seg_k == n_segs - 1:
                seg_rel = 'pin_j'
            else:
                seg_rel = 'none'

            pts = [(t, nid) for nid in
other_eps
                   for t in
[_t_on_segment(nodes[nid].x, nodes[nid].y,
ni_nd.x, ni_nd.y, nj_nd.x, nj_nd.y)]
                   if t is not None]
            pts.sort()
            chain = [ni_id] + [nid for _,
nid in pts] + [nj_id]
            _add_split_elements(chain,
'branch', b_idx, seg_rel)

    #
------------------------------------------------------------------
    # Assemble and return
    #
------------------------------------------------------------------
    timber = TimberProps(
        E=E,
        density_kg_m3=density_kg_m3,
frame_section=SectionProps(frame_section_width,
frame_section_depth),
branch_section=SectionProps(branch_section_width,
branch_section_depth),
    )

    return StructuralModel(
        nodes=nodes,
        elements=elements,
        supports=supports,
        timber=timber,
        frame_node_ids=frame_node_ids,
        branch_node_ids=branch_node_ids,
    )
```

```
#
------------------------------------------------------------------
# Solver
#
------------------------------------------------------------------

_RELEASE_CODE = {'none': None, 'pin_i': 1,
'pin_j': 2, 'pin_both': 3}


def analyze(
    model: StructuralModel,
    gravity_load: float =
config.FEA_GRAVITY_LOAD_N,
    wind_load: float =
config.FEA_WIND_LOAD_N,
    wind_suction_ratio: float =
config.FEA_WIND_SUCTION_RATIO,
) -> dict:
    """Run a linear static FEA on a
StructuralModel using OpenSeesPy.

    Parameters
    ----------
    model:               StructuralModel
from build_structural_model().
    gravity_load:        Total vertical
load [N] applied as a UDL on the top beam
                         elements
(intensity = gravity_load / beam_width).
Negative = downward.
    wind_load:           Windward pressure
magnitude [N] (intensity = |wind_load| /
frame_height).
                         Positive → wind
from left: pressure +x on left column,
suction -x on right column.
                         Negative → wind
from right: pressure -x on right column,
suction +x on left column.
    wind_suction_ratio:  Leeward suction as
a fraction of the windward pressure
magnitude.

    Returns
    -------
    dict with keys: converged,
max_displacement, max_vertical_disp,
    max_lateral_disp, node_displacements
{node_id: (ux, uy)} in mm.
    Sentinel values (999999.0) returned on
any failure.
    """
```

```
    _sentinel = {
        'converged': False,
        'max_displacement': 999_999.0,
        'max_vertical_disp': 999_999.0,
        'max_lateral_disp': 999_999.0,
        'max_utilization': 999_999.0,
        'mean_branch_util': 0.0,
        'node_displacements': {},
        'element_utilizations': {},
    }

    try:
        ops.wipe()
        ops.model('basic', '-ndm', 2, '-
ndf', 3)
        ops.geomTransf('Linear', 1)

        E = model.timber.E

        # Nodes (OpenSees tags are 1-based)
        for nd in model.nodes.values():
            ops.node(nd.id + 1, nd.x, nd.y)

        # Supports
        support_ids = {s.node_id for s in
model.supports}
        for s in model.supports:
            ops.fix(s.node_id + 1,
int(s.ux), int(s.uy), int(s.rz))

        # Identify pure-hinge nodes: nodes
where every connected element has a
        # moment release at that end. These
have an unrestrained rotation DOF
        # which makes the stiffness matrix
singular. Fix only rz there.
        ele_at_node: dict[int, list] =
{nid: [] for nid in model.nodes}
        for e in model.elements:
            ele_at_node[e.node_i].append(e)
            ele_at_node[e.node_j].append(e)

        def _released_at(e, nid):
            return ((e.node_i == nid and
e.release in ('pin_i', 'pin_both')) or
                    (e.node_j == nid and
e.release in ('pin_j', 'pin_both')))

        for nid, eles in
ele_at_node.items():
            if nid in support_ids or not
eles:
                continue
            if all(_released_at(e, nid) for
e in eles):
                ops.fix(nid + 1, 0, 0, 1)

        for e in model.elements:
            sec =
model.timber.section(e.member_type)
            release =
_RELEASE_CODE[e.release]
            args = ['elasticBeamColumn',
e.id + 1, e.node_i + 1, e.node_j + 1,
                    sec.A, E, sec.Iz, 1]
            if release is not None:
                args += ['-release',
release]
            ops.element(*args)

        # Loads — elemental UDL on frame
edges so intensity is fixed regardless
        # of how many branch connections
split the beam/column into sub-elements.
        tol   =
config.FEA_NODE_TOLERANCE_MM
        max_y = max(model.nodes[nid].y for
nid in model.frame_node_ids)
        max_x = max(model.nodes[nid].x for
nid in model.frame_node_ids)
        min_x = min(model.nodes[nid].x for
nid in model.frame_node_ids)
        min_y = min(model.nodes[nid].y for
nid in model.frame_node_ids)
        beam_length   = max_x - min_x
        column_height = max_y - min_y

        # Top beam and right column frame
elements (may be split by branch endpoints)
        top_beam_eles = [e for e in
model.elements
                         if e.member_type
== 'frame'
                         and
abs(model.nodes[e.node_i].y - max_y) <= tol
                         and
abs(model.nodes[e.node_j].y - max_y) <=
tol]
        right_col_eles = [e for e in
model.elements
                          if e.member_type
== 'frame'
                          and
abs(model.nodes[e.node_i].x - max_x) <= tol
                          and
abs(model.nodes[e.node_j].x - max_x) <=
tol]
        left_col_eles = [e for e in
```

```
model.elements
                        if e.member_type
== 'frame'
                        and
abs(model.nodes[e.node_i].x - min_x) <= tol
                        and
abs(model.nodes[e.node_j].x - min_x) <=
tol]

        if not top_beam_eles:
            ops.wipe()
            return _sentinel

        def _wy(ni: Node, nj: Node,
fx_global: float, fy_global: float) ->
float:
            """Project a global force/
length onto the element's local y axis."""
            dx = nj.x - ni.x
            dy = nj.y - ni.y
            L  = math.sqrt(dx * dx + dy *
dy)
            # local y unit vector = (-dy/L,
dx/L)
            return fx_global * (-dy / L) +
fy_global * (dx / L)

        ops.timeSeries('Linear', 1)
        ops.pattern('Plain', 1, 1)

        # Gravity: UDL along the top beam
(intensity constant regardless of sub-
elements)
        q_gravity = gravity_load /
beam_length
        for e in top_beam_eles:
            wy = _wy(model.nodes[e.node_i],
model.nodes[e.node_j], 0.0, q_gravity)
            ops.eleLoad('-ele', e.id + 1,
'-type', '-beamUniform', wy)

        # Wind: combined pressure
(windward) + suction (leeward) in one load
pattern.
        # Both forces act in the same
direction (wind direction): pressure pushes
windward
        # column inward, suction pulls
leeward column outward — both laterally
load the frame.
        # wind_load > 0 → wind from left:
+x on left column, +x on right column.
        # wind_load < 0 → wind from right:
-x on right column, -x on left column.
        if wind_load != 0.0:
            q_pressure = wind_load /
column_height
            q_suction  = wind_suction_ratio
* wind_load / column_height
            pressure_eles = left_col_eles
if wind_load > 0 else right_col_eles
            suction_eles  = right_col_eles
if wind_load > 0 else left_col_eles
            for e in pressure_eles:
                wy =
_wy(model.nodes[e.node_i],
model.nodes[e.node_j], q_pressure, 0.0)
                ops.eleLoad('-ele', e.id +
1, '-type', '-beamUniform', wy)
            for e in suction_eles:
                wy =
_wy(model.nodes[e.node_i],
model.nodes[e.node_j], q_suction, 0.0)
                ops.eleLoad('-ele', e.id +
1, '-type', '-beamUniform', wy)

        # Analysis
        ops.system('BandGeneral')
        ops.numberer('RCM')
        ops.constraints('Plain')
        ops.integrator('LoadControl', 1.0)
        ops.algorithm('Linear')
        ops.analysis('Static')
        converged = (ops.analyze(1) == 0)
        if not converged:
            return _sentinel

        # Extract displacements
        node_disps = {}
        for nid in model.nodes:
            ux, uy, _ = ops.nodeDisp(nid +
1)
            node_disps[nid] = (ux, uy)

        max_disp     = max(math.sqrt(ux**2
+ uy**2) for ux, uy in node_disps.values())
        max_vert     = max(abs(uy) for _,
uy in node_disps.values())
        max_lat      = max(abs(ux) for ux,
_ in node_disps.values())

        # Utilization — EC5 simplified
interaction for each element
        max_util = 0.0
        element_utilizations: dict[int,
float] = {}
        branch_util_x_len = 0.0     # Σ
util_e · L_e   (branches only)
```

```
        branch_total_len  = 0.0     # Σ L_e
(branches only)
        for e in model.elements:
            ni, nj = model.nodes[e.node_i],
model.nodes[e.node_j]
            dx, dy = nj.x - ni.x, nj.y -
ni.y
            L = math.sqrt(dx * dx + dy *
dy)
            if L < 1e-6:
                element_utilizations[e.id]
= 0.0
                continue
            gf = ops.eleForce(e.id + 1)  #
[Fx1, Fy1, Mz1, Fx2, Fy2, Mz2] global
            cx, cy = dx / L, dy / L
            N = cx * gf[0] + cy * gf[1]  #
local axial force at node i
            M = max(abs(gf[2]), abs(gf[5]))
# governing end moment
            sec =
model.timber.section(e.member_type)
            Wz = sec.width * sec.depth **
2 / 6
            sigma_m = M / Wz
            sigma_n = abs(N) / sec.A
            if N <= 0:  # compression +
bending
                util = (sigma_n /
config.TIMBER_FC0K) ** 2 + sigma_m /
config.TIMBER_FMK
            else:       # tension + bending
                util = sigma_n /
config.TIMBER_FT0K + sigma_m /
config.TIMBER_FMK
            element_utilizations[e.id] =
util
            if util > max_util:
                max_util = util
            if e.member_type == 'branch':
                branch_util_x_len += util *
L
                branch_total_len  += L

        mean_branch_util =
(branch_util_x_len / branch_total_len) if
branch_total_len > 0 else 0.0

        return {
            'converged': converged,
            'max_displacement': max_disp,
            'max_vertical_disp': max_vert,
            'max_lateral_disp': max_lat,
            'max_utilization': max_util,
            'mean_branch_util':
mean_branch_util,
            'node_displacements':
node_disps,
            'element_utilizations':
element_utilizations,
        }

    except Exception as exc:
        print(f'[fea.analyze] error:
{exc}')
        return _sentinel

    finally:
        ops.wipe()


#
------------------------------------------------------------

# [demo / visualisation harness (if
__name__ == '__main__') omitted — not part
of the analysis method]
```

A.2.6 digital_fachwerk/fachwerk/optimizer.py

FachwerkProblem (pymoo ElementwiseProblem) and the NSGA-II run loop: 2 objectives, 1 constraint, interleaved genome, worst-case wind symmetry.

512 lines

```
import json
from datetime import datetime
from pathlib import Path

import matplotlib.pyplot as plt
import numpy as np
from pymoo.algorithms.moo.nsga2 import
NSGA2
from pymoo.core.problem import
ElementwiseProblem
from pymoo.operators.crossover.pntx import
TwoPointCrossover
from pymoo.operators.crossover.sbx import
SBX
from pymoo.operators.crossover.ux import UX
from pymoo.operators.mutation.pm import PM
from pymoo.operators.repair.rounding import
RoundingRepair
from pymoo.operators.sampling.rnd import
IntegerRandomSampling
from pymoo.core.callback import Callback
from pymoo.indicators.hv import HV
```

```
from pymoo.optimize import minimize

from fachwerk import config
from fachwerk.fea import analyze,
build_structural_model
from fachwerk.placement import replay

# Fixed reference point for hypervolume and
plot axes.
# Axes order matches pymoo objectives:
[max_displacement_mm,
total_branch_length_mm]
HV_REF_POINT = np.array([150.0, 25_000.0])
PLOT_XLIM = (0, float(HV_REF_POINT[1]))   #
total branch length
PLOT_YLIM = (0, float(HV_REF_POINT[0]))   #
max displacement


class ParetoHistoryCallback(Callback):
    def __init__(self):
        super().__init__()
        self.pareto_F = []

    def notify(self, algorithm):
self.pareto_F.append(algorithm.opt.get('F').copy())


class FachwerkProblem(ElementwiseProblem):
    def __init__(self, frame, branches,
n_steps=10, wind_load=None):
        n = n_steps
        xl = np.zeros(4 * n, dtype=int)
        # Interleaved layout: [gp_0, pa_0,
pb_0, fl_0, gp_1, pa_1, pb_1, fl_1, ...].
        # Keeps each step's 4 genes
adjacent so 2pt crossover swaps whole step-
tuples
        # rather than scrambling them
across the gp/pa/pb/fl block boundaries.
        xu = np.array(
            [config.GRID_NX *
config.GRID_NY - 1,
             config.MAX_BOUNDARY_PTS - 1,
             config.MAX_BOUNDARY_PTS - 1,
             1] * n,
            dtype=int,
        )
        super().__init__(n_var=4 * n,
n_obj=2, n_ieq_constr=1, xl=xl, xu=xu,
vtype=int)
        self.frame = frame
        self.branches = branches
        self.n_steps = n
        self.wind_load = wind_load if
wind_load is not None else
config.FEA_WIND_LOAD_N
        self.total_evals = 0
        self.total_placed = 0
        self.total_skipped = 0

    def _evaluate(self, x, out, *args,
**kwargs):
        x = x.astype(int)
        gp = x[0::4].tolist()
        pa = x[1::4].tolist()
        pb = x[2::4].tolist()
        fl = x[3::4].tolist()

        placed, _, _, _, n_placed,
n_skipped = replay(self.frame,
self.branches, gp, pa, pb, fl)
        self.total_evals += 1
        self.total_placed += n_placed
        self.total_skipped += n_skipped

        if not placed:
            out['F'] =
[float(HV_REF_POINT[0]),
float(HV_REF_POINT[1])]
            out['G'] = [999.0]
            return

        total_length = sum(ls.length for ls
in placed)
        model =
build_structural_model(self.frame, placed)

        # Reject before calling OpenSeesPy
if any element has zero length —
        # the C-level error cannot be
caught by Python's except.
        if any(model.element_length(e) <
config.FEA_NODE_TOLERANCE_MM
               for e in model.elements):
            out['F'] =
[float(HV_REF_POINT[0]), total_length]
            out['G'] = [999.0]
            return

        result_left  = analyze(model,
wind_load=+self.wind_load)   # wind from
left:  pressure on left col, suction on
right
        result_right = analyze(model,
wind_load=-self.wind_load)   # wind from
right: pressure on right col, suction on
left
```

```
        worst_disp =
max(result_left['max_displacement'],
result_right['max_displacement'])
        worst_util =
max(result_left['max_utilization'],
result_right['max_utilization'])

        out['F'] = [worst_disp,
total_length]
        out['G'] = [worst_util - 1.0]


def run_optimization(
    frame,
    branches,
    n_gen: int = 50,
    pop_size: int = 350,
    n_steps: int = 10,
    crossover: str = '2pt',   # 'sbx',
'ux', or '2pt'
    sbx_eta: float = 15.0,
    mutation_eta: float = 3.0,
    mutation_prob_mult: float = 6.0,
    seed: int = 30,
    results_dir: str | None = None,
    stock_file: str | None = None,
    wind_load: float | None = None,
) -> dict:
    """Run NSGA-II to minimise deflection
and total branch length.

    Returns a dict with metadata and the
Pareto front solutions.
    Saves results/run_{timestamp}.json and
results/run_{timestamp}_pareto.png.

    crossover: 'sbx' (Simulated Binary),
'ux' (Uniform), or '2pt' (Two-Point, best
for integer genomes).
    sbx_eta / mutation_eta: distribution
index — lower = more exploratory, higher =
more local.
    mutation_prob_mult: multiplier on the
base per-gene mutation rate 1/(4*n_steps).
    """
    timestamp =
datetime.now().strftime('%Y%m%d_%H%M%S')
    results_path = Path(results_dir) if
results_dir else
Path(__file__).parent.parent / 'results'
    results_path.mkdir(parents=True,
exist_ok=True)

    n_branches = len(branches)
    total_stock_length_mm =
round(sum(b['geometry'].length for b in
branches), 1)

    if crossover == 'sbx':
        crossover_op = SBX(prob=0.9,
eta=sbx_eta, vtype=float,
repair=RoundingRepair())
    elif crossover == 'ux':
        crossover_op = UX()
    else:  # '2pt'
        crossover_op =
TwoPointCrossover(prob=0.9)

    actual_wind = wind_load if wind_load is
not None else config.FEA_WIND_LOAD_N
    callback = ParetoHistoryCallback()
    problem = FachwerkProblem(frame,
branches, n_steps=n_steps,
wind_load=actual_wind)

    algorithm = NSGA2(
        pop_size=pop_size,
        sampling=IntegerRandomSampling(),
        crossover=crossover_op,
mutation=PM(prob=mutation_prob_mult / (4 *
n_steps), eta=mutation_eta, vtype=float,
repair=RoundingRepair()),
        eliminate_duplicates=True,
    )

    res = minimize(
        problem,
        algorithm,
        ('n_gen', n_gen),
        seed=seed,
        verbose=True,
        callback=callback,
    )

    meta = {
        'n_gen':                  n_gen,
        'pop_size':               pop_size,
        'n_steps':                n_steps,
        'genome_layout':
'interleaved',
        'crossover':
crossover,
        'sbx_eta':                sbx_eta,
        'mutation_eta':
mutation_eta,
        'mutation_prob_mult':
mutation_prob_mult,
        'seed':                   seed,
```

```
        'stock_file':
stock_file if stock_file is not None else
Path(config.TREE_OBJ_PATH).name,
        'n_branches':
n_branches,
        'total_stock_length_mm':
total_stock_length_mm,
        'fea_gravity_load_n':
config.FEA_GRAVITY_LOAD_N,
        'fea_wind_load_n':
actual_wind,
        'frame_section_mm':
f'{config.FRAME_SECTION_WIDTH_MM}
x{config.FRAME_SECTION_DEPTH_MM}',
        'branch_section_mm':
f'{config.BRANCH_SECTION_WIDTH_MM}
x{config.BRANCH_SECTION_DEPTH_MM}',
        'timber_E_mpa':
config.TIMBER_E,
    }

    total_steps = problem.total_placed +
problem.total_skipped
    placement_rate = problem.total_placed /
total_steps * 100 if total_steps else 0.0
    print(
        f'\nPlacement stats across
{problem.total_evals} evaluations: '
        f'{problem.total_placed} placed,
{problem.total_skipped} skipped '
        f'({placement_rate:.1f}% success
rate)'
    )
    meta['placement_total_evals']   =
problem.total_evals
    meta['placement_total_placed']  =
problem.total_placed
    meta['placement_total_skipped'] =
problem.total_skipped

    # Build Pareto front entries (re-run
replay to get n_placed / n_skipped)
    if res.X is None:
        print('Warning: no valid solutions
found — Pareto front is empty.')
        return {'timestamp': timestamp,
'obj_path': str(config.TREE_OBJ_PATH),
                'hv_ref_point': None,
'hv_history': [], 'pareto_front': [],
**meta}

    pareto_front = []
    for genome_arr in res.X:
        genome =
genome_arr.astype(int).tolist()
        gp = genome[0::4]
        pa = genome[1::4]
        pb = genome[2::4]
        fl = genome[3::4]
        placed_pf, _, cells_pf, _,
n_placed, n_skipped = replay(frame,
branches, gp, pa, pb, fl)
        worst_util_pf = 999_999.0
        worst_disp_pf =
float(HV_REF_POINT[0])
        total_length_pf = 0.0
        fea_nodes = fea_disps =
fea_elements = []
        if placed_pf:
            total_length_pf = sum(ls.length
for ls in placed_pf)
            model_pf =
build_structural_model(frame, placed_pf)
            if not
any(model_pf.element_length(e) <
config.FEA_NODE_TOLERANCE_MM
                       for e in
model_pf.elements):
                r_l = analyze(model_pf,
wind_load=+actual_wind)
                r_r = analyze(model_pf,
wind_load=-actual_wind)
                worst_util_pf =
max(r_l['max_utilization'],
r_r['max_utilization'])
                worst_disp_pf =
max(r_l['max_displacement'],
r_r['max_displacement'])
                r_worst = r_l if
r_l['max_displacement'] >=
r_r['max_displacement'] else r_r
                node_ids  =
sorted(model_pf.nodes.keys())
                nid_to_k  = {nid: k for k,
nid in enumerate(node_ids)}
                fea_nodes    =
[[model_pf.nodes[nid].x,
model_pf.nodes[nid].y] for nid in node_ids]
                fea_disps    =
[list(r_worst['node_displacements'].get(nid,
[0.0, 0.0])) for nid in node_ids]
                fea_elements =
[[nid_to_k[e.node_i], nid_to_k[e.node_j]]
for e in model_pf.elements]
        if n_placed == 0 or worst_util_pf >
1.0:
            continue  # drop infeasible /
empty solutions from the saved front
```

```
        pareto_front.append({
            'genome':
genome,
            'max_displacement':
worst_disp_pf,
            'total_material_length':
total_length_pf,
            'max_utilization':
worst_util_pf,
            'n_placed':
n_placed,
            'n_skipped':
n_skipped,
            'branches_coords':
[list(ls.coords) for ls in placed_pf],
            'cells_coords':
[[list(xy) for xy in p.exterior.coords] for
p in (cells_pf or [])],
            'fea_nodes':
fea_nodes,
            'fea_disps':
fea_disps,
            'fea_elements':
fea_elements,
        })

    # Compute hypervolume per generation
from callback
    all_F = np.vstack(callback.pareto_F)
    valid_F = all_F[np.all(all_F < 999_000,
axis=1)]
    hv_history = []
    ref_point = HV_REF_POINT.tolist()
    if len(valid_F) > 0:
        hv_indicator =
HV(ref_point=HV_REF_POINT)
        for gen_F in callback.pareto_F:
            valid_gen = gen_F[np.all(gen_F
< 999_000, axis=1)]
hv_history.append(float(hv_indicator(valid_gen))
if len(valid_gen) > 0 else 0.0)

    pareto_front_history = [
        gen_F[np.all(gen_F < 999_000,
axis=1)].tolist()
        for gen_F in callback.pareto_F
    ]

    trend_history = [
_extract_anchor_points(gen_F[np.all(gen_F <
999_000, axis=1)])
        for gen_F in callback.pareto_F
    ]

    result_dict = {
        'timestamp':            timestamp,
        'obj_path':
str(config.TREE_OBJ_PATH),
        **meta,
        'hv_ref_point':         ref_point,
        'hv_history':           hv_history,
        'pareto_front_history':
pareto_front_history,
        'trend_history':
trend_history,
        'pareto_front':
pareto_front,
    }

    json_path = results_path /
f'run_{timestamp}.json'
    with open(json_path, 'w') as f:
        json.dump(result_dict, f, indent=2)
    print(f'Results saved to {json_path}')

    _save_pareto_plot(pareto_front,
results_path / f'run_{timestamp}
_pareto.png', meta=meta)
    if hv_history:
        _save_convergence_plot(hv_history,
results_path / f'run_{timestamp}
_convergence.png', meta=meta)
    if callback.pareto_F:
_save_pareto_history_plot(callback.pareto_F,
results_path / f'run_{timestamp}
_pareto_history.png', meta=meta)
    if any(t is not None for t in
trend_history):
        _save_trendline_plot(trend_history,
results_path / f'run_{timestamp}
_trendlines.png', meta=meta)
_save_trajectory_plot(trend_history,
results_path / f'run_{timestamp}
_trajectory.png', meta=meta)

    return result_dict


def _meta_label(meta: dict) -> str:
    cx = meta.get('crossover', '?')
    cx_str =
f"SBX(η={meta.get('sbx_eta')})" if cx ==
'sbx' else cx.upper()
    lines = [
        f"stock : {meta.get('stock_file')}
"
```

```
        f"{meta.get('n_branches')} branches
"
f"{meta.get('total_stock_length_mm'):,.0f}
mm total",
        f"GA    : n_gen={meta.get('n_gen')}
pop={meta.get('pop_size')}  "
        f"steps={meta.get('n_steps')}
{cx_str}  "
        f"mut_η={meta.get('mutation_eta')}
p×{meta.get('mutation_prob_mult')}  "
        f"seed={meta.get('seed')}",
        f"loads :
G={abs(meta.get('fea_gravity_load_n',
0)):,.0f} N ↓   "
        f"W={meta.get('fea_wind_load_n',
0):,.0f} N →",
        f"section: frame
{meta.get('frame_section_mm')}  "
        f"branch
{meta.get('branch_section_mm')}  "
        f"E={meta.get('timber_E_mpa'):,.0f}
MPa",
    ]
    return '\n'.join(lines)


def _annotate(ax, meta: dict, x=0.01,
y=0.01, va='bottom', ha='left') -> None:
    ax.text(x, y, _meta_label(meta),
            transform=ax.transAxes,
fontsize=6, va=va, ha=ha,
            fontfamily='monospace',
bbox=dict(boxstyle='round,pad=0.3',
fc='white', alpha=0.8, ec='#cccccc'))


def _save_convergence_plot(hv_history:
list, path: Path, meta: dict | None = None)
-> None:
    fig, ax = plt.subplots(figsize=(8, 4))
    ax.plot(range(1, len(hv_history) + 1),
hv_history, color='steelblue',
linewidth=1.5)
    ax.set_xlabel('Generation')
    ax.set_ylabel('Hypervolume')
    ax.set_title('Convergence — hypervolume
per generation')
    ax.grid(True, alpha=0.3)
    if meta:
        _annotate(ax, meta)
    fig.tight_layout()
    fig.savefig(path, dpi=150)
    plt.close(fig)
    print(f'Convergence plot saved to
{path}')


def _save_pareto_history_plot(pareto_F:
list, path: Path, n_snapshots: int = 10,
meta: dict | None = None) -> None:
    """Plot Pareto fronts at evenly spaced
generations, coloured light→dark."""
    indices = np.linspace(0, len(pareto_F)
- 1, min(n_snapshots, len(pareto_F)),
dtype=int)
    cmap = plt.get_cmap('Blues')
    colours = [cmap(0.25 + 0.75 * i /
max(len(indices) - 1, 1)) for i in
range(len(indices))]

    fig, ax = plt.subplots(figsize=(8, 5))
    for colour_idx, gen_idx in
enumerate(indices):
        opt_F = pareto_F[gen_idx]
        valid = opt_F[np.all(opt_F <
999_000, axis=1)]
        if len(valid) == 0:
            continue
        gen_num = gen_idx + 1
        label = f'Gen {gen_num}' if
colour_idx in (0, len(indices) - 1) else
None
        ax.scatter(valid[:, 1], valid[:,
0],
color=colours[colour_idx], s=20, alpha=0.8,
label=label, zorder=2)

    ax.set_xlabel('Total branch length
(mm)')
    ax.set_ylabel('Max displacement (mm)')
    ax.set_title('Pareto front history —
light = early, dark = final')
    ax.set_xlim(*PLOT_XLIM)
    ax.set_ylim(*PLOT_YLIM)
    ax.legend(loc='upper right')
    ax.grid(True, alpha=0.3)
    if meta:
        _annotate(ax, meta, x=0.99, y=0.99,
va='top', ha='right')
    fig.tight_layout()
    fig.savefig(path, dpi=150)
    plt.close(fig)
    print(f'Pareto history plot saved to
{path}')


def _extract_anchor_points(F: np.ndarray) -
```

```
> dict | None:
    """Return three anchor points from one
generation's filtered Pareto front.

    F: (k, 2) array of
[max_displacement_mm, total_length_mm] for
valid solutions.
    Returns dict with 'min_disp',
'min_length', 'knee' (each [disp, length]),
    or None if the front has fewer than 2
points.
    Knee = point with max perpendicular
distance, in normalised [0,1] space,
    to the chord connecting the two front
extremes (standard convex-front
definition).
    """
    if len(F) < 2:
        return None
    min_disp_idx   = int(F[:, 0].argmin())
    min_length_idx = int(F[:, 1].argmin())
    A = F[min_disp_idx]
    B = F[min_length_idx]
    if min_disp_idx == min_length_idx or
len(F) == 2:
        knee = A
    else:
        F_min = F.min(axis=0)
        span  = F.max(axis=0) - F_min
        span[span == 0] = 1.0
        Fn    = (F - F_min) / span
        chord = Fn[min_length_idx] -
Fn[min_disp_idx]
        chord_len =
float(np.linalg.norm(chord))
        if chord_len < 1e-12:
            knee = A
        else:
            v = Fn - Fn[min_disp_idx]
            dist = np.abs(v[:, 0] *
chord[1] - v[:, 1] * chord[0]) / chord_len
            knee = F[int(dist.argmax())]
    return {
        'min_disp':   A.tolist(),
        'min_length': B.tolist(),
        'knee':       knee.tolist(),
    }


_TREND_KEYS    = ('min_disp', 'knee',
'min_length')
_TREND_COLOURS = {'min_disp': '#1f77b4',
'knee': '#ff7f0e', 'min_length': '#d62728'}
_TREND_CMAPS   = {'min_disp': 'Blues',
'knee': 'Oranges',  'min_length': 'Reds'}
_TREND_LABELS  = {'min_disp': 'Min
displacement', 'knee': 'Knee',
'min_length': 'Min length'}


def _save_trendline_plot(trend_history:
list, path: Path, meta: dict | None = None)
-> None:
    """Stacked subplots: each objective vs
generation, one line per anchor point."""
    n = len(trend_history)
    gens = np.arange(1, n + 1)
    disp   = {k: np.full(n, np.nan) for k
in _TREND_KEYS}
    length = {k: np.full(n, np.nan) for k
in _TREND_KEYS}
    for i, t in enumerate(trend_history):
        if t is None:
            continue
        for k in _TREND_KEYS:
            disp[k][i]   = t[k][0]
            length[k][i] = t[k][1]

    fig, (ax_d, ax_l) = plt.subplots(2, 1,
figsize=(8, 7), sharex=True)
    for k in _TREND_KEYS:
        ax_d.plot(gens, disp[k],
color=_TREND_COLOURS[k],
label=_TREND_LABELS[k], linewidth=1.5)
        ax_l.plot(gens, length[k],
color=_TREND_COLOURS[k],
label=_TREND_LABELS[k], linewidth=1.5)
    ax_d.set_ylabel('Max displacement
(mm)')
    ax_d.set_title('Anchor-point trendlines
across generations')
    ax_d.grid(True, alpha=0.3)
    ax_d.legend(loc='upper right')
    ax_l.set_xlabel('Generation')
    ax_l.set_ylabel('Total branch length
(mm)')
    ax_l.grid(True, alpha=0.3)
    if meta:
        _annotate(ax_l, meta)
    fig.tight_layout()
    fig.savefig(path, dpi=150)
    plt.close(fig)
    print(f'Trendlines plot saved to
{path}')


def _save_trajectory_plot(trend_history:
list, path: Path, meta: dict | None = None)
```

```
-> None:
    """2D objective-space trajectories for
each anchor point, light->dark over
gens."""
    fig, ax = plt.subplots(figsize=(8, 6))
    for k in _TREND_KEYS:
        xs, ys = [], []
        for t in trend_history:
            if t is None:
                continue
            xs.append(t[k][1])
            ys.append(t[k][0])
        if not xs:
            continue
        xs = np.array(xs)
        ys = np.array(ys)
        cmap =
plt.get_cmap(_TREND_CMAPS[k])
        n_pts = len(xs)
        for i in range(n_pts - 1):
            shade = 0.25 + 0.75 * i /
max(n_pts - 1, 1)
            ax.plot(xs[i:i+2], ys[i:i+2],
color=cmap(shade), linewidth=1.0, zorder=2)
        ax.scatter(xs[-1], ys[-1],
color=cmap(1.0), s=70,
                   edgecolor='black',
linewidth=0.5, zorder=3,
label=_TREND_LABELS[k])

    ax.set_xlabel('Total branch length
(mm)')
    ax.set_ylabel('Max displacement (mm)')
    ax.set_title('Anchor-point trajectories
— light = early gens, dark = late')
    ax.set_xlim(*PLOT_XLIM)
    ax.set_ylim(*PLOT_YLIM)
    ax.legend(loc='upper right')
    ax.grid(True, alpha=0.3)
    if meta:
        _annotate(ax, meta)
    fig.tight_layout()
    fig.savefig(path, dpi=150)
    plt.close(fig)
    print(f'Trajectory plot saved to
{path}')


def _save_pareto_plot(pareto_front: list,
path: Path, meta: dict | None = None) ->
None:
    xs = [s['total_material_length'] for s
in pareto_front]
    ys = [s['max_displacement'] for s in
pareto_front]

    fig, ax = plt.subplots(figsize=(8, 5))
    ax.scatter(xs, ys, c='crimson', s=60,
zorder=3, label=f'Pareto front
(n={len(pareto_front)})')
    ax.set_xlabel('Total branch length
(mm)')
    ax.set_ylabel('Max displacement (mm)')
    ax.set_title('Pareto Front — deflection
vs material')
    ax.set_xlim(*PLOT_XLIM)
    ax.set_ylim(*PLOT_YLIM)
    ax.legend()
    ax.grid(True, alpha=0.3)
    if meta:
        _annotate(ax, meta)
    fig.tight_layout()
    fig.savefig(path, dpi=150)
    plt.close(fig)
    print(f'Pareto plot saved to {path}')


if __name__ == '__main__':
    import sys
    sys.path.insert(0,
str(Path(__file__).parent.parent))

    from fachwerk.geometry import
create_frame, load_branches_from_obj

    OBJ_PATH = config.TREE_OBJ_PATH
    frame    =
create_frame(config.FRAME_WIDTH_MM,
config.FRAME_HEIGHT_MM)
    branches =
load_branches_from_obj(OBJ_PATH)

    result = run_optimization(frame,
branches, crossover='2pt',
mutation_prob_mult=2.0, mutation_eta=10.0)

    print(f'\nPareto front
({len(result["pareto_front"])}
solutions):')
    for sol in result['pareto_front']:
        print(
            f"
disp={sol['max_displacement']:.2f}mm  "
f"material={sol['total_material_length']:.0f}
mm  "
            f"placed={sol['n_placed']}
skipped={sol['n_skipped']}"
        )
```