GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

ISSN 1612-6793

Bachelor's Thesis

submitted in partial fulfilment of the
requirements for the course "Applied Computer Science"

Implementation And Comparison Of Two Simulation Methods For The Bending Of Cylindrical Objects Under Their Own Weight

Lukas Gürtler

Institute of Computer Science

Bachelor's and Master's Theses
of the Center for Computational Sciences
at the Georg-August-Universität Göttingen

18 May 2018

Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

☎ +49 (551) 39-172000
FAX +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de
🌍 `www.informatik.uni-goettingen.de`

First Supervisor:	Prof. Dr. Winfried Kurth
Second Supervisor:	Prof. Dr. Stephan Waack

I hereby declare that I have written this thesis independently without any help from others and without the use of documents or aids other than those stated. I have mentioned all used sources and cited them correctly according to established academic citation rules.

Göttingen, 18 May 2018

Abstract

There is a need for a realistic physics based behavior of computer tree models. In this work two methods from literature to calculate the behaviors of cylinder-shaped objects being subject to self-bending under their own weight are implemented in XL. The rule-based modeling language XL is especially used for functional-structural plant modeling. The work's concrete application is located in modeling tree branch structures with ramification of higher orders. An approximative force-applying approach and a finite element approach, which also considers reaction wood building and phototropism, are presented. Both methods are applied to three existing tree models. Furthermore a comparison in terms of phenotype, analysis of sensitivity, runtime and memory consumption is accomplished.

Contents

List of Figures

Chapter 1

Introduction

Tree simulation models that intend to be realistic need to consider the laws of physics and biomechanics. Tree models primarily are not static, they are thus dynamic systems. That means their appearance changes within a more or less complex development process. Because of that, fixed shapes of branches or fixed lists of rotation angles, even if they are taken from mechanical calculations or experimental data, can not fully match the request for modeling living organisms. A programming language that suits the call for the combination of structure and function and is outstandingly appropriate for plant modeling is the rule-based language XL. XL is a Java-based implementation of relational growth grammars. The concept of the underlaying relational growth grammar for functional-structural plant modeling has been developed by [1]. For this work all implementations are written in XL. Owning a suitable language, tree models with their development process, which consist of primary and secondary growth, different tropisms (gravitropism, phototropism) and other reaction mechanisms (reaction wood, leafs, needles), need to be expanded with sufficient algorithms that suit these demands for a physics based mechanical shape evolution. This work deals with two such algorithms: an approximative approach and an incremental finite element formulation.

An approach for a simple force applying bending mechanism for branches has been proposed by [2]. Several parameters such as discretization, the initial angles of deflection, a material constant and finally a conicity value can be set. Another parameter that was used in [3] also enables the possibility of the branch to bend upward at a certain point, with the meaning of phototropism. In favour of simplicity this method makes some approximation assumptions, hence it has only few parameters.

An incremental finite element approach has been formulated by [4] and [5]. It completely fulfills the obligation of connection to physics. Fundamental branch properties besides its length or radius like material characteristics e.g. the Young's modulus, density distribution etc. are adjustable. The word *incremental* in this algorithm is not only based on the fragmentation in finite elements but also on the feature of an evolution over discrete time steps where branch growth (primary,

secondary) takes place. It also refers to the feature of this algorithm that the branch shape is updated with the new structure data (weight, length, rotation angles) after every growth step and by that the correct physical development process occurs within the growth process.
In this work the two mentioned methods from [2] and [4] [5] are implemented in XL. Furthermore both methods are applied to three existing tree models. After that the algorithms are compared in terms of phenotype differences, analysis of sensitivity, runtime and memory consumption.

The document structure reveals as follows:

- In chapter 2 the theoretical base of the two methods is expounded.
- In chapter 3 the two methods are implemented with references to the theoretical base handled in chapter 2.
- In chapter 4 both developed algorithms are applied to three existing tree models. All adaptation steps are described.
- In chapter 5 a comparison considering the phenotype differences of the two algorithms is carried out examining various parameter ranges.
- In chapter 6 an analysis of sensitivity examining various parameter ranges for the two algorithms is accomplished. The working basis is given by two tree models from chapter 4.
- In chapter 7 the differences considering the runtime and memory consumption between the two algorithms are examined.
- In chapter 8 the overall result is summed up and an outlook to further adaptation possibilities is given.

Chapter 2

Basics

In this chapter the basics of the underlying physics for the force-applying approximative method and for the incremental finite element formulation is presented. All methods lean on some assumptions that specify the character of these methods in terms of physical, mathematical or biological approximations. The reason for this is no matter which simulation is considered all of them need to make assumptions of that kind because none of them can fully cover reality and even needs to do so due to the fact that not all aspects are necesary, due to the limitation of computational ressources etc. For example both methods approximate branches as cylinder-shaped objects. The following explanations can be seen as summaries from the principal sources [2] and [4] [5].

2.1 Force-applying approximative method

Gravity is the main physical law plants grow under. The tree shape is fully specified by its organ parts and every part changes the whole structure as a system. The calculation of a curve formed by an axis bending can be done with knowledge of the axis profile (length, diameter development), material characteristics (Young's modulus, density distribution), the initial angle of deflection (with respect to the vertical) and finally the direction and the absolute value of the force that is applied. Formula and solution to this problem is given by the theory of elasticity. A problem to this approach is the fact that some tree characteristics are not reckoned, for example the growth process collides with the mechanical law of mass conservation, or the habit of trees to withstand high deformations by an adaptation of their growth process is neglected. Furthermore wood can not be seen as homogeneous. The complexity of the growth process does not only depend on the evolution of length and diameter but also on the accumulation of tree year rings in the growth process that support a construction differing from a simple homogeneous material distribution. Thus the real shape of a bending branch is more complex than the pure flexion that occurs by the straightforward application of force to the end of a homogeneous cylinder-shaped object. Another

phenomenon that is not considered by this approach is the so called orthotropy. It describes the tendency of a branch to invert the bending against the gravity direction to counteract this gravity. It is done by changing the growth direction and by the formation of reaction wood. To do a correct simulation it needs to pay attention to these processes.
In this section an approach for the deformation of conic beams is proposed. Wood is considered as homogeneous. As a contribution to the mentioned tree reaction phenomena the literature from [3] proposes an orthotropy parameter which will be implemented to this approach in order to cover it as well.

Model of pure beam bending
Elementary physical sizes for the model are: **E**, the Young's modulus as a material constant for the elasticity of a material. **I(s)**, the second moment of area for a beam section. **r(s)**, the radius of curvature at a position on the beam. If $r(s) = \infty$ the beam does not bend. For a circular section it holds true that

$$I(s) = \pi \frac{R_s^4}{4}. \tag{2.1}$$

Where R_s is the radius of the beam at a distance of s from the beginning. The theory of elasticity also delivers the relation

$$\frac{1}{r} = \frac{d\theta}{ds}. \tag{2.2}$$

That means the curvature is equal to the rate of change of the flexion angle θ. Besides there is a relation between the moment of force, the second moment of area and the rate of change of the flexion angle given by the formula

$$M = EI\frac{d\theta}{ds}. \tag{2.3}$$

This equation is the link between force and the bending of the beam. The notations for the following variables are shown in the figure 2.1.

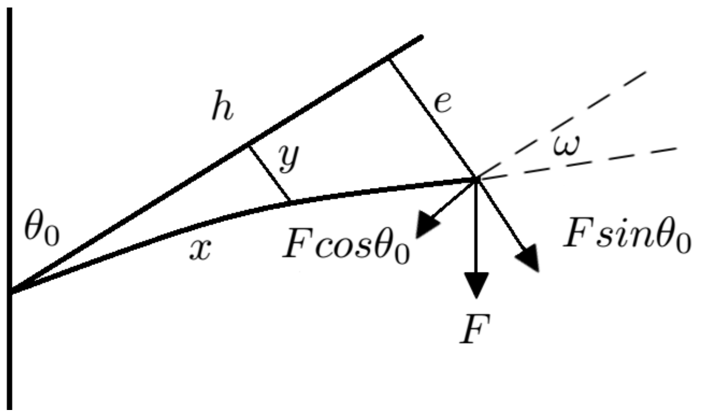

Figure 2.1: Notations of the model

It is supposed that a vertical force F is applied on the tip of the beam, whereby the force vector has the same direction as the gravity. The beam itself is fixed and has an initial angle of deflection which is defined by θ_0 which is measured relative to the vertical axis. The applied force then can be decomposed into its orthogonal and parallel contributions $F \sin\theta_0$ and $F \cos\theta_0$, the first causes pure flexion and the second causes compression. The length of the beam is given by h. First of all the beam is supposed to be a cylinder that means for the second moment of area $I = const.$ ω defines an angle which is an indicator for the overall deformation. The displacement due to the deformation is measured by e. Corresponding to the figure 2.1 the fundamental equations can be written directly considering (2.3) and beam theory as:

$$EI\frac{d\theta}{ds} = F\cos\theta_0\,(e-y) + F\sin\theta_0\,(h-x) \tag{2.4}$$

$$dx = ds\cos\theta \tag{2.5}$$

$$dy = ds\sin\theta \tag{2.6}$$

By the derivation of (2.4) it is obtained that

$$EI\frac{d^2\theta}{ds^2} = F\cos\theta_0\,(-\sin\theta) + F\sin\theta_0\,(-\cos\theta)\,. \tag{2.7}$$

Multiplying both sides of equation (2.7) with $\left(\frac{d\theta}{ds}\right)$ and integrating with θ from 0 to ω leads to

$$\frac{1}{2}EI\left(\frac{d\theta}{ds}\right)^2 = F\cos\theta_0\,(\cos\theta - \cos\omega) + F\sin\theta_0\,(\sin\theta - \sin\omega)\,. \tag{2.8}$$

Introducing the variable K and defining it by writing

$$K^2 := 2\frac{F}{EI} \tag{2.9}$$

makes it possible to denote $d\theta$ as

$$d\theta = K\sqrt{\cos(\theta+\theta_0) - \cos(\omega+\theta_0)}\,ds. \tag{2.10}$$

The curve's shape function which is a result of the flexion of the beam is obtained by the integration of ds:

$$\int ds = h = \int_0^\omega \frac{1}{K\sqrt{cos\,(\theta+\theta_0) - cos\,(\omega+\theta_0)}}\,d\theta. \tag{2.11}$$

Furthermore it is possible to calculate the coordinates x and y at each position s on the beam by

$$X_s = \int_0^s \cos(\theta_0 + \theta)\ ds \tag{2.12}$$

$$Y_s = \int_0^s \sin(\theta_0 + \theta)\ ds \tag{2.13}$$

$$\theta_s = \int_0^s K\ \sqrt{\cos(\theta_0 + \theta) - \cos(\omega + \theta_0)}\, ds. \tag{2.14}$$

Numerically this equation system can be solved without effort. To do this ds would be chosen small enough and the start point would be $\theta = 0$. In the process after a defined number of iterations the curve is drawn step by step. Let x be the step index the sums reveals as:

$$d\theta_x = K\ \sqrt{\cos(\theta_0 + \theta_x) - \cos(\omega + \theta_x)}\, ds \tag{2.15}$$

$$\theta_x = \sum_{i=1}^{x} d\theta_i \tag{2.16}$$

$$X_s = \sum_{i=1}^{s} \cos(\theta_0 + \theta_i)\ ds \tag{2.17}$$

$$Y_s = \sum_{i=1}^{s} \sin(\theta_0 + \theta_i)\ ds. \tag{2.18}$$

Considering this approach the variable ω which was defined as the angle of the overall deformation (see figure 2.1) is still undefined. In addition the beam's possible cone-shape has also not been considered yet. These missing regards will be dealt in the following.

Conicity
Let α be the conicity parameter of the beam. It is defined by

$$\alpha := 1 - \frac{R_{end}}{R_{start}} = \frac{h}{H}. \tag{2.19}$$

Remember that h represents the length of the beam. R_{start} is the radius at the beginning of the beam. R_{end} denotes the radius at the end. It is not allowed that $R_{end} > R_{start}$ and by that $0 \leq \alpha \leq 1$. Thus for $\alpha = 0$ a cylinder-shape, for $\alpha = 1$ a cone and for $0 < \alpha < 1$ a frustum is preserved. The variable H has no further meaning and was introduced for easier use of conicity later. The definition from (2.19) implies the relation that at the distance s:

$$R_s = R_{start}\left(1 - \frac{s}{H}\right) \tag{2.20}$$

and

$$R_{end} = (1-\alpha)\, R_{start}. \tag{2.21}$$

Now when reconsidering the definition from (2.9) and the fact that for a cylinder the second moment of area is

$$I = \pi \frac{R^4}{4}. \tag{2.22}$$

I can be made dependent from the distance s:

$$I_s = \pi \frac{R_s^4}{4}. \tag{2.23}$$

where R_s is defined by (2.20). So the parameter K becomes s-dependent and so

$$K_s = \sqrt{2 \frac{F}{E\pi \frac{R_{start}^4 \left(1-\frac{s}{H}\right)^4}{4}}} = \frac{K_0}{\left(1-\frac{s}{H}\right)^2}. \tag{2.24}$$

By that equations (2.10), (2.11) and (2.14) become

$$K_0 \frac{ds}{\left(1-\frac{s}{H}\right)^2} = \frac{d\theta}{\sqrt{\cos(\theta+\theta_0) - \cos(\omega+\theta_0)}}, \tag{2.25}$$

$$\frac{h}{1-\alpha} = \int_0^\omega \frac{1}{K\sqrt{\cos(\theta+\theta_0) - \cos(\omega+\theta_0)}}\, d\theta. \tag{2.26}$$

and

$$\theta_s = \int_1^s \frac{K_0}{\left(1-\frac{s}{H}\right)^2} \sqrt{\cos(\theta_0+\theta) - \cos(\omega+\theta)}\, ds. \tag{2.27}$$

Calculating the angle of the overall deformation ω

Supposing the angle θ remains small enough, the approximations

$$\sin\theta \approx \theta \tag{2.28}$$

and

$$\cos\theta \approx 1 - \frac{\theta}{2} \tag{2.29}$$

make it possible to write equation (2.11) as

$$h = \int_0^\omega \frac{1}{K\sqrt{A + B\theta + C\theta^2}}\, d\theta. \tag{2.30}$$

The Taylor series of $\cos(\theta+\theta_0) - \cos(\omega+\theta_0)$ is used to compute the parameters A, B and C. The integration of (2.30) finally leads to the solution

$$\omega = \frac{\sin\theta_0\left(1-\cos\left(Kh\sqrt{\cos\theta_0}\right)\right)}{\cos\theta_0\cos\left(Kh\sqrt{\cos\theta_0}\right)}. \tag{2.31}$$

Nevertheless the formula (2.31) contains one problem: the argument $\cos\theta_0$ for the square root function could be less than zero that means for $\theta_0 > \frac{\pi}{2}$ it returns $\cos\theta_0 < 0$. This is a problem for most compilers. But it is known that the following holds true:

$$\underbrace{\cos\left(i * Kh\sqrt{|\cos\theta_0|}\right)}_{\in\mathbb{R}} = \frac{\exp\left(-Kh\sqrt{\cos\theta_0}\right)+\exp\left(Kh\sqrt{\cos\theta_0}\right)}{2}. \tag{2.32}$$

Therefore in such case the $\cos(...)$ expression in equation (2.31) must be replaced by the right term of formula (2.32). In terms of accuracy the calculation of ω by equation (2.31) is very good for angles $< 45°$. Beyond that ω must be calculated numerically by using equation (2.11).

This calculation for ω neglects the possibility of conicity, but it is very simple to adapt it as well. The only variable that needs to be replaced is h. Like I in equation (2.23) K must be also dependent form s. This leads to:

$$K_s = \frac{K_0}{\left(1-\frac{s}{H}\right)^2}. \tag{2.33}$$

The variable h through this adaptation then needs to be replaced by $\frac{h}{1-\alpha}$.

2.2 Incremental finite element method

The use of finite elements for the analysis of mechanical structures is most common. Living trees can also be seen as mechanical structures with their internal stresses caused by self-weight and internal growth. Both processes constitute a cumulative process of internal stresses and deformation of tree parts. This correlation has been variously examined in the theoretical field for example by [6] [7] [8]. Experimental test have been carried out for example by [9] [10]. In this approach not only the real physics based mechanical evolution by using finite elements has been accomplished but also the tree reaction is considered concerning the fact that under its load history a branch adapts its growth process by changing its growth direction or by the building of reaction wood. In general the numerical finite element method itself does not contain an adaption to the biological behavior of living trees. This approach takes into consideration the growth-induced progressive volume extension process with regards to primary (elongation) and secondary (thickening) growth. The basic shape is represented by a Bernoulli 3D multi-layer beam element. That means the branch's shape is again like in the force-applying method

approximated by cylinder-shaped objects. The multi-layer context is related to the incremental formulation that means the beam is divided into layers that represent the tree's year rings. This convention furthermore makes it possible to set different properties for each year ring as they generally do not hold the same material characteristics. Furthermore there is a possibility to add weight-contributions from entities like snow, wind, needles or leaves to the branch. Another factor that was discretized for the incremental formulation is the time. At any time step a new domain which fulfills the equilibrium equations and defines the starting point for the next development step is reached.
For this work the internal stresses are not examined. The main focus is on the deformation process caused by self-weight and reaction processes.

Theoretical formulation
For the following notation capital letters indicate matrices, minuscule letters represent vectors or constants. First of all the definition base for the finite element formulation is the principle of virtual work (see [11]). The field for the displacement and its discretization is formed by the choice of control points or nodes and writing it in the form

$$U = Nq. \tag{2.34}$$

The column vector U contains U_x, U_y and U_z, the three components of displacement relative to the global referential axis. The unknown nodal displacements are represented by q. N is the matrix which holds the shape functions that are polynomials and the degree is determined by the number of nodes. According to the theory of the virtual work principle the strain field is expressed by

$$\epsilon = Bq. \tag{2.35}$$

The matrix B is the derivative of matrix N. Owing the definition of matrix B and discretizing the time in time slots defined by $\Delta t_n = t_{n-1} - t_n$, a new domain Ω_n must satisfy the equilibrium requirement after any time t_n. This can be expressed by

$$\int_{\Omega_n} B^T \sigma_n d\Omega = f_n. \tag{2.36}$$

Where the column vector σ_n is holding the stress field components. A main vector appearing in the equation (2.36) is f_n. This vector contains the nodal loads originating for example from the self-weight of the branches or additional loads like needles or snow. It represents the load that is applied to the branch at the state Ω_n. The process of the domain evolution is expressed by the relation

$$\Omega_{n+1} = \Omega_n + \Delta\Omega_n. \tag{2.37}$$

This domain evolution process represents the growth of the branch. After every time step Δt_n

wood material is elaborated and contributes to the weight and shape development. So the new equilibrium equation according to (2.36) can be written as

$$\int_{\Omega_{n+1}} B^T \sigma_{n+1} d\Omega = f_{n+1}. \tag{2.38}$$

Considering the requirement of incrementalism the equations (2.38) and (2.36) can be substracted and with respect to the formulation (2.37) it is obtained

$$\int_{\Omega_n} B^T \left(\sigma_{n+1} - \sigma_n\right) d\Omega + \int_{\Delta\Omega_n} B^T \sigma_{n+1} d\Omega = f_{n+1} - f_n. \tag{2.39}$$

This also makes it appropriate to write

$$\sigma_{n+1} = \sigma_n + \Delta\sigma_n \Leftrightarrow \Delta\sigma_n = \sigma_{n+1} - \sigma_n \tag{2.40}$$

and

$$f_{n+1} = f_n + \Delta f_n \Leftrightarrow \Delta f_n = f_{n+1} - f_n. \tag{2.41}$$

Furthermore considering for domain $\Delta\Omega_n$ that $\sigma_n = 0$ and $\sigma_{n+1} = \Delta\sigma_n$, because $\Delta\Omega_n$ did not exist before time t_{n+1}, it is possible to formulate the equilibrium equation after elapsed time Δt_n in the form

$$\int_{\Omega_n} B^T \Delta\sigma_n d\Omega + \int_{\Delta\Omega_n} B^T \Delta\sigma_n d\Omega = \Delta f_n. \tag{2.42}$$

Looking again at equation (2.35) which was the relation between strain and displacement it is possible to write

$$\Delta\epsilon_n = B\Delta q_n. \tag{2.43}$$

The strain then must be divided into its components of elastic and maturation strain

$$\Delta\epsilon_n = \Delta\epsilon_n^{el} + \Delta\epsilon_n^{MS}. \tag{2.44}$$

Maturation strains come from the building of new wood cells while growing. Referring to the concept of time discretization and domain evolution maturation strain is defined for the domains by

$$\Delta\epsilon_n^{MS} = \alpha_n \quad \text{for } \Delta\Omega_n \tag{2.45}$$

and

$$\Delta\epsilon_n^{MS} = 0 \quad \text{for } \Omega_n. \tag{2.46}$$

Let D be the material stiffness matrix at the current material point then using (2.44),(2.45) and (2.46) leads to

$$\Delta\sigma_n = D\Delta\epsilon_n - D\alpha_n \quad \text{for } \Delta\Omega_n \tag{2.47}$$

and

$$\Delta\sigma_n = D\Delta\epsilon_n \quad \text{for } \Omega_n. \tag{2.48}$$

Substituting the strain increment in equation (2.43) by the one from equation (2.47) and (2.48) it is possible to subsume the incremental formulation from equation (2.42) to the linear system

$$K_n \Delta q_n = \Delta F_n. \tag{2.49}$$

K_n denotes the global stiffness matrix, the vector ΔF_n is the nodal load vector as a sum of elementary loads issuing from the self-weight increment Δf_n and maturation strain $\Delta\Lambda_n$. $\Delta\Lambda_n$ and K_n are defined by

$$\Delta\Lambda_n = \int_{\Delta\Omega_n} B^T D\alpha_n d\Omega \tag{2.50}$$

and

$$K_n = \int_{\Omega_n} B^T DBd\Omega + \int_{\Delta\Omega_n} B^T DBd\Omega. \tag{2.51}$$

However, this finite element approach needs a mesh, a geometrical domain that is discretized by simply shaped elements. In this method an element-wise subdivision has been chosen. That implies every element is connected to an elementary stiffness matrix. Formally it is defined by

$$k_n^e = \int_{\Omega_n^e} (B^e)^T D^e B^e d\Omega + \int_{\Delta\Omega_n^e} (B^e)^T D^e B^e d\Omega. \tag{2.52}$$

That however implies that the global stiffness matrix and the global load vector are compounded from their elementary relatives. This assembly procedure will be shown at the end. Modeling this mesh described above using beam elements as the smallest unit leads to the following derivation of the elementary units:
Let the definition of B^e be

$$B^e := Gb^e. \tag{2.53}$$

Beam elements e are specified by their length and their cross-section area. Let them be denoted by L^e and S_n^e. Then the elementary stiffness matrix becomes

$$k_n^e = \int_{L^e} \int_{S_n^e + \Delta S_n^e} (B^e)^T D^e B^e d\Omega. \tag{2.54}$$

Let $p(e)$ be the date of the creation of an element e, then the definition

$$S_n^e := \sum_{j=p(e)}^{n} \Delta S_j^e \tag{2.55}$$

denotes the cross-section area of a beam element e. This formulation is a tribute to the secondary growth. Then with the help of formula (2.55) the equation (2.54) can be written as

$$k_n^e = \int_{L^e} (b^e)^T \left[\sum_{j=p(e)}^{n} \int_{\Delta S_j^e} G^T D_j^e G dS \right] b^e dx. \tag{2.56}$$

D_j^e is the material stiffness matrix in the layer evolved at time t_j. The terms for the nodal loads for self-weight and maturation strain are given by

$$\Delta f_n^e = \int_{L^e} (N^e)^T \Delta p_n^e dx_n^e \quad \text{with } \Delta p_n^e = \begin{pmatrix} \int_{\Delta S_n^e} P_x^e dS & \int_{\Delta S_n^e} P_y^e dS & \int_{\Delta S_n^e} P_z^e dS & 0 \end{pmatrix}^T \tag{2.57}$$

and

$$\Delta\Lambda_n^e = \int_{L^e} (b^e)^T \left[\int_{\Delta S_j^e} G^T D_n^e \alpha_n^e dS \right] dx. \tag{2.58}$$

The vector Δp_n^e is holding the distributed loads. Remembering equations (2.45) and (2.46) the element-wise maturation strain increments can be defined as

$$(\Delta\epsilon_n^e)^{MS} = \alpha_n^e \quad \text{for } \Delta\Omega_n \tag{2.59}$$

and

$$(\Delta\epsilon_n^e)^{MS} = 0 \quad \text{for } \Omega_n. \tag{2.60}$$

By assuming that no transverse shear is induced by the maturation strain process, we get

$$\alpha_n^e = \begin{pmatrix} \alpha_{L,n}^e & 0 & 0 \end{pmatrix}^T. \tag{2.61}$$

The maturation strain process furthermore is modulated with a hoop variation done by the trigonometric cosine function

$$\alpha_{L,n}^e = a_n^e + \frac{\varsigma_n^e}{2} (b_n^e - a_n^e) \left(1 + \cos\left(\theta - \psi_n^e\right)\right). \tag{2.62}$$

Where $\theta \in [0, 2\pi]$ represents the hoop variation variable. It is also defined that

$$\psi_n^e = \begin{cases} \varphi_n^e + \pi & \text{if } b_n^e \geq 0 \\ \varphi_n^e & \text{if } b_n^e < 0 \end{cases}. \tag{2.63}$$

The variables a_n^e and b_n^e define the extreme values of maturation strain. Equation (2.63) contributes to the fact that trees follow different strategies for the straightening up reaction: reaction wood can be formed at the upper or lower part of the stem depending on the extreme values of maturation strain. These values can be found in literature for different trees (see [12]). φ_n^e is the angle for the straightening up reaction measured with respect to the local axis attached to the beam element.

In order to control the sensitivity of this reaction process the variable ς_n^e is introduced and it is defined that $\varsigma_n^e \in [0,1] \subseteq \mathbb{R}$. So for $\varsigma_n^e = 0$ no reaction and for $\varsigma_n^e = 1$ the full reaction takes place. The previous considerations dealt with the theoretical expressions for the vectors and matrices involved in this finite element approach. The following paragraph will give the analytical expressions for them.

Analytical matrix and vector definitions

Starting with equation (2.34) the main vectors and matrices are analytically defined by

$$u^e = \begin{pmatrix} u_x^e & u_y^e & u_z^e & \omega^e \end{pmatrix}^T, \tag{2.64}$$

$$q^e = \begin{pmatrix} q_1 & q_2 & \dots & q_{12} \end{pmatrix}^T \tag{2.65}$$

and

$$N^e = \begin{pmatrix} L_1^e & 0 & 0 & 0 & 0 & 0 & L_2^e & 0 & 0 & 0 & 0 & 0 \\ 0 & H_1^e & 0 & 0 & 0 & H_2^e & 0 & H_3^e & 0 & 0 & 0 & H_4^e \\ 0 & 0 & H_1 & 0 & -H_2^e & 0 & 0 & 0 & H_3^e & 0 & -H_4^e & 0 \\ 0 & 0 & 0 & L_1^e & 0 & 0 & 0 & 0 & 0 & L_2^e & 0 & 0 \end{pmatrix}, \tag{2.66}$$

where

$$\begin{aligned} L_1^e = 1 - \xi, \;\; L_2^e = \xi, \;\; H_1^e = 1 - 3\xi^2 + 2\xi^3, \;\; H_2^e = L^e \xi \left(\xi - 1\right)^2 \\ H_3^e = 3\xi^2 - 2\xi^3, \;\; H_4^e = L^e \xi^2 \left(\xi - 1\right), \end{aligned} \tag{2.67}$$

and

$$\xi = \frac{x}{L^e}. \tag{2.68}$$

For the decomposition in equation (2.53) it holds true that

$$b^e(x) = \begin{pmatrix} \frac{dL_1^e}{dx} & 0 & 0 & 0 & 0 & 0 & \frac{dL_2^e}{dx} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{d^2H_1^e}{dx^2} & 0 & 0 & 0 & \frac{d^2H_2^e}{dx^2} & 0 & \frac{d^2H_3^e}{dx^2} & 0 & 0 & 0 & \frac{d^2H_4^e}{dx^2} \\ 0 & 0 & -\frac{d^2H_1^e}{dx^2} & 0 & \frac{d^2H_2^e}{dx^2} & 0 & 0 & 0 & -\frac{d^2H_3^e}{dx^2} & 0 & \frac{d^2H_4^e}{dx^2} & 0 \\ 0 & 0 & 0 & \frac{dL_1^e}{dx} & 0 & 0 & 0 & 0 & 0 & \frac{dL_2^e}{dx} & 0 & 0 \end{pmatrix} \tag{2.69}$$

and

$$G(y,z) = \begin{pmatrix} 1 & -y & z & 0 \\ 0 & 0 & 0 & -z \\ 0 & 0 & 0 & y \end{pmatrix}. \tag{2.70}$$

Let $t_{p(e)}$ be the date of creation of an element e. Furthermore let $r_{j.ext}^e$ and $r_{j.int}^e$ denote the outer and inner radii of the tree layers that have been elaborated during the time Δt_j. Also let E_j^e and

G_j^e be the longitudinal Young's modulus and the shear modulus of the wood elaborated during the time Δt_j. Then the elementary stiffness matrix (see equation (2.56)) is defined by

$$k_n^e = \begin{pmatrix} a & 0 & 0 & 0 & 0 & 0 & -a & 0 & 0 & 0 & 0 & 0 \\ 0 & b & 0 & 0 & 0 & c & 0 & -b & 0 & 0 & 0 & c \\ 0 & 0 & d & 0 & -e & 0 & 0 & 0 & -d & 0 & -e & 0 \\ 0 & 0 & 0 & f & 0 & 0 & 0 & 0 & 0 & -f & 0 & 0 \\ 0 & 0 & -e & 0 & j & 0 & 0 & 0 & 0 & 0 & h & 0 \\ 0 & c & 0 & 0 & 0 & i & 0 & -c & 0 & 0 & 0 & g \\ -a & 0 & 0 & 0 & 0 & 0 & a & 0 & 0 & 0 & 0 & 0 \\ 0 & -b & 0 & 0 & 0 & -c & 0 & b & 0 & 0 & 0 & -c \\ 0 & 0 & -d & 0 & e & 0 & 0 & 0 & d & 0 & e & 0 \\ 0 & 0 & 0 & -f & 0 & 0 & 0 & 0 & 0 & f & 0 & 0 \\ 0 & 0 & -e & 0 & h & 0 & 0 & 0 & e & 0 & j & 0 \\ 0 & c & 0 & 0 & 0 & g & 0 & -c & 0 & 0 & 0 & i \end{pmatrix}, \tag{2.71}$$

where

$$\begin{aligned} a &= \frac{(ES)_n^e}{L^e},\ b = \frac{12(EI_z)_n^e}{(L^e)^3},\ c = \frac{6(EI_z)_n^e}{(L^e)^2},\ d = \frac{12(EI_y)_n^e}{(L^e)^3},\ e = \frac{6(EI_y)_n^e}{(L^e)^2}, \\ f &= \frac{(GJ)_n^e}{L^e},\ g = \frac{2(EI_z)_n^e}{L^e},\ h = \frac{2(EI_y)_n^e}{L^e},\ i = \frac{4(EI_z)_n^e}{L^e},\ j = \frac{4(EI_y)_n^e}{L^e}, \end{aligned} \tag{2.72}$$

$$(ES)_n^e = \pi \sum_{j=p(e)}^{n} E_j^e \left(\left(r_{j.ext}^e\right)^2 - \left(r_{j.int}^e\right)^2 \right), \tag{2.73}$$

$$(EI_y)_n^e = (EI_z)_n^e = \frac{\pi}{4} \sum_{j=p(e)}^{n} E_j^e \left(\left(r_{j.ext}^e\right)^4 - \left(r_{j.int}^e\right)^4 \right), \tag{2.74}$$

and

$$(GJ)_n^e = \frac{\pi}{2} \sum_{j=p(e)}^{n} G_j^e \left(\left(r_{j.ext}^e\right)^4 - \left(r_{j.int}^e\right)^4 \right). \tag{2.75}$$

Let $\begin{pmatrix} Z_x^e & Z_y^e & Z_z^e \end{pmatrix}^T$ be the vertical unit vector that is attached to the beam indicating its local axis. Furthermore let ρ_n^e be the density of the wood elaborated during the time Δt_n. Let the gravity constant be given by g. Then the nodal load increment vector (equation (see 2.57)) is defined by

$$\Delta f_n^e = -\frac{\rho_n^e g L^e \Delta S_n^e}{12} \begin{pmatrix} 6Z_x & 6Z_y & 6Z_z & 0 & -L^e Z_z & L^e Z_y & 6Z_x & 6Z_y & 6Z_z & 0 & L^e Z_z & -L^e Z_y \end{pmatrix}^T, \tag{2.76}$$

where

$$\Delta S_n^e = \pi \left(\left(r_{j.ext}^e\right)^2 - \left(r_{j.int}^e\right)^2 \right). \tag{2.77}$$

Reconsidering equations (2.58) and (2.62) the analytical expression for the maturation strain increment is given by

$$\Delta\Lambda_n^e = \begin{pmatrix} -N^e & 0 & 0 & -M_x^e & -M_y^e & -M_z^e & N^e & 0 & 0 & M_x^e & M_y^e & M_z^e \end{pmatrix}, \tag{2.78}$$

where

$$N^e = \pi E_n^e \left(\left(r_{j.ext}^e\right)^2 - \left(r_{j.int}^e\right)^2 \right) \left(a_n^e + \frac{\varsigma_n^e}{2} \left(b_n^e - a_n^e\right) \right), \tag{2.79}$$

$$M_x^e = 0, \tag{2.80}$$

$$M_y^e = -\frac{\pi}{6} E_n^e \varsigma_n^e \left(b_n^e - a_n^e\right) \left(\left(r_{j.ext}^e\right)^3 - \left(r_{j.int}^e\right)^3 \right) \cos\psi_n^e, \tag{2.81}$$

and

$$M_z^e = \frac{\pi}{6} E_n^e \varsigma_n^e \left(b_n^e - a_n^e\right) \left(\left(r_{j.ext}^e\right)^3 - \left(r_{j.int}^e\right)^3 \right) \sin\psi_n^e. \tag{2.82}$$

Global assembly of the elementary vectors and matrices

The elementary vectors and matrices given in their analytical form (equations (2.71), (2.76) and (2.78)) are given with respect to the local referential axis that is attached to the beam element. But for the global formulation of the linear equation system formed by formula (2.49) they are needed in their global form. To achieve this a change of the basis matrix (rotation matrix) is necessary to transform the local referential to the global coordinate system. This matrix named Q^e is defined by the relation

$$\Delta q^e = Q^e (\Delta q^e)_{global} \tag{2.83}$$

The vector q^e is the displacement vector for one beam element e. This vector for every node contains six displacement components according to the following notation (figure taken from [5]).

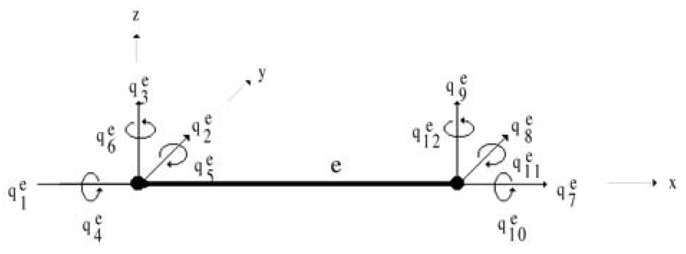

Figure 2.2: Beam element displacements

Let the number of beam elements (finite elements) be denoted by N. Then the vector that contains all displacement components is given by

$$q = \begin{pmatrix} q_1 & q_2 & \dots & q_{6(N+1)} \end{pmatrix}^T . \tag{2.84}$$

Every beam element is attached to its own coordinate system. Let this basis be denoted by

$$R^e = \begin{pmatrix} \vec{x_e} & \vec{y_e} & \vec{z_e} \end{pmatrix} = \begin{pmatrix} x_e^x & y_e^x & z_e^x \\ x_e^y & y_e^y & z_e^y \\ x_e^z & y_e^z & z_e^z \end{pmatrix} . \tag{2.85}$$

Then the change of basis matrix is defined by

$$Q^e = \begin{pmatrix} R^e & 0 & 0 & 0 \\ 0 & R^e & 0 & 0 \\ 0 & 0 & R^e & 0 \\ 0 & 0 & 0 & R^e \end{pmatrix} = \begin{pmatrix} x_e^x & y_e^x & z_e^x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ x_e^y & y_e^y & z_e^y & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ x_e^z & y_e^z & z_e^z & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_e^x & y_e^x & z_e^x & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_e^y & y_e^y & z_e^y & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_e^z & y_e^z & z_e^z & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & x_e^x & y_e^x & z_e^x & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & x_e^y & y_e^y & z_e^y & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & x_e^z & y_e^z & z_e^z & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_e^x & y_e^x & z_e^x \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_e^y & y_e^y & z_e^y \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_e^z & y_e^z & z_e^z \end{pmatrix} . \tag{2.86}$$

It is supposed that the basis vectors are orthogonal in terms of the standard scalar product. That implies that Q^e is an orthogonal matrix and it holds true that $(Q^e)^T = (Q^e)^{-1}$. Knowing the matrix Q^e a definition for all the global elementary units can be given as

$$(f^e)_{global} = (Q^e)^T f^e, \tag{2.87}$$

$$(\Lambda^e)_{global} = (Q^e)^T \Lambda^e, \tag{2.88}$$

and

$$(k^e)_{global} = (Q^e)^T k^e Q^e. \tag{2.89}$$

To link all beam elements together an interconnection matrix has to be defined due to correct placement of the element-wise contribution in the global stiffness matrix for the final linear equation system. Such matrix denoted by C^e can be gained by the extraction matrix for getting the beam elements' displacements from the vector (2.84) by

$$q^e = C^e q. \tag{2.90}$$

This matrix fulfills $C^e \in \mathbb{R}^{12\times 6(N+1)}$. For the whole assembly procedure it is supposed that all beam elements are connected one after the other with the meaning $e_1 - e_2 - \ldots - e_N$. This assumption leads to the analytical definition of C^e

$$C^{e_i} = \begin{pmatrix} 0 & \ldots & 0 & 1 & 0 & \ldots & 0 & \ldots & 0 \\ \vdots & \ddots & \vdots & \ddots & \ddots & \ddots & \vdots & \ddots & \vdots \\ \underbrace{0 \quad \ldots \quad 0}_{6(i-1)\text{ zeros}} & & & \ldots & 0 & 1 & \underbrace{0 \quad \ldots \quad 0}_{6(N-1)\text{ zeros}} & & \end{pmatrix}. \tag{2.91}$$

Finally after every time step t_n the global vectors and matrices are assembled by

$$K_n = \sum_{i=1}^{n} (C^{e_i})^T (k_n^{e_i})_{global} C^{e_i} \tag{2.92}$$

and

$$\Delta F_n = \sum_{i=1}^{n} (\Delta f_n^{e_i})_{global} C^{e_i} + \sum_{i=1}^{n} (\Delta \Lambda_n^{e_i})_{global} C^{e_i}. \tag{2.93}$$

So the unknown nodal displacement increments Δq_n which are the solutions to the finite element formulation can be obtained by solving the linear equation system

$$K_n \Delta q_n = \Delta F_n \tag{2.94}$$

after every elapsed time step.

Chapter 3

Implementation

In this chapter the two methods presented in chapter 2 are implemented in XL. References to the theoretical basis are given. The necessary internal constructions (for example the class for a finite element) for the code that are needed for the implementation are carried out. Both implementations for the two methods furthermore need some constructive assumptions. These are explained ahead of every implementation. As their result of calculations both algorithms return arrays of rotation angles which define the curvature of the branch. The main advantage of this convention is that in contrast to 2D-coordinates rotation angles are independent of any choice of coordinate system. By that a transformation easily can be performed by just adding or subtracting angles to every entry in the array.

3.1 Force-applying approximative method

The following parameters for this method can be set:

- *int disc* :
  This parameter sets the number of rotation angles (and thus subsections) calculated by the model. Because of that, the parameter also is equivalent to the number of discretization steps.
- *double t0* :
  This parameter sets the initial angle of deflection. The angle, in degrees, is measured relatively to the vertical z-axis. The definition range for this parameter is $0.0 \ldots 180.0$.
- *double K0* :
  This non-negative parameter is the model's material constant. It is related to the physical term elastic modulus, which defines a material constant for the resistance of a material against bending deformation.

- *double coni* :
  This parameter sets the conicity of the branch. The definition range is $0.0 \dots 1.0$. 0.0 means the branch has constant diameter, hence it has cylinder shape. 1.0 means it has zero diameter at the tip and therefore a total cone shape. For values $0.0 < coni < 1.0$, the branch is a frustum.

- *double p* :
  This parameter defines the relative point where the branch starts to bend upwards with the meaning of phototropism. The definition range is $0.0 \dots 1.0$, with 0.0 referring to the base and 1.0 to the tip.

Source code:

```
double[] GetRotationAnglesApprox(int disc, double t0, double K0, double coni, double p)
    {

   double res[] = new double[disc+1];
   double dx = (double) disc / 10.0;
   double h = (double) disc / (1 - coni);

   //constants
   double rad = 180.0/Math.PI;
   double t0_rad = t0/rad;
   double cost0 = Math.cos(t0_rad);
   double sint0 = Math.sin(t0_rad);
   double eps = 0.17;  // corresp. to 10 degree

   //variables
   double x, t, w=-1.0, dt, t1;

   if (coni == 0.0) {
      if (cost0 > 0.0) {
         x = K0 * h * Math.sqrt(cost0);
         w = sint0 * (1.0 - Math.cos(x))/(cost0 * Math.cos(x));
      } else {
         x = K0 * h * Math.sqrt(Math.abs(cost0));
         w = sint0 * (1.0 - (Math.exp(-x) + Math.exp(x))/2.0) / (cost0 * (Math.exp(-x) +
              Math.exp(x))/2.0);
      }
   }

   if (coni > 0.0 || K0 * h * Math.sqrt(cost0) * 2.0/Math.PI > 0.8 || Math.abs(w) > 0.8
       || w < 0.0) {

      double a = 0.0;
      double b = Math.PI - t0_rad;

32       do {
33          w = (a+b)/2.0;
34          t = 0.0; x = 1.0;
35
36          do {
37             dt = dx * Math.sqrt(2.0) * K0 * Math.sqrt(Math.abs(Math.cos(t0_rad + t) -
                   Math.cos(t0_rad + w)));
38             t += dt;
39             x += dx;
40          } while (t < w);
41
42          if (x < h - dx)
43             a = w;
44          else
45             b = w;
46        } while (b-a > eps);
47        w = (a + b)/2.0;
48    }
49
50    dx = 1.0;
51    t = 0.0;
52    t1 = 0.0;
53
54    res[0] = t0;
55
56    for (int i = 0; i < disc; i++) {
57
58       if (t < w) {
59          t1 = (K0/(1.0 - (coni*(i+1))/disc)**2.0) * Math.sqrt(2.0 * (Math.cos(t + t0_rad
                ) - Math.cos(w + t0_rad))) * dx;
60          if (i > (double) disc * p)
61             t1 = -t1;
62          t += t1;
63       }
64       res[i+1] = t1 * rad;
65    }
66    return res;
67 }
```

In line 1 the definition of the function $GetRotationAnglesApprox(\ldots)$ takes place. It returns an array of doubles with the rotation angles that specify the curvature of the branch. In the argument list the parameters described above the source code can be found again. $disc$ is related to the variable h which was the length of the branch. The double $t0$ is the equivalent to the initial angle of deflection θ_0. $K0$ represents the material constant K_0. The double $coni$ is the equivalent to the conicity parameter α (see definition (2.19)). As an addition to the basic formulation from chapter 2 a phototropism parameter p is introduced (referring to [3]).

An array for picking up the solution, the rotation angles, can be found in line 3. This array's length is given by $disc + 1$ because as a convention the initial deflection angle is also picked up at the first array index. For its first usage the variable dx (line 4) is needed as a step width for the calculation of the overall deformation ω (in the code denoted by w) in the case that for the first attempt $\omega \geq 45°$ (see statements after equation (2.32)). Since conicity is considered the internal definition for h is not equivalent to $disc$. That is why in line 5 h is defined as the left side of the equation (2.26). After that the constants for the calculation are set in order to prevent unnecessary recalculations, for example in the argument list the parameter θ_0 is given in degrees but for all internal trigonometric functions the unit must be radian (line 8-12). The double eps, defined in line 12, constitutes the accuracy for the calculation of ω (see line 46). In line 15 space for a bunch of working variables that are needed for the iterative procedure is reserved.

After that the first calculation of the angle of the overall deformation ω is accomplished (line 17-25). Since the value from equation (2.31) is only valid for a total cylinder shape it merely makes sense to perform the calculations when it holds true that $coni = 0$ (line 17). In order to respond to the problem with imaginary numbers stated in formula (2.32) the two different calculation ways for ω are carried out (line 19-20 and line 22-23).

The next section (line 27-48) is about the numerical calculation of ω for the case when the accuracy of equation (2.31) is not enough or there is no other way than doing so. For the case that $coni > 0$ (line 27) the shape is not a cylinder. Thus ω must be calculated like this. The accuracy condition $\omega < 45°$ (see after equation (2.32)) can be found in the second and third term of the boolean construction (line 27). Furthermore for negative angles of the overall deformation it is also mandatory. This method in line 27-48 is a search algorithm to find the solution for ω. It is known that $\omega \in [0, \pi - \theta_0]$. The variables for this seek area ($[a, b]$) are defined in line 29-30. By successively dividing the seek area (line 33 and line 42-46) ω is found when the defined accuracy is reached (line 46).

Having calculated ω it is now possible to calculate the branch's curvature: the array of rotation angles. This is done in line 50-65. The iteration variables are reset in line 50-52. To achieve sufficient precision the value of dx (line 50) must be chosen small enough in contrast to the dx value for the calculation of ω where it is permitted to be significantly higher. Besides in line 54 the first rotation angle is fixed to the initial angle of deflection θ_0 as mentioned at the beginning.

The iteration loop in line 56-65 is the numerical calculation of equation (2.15) and (2.27). Numerical integration is accomplished in the most simple way by the use of a Riemann sum. The phototropism parameter is responsible for the branch to bend upward at a certain relative point. This behavior is ensured in line 60-61 by changing the growth direction after reaching the position $disc * p$ which is the inflection point. All gained rotation angles, measured in degrees, are stored to the solution array in line 64. Finally this array of rotation angles is returned in line 66.

3.2 Incremental finite element method

The following parameters for this method can be set:

- *int n_FE* :
  This parameter sets the number of finite elements. Thus it is also the number of discretization steps for the time. Moreover it is the maximum number of tree year rings. The parameter furthermore sets the resolution of the finite mesh. In order to achieve reasonable runtime and sufficient correctness it should hold true that $n\_FE \in [10, 40] \subseteq \mathbb{N}$.
- *int n_GU* :
  This parameter sets the number of growth units. It is synonymous to the extent of interpolation between the displacement solutions of two finite elements. That means the number of finite elements is not inevitably equivalent to the number of growth units. Thus it determines the number of rotation angles in the solution array. The lower limit for this parameter is the number of finite elements: $n\_GU \geq n\_FE$. For the function it is recommended that $n\_GU \in [n\_FE, 1000] \subseteq \mathbb{N}$.
- *double t0* :
  This parameter sets the initial angle of deflection. The angle, in degrees, is measured relatively to the vertical z-axis. The definition range for this parameter is $0.0 \ldots 180.0$.
- *double rho* :
  This parameter sets the material density. For the calculations a homogeneous density distribution is supposed. The unit is $\frac{kg}{m^3}$.
- *double E* :
  This parameter sets the Young's modulus. It is the material constant of the branch, which defines the resistance of a material against bending deformation. Again a homogeneous distribution of the Young's modulus is supposed. The unit is MPa.
- *double grav* :
  This parameter sets the gravity constant. The unit is $\frac{m}{s^2}$.
- *double Le* :
  This parameter sets the length of a finite element. Consequently the length of the branch is defined by $n\_FE\ * Le$. The unit is m.
- *double r_start* :
  This parameter sets the initial radius of the branch. The unit is m.

- *double r_end* :
  This parameter sets the end radius of the branch. The unit is m.
- *double sigma* :
  This parameter controls the sensitivity of the reaction wood building process initiated by maturation strain. The definition range is $sigma \in [0,1] \subseteq \mathbb{R}$. For $sigma = 0.0$ no reaction and for $sigma = 1.0$ the full reaction takes place.
- *double phi* :
  This parameter sets the angle for the direction of the reaction process controlled by $sigma$. The angle, in degrees, is measured relatively to the x-y-plane. The definition range for this parameter is $0.0 \ldots 180.0$.
- *double a* :
  This parameter sets the minimum value of maturation strain. The unit is μdef (this is equivalent to $1\,ppm$ that means $1000\mu def = 0.1\%$).
- *double b* :
  This parameter sets the maximum value of maturation strain. The unit is μdef.
- *boolean primaryTropism* :
  This parameter enables the possibility to fix the growth direction of the branch with the angle $t0$. That means every new growth unit is set fixed in its growth direction with the meaning of phototropism.

Source code:

```
const double rad = 180.0/Math.PI;   //radian in degrees
const int MAX_FE_COUNT = 40;        //maximal number of finite elements

//class for one finite elemenent
public class FE {

   double length;           //length of the finite element
   double[] radii;          //array of radii for the wood layers
   double[] densities;      //densities for each wood layer
   double[] young_moduli;   //Young's moduli for the wood layers
   double GlobalRotation;   //angle relative to the global vertical reference
   double sigma;            //growth strategy parameter
   double phi;              //direction of the growth strategy
   double a;                //minimum of the growth stress
   double b;                //maximum of the growth stress
   double grav;             //gravity constant
   double extraLoad;        //extra weight
   int Layer_count = 0;     //number of wood layers

   //constructor
```

```
    public FE(  double length,
                double init_radius,
                double init_density,
                double init_YoungModulus,
                double GlobalRotation,
                double sigma,
                double phi,
                double a,
                double b,
                double grav
                double extraLoad   )
             {

       this.length = length;
       this.GlobalRotation = GlobalRotation;
       this.sigma = sigma;
       this.phi = phi;
       this.a = a;
       this.b = b;
       this.grav = grav;
       this.extraLoad = extraLoad;
       this.radii = new double[MAX_FE_COUNT+1];
       this.densities = new double[MAX_FE_COUNT];
       this.young_moduli = new double[MAX_FE_COUNT];
       //first radius must be 0.0
       this.radii[0] = 0.0;
       this.radii[1] = init_radius;
       this.densities[0] = init_density;
       this.young_moduli[0] = init_YoungModulus;
       this.Layer_count++;
    }

    //add wood layer to the element
    public void addLayer(double thickness, double density, double YoungModulus) {
       radii[Layer_count+1] = radii[Layer_count] + thickness;
       densities[Layer_count] = density;
       young_moduli[Layer_count] = YoungModulus;
       Layer_count++;
    }

    //get StiffnessMatrix for the element
    public RealMatrix StiffnessMatrix() {
       double tmp1 = 0.0;
       double tmp2 = 0.0;
       double tmp3 = 0.0;
       for (int i = 0; i < Layer_count; i++) {
          tmp1 += (young_moduli[i] * (radii[i+1] ** 2 - radii[i] ** 2));
          tmp2 += (young_moduli[i] * (radii[i+1] ** 4 - radii[i] ** 4));
          tmp3 += ((young_moduli[i]/3) * (radii[i+1] ** 4 - radii[i] ** 4));
```

```
      }
      double ES = Math.PI * tmp1;
      double EI = Math.PI * 0.25 * tmp2;
      double GJ = Math.PI * 0.5 * tmp3;

      double var1 = ES/length;
      double var2 = 12 * EI/(length*length*length);
      double var3 = 6 * EI/(length*length);
      double var4 = 12 * EI/(length*length*length);
      double var5 = 6 * EI/(length*length);
      double var6 = GJ/length;
      double var7 = 2 * EI/length;
      double var8 = 2 * EI/length;
      double var9 = 4 * EI/length;
      double var10 = 4 * EI/length;

      double[][] matrix = {    {var1,0,0,0,0,0,-var1,0,0,0,0,0},
                  {0,var2,0,0,0,var3,0,-var2,0,0,0,var3},
                  {0,0,var4,0,-var5,0,0,0,-var4,0,-var5,0},
                  {0,0,0,var6,0,0,0,0,0,-var6,0,0},
                  {0,0,-var5,0,var10,0,0,0,0,0,var8,0},
                  {0,var3,0,0,0,var9,0,-var3,0,0,0,var7},
                  {-var1,0,0,0,0,0,var1,0,0,0,0,0},
                  {0,-var2,0,0,0,-var3,0,var2,0,0,0,-var3},
                  {0,0,-var4,0,var5,0,0,0,var4,0,var5,0},
                  {0,0,0,-var6,0,0,0,0,0,var6,0,0},
                  {0,0,-var5,0,var8,0,0,0,var5,0,var10,0},
                  {0,var3,0,0,0,var7,0,-var3,0,0,0,var9}     };

      RealMatrix Q = ChangeOfBasisMatrix();
      return Q.transpose().multiply(MatrixUtils.createRealMatrix(matrix)).multiply(Q);
   }

   //get the change in nodal loads after primary and secondary growth
   public ArrayRealVector deltaNodalLoads() {

      ArrayRealVector res = new ArrayRealVector(12);
      double r_ext = radii[Layer_count];
      double r_int = radii[Layer_count-1];
      double fac = -densities[Layer_count-1]*grav*length*Math.PI*(r_ext ** 2 - r_int **
          2)/12;
      double Zx = Math.cos(GlobalRotation);
      double Zy = 0;
      double Zz = -Math.sin(GlobalRotation);

      res.setEntry(0, fac*6*Zx);
      res.setEntry(1, fac*6*Zy);
      res.setEntry(2, fac*6*Zz);
      res.setEntry(4, -fac*length*Zz);
      res.setEntry(5, fac*length*Zy);
      res.setEntry(6, fac*6*Zx);
      res.setEntry(7, fac*6*Zy);
      res.setEntry(8, fac*6*Zz);
      res.setEntry(10, fac*length*Zz);
      res.setEntry(11, -fac*length*Zy);

      RealMatrix Q = ChangeOfBasisMatrix();
      return new ArrayRealVector(Q.transpose().operate(res));
   }

   //get the change in maturation strain for the new layer elaborated after primary and
       secondary growth
   public ArrayRealVector deltaMaturationStrain() {

      ArrayRealVector res = new ArrayRealVector(12);
      double r_ext = radii[Layer_count];
      double r_int = radii[Layer_count-1];
      double d = (phi/rad) - GlobalRotation;
      double psi = b < 0 ? d : d + Math.PI;
      double Ne = Math.PI * young_moduli[Layer_count-1] * (r_ext ** 2 - r_int ** 2) * (
          sigma * (a + 0.5 * (b-a)));
      double Mx = 0;
      double My = -Math.PI * young_moduli[Layer_count-1] * sigma * (b-a) * (r_ext ** 3 -
           r_int ** 3) * Math.cos(psi)/6;
      double Mz = Math.PI * young_moduli[Layer_count-1] * sigma * (b-a) * (r_ext ** 3 -
          r_int ** 3) * Math.sin(psi)/6;
      res.setEntry(0, -Ne);
      res.setEntry(3, -Mx);
      res.setEntry(4, -My);
      res.setEntry(5, -Mz);
      res.setEntry(6, Ne);
      res.setEntry(9, Mx);
      res.setEntry(10, My);
      res.setEntry(11, Mz);

      RealMatrix Q = ChangeOfBasisMatrix();
      return new ArrayRealVector(Q.transpose().operate(res));
   }

   //get the change in extra loads
   ArrayRealVector deltaExtraLoad(int FE_count) {

      ArrayRealVector res = new ArrayRealVector(12);

      double Zx = Math.cos(GlobalRotation);
      double Zy = 0;
      double Zz = -Math.sin(GlobalRotation);

      double fac = -grav*extraLoad/(12.0*FE_count);

      res.setEntry(0, fac*6*Zx);
      res.setEntry(1, fac*6*Zy);
      res.setEntry(2, fac*6*Zz);
      res.setEntry(4, fac*-length*Zz);
      res.setEntry(5, fac*length*Zy);
      res.setEntry(6, fac*6*Zx);
      res.setEntry(7, fac*6*Zy);
      res.setEntry(8, fac*6*Zz);
      res.setEntry(10, fac*length*Zz);
      res.setEntry(11, fac*-length*Zy);

      RealMatrix Q = ChangeOfBasisMatrix();
      return new ArrayRealVector(Q.transpose().operate(res));
   }

   //change of basis matrix (rotation matrix), transforms the local referential axis to
       the gloabal referential axis
   public RealMatrix ChangeOfBasisMatrix() {
      double var11 = Math.sin(GlobalRotation);
      double var12 = Math.cos(GlobalRotation);
      double[][] yRotM = { {var12,0,var11}, {0,1,0}, {-var11,0,var12} };

      RealMatrix res = MatrixUtils.createRealMatrix(12,12);
      res.setSubMatrix(yRotM,0,0);
      res.setSubMatrix(yRotM,3,3);
      res.setSubMatrix(yRotM,6,6);
      res.setSubMatrix(yRotM,9,9);

      return res;
   }

   //matrix for connecting the finite elements for the later assembly process
   public RealMatrix getConnexionMatrix(int position, int FE_count) {
      RealMatrix res = MatrixUtils.createRealMatrix(12,6*(FE_count+1));
      for (int i=0; i<12; i++) {
         res.setEntry(i,6*position+i,1.0);
      }
      return res;
   }
}

public double[] getRotationAnglesFE(int n_FE, int n_GU, double t0, double rho, double E,
     double grav, double Le, double r_start, double r_end, double sigma, double phi,
    double a, double b, boolean primaryTropism) {

   //initial finite element list
   FE[] FiniteElements = new FE[n_FE];
```

```
int element_count = 0;

//thickness of new wood layers for secondary growth
double delta_radii = (r_start - r_end) / (n_FE-1);

do {
   //first node of first finite element has fixed rotation (initial angle of
       deflection)
   //all other nodes are linked to the previous finite element (growth unit) or to a
       fixed
   //direction with the meaning of a primary tropism
   double rot;

   if (element_count==0 || primaryTropism)
      rot = t0/rad;
   else
      rot = FiniteElements[element_count-1].GlobalRotation;

   //primary growth
   FiniteElements[element_count] = new FE(Le, r_end, rho, E, rot, sigma, phi, a, b);
   element_count++;

   //secondary growth
   for (int i=0; i<(element_count-1); i++) {
      FiniteElements[i].addLayer(delta_radii,rho,E);
   }

   //elemtary matrices
   RealMatrix GlobalStiffnessMatrix = MatrixUtils.createRealMatrix(6*(element_count
       +1), 6*(element_count+1));
   RealVector NodalLoadsExternal = new ArrayRealVector(6*(element_count+1));
   RealVector NodalLoadsInternal = new ArrayRealVector(6*(element_count+1));
   RealVector NodalLoadsExtra = new ArrayRealVector(6*(element_count+1));

   //assembly
   for (int i = 0; i < element_count; i++) {
      FE el = FiniteElements[i];
      RealMatrix StiffnessMatrix = el.StiffnessMatrix();
      RealMatrix C = el.getConnextionMatrix(i,element_count);

      //assembly of global stiffness matrix
      GlobalStiffnessMatrix = GlobalStiffnessMatrix.add(C.transpose().multiply(
          StiffnessMatrix).multiply(C));

      //assembly of nodal load vector
      double[] loaddata = el.deltaNodalLoads(grav).toArray();
      RealMatrix A = MatrixUtils.createRealMatrix(1,12);
      A.setRow(0,loaddata);
      NodalLoadsExternal = NodalLoadsExternal.add(new ArrayRealVector(A.multiply(C).
          getRow(0)));

      //assembly of maturation strain vector
      loaddata = el.deltaMaturationStrain().toArray();
      A.setRow(0,loaddata);
      NodalLoadsInternal = NodalLoadsInternal.add(new ArrayRealVector(A.multiply(C).
          getRow(0)));

      //assembly of extra load vector
      loaddata = el.deltaExtraLoad().toArray();
      A.setRow(0,loaddata);
      NodalLoadsExtra = NodalLoadsExtra.add(new ArrayRealVector(A.multiply(C).getRow
          (0)));
   }

   //solve the LES
   DecompositionSolver solver = new SingularValueDecomposition(GlobalStiffnessMatrix)
       .getSolver();
   RealVector Solution = solver.solve(NodalLoadsExternal.add(NodalLoadsInternal).add(
       NodalLoadsExtra));

   //apply the displacements to all finite elements
   for (int i = 0; i < element_count; i++) {
      FE el = FiniteElements[i];
      RealVector dqe = el.getConnexionMatrix(i, element_count).operate(Solution);
      double rot_y = dqe.getEntry(4);
      el.GlobalRotation += rot_y;
   }

} while (n_FE != element_count);

//extract rotation angles from finite elements
double[] angles = new double[n_FE];
double[] range = new double[n_FE];

for (int i = 0; i < n_FE; i++) {
   FE el = FiniteElements[i];
   angles[i] = rad * el.GlobalRotation;
   range[i] = ((double)i)/(n_FE-1);
}

//interpolation
AkimaSplineInterpolator Interpolator = new AkimaSplineInterpolator();
PolynomialSplineFunction function = Interpolator.interpolate(range, angles);

//number of growth units not less than number of finite elements
n_GU = n_GU < n_FE ? n_FE : n_GU;

double[] res = new double[n_GU];
```

```
298      double prev = 0.0;
299
300      for (int i = 0; i < n_GU; i++) {
301         double angle = function.value(((double)i)/(n_GU-1));
302         res[i] = angle - prev;
303         prev = angle;
304      }
305
306      return res;
307   }
```

In line 1-2 the definition for the constants that are frequently needed for the code is accomplished. Like stated above the number of finite elements is limited. The constant MAX_FE_COUNT holds this number. It is needed for array definitions in line 42-44.

Every finite element is represented by an own object as an instance of a special class for a finite element. This class is named FE (line 5-177). Every finite element consists of its own properties and elementary vector and matrix definitions that are especially connected to it. Furthermore the tree layer structure and the incremental evolution must be implemented as well. This construction facilitates the overall view and the storage of structural information as well as the assembly of the linear equation system. Since the finite elements are beam elements, for the following these designations will be used synonymously.

In line 7-18 the properties of the class FE are defined. The double $length$ denotes the parameter L^e which was the length of a finite element. Considering the fact that the tree's year rings do not all possess the same properties the arrays in line 8-10 pick up the layer specific properties. For the thickness, an array of doubles (line 8) picks up the radii evolution of the finite element where the thickness of a layer elaborated at the time t_n is given by $radii[n+1] - radii[n]$. The same train of thoughts leads to the arrays for the density and the Young's modulus. The first item in the arrays in line 9-10 holds the properties for the first inner year ring. A main property that is needed for the translation of local to global formulation of elementary vectors (see (2.85)-(2.89)) and matrices is the global orientation angle $GlobalRotation$ (line 11). This one-dimensional value is completely sufficient to determine the local referential axis and by that the change of basis matrix Q^e (see (2.83)). The next properties $sigma$, phi, a, b (line 12-15) are the equivalent to the parameters ς^e, φ^e, a^e and b^e (see (2.62) ff.). They were defined as the variables that control the sensitivity of the reaction wood formation process, the angle for the direction of the reaction process, the minimum and maximum value of maturation strain. The gravity constant can be found in line 16. A possibility to add extra loads like snow or needles to the beam element is given by the variable $extraLoad$ (line 17). After that the total number of wood layers is picked up by the integer $Layer\_count$ (line 18).

The constructor for an object of type FE can be found in line 21-51. By calling with the argument list from line 21-31 it initially creates the first wood layer and sets its common properties (line 34-50). The arrays, sized by the constant MAX_FE_COUNT, for the layer properties are initialized in line 42-44. As a particularity the array for the radii (line 42) contains one item more. This is due to the reason of the differential formulation for the radii evolution. That is why in line 46 the first item in this array must be set to 0.0. Finally after setting all properties the counter for the number of layers can be increased (line 50). For modeling the process of secondary growth the routine $addLayer(..)$ (line 54-59) is responsible. In line 55 the radii thickening is accomplished. After that the density and the Young's modulus of the new developed layer are set. At the end the counter for the number of layers must be increased (line 58).

For the implementation in XL the class definitions for matrices, vectors and operations among them, especially matrix multiplication and solving linear equation systems, are carried out by the use of the "Apache Commons Math 3.6.1 API" library. When creating matrices or vectors with class definitions from this library the entries are initialized with zeros.

The function $StiffnessMatrix()$ (line 62-101) returns the global definition of the elementary stiffness matrix $(k_n^e)_{global}$ according to equation (2.89). First of all it is necessary to construct the local definition of the elementary stiffness matrix considering equations (2.71)-(2.75). This is done in line 63-97. The variables $tmp1$, $tmp2$ and $tmp3$ are used to calculate the sum in the equations (2.73)-(2.75) for $(ES)_n^e$, $(EI_y)_n^e = (EI_z)_n^e$ and $(GJ)_n^e$. The equivalents in line 71-73 are the doubles ES, EI and GJ. The modulus of rigidity G_j^e is set to $\frac{E_j^e}{3}$ after the Poisson's ratio relation between Young's modulus and shear modulus (line 69). In the iteration loop in line 66-70 the iteration variable i runs from 0 to $Layer\_count - 1$. This formulation is equivalent to the one from equations (2.73)-(2.75) where the indices of the sums start at $p(e)$, which indicated the date of creation of an finite element, and run until n, the current time step index from time t_n. Afterwards the entries for the elementary stiffness matrix a, b, c, d, e, f, g, h, i and j according to the equations in (2.72) can be found in the declaration of the variables $var1$, $var2$, $var3$, $var4$, $var5$, $var6$, $var7$, $var8$, $var9$ and $var10$ (line 75-84). The final local elementary stiffness matrix then is defined in line 86-97. This represents the analytic definition referring to equation (2.71).

In line 99 the function that returns the current change of basis matrix $ChangeOfBasisMatrix()$ is called. The result of such call is dependent of the finite element's state of global position. The return value (line 100) is determined like in equation (2.89) and it transforms the local to the global formulation of the elementary stiffness matrix.

The equivalent function for the global nodal loads increment $(\Delta f_n^e)_{global}$ (equation (2.87)) can be found in line 104-127 with $deltaNodalLoads()$. The Δ refers to the incremental formulation and means only the load for the last wood layer of this finite element is returned. Moreover the return value of this function is, like the global elementary stiffness matrix, dependent on the current state of the finite element's global orientation and on the time (secondary growth). In line 106 space for the vector Δf_n^e with its 12 entries is reserved. After that the variable fac (line 109) denotes the pre-factor from equation (2.76) for the vector Δf_n^e. The components of the vector $\begin{pmatrix} Z_x^e & Z_y^e & Z_z^e \end{pmatrix}^T$ used in equation (2.76) are defined in line 110-112 by the variables Zx, Zy and Zz. As stated above the single value for the relative global rotation angle between the finite element and the global z-axis is completely sufficient to calculate the ascending vertical unity vector that is attached to the beam element by $\begin{pmatrix} Z_x^e & Z_y^e & Z_z^e \end{pmatrix}^T = \begin{pmatrix} \cos(globalRotation) & 0 & -\sin(globalRotation) \end{pmatrix}^T$. The entries of the local definition of Δf_n^e are set in line 114-123. Obtaining the change of basis matrix Q^e by calling the appropriate function in line 125 the final return value (line 126) of the function $deltaNodalLoads()$ is the global nodal loads increment vector following equation (2.87).

Maturation strain increments $(\Delta\Lambda_n^e)_{global}$ (equation (2.88)) are modeled by the function $deltaMaturationStrain()$ (line 130-152). Again it is first essential to define the local elementary definition of maturation strain $\Delta\Lambda_n^e$ (equation (2.78)). This is done in line 132-148. In line 132 space for the vector $\Delta\Lambda_n^e$ with its 12 entries is reserved. Afterwards the variables that are needed for the construction of the solution vector are defined (line 133-140). The external and internal radii r_{ext}^e and r_{int}^e considering the concept of layering find their representation with the doubles r_ext and r_int (line 133-134). In line 135 the angle for the straightening up reaction φ_n^e is given by d. Paying attention to the fact that φ_n^e is given relatively to the global z-axis it must be transformed in a local direction relative to the beam element. This is done by subtracting the finite element's global rotation angle (line 135) and using the variable d instead of phi. Regarding equation (2.63) ψ_n^e, which was the shift of the angle for the straightening up reaction considering the different ways of reaction wood building, is given by the variable psi (line 136). The construction of the variables Ne, Mx, My and Mz follows equations (2.79)-(2.82). The individual entries of the vector Λ_n^e are set in line 141-148 by the use of equation (2.78). Getting the change of basis matrix Q^e in line 150 the final return value (line 151) of the function $deltaMaturationStrain()$ is the global maturation strain increment vector following equation (2.88).

Looking upon the possibility to consider extra weights of a finite element the function $deltaExtraLoad(\ldots)$ (line 155- 178) is appropriate. In order to distribute the weight and by that the resulting force equally to all wood layers the function $deltaExtraLoad(\ldots)$ needs an argument that specifies the total end number of layers when the growth process is fully accomplished. This is represented by the integer FE_count (line 155). In line 157 space for the local incremental extra load vector with its 12 entries is reserved. The components $\begin{pmatrix} Z_x^e & Z_y^e & Z_z^e \end{pmatrix}^T$ that are needed for the vertical ascending unity vector can be found in line 159-161. Depending on the value of the variable $extraLoad$, the gravity constant $grav$ and the number of layers the pre-factor for the incremental extra load vector in its local form is defined in line 163 by the double fac. The entries of this vector are set in line 165-174. The transformation from local to global is done in line 176-177.

In line 177 the final return value is the global formulation of the incremental extra load vector.

The functional representation of the change of basis matrix Q^e (see equations (2.83)-(2.86)) is given by $ChangeOfBasisMatrix()$ in line 181-203. As already mentioned the one dimensional value of the relative angle between the global z-axis and the finite element is sufficient to determine the ascending vertical unity vector and also the rotation matrix that transforms the local to the global referential axis. The basis of the coordinate system referring to equation (2.85) is constructed in line 182-184. The matrix in line 184 is the representation of R^e. In line 186 space for the 12×12 sized result matrix is reserved. The diagonal change of basis matrix Q^e, regarding equation (2.86), is put together in line 187-190 by the matrix-wise setting of its diagonal entries. Finally it is returned in line 192.

The interconnection matrix that was introduced in equation (2.90) can be obtained by the call of the function $getConnectionMatrix(\ldots)$ (line 196-202). The resulting matrix is dependent on the index position of the finite element and on the total number of finite elements. In equation (2.91) the index of a finite element e_i was denoted by i and the total number of finite elements by N. Both parameters are represented as arguments of the function $getConnectionMatrix(\ldots)$ by the integers $position$ and FE_count (line 196). In line 197 space for the $12 \times 6(N+1)$ dimensioned result matrix is reserved. Afterwards (line 198-200) the diagonal entries with ones are set according to the position information from equation (2.91). Finally the interconnection matrix is returned in line 201. The definition of class FE is finished in line 203.

Owning the representation of a finite element, the main concept of primary and secondary growth as well as the assembly operations for the linear equation system can be carried out. At the end it leads to the function $GetRotationAnglesFE(\ldots)$ (line 205-306) delivering an array of rotation angles that defines the curvature of the branch.

In the function's argument list (line 205) the parameters defined above the source code can be found again. As stated after equation (2.90) it is supposed that all finite elements are connected one after the other: $e_1 - e_2 - \ldots - e_N$. A simple array is the natural choice for such arrangement. This array can be found in line 208. As the maximum number of finite elements is restricted and a counter $element\_count$ (line 209) for the current number of finite elements is established, a list is not necessary and an array is completely sufficient. Since a constant growth velocity is supposed the change rate of the radii $delta\_radii$ (line 212) can be set. The growth process and with it the successive application of the incremental displacement results, coming from the solutions of the time-dependent linear equation systems, to the finite elements is accomplished in the loop in line 214-277. This loop is performed until all beam elements have been elaborated and by that the growth process ends (line 277).

First of all before its creation the relative position of a finite element must be determined. The variable rot (line 217) is the initial angle between a new finite element and the global z-axis. The first beam element's direction is given by the initial angle of deflection $t0$. That's why in line 220-221 rot is set to $t0$ when it holds true that $element\_count = 0$. Reconsidering the parameter $primaryTropism$, which fixed the growth direction to the value of $t0$, rot must also set to $t0$ when it is $true$ (line 220). All other new created beam elements are linked to the predecessor. Thus the angle must be set to the value from the previous one (line 223).

Primary growth is modeled in line 226-227 by creating a new finite element with its properties from the argument list (line 205) and adding it to the array $FiniteElements$. Descriptively this array can be seen as the branch. In order to model the secondary growth (radial thickening) for all beam elements, except the one originated from the primary growth before, the function $addLayer(\ldots)$ is called in a loop in line 230-232. As already mentioned homogeneous material distribution and properties are supposed. That is why again the radii constant thickening parameter as well as the density and Young's modulus can be found as arguments for the function $addLayer(\ldots)$ in line 231.

Afterwards in line 235-238 space for the global stiffness matrix (see equation (2.92)) and for the load vectors is reserved. It should be noted that the vector $NodalLoadsExternal$ is responsible for the load induced by self-weight, the vector $NodalLoadsInternal$ for the maturation strain and the vector $NodalLoadsExtra$ for the additional weight. The size of the global stiffness matrix depends on the current number of finite elements and is given by $6(element\_count + 1) \times 6(element\_count + 1)$ (line 235).

The assembly procedure, in order to preserve the linear equation system, is carried out according to the formulations (2.92)-(2.94) in a loop in line 241-264. For every finite element (line 242) the elementary stiffness matrix in its global form and the interconnection matrix C^e are obtained by calling the appropriate function of this element (line 243-244). Following equation (2.92) the global stiffness matrix K_n is assembled in the loop (line 247). Then the three components (self-weight, maturation strain and extra load) for the vector ΔF_n (see equation (2.93)) are calculated in line 250-263. Concerning the vector by matrix multiplication it is necessary to create a row vector represented by the matrix A in line 251 to be able to build the product of the row vector of the loads with the interconnection matrix C. This is done for all three load components (line 253, 258 and 263).

After exiting the loop all incremental load vectors are present in their global form. Thus the linear equation system can be formulated and finally solved. A singular value decomposition of the global stiffness matrix has been chosen in order to solve the linear equation system. That is why in line 267 the global stiffness matrix is decomposed correspondingly. The solution vector that contains all nodal displacement components is obtained in line 268. The argument vector in this linear equation system is the sum of the global incremental load vectors (line 268) calculated in the loop before. After solving this system the vector $Solution$ represents the increment in displacements $\begin{pmatrix}\Delta q_1 & \Delta q_2 & \dots & \Delta q_{6(element\_count+1)}\end{pmatrix}$. It should be remembered that for a finite element e_i it is defined $\Delta q^{e_i} = \begin{pmatrix}\Delta q_{6i-5} & \Delta q_{6i-4} & \dots & \Delta q_{6(i+1)}\end{pmatrix}$. It is also recommended to reconsider figure 2.2 for the notation.

To update the whole structure it is required to apply the respective displacement solution to the single finite elements. This is done in a loop in line 271-276. According to equation (2.90) the displacement for the current finite element is extracted from the solution vector with the help of the interconnection matrix (line 272-273). Afterwards the fourth entry of the vector Δq^e is extracted because referring to figure 2.2 this component represents the incremental rotation about the local y-axis. Since the rotation angle is the same regardless of whether global or local coordinate system, it then instantly can be added to the property $GlobalRotation$, which was the relative angle measured between the global z-axis and that of the finite element. When all growth processes are finished (line 278) the curvature of the branch is translated into an array of rotation angles in line 281-304.

Since there is the possibility for interpolation by the variable n_GU it is necessary to define the domain and codomain for the resulting rotation angles. Both are denoted by the arrays $angles$ and $range$ (line 281-282). In the loop in line 284-288 the rotation angles are extracted from the finite elements (line 286). Let $[0, 1]$ be the codomain and let $ang(i)$ be the function that returns the i-th roation angle. Then the simple mapping function $ang(i) \mapsto \frac{i}{n\_FE-1}$ defines the codomain constructed in line 287. The Akima spline interpolation has been used to interpolate between the rotation angles. The interpolation function is defined in line 291-292. Line 295 considers the fact that the number of growth units must not be less than the number of finite elements.

The n_GU-sized solution array is defined in line 297. Since the array of rotation angles is supposed to be differential, that means each item in the list only is the rotational difference to its predecessor, the help variable $prev$ is defined in line 298. Finally with the help of the interpolation function the rotation angles are obtained in the loop in line 300-304. At the end the curvature of the branch represented by an array of rotation angles is returned in line 306.

Chapter 4

Application in three tree models

In this chapter the algorithms developed in chapter 3 are individually applied to three tree models which have been implemented before: a spruce, a beech and a sympodial tree. The original source codes for the spruce and the beech tree can be found in [13] and [1, pp. 317-326]. The model for the sympodial tree was taken from the example section of the GroIMP software ([14]). The differences to the original tree model and the adaptation steps needed in order to apply the branch bending behavior to the tree models are documented. Since tree branches can be seen as ramified structures the successively occurring gain in mass through ramification is also considered. In order to recognize the adaptation steps the code that is supplemented or modified is highlighted bold.

4.1 Spruce tree

4.1.1 Force-applying approximative method

This topic has been accomplished by the author in a previous work and can be found in [15][pp. 18-22].

4.1.2 Incremental finite element method

Source code:

```
module T;
module M1;
module S1;
module M2;
module S2;
module M3;
module S3;
module GU(float incd, int age, float diameter, float length, int order) {
   double init_rot = 0.0;
   double[] angles;
   float volume() {
      return Math.PI * diameter * diameter * 0.25 * length;
   };
};
module GUS(float incd, int age, float diameter, float length) extends F0;
module BA(int age, super.angle) extends RL(angle);
module GA(int age, super.angle) extends RL(angle);
module HA(int age, super.angle) extends RL(angle);

module ROT(super.angle) extends RL(angle);
module G(super.length, super.diameter) extends F(length, diameter);
const int ang = 45;
const int x3 = 0;
//const int[] a  = { 0, 15, 25, 32, 37, 40 };
const int[] gg = { 0, 0, 4 };
const int[] hh = { 0, 0, 2, 4, 8 };
int n, k;
const float[] prob_n = {0.1, 0.4, 0.3, 0.2};
const int[] n_subap = {5, 6, 7, 8};

const int n_FE = 20;
int n_GU = 200;
double t0 = 100;
double rho = 2.0;
double E = 70000;
double grav = 9.81;
const double rad = 180.0/Math.PI;
double sigma = 0.0;
double phi = 60;
double a = -200.0;
double b = 2000.0;
boolean primaryTropism = true;

public void grow() {
[

Axiom ==> P(2) D(1) L(100) T;

x:T ==> Nl(80*TurtleState.length(x)) GUS(2.2, 0, TurtleState.diameter(x), TurtleState.length(x))
            RH(random(0, 360)) { k = 0; }
                for ((1:3))
             ( [ MRel(random(0.2, 0.85)) RH(k*120+normal(0, 5.5))
               { k++; } RL(x3+normal(0, 2.2)) BA(0, 0) LMul(0.4) M1 ] )
            RH(random(0, 360)) { n = n_subap[distribution(prob_n)]; k = 0; }
                for ((1:n))
```

```
                  ( [ MRel(random(0.85, 1)) RH(k*360/n+normal(0, 3.1))
                    { k++; } RL(x3+normal(0, 2.2)) BA(0, 0) LMul(0.65) S1 ] )
            T;
x:S1 ==> Nl(80*TurtleState.length(x)) GU(1.3, 0, TurtleState.diameter(x), TurtleState.length(x),1)
               [ MRel(random(0.85, 1)) RH(15)
                 RU(ang+normal(0, 2.2)) AdjustLU LMul(0.7) S2 ]
          [ MRel(random(0.85, 1)) RH(-15)
            RU(-ang+normal(0, 2.2)) AdjustLU LMul(0.7) S2 ] GA(0, 0) S1;
x:M1 ==> Nl(80*TurtleState.length(x)) GU(0.8, 0, TurtleState.diameter(x), TurtleState.length(x),1)
               [ MRel(random(0.85, 1)) RH(15)
            RU(ang+normal(0, 2.2)) AdjustLU LMul(0.7) M2 ]
          [ MRel(random(0.85, 1)) RH(-15)
            RU(-ang+normal(0, 2.2)) AdjustLU LMul(0.7) M2 ] HA(0, 0) M1;
x:S2 ==> Nl(80*TurtleState.length(x)) GU(1.3, 0, TurtleState.diameter(x), TurtleState.length(x),2)
      [ MRel(random(0.85, 1)) RH(10)
                 RU(ang) AdjustLU LMul(0.7) S3 ]
          [ MRel(random(0.85, 1)) RH(-10)
            RU(-ang) AdjustLU LMul(0.7) S3 ] S2;
x:M2 ==> Nl(80*TurtleState.length(x)) GU(0.8, 0, TurtleState.diameter(x), TurtleState.length(x),2)
                [ MRel(random(0.85, 1)) RH(10)
                 RU(ang) AdjustLU LMul(0.7) M3 ]
                [ MRel(random(0.85, 1)) RH(-10)
            RU(-ang) AdjustLU LMul(0.7) M3 ] M2;
x:S3 ==> Nl(80*TurtleState.length(x)) GU(1.3, 0, TurtleState.diameter(x), TurtleState.length(x),3);
x:M3 ==> Nl(80*TurtleState.length(x)) GU(0.8, 0, TurtleState.diameter(x), TurtleState.length(x),3);

x:GU(incd, a, d, l, o) ==> DlAdd(incd*(a+1)) GU(incd, a+1, TurtleState.diameter(x)+incd*(a+1), l, o);
x:GUS(incd, a, d, l) ==> DlAdd(incd*(a+1)) GUS(incd, a+1, TurtleState.diameter(x)+incd*(a+1), l);
DlAdd(arg) ==> ;

BA(age, angle) ==> BA(age+1, angle);
GA(age, angle) ==> GA(age+1, angle);
HA(age, angle) ==> HA(age+1, angle);

]

erase();
derive();
for (int i=1; i<5; i++) {
   replace(i);
   derive();
```

```
}
}

public void erase() {
[
   G(a,b) ==> ;
   ROT ==> ;
   g:GU ::> {g[angles]= null;}
]
}

public void replace(int order) {
[
   x:GU, (x[order] == order) ==> x

      {GU rootGU = selectWhere((* x (<--)* y:GU, (y[order] == x[order]) *), (1==count((*
           y (<--)* z:GU, (z[order]==x[order]) *))));

      GU[] branch = array((* rootGU (-->)* y:GU, (y[order] == rootGU[order]) *));

      double branch_length = sum((* rootGU (-->)* y:GU, (y[order] == rootGU[order]) *)[
           length]);
      int GU_count = (int)count((* rootGU (-->)* y:GU, (y[order] == rootGU[order]) *));
      double r_start = 0.5 * rootGU[diameter];
      double r_end = 0.5 * branch[branch.length-1][diameter];
      double Le = branch_length/n_FE;

      double left = sum((* x (<--)+ y:GU, (y[order] == x[order]) *)[length]);
      double start = (n_GU-1) * left/branch_length;
      double end = (n_GU-1) * (left + x[length])/branch_length;

      double[] extraLoads = new double[n_FE];
      double ll = 0.0;

      GU[] hn = new GU[2*GU_count];
      int[] rel_hn_index = new int[2*GU_count];
      int hn_count = 0;

      for (int i = 0; i < branch.length; i++) {
         GU gr = branch[i];
         GU[] next_hn = array((* gr -minDescendants-> y:GU, (y[order] > rootGU[order])
              *));
         boolean branched = (next_hn.length != 0);
         ll += gr.length;

         if (branched) {
            double extraVolume = 0.0;

            for (int j = 0; j < next_hn.length; j++) {
```

```
         GU nb = next_hn[j];
         hn[hn_count] = nb;
         rel_hn_index[hn_count] = (int) ((n_GU-1)*ll/branch_length);
         hn_count++;

         extraVolume += sum((* nb (-->)* GU *).volume());
      }

      int index = (int) Math.ceil((n_FE-1)*ll/branch_length);

      double weight = rho * extraVolume/(index+1);
      for (int j = 0; j <= index; j++)
         extraLoads[j] += weight;
   }
}

double t0_;
if (x[order] == 1)
   t0_ = t0;
else
   t0_ = rootGU[init_rot];

double[] angles;
if (rootGU[angles] != null) {
   angles = rootGU[angles];
} else {
   angles = getRotationAngles(n_FE, n_GU, t0_, rho, (rootGU[age]+1)*E, grav, Le,
       r_start, r_end, sigma, phi, a, b, primaryTropism, extraLoads);
   rootGU[angles] = angles;
}

for (int i = 0; i < hn_count; i++) {
   GU gru = hn[i];
   double f = 0.0;
   for (int j = 0; j <= rel_hn_index[i]; j++)
      f += angles[j];
   gru.init_rot = f;
}

double ring_increment = (r_start - r_end)/(n_GU-1);
double l = Le/((double)n_GU/n_FE);
}
if (left==0 && x[order] > 1) (
   AdjustLU
   ROT(-t0_)
)

for (int i = (int)start; i < end; i++) (
   ROT(angles[i])
```

```
191           G(1, 2*r_start - 2*i*ring_increment)
192         );
193   ]
194   }
```

The first thing necessary is the identification of the tree model's growth units. The ones of the spruce tree are named *GU* (line 8). They had inherited from the class $F0$, thus there were no diameter and length information within the module *GU* but this information is needed for the later replacement with the new growth units forming the branch's curvature. That is why the float parameters $diameter$ and $length$ have been added to the parameter list of GU. The determination of the order of the growth units (hierarchic branch structure) is accomplished by also adding the integer $order$ (line 8) to the parameter list. Therefore in all appearances of the module *GU* in the production rules from function $grow()$ the order information must be added. Moreover the heredity from the class $F0$ was deleted, because *GU* now constitutes a control point, and the entirety of all control points represent and store the structure information of the tree. Through this fact the control points of type *GU* can be expanded with other properties that are needed for the later construction. The initial angle of deflection must be stored in a *GU* control point as this information is an essential parameter for the function $GetRotationAnglesFE(\ldots)$. This property can be found in line 9 with the double $init\_rot$.

Considering unnecessary recalculations it is useful to save the already calculated rotation angles in a control point. This is carried out in line 10 with the array $angles$. By that when replacing the part-wise *GU* sections there are no further calculations performed, it is needed not more than using the present angles. For the successively occurring gain in branch mass there must be a volume function for every control point. In line 11-13 such function can be witnessed.

A distinction between the tree's stem and branch parts is necessary, because only the branches are meant for bending. That is why in line 15 a module for the stem parts, named GUS, can be found. This module is a full copy of *GU* but with the difference that GUS can not be ranked and so does not hold the parameter $order$ and furthermore it inherits from $F0$, thus possesses a geometrical interpretation.

The growth units and rotation commands from the new branch need to be identified by their own modules. Because of that in line 20-21 the modules ROT and G are defined. The module separation is done for identification due to later applied production rules. The parameters for the function $GetRotationAnglesFE(\ldots)$ can be found in line 31-42. The setting of these parameters determines the amount of discretization, of interpolation and also the branch properties and by that its bending behavior.

The growth function of the tree is defined in line 44-98 by $grow()$. In every production rule in line 49-83 all occurring GU statements were modified, due to the reason that the tree's replacement rules make use of turtle state changing commands (for example $MRel$ or $LMul$). Every time a GU control point is set, the turtle states for diameter and length need to be stored in the module. The tree's own branch bending mechanism in line 86-88 is abrogated.

Since relational growth grammars are parallel graph replacement systems, there is a need for a sequential rule application. Thus the functions $replace(\ldots)$ (line 108-194) and $erase()$ (line 100-105) separate the different replacement rules, for later sequential invocation. The function $erase()$ deletes all ROT rotation commands and G growth units. Furthermore for all GU control points it sets the array of rotation angles to $null$. This is needed because the tree grows in every derivation step, thus the branches also need to be updated with a completely new construction.

The function $replace(\ldots)$ replaces the original branch growth units by the new ones with the rotation angles in between constituting the branch's curvature. This function also contains a dependency on the branch order. That means only the branches of the specified order are replaced. This construction pays attention to the later considered implementation of the gain in branch mass while growing. Having defined functions of that kind after the tree's production rules the functions $erase()$ and $replace(\ldots)$ can be invoked (line 92-97). The replacement function must be called in the loop in line 94-97 for all possible ramification orders due to the mentioned reason of gain in branch mass. The XL function $derive()$ applies all remaining productions to the current graph and hence ensures the sequential execution of the functions containing the replacement rules. The function $replace(\ldots)$ does only contain one production rule. This rule (line 110-192) replaces all nodes of type GU with the specified order.

Since the GU nodes represent the control points and all of them constitute the branch structure they must be sustained (line 110). The graph query in line 112 determines the root growth unit, which is the most left one having the same order. Starting from the root growth unit it is possible to obtain all control points which determine the structural branch information. This is done with the graph query in line 114. Afterwards other branch information like its length or radii are gathered again by the use of graph queries (line 116-119). The length of the finite elements is defined in line 120 depending on the length of the branch and the number of finite elements set by the user. The variable $left$ (line 122) represents the leftward length of the branch measured from the current control point. It is needed as a position information for the part-wise replacement of the control points with the matching branch section. That is why in line 123-124 the index variables $start$ and end determine the range of rotation angles matching the relative control point position and length.

The additional branch mass that comes from ramification is calculated in line 126-157. The array $extraLoads$ corresponds to the extra load that can be added to every finite element.

The array hn (line 129) picks up all first successor control points with a higher order and the array rel_hn_index (line 130) the indices, regarding the new growth unit number, where the ramification takes place. The integer hn_count (line 131) counts the number of ramification events. To examine where ramification occurs the loop in line 133-157 runs through all control points of the branch. The graph query in line 135 generates an array with all higher order possessing control points. If this array is empty at this position the branch is not branched. This is expressed by the boolean in line 136. If branched it is necessary to iterate over all following control points after the ramification positions in order to calculate the additional weight. A variable considering the volume increase is initialized in line 140. After that in line 143-146 the array hn is supplemented with the new ramification control points. The volume which is obtained by graph queries is added in line 148. Considering the density the entries for the vector carrying the additional loads for the finite elements then can be set (line 151-156).

For the calculations in line 159-163 the initial angle of deflection is determined. This angle $t0$ is only valid for branches of the first order but the side branches' initial angle of deflection depends on the orientation of their predecessor growth unit. That is why these angles are maintained in line 173-179 using the present array hn of successor control points with higher orders. The mentioned way to prevent unnecessary recalculations can be found in line 165-171. In the case when the root growth unit contains angles then these are used, otherwise (line 169-170) the angles are calculated with the appropriate arguments and then stored in the root growth units for further utilization.

In line 181-182 the values for the radial thickening and the length of a new growth unit are defined. The condition ($left == 0$ && $x[order] > 1$) in line 184 is for the case when the root growth unit of a side branch is replaced. Then the turtle's position needs to be corrected, so that the gravity vector always is identical to the vertical axis and the new growth units are spatially placed correctly. Afterwards in line 189-192 within the indices range $start$ and end, defined in line 123-124, the rotations are performed and the new growth units are set.

4.2 Beech tree

4.2.1 Force-applying approximative method

This topic has been accomplished by the author in a previous work and can be found in [15][pp. 25-30].

4.2.2 Incremental finite element method

Source code:

```
[...]

@de.grogra.xl.lang.ImplicitDoubleToFloat
public class BeechModel extends RGG// implements TreeModel
{
   module ROT(super.angle) extends RL(angle);
   module G(super.length, super.diameter) extends F(length, diameter).(setShader(
       branchShader));

   const int n_FE = 20;
   int n_GU = 100;
   double rho = 2.0;
   double E = 300;
   double grav = 9.81;
   const double rad = 180.0/Math.PI;
   double sigma = 0.0;
   double phi = 60;
   double a = -200.0;
   double b = 2000.0;
   boolean primaryTropism = false;

   const boolean USE_RADIATION = true;

   const Shader leafShader = shader("Beech leaf");
   const Shader stemShader = shader("Beech stem");
   const Shader branchShader = shader("Beech branch");

   [...]

   module Organ(super.length, int order, boolean isInternode) extends Cylinder(length,
       0.0001)
   {
      float len = length;
      double angles[];
      double init_rot = 0.0;
      { if (isInternode && order > 0)
         length = 0;
      };
      float allocatedCarbon;
      float producedCarbon;
      float exportedCarbon;

      @Editable
      public float preference = 1;

      float volume()
      {
         return Math.PI * radius * radius * len;
      }

      float maintenanceRespiration() // mol CO2 in current year
      {
         return 3600 * 24 * 365 * MAINTENANCE_RESPIRATION * 2 * Math.PI
            * radius * len;
      }

      void transportCarbon(float imported, float prod, float above)
      {
         imported += prod;
         above += prod;
         float q = radius / 0.0025;
         float ex = imported * Math.exp(-0.7 * len * (1 + q) / q);
         this[allocatedCarbon] = imported - ex;
         this[exportedCarbon] = ex;
         this[producedCarbon] = above;
      }

      void grow(float distributedCarbon)
      {
         float input = allocatedCarbon + distributedCarbon;
         float mr = maintenanceRespiration();
         float c = input - mr;
         if (c >= 0)
         {
            float m = c * (GROWTH_RESPIRATION_FRACTION * C_MASS / C_FRACTION);
            this[radius] = Math.sqrt(this[radius]**2 + m/(DENSITY*Math.PI*len));
         }
         this[mark] = (order > 0) && (c < 0);
      }
      [...]
   }

   [...]

   module Internode(super.length, super.order) extends Organ(length,order,true)
      .(setShader((order==0) ? stemShader : branchShader))
   {
      {setScaleV(true);setLayer(8);}
   }

   [...]

   public void step ()
   {
      radiation.compute();
      transportCarbon();
      distributeCarbon();
      [
         o:Organ, (o[mark]) ==>> ;
         BeechLeaf ==>> ;
      ]
//    sun = direction(graph().getNodeForName("Sun"));
      grow();

      erase();
      derive();
      for (int i=1; i<4; i++) {
         replace(i);
         derive();
      }
   }

   [...]

   protected void grow ()
   [
      b:Bud(o, v, s,,, t) ==>
      {
      [...]
      }
         for (int i : (1 : count))
         (
         {
            int sign = s * (1 - (i&1)*2);
            boolean terminal = i == count;
            float q = ((float) i / count) ** VIT_POWER_0;
            float vit = Math.max(v * VIT_A * q  / (1 + VIT_B * q), VIT_MIN);
         }
            x:Internode(createShort ? 0.002 : lenDist[i-1]*len, o)

            RU(sign * t.axisAngle(o))
            //RH(t.twist(o))
            [...]
         );
   ]

   [...]

public void erase() {
[
   G(a,b) ==> ;
   ROT ==> ;
   g:Internode ::> {g[angles]= null;}
```

```
]
}

public void replace(int order) {
[
   x:Internode, (x[order] == order) ==> x

      {Internode rootGU = selectWhere((* x (<--)* y:Internode, (y[order] == x[order]) *)
          , (1==count((* y (<--)* z:Internode, (z[order]==x[order]) *))));

      Internode[] branch = array((* rootGU (-->)* y:Internode, (y[order] == rootGU[order
          ]) *));

      double branch_length = sum((* rootGU (-->)* y:Internode, (y[order] == rootGU[order
          ]) *)[len]);
      int Internode_count = (int)count((* rootGU (-->)* y:Internode, (y[order] == rootGU
          [order]) *));
      double r_start = rootGU[radius];
      double r_end = branch[branch.length-1][radius];
      double Le = branch_length/n_FE;

      double left = sum((* x (<--)+ y:Internode, (y[order] == x[order]) *)[len]);
      double start = (n_GU-1) * left/branch_length;
      double end = (n_GU-1) * (left + x[len])/branch_length;

      double[] extraLoads = new double[n_FE];
      double ll = 0.0;

      Internode[] hn = new Internode[2*Internode_count];
      int[] rel_hn_index = new int[2*Internode_count];
      int hn_count = 0;

      for (int i = 0; i < branch.length; i++) {
         Internode gr = branch[i];
         Internode[] next_hn = array((* gr -minDescendants-> y:Internode, (y[order] >
             rootGU[order]) *));
         boolean branched = (next_hn.length != 0);
         ll += gr.len;

         if (branched) {
            double extraVolume = 0.0;

            for (int j = 0; j < next_hn.length; j++) {
               Internode nb = next_hn[j];
               hn[hn_count] = nb;
               rel_hn_index[hn_count] = (int) ((n_GU-1)*ll/branch_length);
               hn_count++;

               extraVolume += sum((* nb (-->)* Internode *).volume());
```

```
                }

                int index = (int) Math.ceil((n_FE-1)*ll/branch_length);

                double weight = rho * extraVolume/(index+post+1);
                for (int j = 0; j <= (index+post); j++)
                    extraLoads[j] += weight;
            }
        }

        double t0_;
        if (x[order] == 1) {
            RL[] a = array((* rootGU -ancestor-> RL *));
            t0_ = a[0].angle;
        } else {
            t0_ = rootGU[init_rot];
        }

        double[] angles;
        if (rootGU[angles] != null) {
            angles = rootGU[angles];
        } else {
            angles = getRotationAngles(n_FE, n_GU, t0_, rho, E, grav, Le, r_start, r_end,
                 sigma, phi, a, b, primaryTropism, extraLoads);
            rootGU[angles] = angles;
        }

        for (int i = 0; i < hn_count; i++) {
            Internode g = hn[i];
            double f = 0.0;
            for (int j = 0; j <= rel_hn_index[i]; j++)
                f += angles[j];
            g.init_rot = f;
        }

        double ring_increment = (r_start - r_end)/(n_GU-1);
        double l = Le/((double)n_GU/n_FE);

        }
        if (left==0) (
            AdjustLU
            ROT(-t0_)
        )

        for (int i = (int)start; i < end; i++) (
            ROT(angles[i])
            G(l, 2*r_start - 2*i*ring_increment)
        );
]
```

```
234 }
235 }
```

On account of a complex construction, we limit the source code to the necessary and modified parts. Code that is not shown is replaced by a [...] statement. The module definitions for the rotation command and growth units can be found in line 6-7. The $setShader(branchShader)$ statement (line 7) sets the surface coloring of the branch parts. This is done because the original growth units came up with this coloring and it is not supposed to change.

In line 9-19 the parameters for the branches can be found. The original growth unit of this beech tree is the module $Internode$ (line 83-87). It inherits from the module $Organ$ (29-79). On account of the already existing distinguishable module for the growth units, there is no need for the GU module. Besides, the identification of the stem does not need a separate module. It can be done with the help of the parameter $order$ (line 83). When $order$ equals zero, it is a growth unit of the stem. However, the parent class $Organ$ itself contains the geometric interpretation by inheriting from the class $Cylinder$. This is the reason why it is not possible to cut the inheritance from $Organ$ in the module $Internode$, to make the nodes of type $Internode$ invisible.

The solution for this problem is the additional boolean parameter $isInternode$ in the definition (line 29). When $Internode$ inherits from $Organ$, this parameter must be set to true. Now in the mother class $Organ$ the length of this shape, a cylinder, can be set to zero, when it is of type $Internode$ and also not a stem part, hence having an $order$ greater than zero (line 34-36). Furthermore we need a copy of the length parameter, owing to the fact that this parameter is needed for the internal functions $volume()$, $maintenanceRespiration()$, $transportCarbon()$ and $grow()$ and thus can not be simply set to zero. This duty is done by the float len (line 31), a full copy of length. All appearances of $length$ must be replaced by len (line 46, 52, 60, 74). In the parent class of $Internode$, the module $Organ$, the properties for the initial angle of deflection and for the rotation angles have been added in line 32-33 due to the same reason as in the spruce tree: maintaining the right branch construction details and avoiding unnecessary recalculations.

The function for the tree's growing $step$ (line 91-109) is enriched by the invocations of the functions $erase()$ and $replace(\ldots)$ in order to replace the current branch construction by first deleting the old growth units and then replacing the control points with the growth units of the new branch. The XL function $derive()$ (line 104,107) applies all remaining productions to the current graph and hence it ensures the sequential execution of the functions containing the replacement rules. Like in the spruce tree the function $replace(\ldots)$ again has a dependency on the branch order due to the replacement rule and successively occurring gain in branch mass caused by ramification. That's why this function is called in a loop in line 105-108.

The beech tree's internal vertical branch rotations are commented out (line 130) in the function $grow()$, because the bending is only supposed to be determined by our new model. The definition of the function $erase()$ is identical to the one from the spruce tree. The same nearly holds true for the function $replace(\ldots)$, because both functions rely on the same train of thoughts. Therefore just marginal changes need to be carried out. The module for the control points is, like mentioned, of type $Internode$. Because of that all appearances of GU need to be replaced by $Internode$. Due to the fact that stem parts also are of type $Internode$ this may be seen as a problem. But the function $replace(\ldots)$ is only called for $order \geq 1$ and because stem parts possess order zero there is no identification problem.

Another difference is the definition of the variables r_start and r_end (line 155-156). The module $Internode$ makes use of radii whereas GU uses diameters. That is why in the spruce tree they had been divided by two. Furthermore the last adaptation point comes from the circumstance that for the beech tree a general initial angle of deflection can not be given. Owing to the fact that this tree contains some random stochastic behavior, the angle just can be gained from the tree structure itself. On this account in line 198 a graph query obtains this angle by getting the argument of the first RL rotation command before the root growth unit, which is equivalent to the initial angle of deflection.

4.3 Sympodial tree

4.3.1 Force-applying approximative method

This topic has been accomplished by the author in a previous work and can be found in [15][pp. 22-25].

4.3.2 Incremental finite element method

Source code:

```
module A(float l, float w);
module B(float l, float w, int o);

module GU(float length, float diameter, int order)
{
   double init_rot = 0.0;
   float volume() {
      return Math.PI * diameter * diameter * 0.25 * length;
   };
};
module GUS(super.length, super.diameter) extends F(length, diameter);
module ROT(super.angle) extends RL(angle);
module G(super.length, super.diameter) extends F(length, diameter);

const float r1 = 0.9f;
const float r2 = 0.8f;
const float a1 = 30;
const float a2 = 30;
const float wr = 0.707f;

protected void init()
[
   Axiom ==> A(1, 0.1f);
]

const int n_FE = 15;
int n_GU = n_FE;
double t0 = a1;
double rho = 2.0;
double E = 20;
double grav = 9.81;
const double rad = 180.0/Math.PI;
double sigma = 0.0;
double phi = 60;
double a = -200.0;
double b = 2000.0;
boolean primaryTropism = false;

public void grow() {
[
   A(l,w) ==> GUS(l,w) [B(l*r1, w*wr,1)]
            RH(180) [B(l*r2, w*wr,1)];

   B(l,w,o) ==> GU(l,w,o) [RU(-a1) AdjustLU B(l*r1, w*wr,o+1)]
                 [RU(a2) AdjustLU B(l*r2, w*wr,o+1)];
]
erase();
derive();
for (int i=1; i<10; i++) {
   replace(i);
   derive();
}
}

public void erase() {
[
   G(a,b) ==> ;
   ROT ==> ;
]
}
```

```
 62  public void replace(int order) {
 63  [
 64     x:GU, (x[order] == order) ==> x
 65
 66        {double r_start = 0.5 * x[diameter];
 67        double r_end = r_start;
 68        double Le = x[length]/n_FE;
 69
 70        double[] extraLoads = new double[n_FE];
 71
 72        double extraVolume = sum((* x (-->)+ GU *).volume());
 73        double weight = rho * extraVolume/n_FE;
 74        for (int j = 0; j < n_FE; j++)
 75           extraLoads[j] += weight;
 76
 77        double _t0 = (x[order] == 1 ? t0 : x[init_rot]);
 78
 79        double[] angles = getRotationAngles(n_FE, n_GU, _t0, rho, E, grav, Le, r_start,
             r_end, sigma, phi, a, b, primaryTropism, extraLoads);
 80
 81        GU[] higher_neighbours = array((* x -minDescendants-> y:GU, (y[order] > x[order])
             *));
 82        for (int i = 0; i < higher_neighbours.length; i++) {
 83           GU g = higher_neighbours[i];
 84           double f = 0.0;
 85           for (int j = 0; j < n_GU; j++)
 86              f += angles[j];
 87           g.init_rot = f;
 88        }
 89
 90        double ring_increment = 2*(r_start - r_end)/(n_GU-1);
 91        double l = Le/((double)n_GU/n_FE);
 92
 93        }
 94        if (x[order] > 1) (
 95           AdjustLU
 96           ROT(-_t0)
 97        )
 98
 99        for (int i = 0; i < n_GU; i++) (
100           ROT(angles[i])
101           G(l, 2*r_start - i*ring_increment)
102        );
103  ]
104  }
```

Like in the spruce tree there are 4 module definitions needed: one for the control points (old growth units), the stem parts and for the new rotation commands and growth units. This has been carried out in line 4-13. Again the parameter *order* has been added to the parameter list of the control points. Hence the module B, which is the module that models the tree's growth (line 44-45), also needs the addition of the parameter $order$. Furthermore a property that is used to store the initial angle of deflection and a volume function is supplemented. Due to a binary ramification behavior of the sympodial tree it is not required to also add an array for the rotation angles, because every control point in this tree corresponds to an own branch. In line 26-37 the parameters for the branches can be found. The appearances of GU statements in the production rule of the tree's growth function $grow()$ (line 39-53) then need to be replenished with the order information (line 44). The function furthermore is enriched by the invocations of the common functions $erase()$ and $replace(\ldots)$ in order to replace the current branch construction by first deleting the old growth units and then replacing the control points with the growth units of the new branch.

The XL function $derive()$ (line 104,107) applies all remaining productions to the current graph and hence it ensures the sequential execution of the functions containing the replacement rules. Like in all other trees the function $replace(\ldots)$ again has a dependency on the branch order due to the replacement rule and successively occurring gain in branch mass caused by ramification. That's why this function is called in a loop in line 49-52. In comparison to the replacement function of the other trees the one for the sympodial tree can be simplified. In line 66-68 the radii and the lengths of the finite elements are calculated. The extra load vector in composed in line 70-75 by just using graph queries to obtain the volume of all successor control points (line 72), multiplying it with the density (line 73) and finally constructing the vector by distributing the load (line 74-75).

After the determination of the initial angle of deflection (line 77) the function $GetRotationAnglesFE(\ldots)$ can be called to get the rotation angles, as they determine the branch's curvature. In line 81-88 the information of the initial angle of deflection for all following control points, that will be replaced afterwards, is maintained by setting the property $init\_rot$. At the end the radius and length of the new growth units is calculated (line 90-91). In the case when not the first control point is replaced, identified by the condition $order > 1$ (line 94), the turtle's orientation must be corrected. After that the final new branch, bending under its self-weight, is put together in line 99-102.

Chapter 5

Phenotype differences

In this chapter the analysis of phenotype differences between the results of the two algorithms is carried out. The working basis is a branch with constant order 1. Since the two methods rely on several different parameters it is necessary to determine the relationship between them and also to define transformations. This is done in order to obtain an equivalent working basis formulation. This mainly is important for the elasticity behavior of both methods. As a convention the parameters of the approximative method shall be given by the matching parameters from the finite element method. The first difference applicable is that the approximative method does not contain radii information. This information is included in the construction of the parameter $K0$. The only time this information is needed is the conicity parameter $coni$. But this problem is solved by the application of the definition from (2.19):

$$coni = 1 - \frac{r\_end}{r\_start}. \tag{5.1}$$

The branch's length information and some others also seem to be missing but they are also included in the $K0$ parameter. That is why $K0$ is defined like in equation (2.9). Through this definition it already contains the parameters defining the radius, length, density, gravity or Young's modulus. Thus K0 can be calculated as:

$$(K0)^2 := 2\frac{F}{EI} \implies K0 = \sqrt{2\frac{F}{EI}} = \sqrt{2\,\frac{grav * rho * \pi * n\_FE * Le * r\_start^2}{E\frac{\pi * r\_start^4}{4}}} \tag{5.2}$$

For examining phenotype differences it is necessary to possess an indicator to express these. For this purpose the minimal difference between a fixed position on the branch built from the approximative algorithm and the one from the finite element algorithm has been measured. Due to the fact that the approximative algorithm does not contain any interpolation possibilities, the discretization given by the parameter $disc$ is equal to the number of fixed positions where the difference is measured. For the following the function measuring this difference will be called

d_{min}. For the interpolation between the values in order to be able to calculate an overall error by an integrative function, the function d_{min} must be expanded to intermediate values. That is why it holds true that $i \in \mathbb{R}$. Let d_{min} be defined by:

$$d_{min}(i) = \begin{cases} \min\left(\,\delta\left((x_i^{app}|y_i^{app}),\, B\right)\right) & \text{for } i \in \{1;\ 2;\ \ldots;\ disc+1\} \\ \min\left(\,\delta\left((x_{\lfloor i\rfloor}^{app}|y_{\lfloor i\rfloor}^{app}),\, B\right)\right) & \\ +\Big[\min\left(\,\delta\left((x_{\lceil i\rceil}^{app}|y_{\lceil i\rceil}^{app}),\, B\right)\right) & \\ -\min\left(\,\delta\left((x_{\lfloor i\rfloor}^{app}|y_{\lfloor i\rfloor}^{app}),\, B\right)\right)\Big]\,(i-\lfloor i\rfloor) & \text{else} \end{cases} \tag{5.3}$$

Where $(x_i^{app}|y_i^{app}) \in A$. The set of points

$$A := \left\{(x_1^{app}|y_1^{app});\ (x_2^{app}|y_2^{app});\ \ldots\ ;\ (x_{disc+1}^{app}|y_{disc+1}^{app})\right\} \tag{5.4}$$

constitutes the branch built by the approximative algorithm and

$$B := \left\{(x_1^{FE}|y_1^{FE});\ (x_2^{FE}|y_2^{FE});\ \ldots\ ;\ (x_{n\_GU}^{FE}|y_{n\_GU}^{FE})\right\} \tag{5.5}$$

the branch built by the finite element algorithm. The distance function δ is defined by

$$\delta\left((x_i^{app}|y_i^{app}), B\right) := \Bigg\{\sqrt{(x_1^{FE}-x_i^{app})^2+(y_1^{FE}-y_i^{app})^2}\ ;\ \sqrt{(x_2^{FE}-x_i^{app})^2+(y_2^{FE}-y_i^{app})^2}\ ; \\ \ldots\ ;\ \sqrt{(x_{n\_GU}^{FE}-x_i^{app})^2+(y_{n\_GU}^{FE}-y_i^{app})^2}\Bigg\}\ . \tag{5.6}$$

Where $i \in \{1;\ 2;\ \ldots;\ disc+1\}$. Finally the error function $error$, that gives a dimension-less value as an indicator for the phenotype difference. It can be defined as

$$error = \int_1^{disc+1} d_{min}(i)\, di\ . \tag{5.7}$$

Owning the function d_{min} it is possible to examine where the phenotype differences occur for a fixed setting. The function $error$ furthermore makes it possible to find out the error dependency of a certain parameter by traveling through various parameter ranges. Both subjects will be addressed in the following. Considering the amount of possible parameter settings only the main results from these examinations are shown.

5.1 Parameter *E*

The parameter E which is the elasticity constant, the Young's modulus, was examined in a range of $50.0 \leq E \leq 100000.0$. It was found that for the range of $500.0 \leq E \leq 6000.0$ and for small initial angles of deflection in the range $0.0 \leq t0 \leq 40.0$ the main error occurred. Figure 5.1 shows the application of the function d_{min} for the setting $E = 800.0$ and $t0 = 20.0$. What can be recognized is that the branches grow apart. The strength of this behavior slightly decreases until reaching $E = 6000$. Over and below this range the values for the differences were much lower, but also dependent on the initial angle of deflection.

For values of $E \leq 500.0$ the overall error was much less than in the mid range defined before. This can be seen in figure 5.2 which plots the function $error$. Nevertheless for deflection angles of $t0 \leq 20.0$ in the lower elasticity ranges it was again high. But afterwards it is significantly lower than the error occurring in the mid range $500.0 \leq E \leq 6000.0$. It must be noted that not all settings provided reasonable calculation results.

For a value $E < 80.0$ the finite element algorithm failed to do so. Thus in that case the phenotype difference data had to be discarded. Another interesting consideration is the elasticity of the branch's first sections: The bending of the branch built by the approximative algorithm is less than the one from the finite element method. That is one reason for the higher overall stiffness of the branch built by the finite element algorithm. For all examinations it holds true that the overall error was zero for the angle $t0 = 180.0$, which means the branch points strictly downwards. That is the only setting where a perfect matching of both branches could be achieved. Generally with rising growth unit id, the minimum difference also rises. But the intensity of this feature decreases with rising initial angle of deflection. This effect can be clearly seen in figure 5.3.

Another very interesting result is that in figure 5.2 some discontinuities are visible at $t0 = 103.0$ and $t0 = 142.0$. The reason for this is that the approximative method's different calculation ways can be seen. When passing a certain limit the angle of the overall deformation is calculated differently and as a result the branch curvature changes. By that the overall difference (error) also changes because the finite elements calculation are totally continuous.

Contrary to the behavior below $E \leq 6000.0$, for values $E > 6000.0$ the error is lower at the edges of the angle domain. This can be seen in figure 5.4. The minimum difference like in all other settings again increased with the growth unit's id (see again figure 5.1).

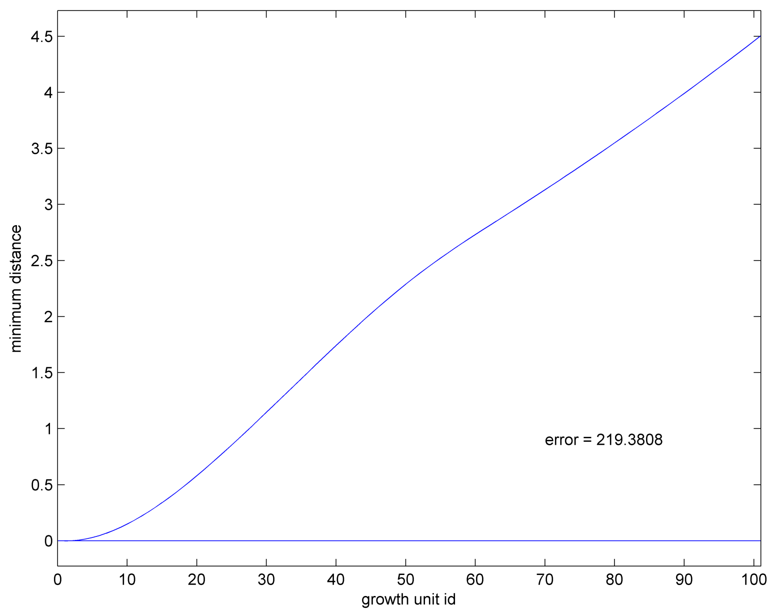

Figure 5.1: Minimum distance, *E*=800.0, *t0*=20.0

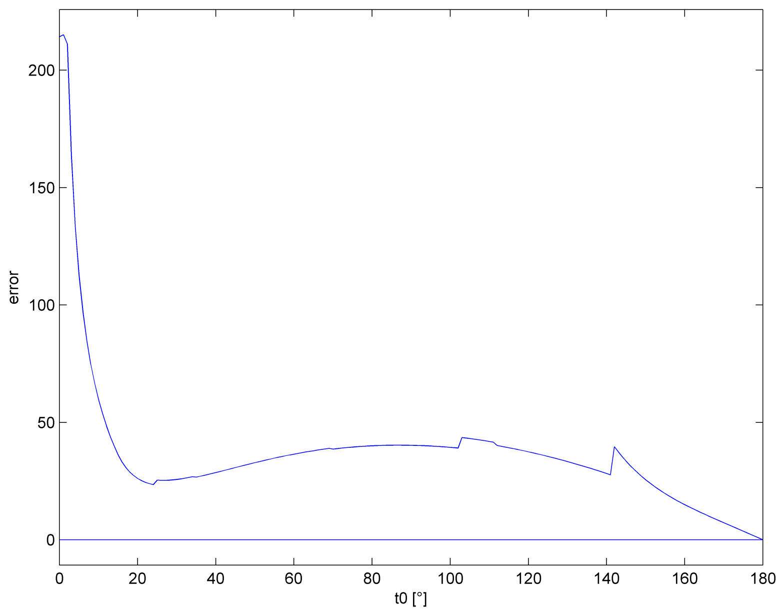

Figure 5.2: Overall error, *E*=200.0

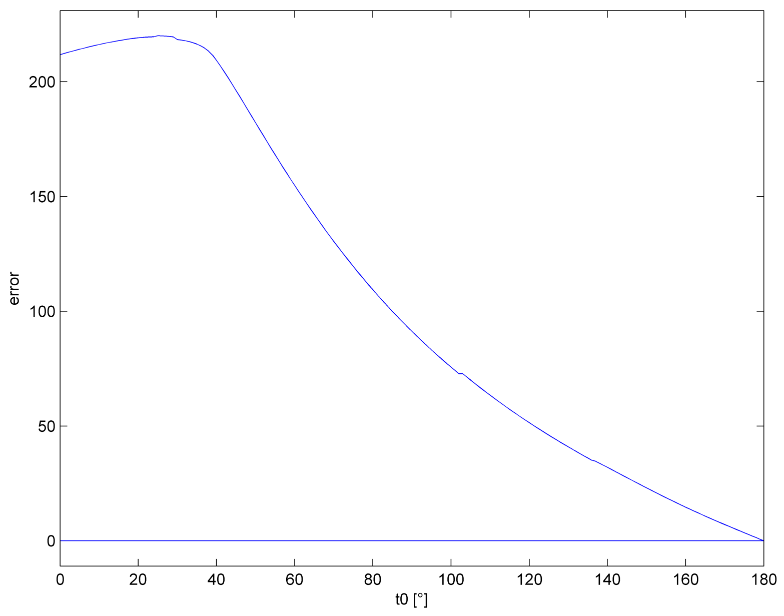

Figure 5.3: Overall error, *E*=1250.0

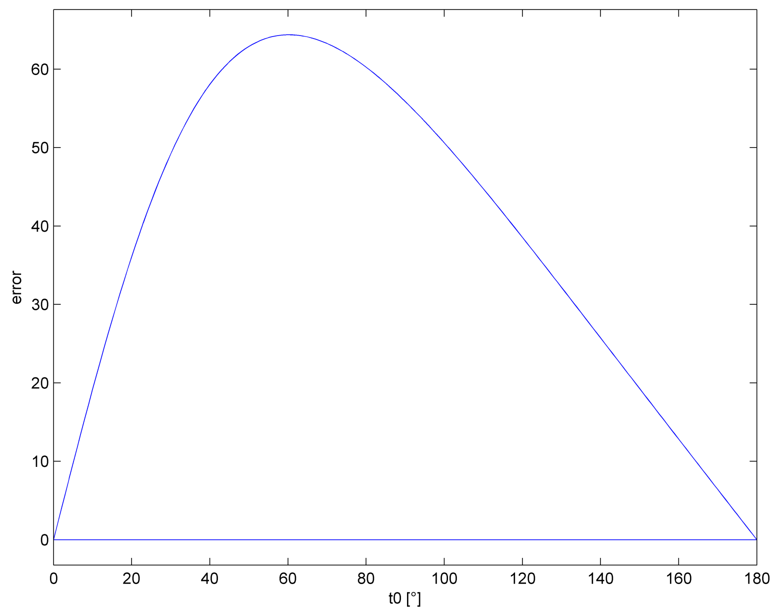

Figure 5.4: Overall error, *E*=10000.0

5.2 Parameter *grav*

The parameter $grav$ is the gravity constant. It has been examined in the range $0.0 \leq grav \leq 12.0$. As this constant sets the rate of force that acts along the branch, the parameter for the density rho has equivalent behavior as the parameter $grav$. Because of that the results for the examination of the parameter rho are equivalent. For zero gravity both algorithms deliver the same result. Thus there is no error.

Considering the error evolution with rising gravity the error decreases for nearly the whole domain of deflection angles. The highest errors occur with low gravity $grav < 5.0$ and deflection angles of less than $40.0°$. Outside this range and especially for $t0 > 20.0$ the errors are within acceptable ranges. An example can be found in figure 5.5.

Considering the minimum distance as an indicator where these differences appear, it is visible that like in the elasticity study the branches grow apart for angles $t0 < 80.0$. Depending on the value of gravity the branches intersect at some point for deflection angles $t0 \geq 90.0$. The higher the gravity the earlier the intersection takes place. This phenomenon can be seen in the comparison of figure 5.6 and 5.7. The minimum at the growth unit with the id 42, which is the intersection point, is shifted to the one with id 24. Like in the examination before some discontinuities are present as well, due to the mentioned reasons.

As a general rule the differences and errors drop with rising deflection angles. The rate of this development increases with the gravity. But the maximum error does not change at all. It remains high in the low deflection angle ranges. Nevertheless an additional slight increase can be noticed after a certain angle of deflection (for example in figure 5.5 from $t0 > 50$). But this effect is so small that it does not put the general statement into doubts.

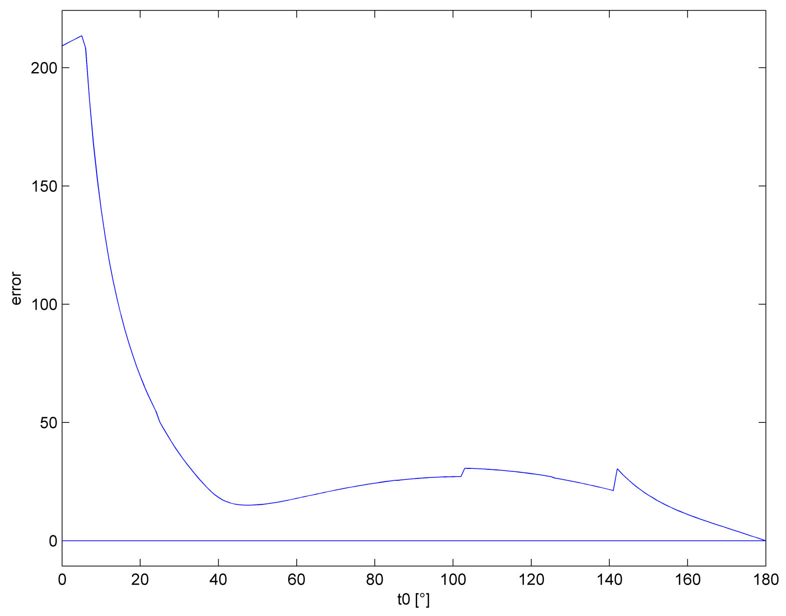

Figure 5.5: Overall error, *grav*=10.0

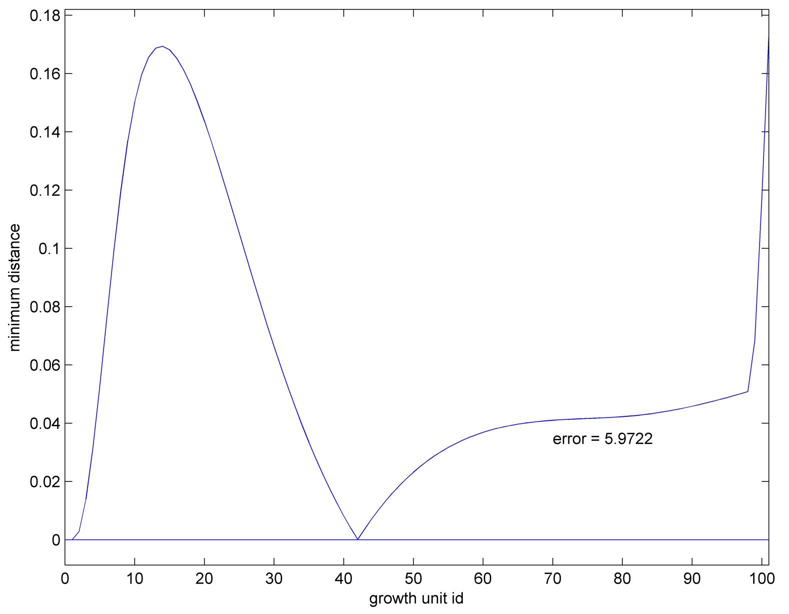

Figure 5.6: Minimum distance, *grav*=7.0, *t0*=90.0

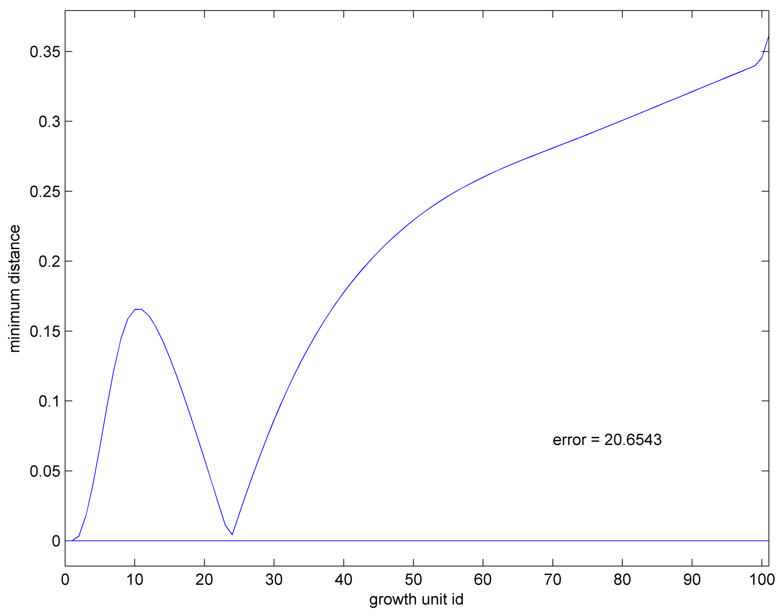

Figure 5.7: Minimum distance, *grav*=9.0, *t0*=90.0

5.3 Parameter *r_start*

The parameter r_start is the base radius of the branch. Due to the reason that the conicity parameter $coni$ forbids the start radius to be less than the end radius, a constant radius is supposed for the examination. That means $r\_end = r\_start$ and $coni = 0$. The parameter has been examined in the range $0.01 \leq r\_start \leq 0.1$. Like the parameter for the gravity $grav$ and the density rho the parameter r_start is also supposed to be responsible for the branch weight. On this account it is obvious that a similar behavior is expected. However this is not exactly what is happening.

Within the range $0.01 \leq r\_start \leq 0.02$ in the front part of the branch an intersection between both simulated shapes occurs. An example for an occurrence can be found in figure 5.8 at the growth unit id 15. The uneven appearance of the curve is due to the reason that through the little diameter the stiffness is very low and by that the finite elements branch's curvature, which points downwards, is wavy.

As a similarity to the other examinations the overall error slightly decreases with rising radius. That even holds true for the maximum error, which was not the case for the gravity parameter. For all examinations it holds true that the overall error is zero for the angle $t0 = 180.0$, which means the branch points strictly downwards. What is very interesting and contrary to the presumption is the evolution of where the errors take place. This development follows the elasticity and not the density or gravity behavior. An instance of this fact is visible in figure 5.9. The discontinuities again can be explained by the different calculation ways in the approximative algorithm.

For $r\_start \geq 0.03$ the branches do not intersect anymore. The minimum distance curve follows the trend of increasing distance for the growth units with the effect that the branches grow apart. This can be seen in figure 5.10. Nevertheless the overall error decreases with rising radius as stiffening and thus less bending happens to the branches.

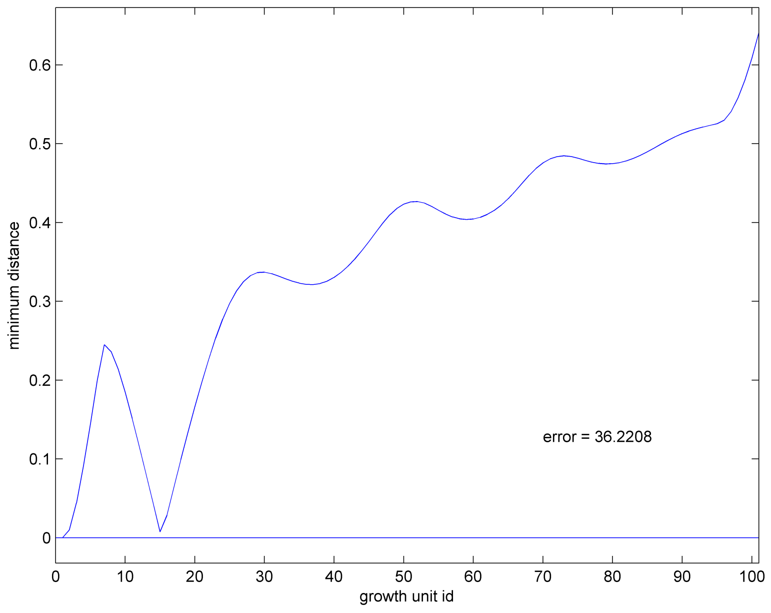

Figure 5.8: Minimum distance, *r_start*=0.01, *t0*=70.0

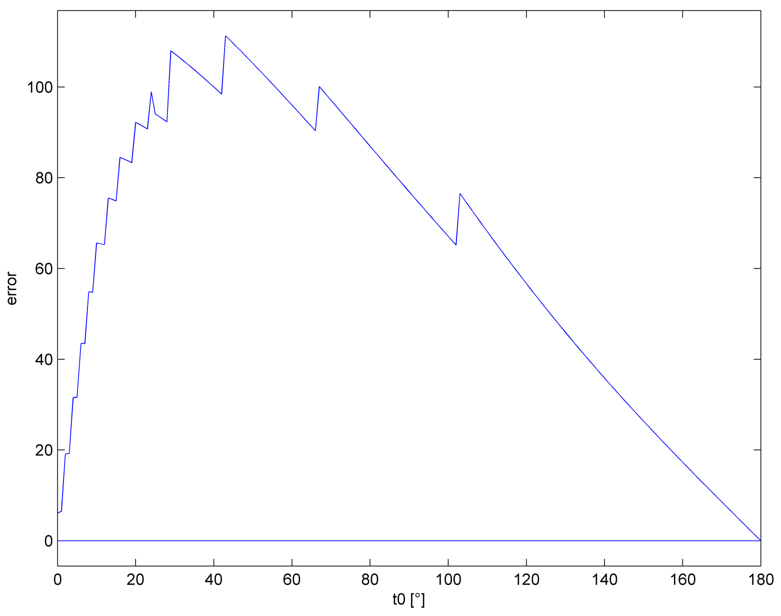

Figure 5.9: Overall error, *r_start*=0.07

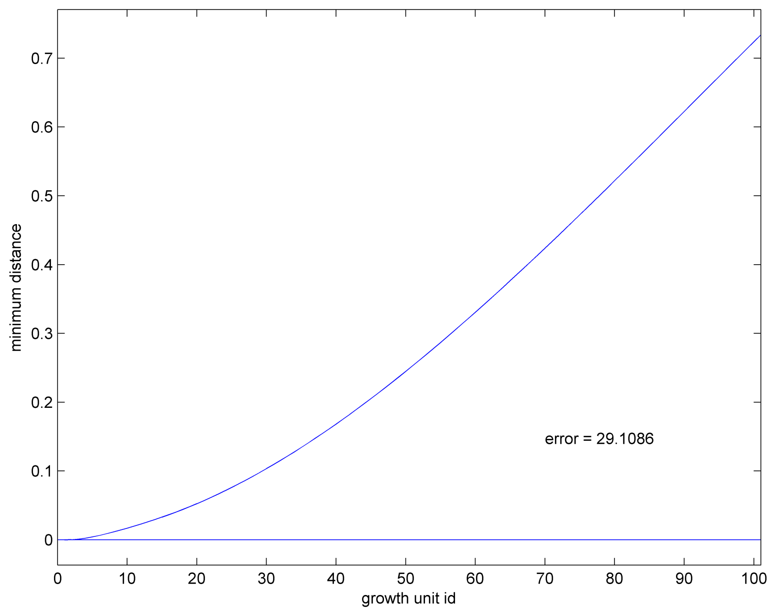

Figure 5.10: Minimum distance, *r_start*=0.06, *t0*=150.0

5.4 Parameter *r_end*

The parameter r_end is the tip radius of the branch. The parameter has been examined in the range $0.01 \leq r\_end \leq 0.1$. By fixing the parameter $r\_start = 0.1$, the maximum of the examination range, the conicity behavior of both models can be compared. As stated before the parameter $coni$ yields the branch's radii information.

For values $r\_end < 0.02$ the branches massively grow apart (see figure 5.11). The one built from the finite element algorithm points straightly upwards and the one from the approximative algorithm straightly downwards. Of all examined parameters these differences and errors are the highest ones measured. So due to the different approaches and assumptions for the algorithms in case of high conicity both deliver completely different results. Nevertheless with rising radii this tendency decreases. An example plot for the error dependency on the deflection angle can be found in figure 5.12.

Passing $r\_end \geq 0.03$ the branches' growth occurs in the same direction and as a result the error rate drops as radius increase leads to stiffening of the branch built by the finite element algorithm. For all examinations it holds true that the overall error is zero for the angle $t0 = 180.0$, which means the branch points strictly downwards.

Considering low conicities, that means $0.06 \leq r\_end < 0.1$, the overall error is about 8 times less than for high conicities. Again some discontinuities are visible for $r\_end \geq 0.04$ by the already mentioned reasons. This can be seen in figure 5.13.

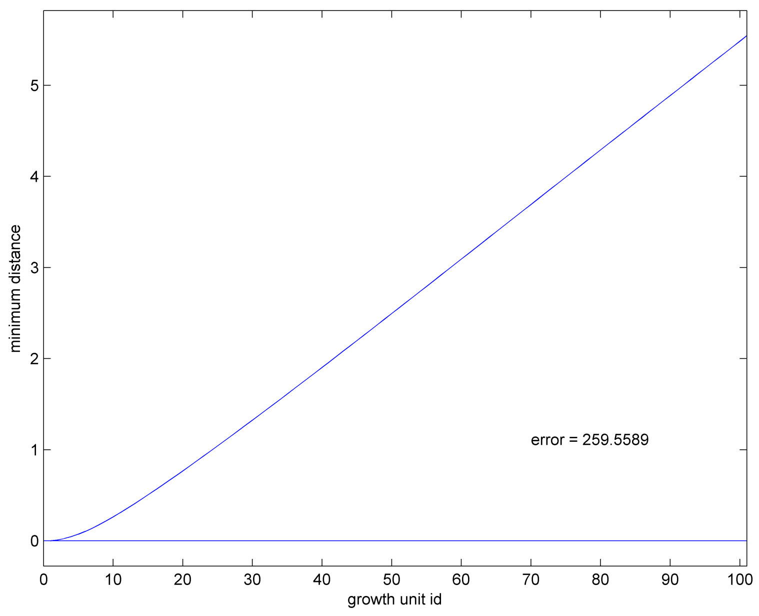

Figure 5.11: Minimum distance, *r_end*=0.01, *t0*=150.0

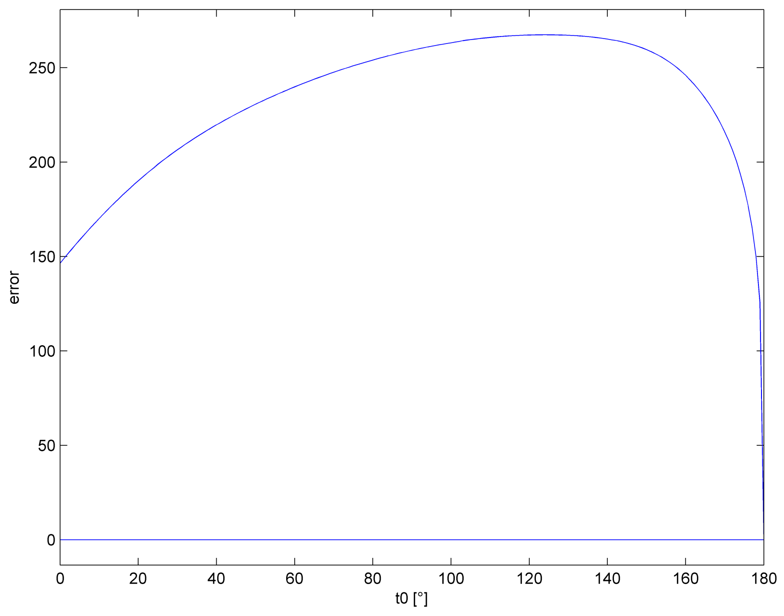

Figure 5.12: Overall error, *r_end*=0.01

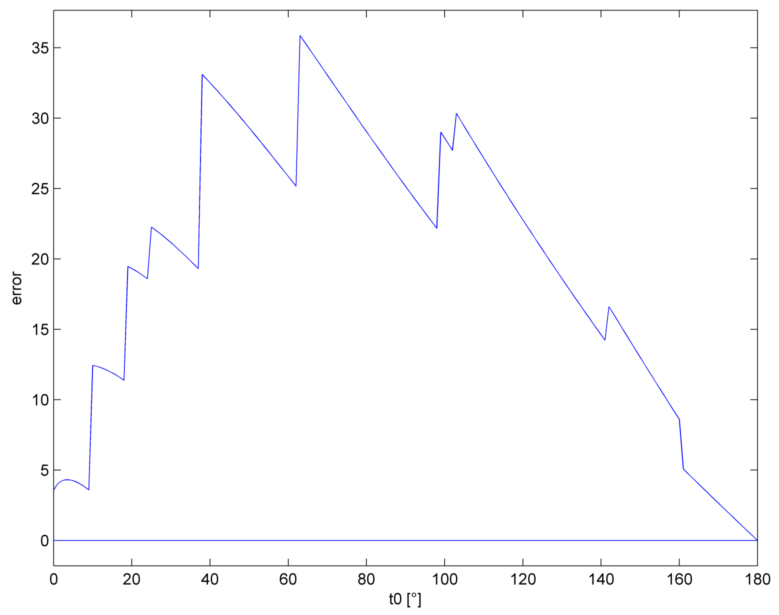

Figure 5.13: Overall error, *r_end*=0.08

5.5 Parameter *p* versus parameter *primaryTropism*

The parameter p from the approximative method is the relative point where the branch starts to invert the bending direction with the meaning of phototropism. The parameter $primaryTropism$ from the finite element method is a switch to fix the growth direction of the branch to the value of the initial angle of deflection.

The examination wants to find if $p = 0.5$ is comparable to $primaryTropism = true$. The first thing noticeable is that the way the branches bend is equivalent. What is different is the amplitude of this behavior. The finite element method's branch has much less curvature and the overall stiffness is higher. Furthermore the start and end directions of the branches also differ: The branches again grow apart (see figure 5.14). However this effect decreases with rising deflection angle. That can clearly be recognized in figure 5.15.

Identical to all other examinations is that for $t0 \geq 90$ the errors and differences are within an acceptable range though. Furthermore it holds true that the overall error is zero for the angle $t0 = 180.0$, which means the branch points strictly downwards. Finally it must be stated that the parameter p can compete with $primaryTropism$ given the right selection of the initial deflection angle.

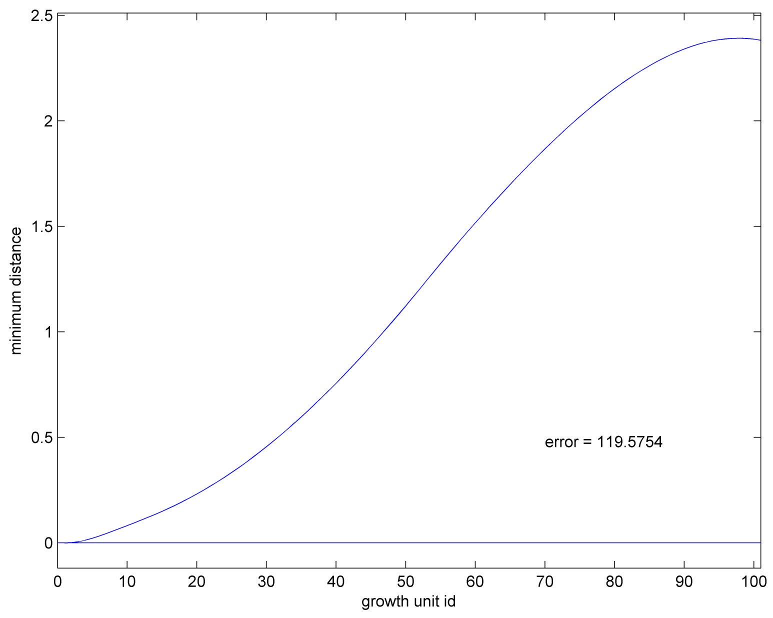

Figure 5.14: Minimum distance, *t0*=90.0, *p*=0.5, *primaryTropism* = true

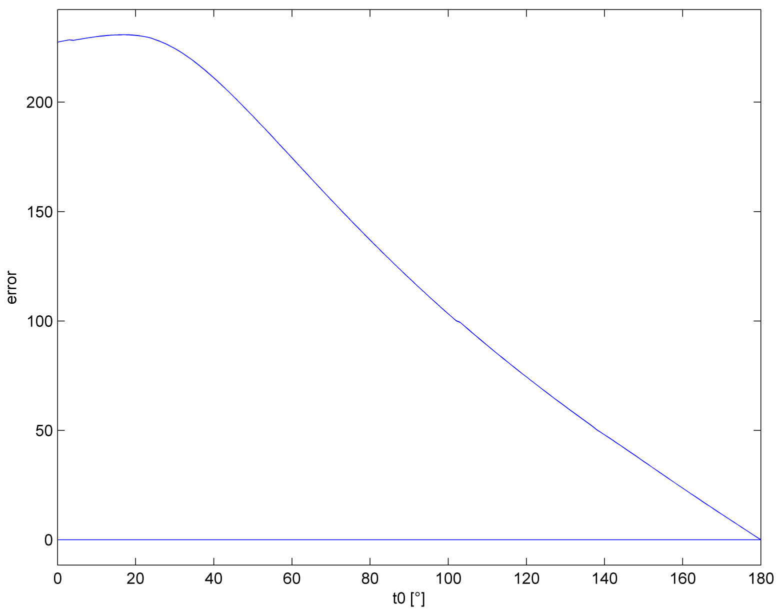

Figure 5.15: Overall error, *p*=0.5, *primaryTropism* = true

Chapter 6

Analysis of sensitivity

This chapter deals with the analysis of sensitivity for the variable parameters of both algorithms. The spruce and the sympodial tree are examined to see how sensitive the tree model's phenotype is when slightly changing the parameters within various ranges. That however does not mean it is desired to precisely evaluate in which parameter range the trees look realistic. It is about to obtain how stable the overall structure is with respect to the parameters of the model and determine the dependency for measurable tree appearance readings namely the maximal elongation in x, y and z direction and the number of discretization steps. All parameters were tested separately. Considering the maximal elongation, due to the fact that the tree models are more or less radially symmetric it is supposed that the plots for the x-axis and y-axis correlate. In all plots the corresponding examination range can be read off the abscissa. Regardless of the examined parameter the other values of the parameters for the different trees can be found in chapter 4.

6.1 Spruce tree

6.1.1 Force-applying approximative method

This topic has been accomplished by the author in a previous work and can be found in [15][pp. 31-36].

6.1.2 Incremental finite element method

6.1.2.1 Parameter *E*

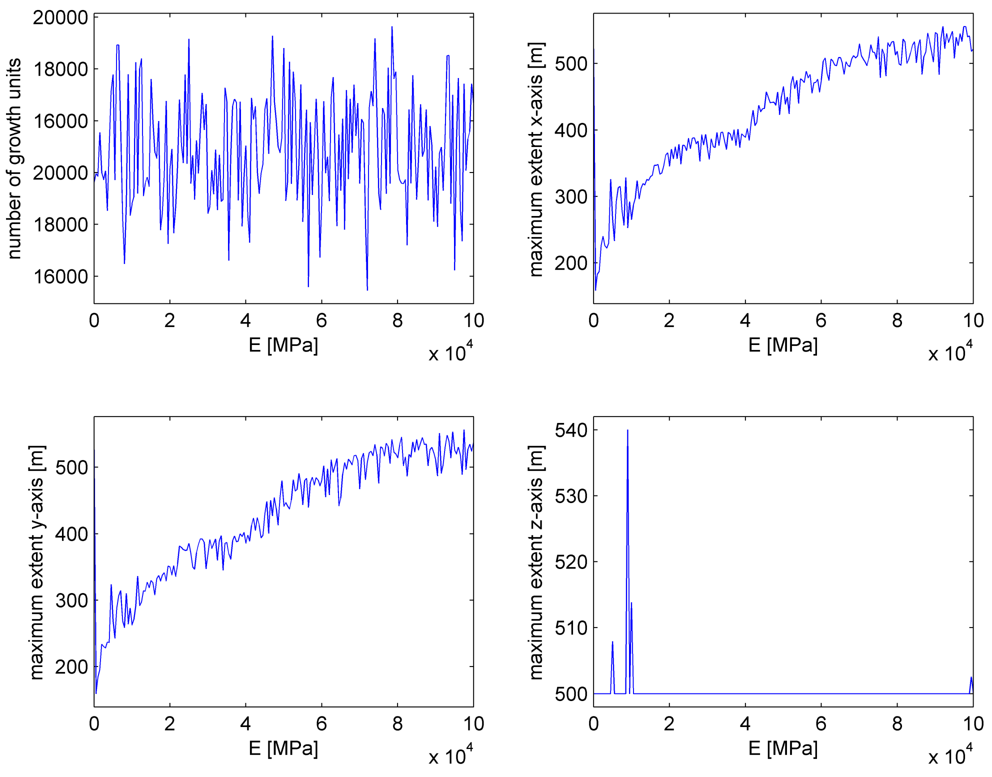

Figure 6.1: Sensitivity for the spruce tree, parameter *E*

The parameter E is the Young's modulus constant. In the examination the first unusual thing noticeable is a stochastic vibration through all plot curves. This is due to the stochastic behavior in the spruce tree model. The number as well as the height of branches are determined within a random range. That is why a dependency between the number of growth units, which is equivalent to the level of discretization, and the Young's modulus is not visible. However such behavior was not even expected.

Nevertheless the elongation curves' vibration does not hide the general trend for the x-axis and y-axis extent: An approximately square root increase in elongation occurs with rising elasticity constant. The maximum x-y-plane extent occurs parallel to the maximum of the examination range with $E = 100000.0$. This is exactly what was supposed. The overall stiffness increases with higher elasticity values. Because of the angle near to orthogonality between stem and branch it leads to flattening of the branch and hence maximal extent is reached.

Considering the tree's height evolution, represented by the maximal z-axis extent, it is clear that this value only varies marginally. The ranges with constant value are due to the fact that the height contribution mainly is from the stem. By that the z-axis development from the branches is hidden. The peaks in the plot curve are due to different branch arrangements derived from the stochastic behavior of the spruce tree.

6.1.2.2 Parameter *grav*

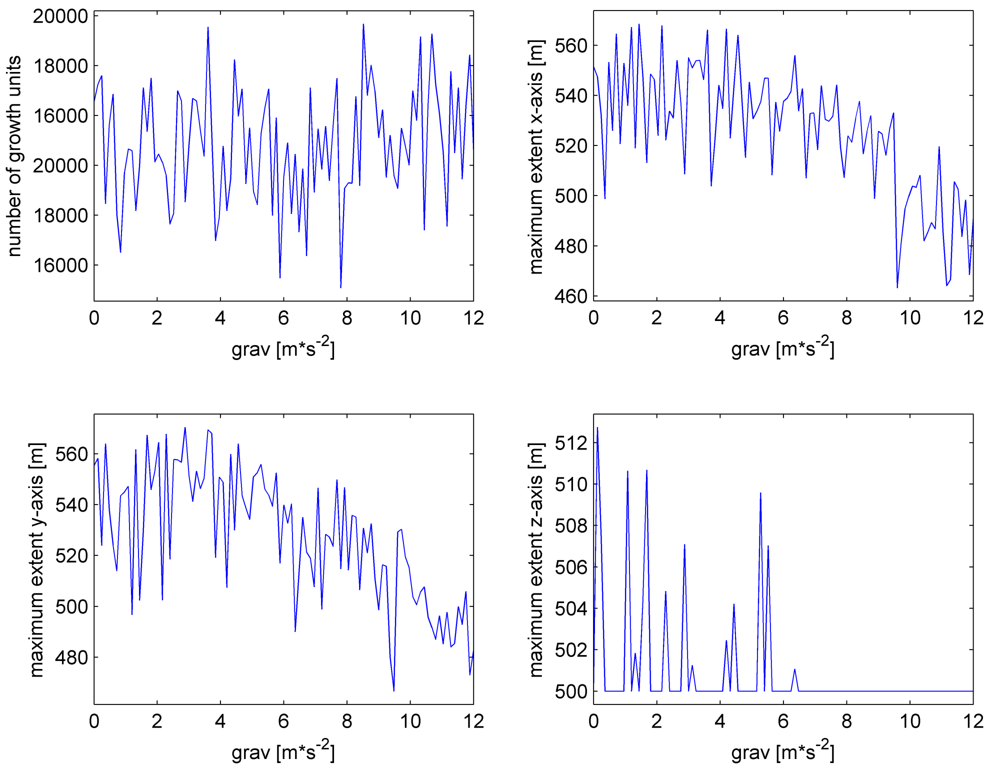

Figure 6.2: Sensitivity for the spruce tree, parameter *grav*

The parameter $grav$ is the gravity constant. Hence this parameter is also responsible for the branch weight. It must be noticed that its behavior is equivalent to the density parameter. That is why the examination of the gravity constant is sufficient. What again leaps out is the stochastic vibration of the curves. There is no dependency between the gravitation and the number of growth units as expected.

The maximal x-axis and y-axis extents show a decrease with rising gravity. This has the simple reason that when the weight increases with rising gravity the branches become heavier and by that sag more downwards. Because of that their main curvature slightly shifts to the z-axis direction. Compared to the elasticity constant the relative extent differences for the gravity are significantly lower. The relative difference for the gravity is about 30%, whereas the value for the elasticity is around 150%. Because of that in this setting the sensitivity is much less which is a very interesting result.

The evolution of the extent along the z-axis again just shows the random behavior of the spruce tree. Nevertheless it should be noted that the z-axis extent also measures the negative contribution to the maximal range along the z-axis. This happens in the case when the branches sag downwards and the tips cross the zero point towards the negative z-axis direction. Relatively the z-axis behavior only fluctuates about 3%. Hence the overall effect is more or less negligible.

6.1.2.3 Parameter *n_FE*

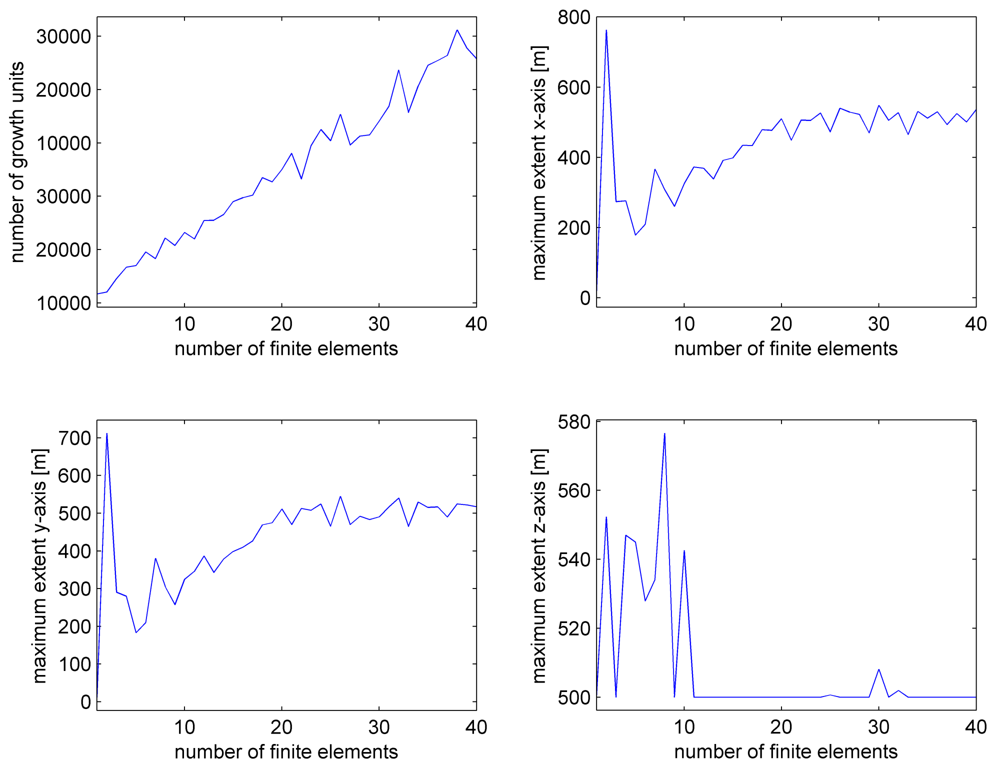

Figure 6.3: Sensitivity for the spruce tree, parameter *n_FE*

The parameter n_FE is the number of finite elements. The number of growth units which is equivalent to the amount of discretization directly correlates with the number of finite elements as expected. This also holds true because interpolation is switched off. This is done by just setting the value of the parameter n_GU fixed to n_FE. The vibration again represents the stochastic behavior of the spruce model as a variation in the number of branches occurs.

The maximal extent in the x-y-plane initially stabilizes after $n\_FE \geq 10$. The reason for this is that if $n\_FE < 10$ the calculations for solving the linear equation system fail and the resulting rotation angles are inaccurate. The same phenomenon can be seen in the z-axis plot. The maximum elongation considerably exceeds the normal stochastic variation effect (3% versus 16%). Just after passing $n\_FE \geq 10$ the normal behavior is applicable again.

Considering sensitivity in the range $10 \leq n\_FE \leq 20$, a linear dependency for the x-y-plane maximum extent can be seen. In that range the branches gain stiffness and the maximum elongation goes up. Afterwards for $n\_FE > 20$ the effect almost is negligible or even disappears. What this examination also demonstrates is the importance of the right choice of discrestization amount given by the number of finite elements. Too low values lead to solutions with high inaccuracies.

6.1.2.4 Parameter *sigma*

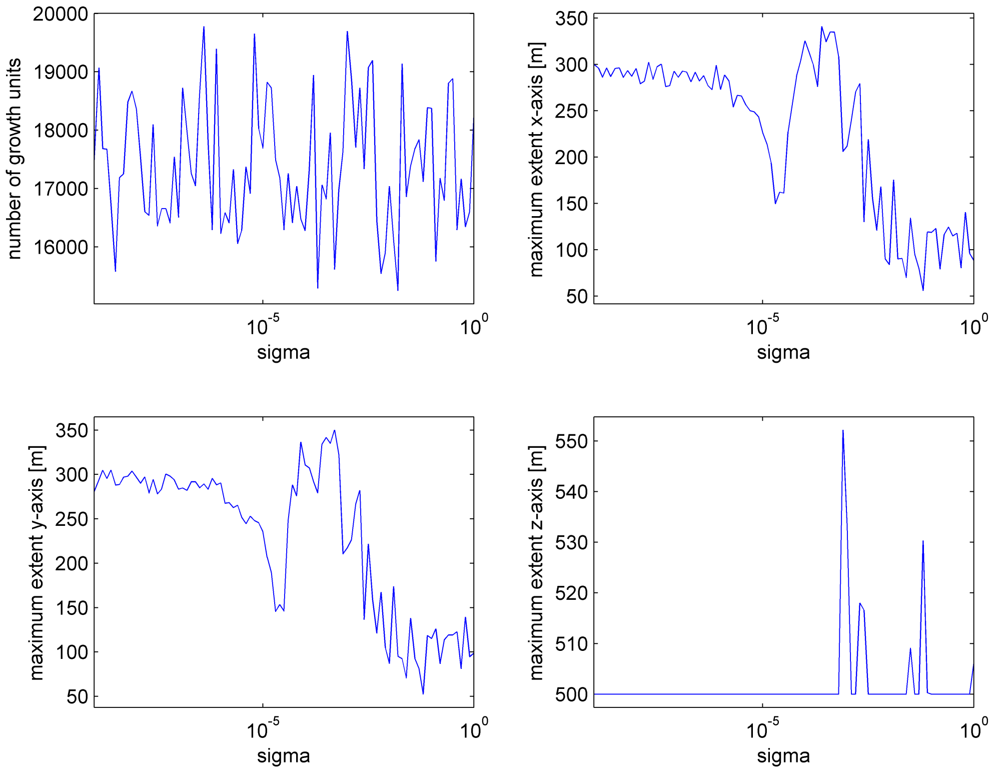

Figure 6.4: Sensitivity for the spruce tree, parameter *sigma*

The parameter $sigma$ controls the sensitivity of the reaction wood formation process initiated by maturation strain. In the examiniation a very high sensitivity was recognized for high values of $sigma$. Because of that for the abzissa axis a logarithmic scale has been chosen for better visibility of the behavior in the lower range for $0 \leq sigma \leq 10^{-4}$. The number of growth units does not follow some trend. It reflects again the randomness of the spruce model.

The maximal x-axis and y-axis extent plots reveal a little decrease in the range $0.0 \leq sigma \leq 10^{-6}$. There a just marginal drop in elongation occurs and hence the sensitivity in that range is very low. However after passing the extent is cut in half within the range $10^{-6} \leq sigma \leq 10^{-4}$. Then, for $sigma > 10^{-4}$ a monotonic decrease can be read off the measurements. Nevertheless from then on the effect of the parameter $sigma$ on the maximal elongation in the x-y-plane heavily fluctuates. The reason for this is that the reasonableness of the calculation results for too high strengths of the reaction wood formation process is not given anymore.

This is also the reason for the behavior of the maximal elongation along the z-axis. The effect from the stochastic vibration is much lower than the effect in the plot caused by too high values of $sigma \geq 10^{-4}$ (3% versus 10%). Another conclusion is that the parameter $sigma$ must be chosen carefully with also considering the other branch properties in order to achieve reasonable calculation results.

6.1.2.5 Parameter *phi*

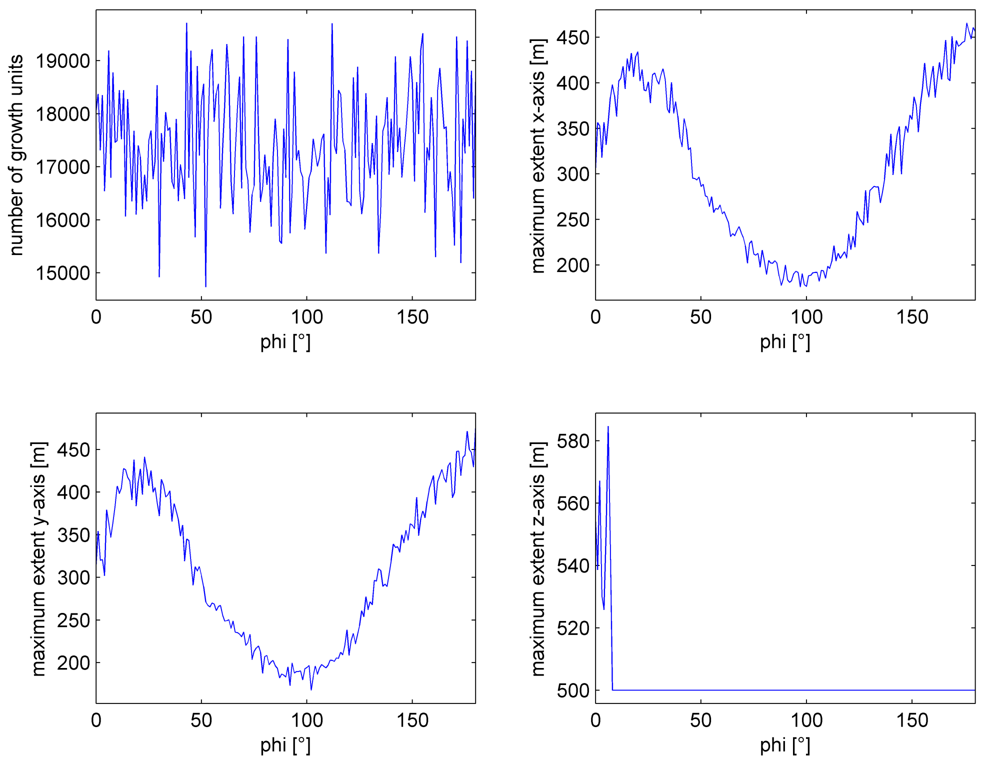

Figure 6.5: Sensitivity for the spruce tree, parameter *phi*

The parameter phi sets the angle for the direction of the reaction process controlled by $sigma$. Again there is no dependency between the number of growth units and phi. The plots for the elongation within the x-y-plane show a really nice sinusoidal curve. This is exactly what was expected.

For the value $phi = 0.0$ the reaction direction is equivalent to the branches' growth direction. Hence it is not surprising that the first maximum in elongation can be found for circa $phi = 20.0$. That is the setting where the reaction wood formation supports the branches so that they can reach their maximal x-y-plane extent by lowering the overall bending.

The same reason leads to the jump in z-axis elongation as little branches that did not bend as much as the bigger ones experience more support in the direction set by phi. Thus their tips point upward and so the maximal extent along the z-axis is not only given by the stem height but also by branches overtowering the stem. After the first maximum the curve falls to the minimum at circa $phi = 100.0$. But such value would indicate that the tropism source is underneath the tree which is unrealistic.

At the end it must be stated that the parameter phi possesses no inconsistency in sensitivity as it works exactly as it is meant to do. Depending on the tree settings, particularly the initial angle of deflection and by that the growth direction of the branches, the parameter phi in combination with $sigma$, a and b considerably influences the branch bending and tropism reaction.

6.1.2.6 Parameter *a*

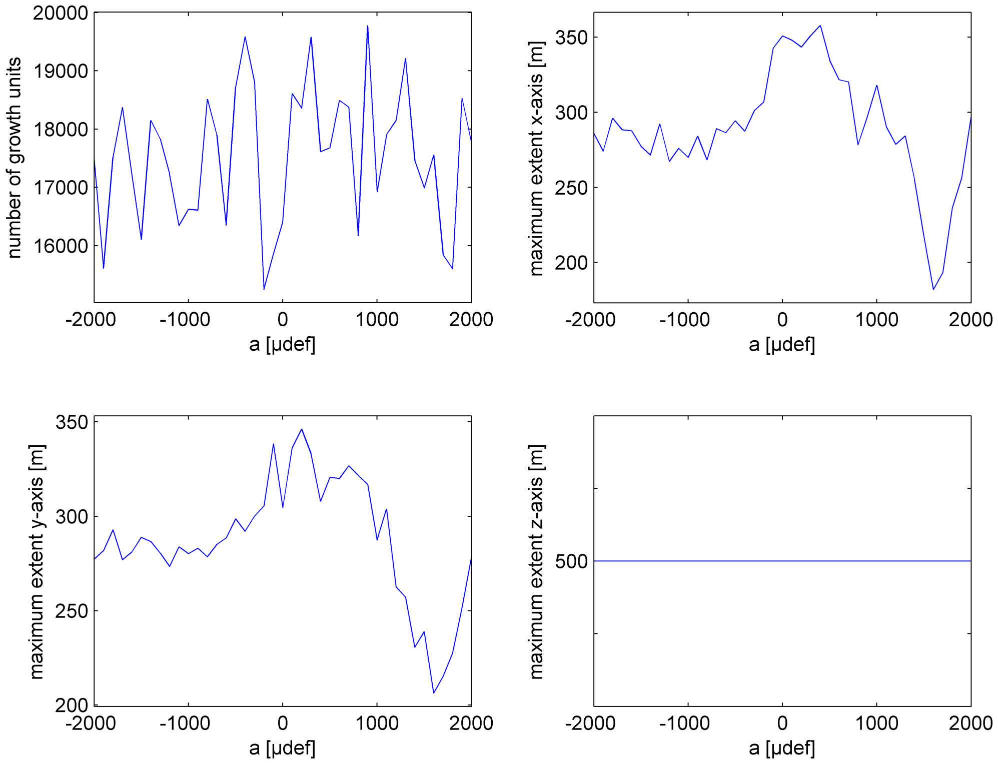

Figure 6.6: Sensitivity for the spruce tree, parameter *a*

The parameter a sets the minimum value of maturation strain. There is no dependency between the parameter a and the number of growth units as expected. The curves for the maximal elongation along the x-axis and y-axis own one maximum and minimum. Starting the examination with $a = -2000.0$ the maximum is reached at circa $a = 0.0$. Nevertheless the strength of the parameter in the range $-2000.0 \leq a \leq 0.0$ is not as much as it is for $a > 0.0$.

The minimum in elongation can be seen at $a = 1600.0$. In the range $0 \leq a \leq 1600.0$ the maximal extent drops about 50%. After that the maximal extent in the x-y-plane goes up to the same value as for $a = -2000.0$.

The z-axis curve just shows a flat line. This means the stem height is the only contribution to this value. Considering the overall sensitivity is must be stated that for positive values it is much higher than for negative values.

6.1.2.7 Parameter b

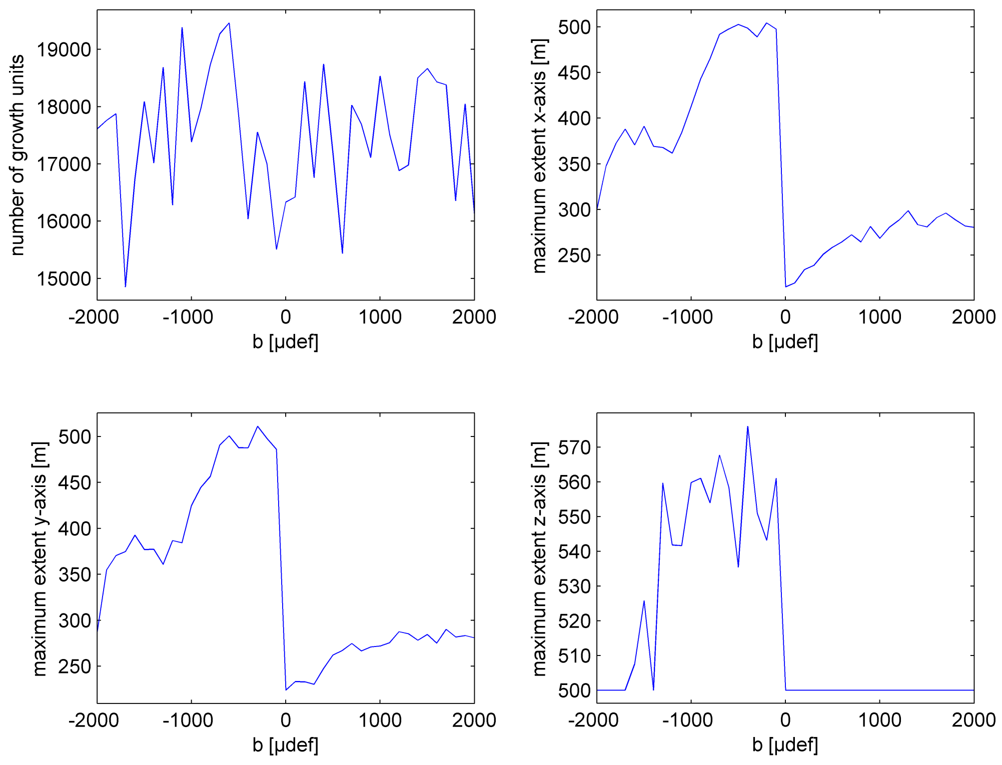

Figure 6.7: Sensitivity for the spruce tree, parameter b

The parameter b sets the maximum value of maturation strain. The vibration in the first plot again represents the stochastic behavior of the spruce model. In the x-axis and y-axis maximal elongation plots a perspicuous discontinuity is visible at $b = 0.0$.

It divides the examination range into two sections where a monotonic slope occurs. In the first section $-2000.0 \leq b \leq 0.0$ the overall level of x-y-plane extent is significantly higher than in the section afterwards. Additionally the gradient as the rate of change for the curve is also higher. At the point $b = 0.0$ which is the minimum the maximal elongation drops more than about 50%.

In the section $0.0 \leq b \leq 2000.0$ the level of x-y-plane elongation remains low and the increase while reaching $b = 2000.0$ is noticeably less than in the section before. The behavior described above also is reflected in the maximal elongation along the z-axis. Corresponding to the first section $-2000.0 \leq b \leq 0.0$ an increase in extent can be seen. It is not exclusively related to the stochastic vibration of the spruce tree but mainly correlates with the parameter b. After passing $b = 0.0$ the elongation in z-direction stays constant.

6.2 Sympodial tree

6.2.1 Force-applying approximative method

This topic has been accomplished by the author in a previous work and can be found in [15][pp. 37-42].

6.2.2 Incremental finite element method

6.2.2.1 Parameter *E*

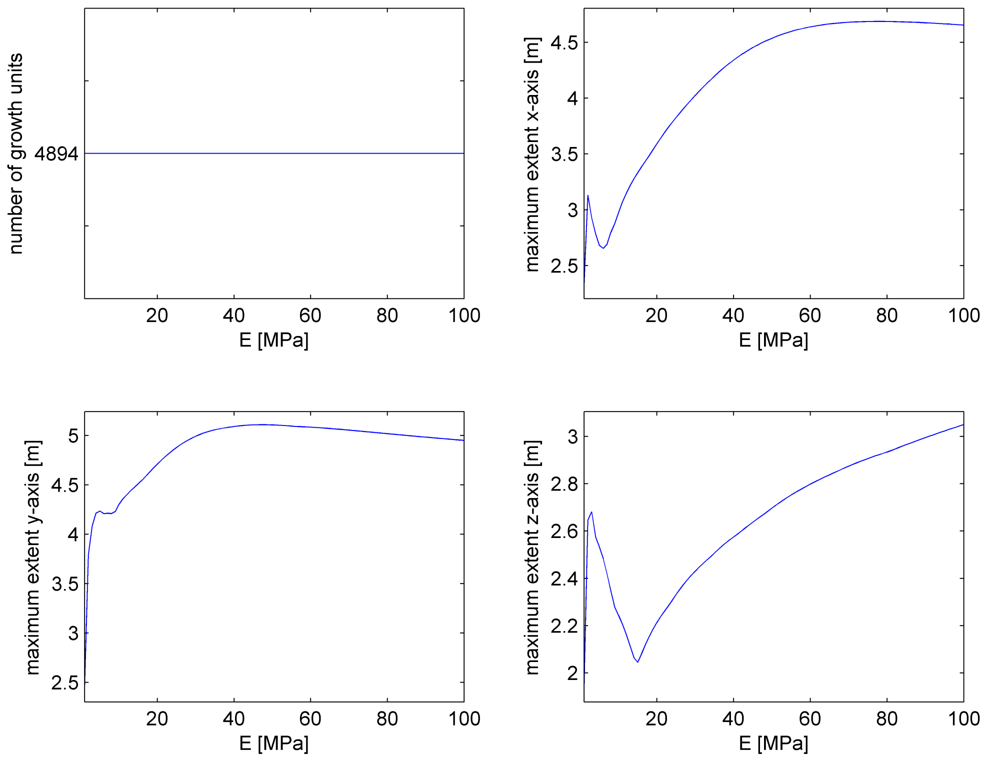

Figure 6.8: Sensitivity for the sympodial tree, parameter E

The parameter E is the Young's modulus constant. Due to the reason that the sympodial tree does not contain any randomness the number of growth units is constant. What is also visible is that the curves are much smoother in contrast to the spruce tree.

In the plots for the maximal extents along the x-axis and y-axis for $E < 10.0$ a fluctuation can be recognized. After that the curve stabilizes. The reason for this discontinuity are instabilities in the calculation of the branch as the elasticity constant is too low and by that the overall stiffness firstly steadies after passing $E = 10.0$. Moreover the plots for both axes do not match entirely. This is due to the fact that the sympodial tree is not fully radially symmetrical. It is a binary structure and as a result both axes experience a little different maximal extent evolution. For $E \geq 10.0$ both plots contain a maximum which indicates that the overall stiffness can not increase anymore. By that the tree is equivalent to its original construction where no bending occurs.

The development of the extent along the z-axis resembles the x-y-plane behavior. This is because the stem height is overtowered completely by the branches and additionally the branches' initial angles of deflection are $30°$. That is why the tree's dimensioning affects all spacial directions and so the z-axis as well.

6.2.2.2 Parameter *grav*

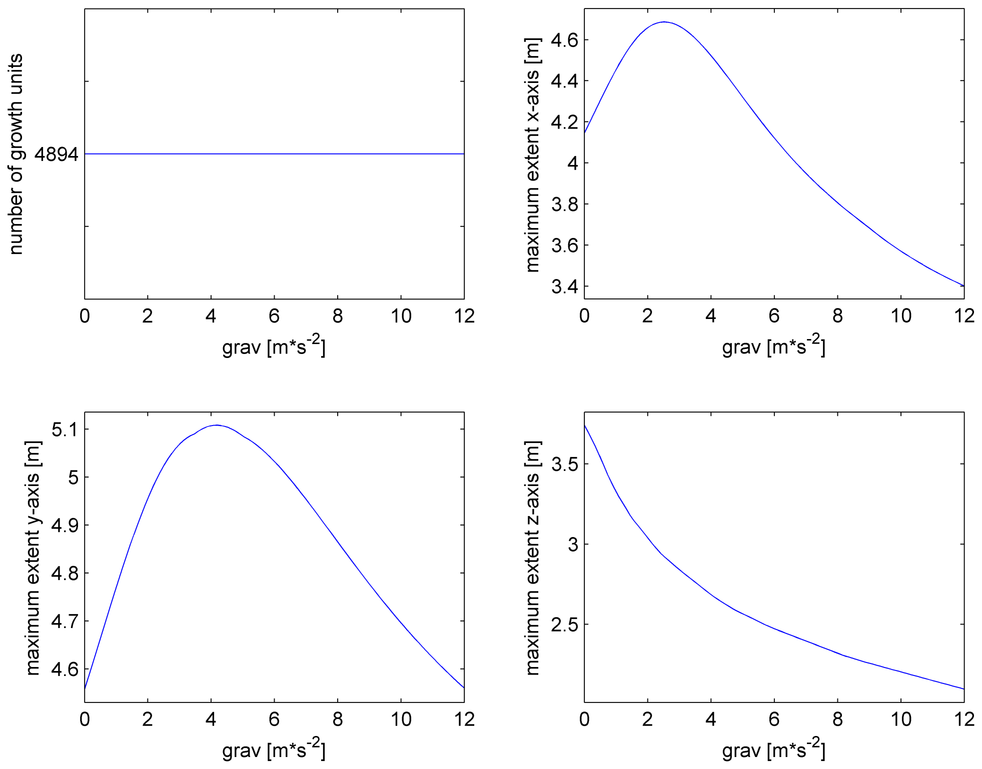

Figure 6.9: Sensitivity for the sympodial tree, parameter *grav*

The parameter $grav$ is the gravity constant. It is directly proportional to the parameter rho. As a result this parameter directly influences the bending behavior of the branches. The number of growth units is constant as expected.

The x-y-plane extent diagrams show an increase with rising gravity until reaching the maximum and afterwards an approximately linear drop in elongation. The position where the extremes occur depend on the tree construction. Considering the sympodial tree the curves are exactly like expected. The results for zero gravity can be discarded as the elongation mainly comes from the branch structure itself and the binary emerging angles of ramification. With rising gravity until reaching the maximum $grav = 2.5$ (x-axis considered only) the branches′ curvature increases and thus the bending goes up and the branches sag downwards until the maximal extent is reached.

Then for $grav \geq 2.5$ the elongation decreases with rising gravity. This is due to the reason that the growth of the branch tips occurs in the downward direction. The same effect also lets the extent in z-direction decrease with rising gravity throughout the whole examination domain.

6.2.2.3 Parameter *n_FE*

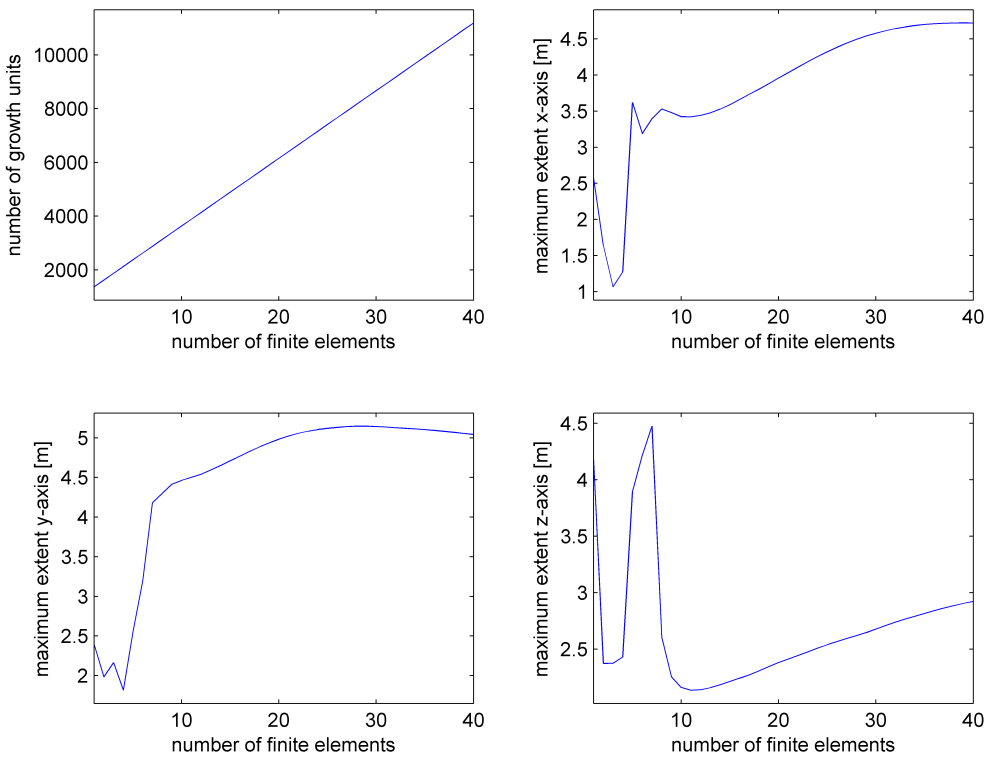

Figure 6.10: Sensitivity for the sympodial tree, parameter *n_FE*

The parameter n_FE is the number of finite elements. This parameter directly correlates with the number of growth units and possesses a linear dependence. This is the case because the number of growth units per branch as an interpolation option, given by n_GU, is fixed to the number of finite elements.

Considering the maximal extent along the x and y-axis it is clearly visible that the curve initially stabilizes after passing $n\_FE = 10$. This is due to calculation inaccuracies when solving the linear equation system. Subsequently with rising number of finite elements the elongation in the x-y-plane just increases about 30%. Another interesting fact is that these curves likely contain a maximum which indicates the accuracy can not be enhanced anymore by increasing the number of finite elements.

The same train of thoughts holds true for the extent in z-direction. For values $n\_FE < 10$ the curve just shows the inaccurate calculation results. After that the increase with rising number of finite elements is slowing down up to the end of the examination range at $n\_FE = 40$.

6.2.2.4 Parameter *sigma*

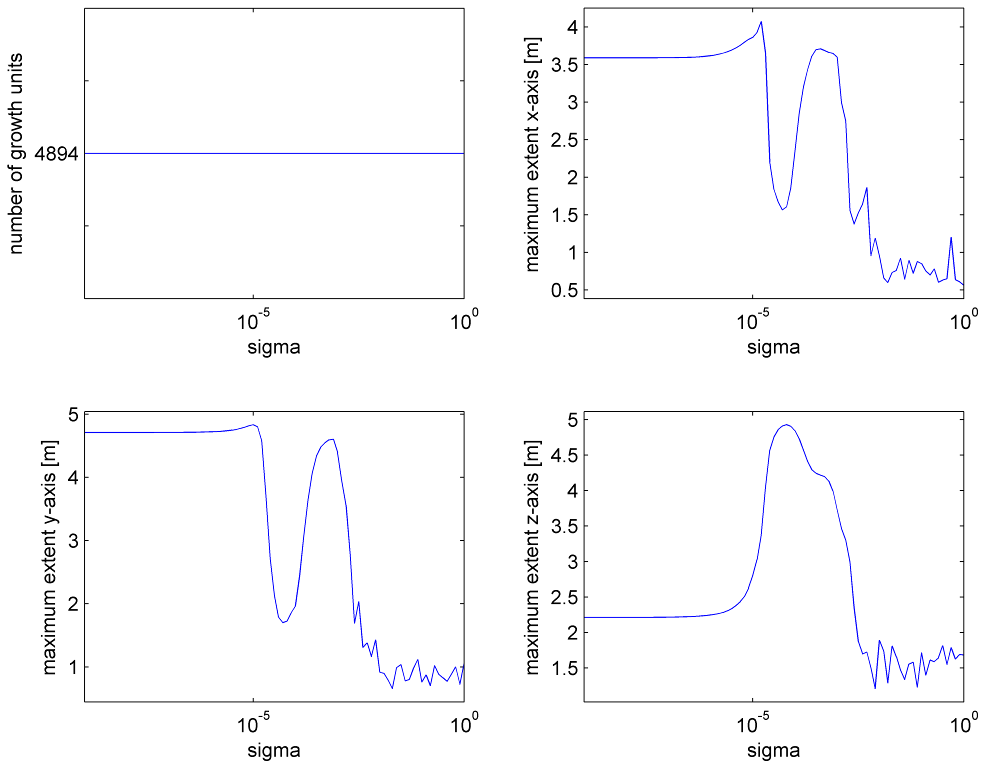

Figure 6.11: Sensitivity for the sympodial tree, parameter *sigma*

The parameter $sigma$ controls the sensitivity of the reaction wood formation process initiated by maturation strain. In the examiniation a very similar behavior like in the spruce tree has been observed. Due to a very high sensitivity for high values of $sigma$ for the abzissa axis a logarithmic scale has been chosen for better visibility of the behavior in the lower range for $0 \leq sigma \leq 10^{-4}$. The number of growth units does not follow any trend. It is constant as expected.

The plots for the maximal extents in the x-y-plane show a marginal increase in the range $0.0 \leq sigma \leq 10^{-5}$. There just a little jump in elongation occurs and hence the sensitivity in that range is very low. However after passing 10^{-5} the extent is cut in half within the range $10^{-5} \leq sigma \leq 10^{-4}$. After jumping high to the previous value for $sigma > 10^{-4}$ a monotonic decrease can be read off the measurements. Nevertheless from then on the effect of the parameter $sigma$ on the maximal elongation in the x-y-plane fluctuates. The reason for this is that the reasonableness of the calculation results for too high strengths of the reaction wood formation process is not given anymore.

This is also the reason for the behavior of the maximal elongation in z-direction. For values in the range $sigma < 10^{-4}$ the curves go the inverted way. This is due to the sympodial tree's initial angles of deflection. Finally it again shows that the parameter $sigma$ must be chosen carefully with also considering the other branch properties in order to achieve reasonable calculation results.

6.2.2.5 Parameter *phi*

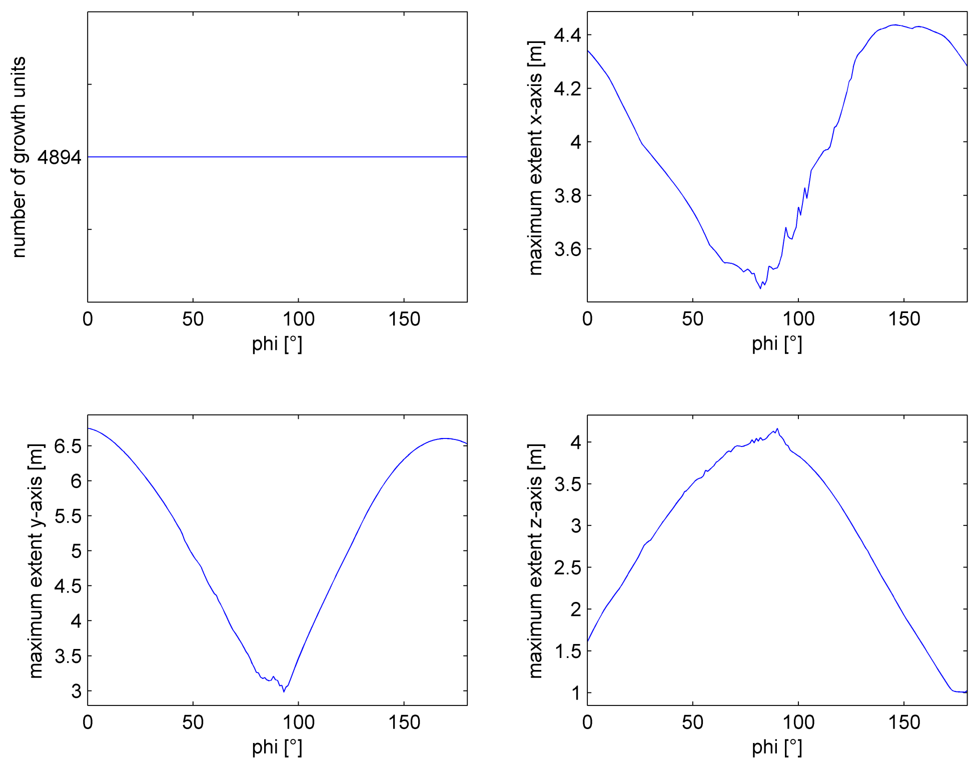

Figure 6.12: Sensitivity for the sympodial tree, parameter *phi*

The parameter phi sets the angle for the direction of the reaction process controlled by $sigma$. A dependency between phi and the number of rotation angles can not be seen. Like in the spruce model a sinusoidal relation for the maximal extent in the x-y-plane is visible.

For the value $phi = 0.0$ the first maximum can be read off. In that case the reaction direction is $90°$, measured relatively to the global z-axis. That is the setting where the reaction wood formation forces the branches to grow in the horizontal direction and by that they can reach their maximal elongation in the x-y-plane. However the minimum correspondingly would have to occur at $phi = 90.0$. This is exactly what is happening.

The same reason leads to opposite behavior in z-direction. When the growth direction is vertical that means $phi = 90.0$ and thus the branches' tips point upward and so the maximal extent along the z-axis is reached. At the borders of the examination range the two minima can be found accordingly.

Summarizing it must be stated that the parameter phi has no inconsistency in sensitivity. In combination with $sigma$, a and b and depending on the tree settings, particularly the initial angles of deflection and by that the growth direction of the branches, the parameter phi considerably influences the growth direction, the bending and the tropism reaction.

6.2.2.6 Parameter *a*

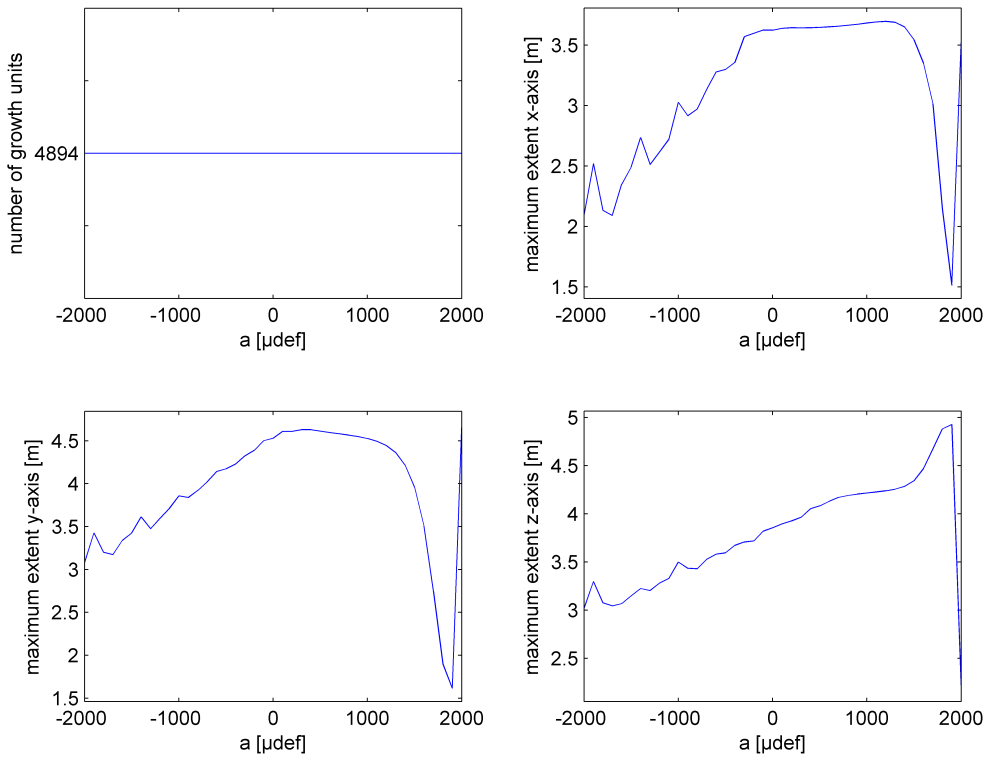

Figure 6.13: Sensitivity for the sympodial tree, parameter *a*

The parameter a sets the minimum value of maturation strain. The number of growth units remains constant throughout the examination. The plots for all spacial directions show similarities.

For almost the whole range an increase in extent can be recognized. The point at $a = 0.0$ again represents a division mark for the examination range.

For $-2000.0 \leq a \leq 0.0$ an approximately linear increase in extent in the x-y-plane takes place. Afterwards in the range $0.0 \leq a < 1400.0$ the curve nearly stays flat, which indicates the maturation strain effect has reached its maximum. But eventually for $a > 1800.0$ the extent dramatically drops to the minimum at $a = 1900.0$. This could indicate that the calculation results are no more within reasonable ranges and the overall stability of the linear equation system collapses.

The same behavior leads to the appearance of the curve for the maximal elongation in z-direction. The difference is that at the point $a = 1900.0$ the drop occurs with a much higher gradient than in the x-y-plane plots. Hence before $a < 1900.0$ an increase in extent can be noticed. That supports the thesis of the maximal achievable maturation strain effect as the maximal possible straightening up of the branches.

6.2.2.7 Parameter b

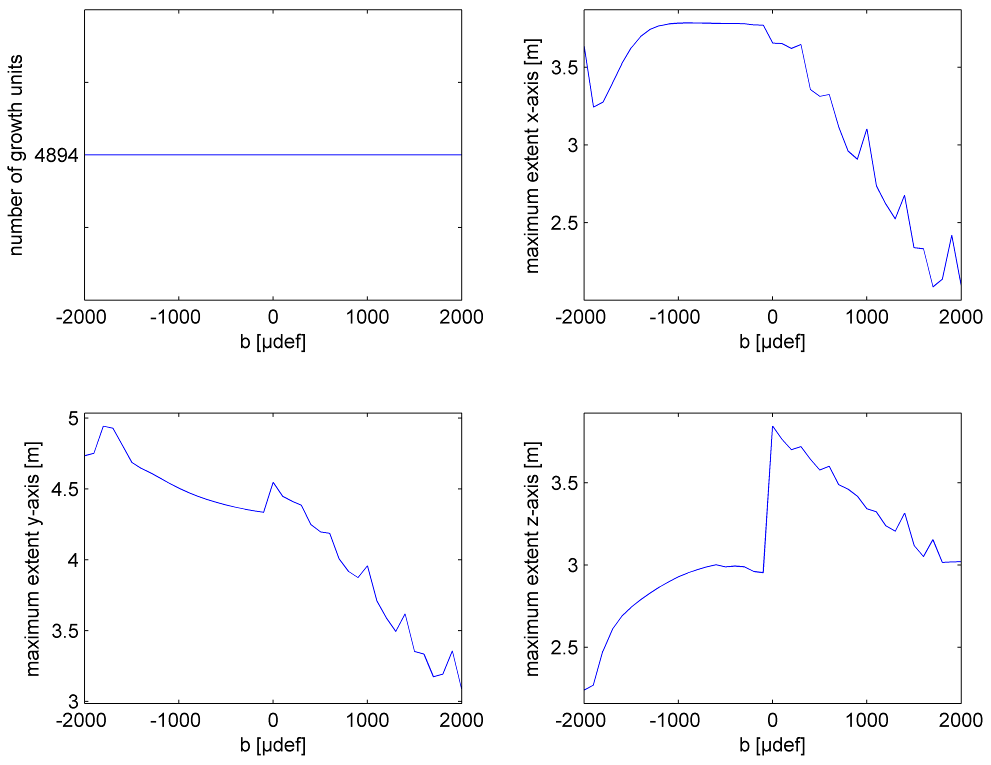

Figure 6.14: Sensitivity for the sympodial tree, parameter b

The parameter b sets the maximum value of maturation strain. A constant number of growth units can be read off the measurements. The value $b = 0.0$ in all three plots represents a point that divides the examination range in two sections of different behavior.

The x-axis plot differs from the y-axis plot in the range $-2000.0 \leq b \leq 0.0$. The curve for the maximal x-axis extent just stabilizes after passing $b = -1900.0$. Then a little increase in extent can be recognized. But in the range $-1000.0 \leq b \leq 0.0$ the curve stays flat. However the y-axis diagram shows a decrease with rising b in the range $-2000.0 \leq b \leq 0.0$. For values $0.0 \leq b \leq 2000.0$ in the whole x-y-plane an approximately linear decrease is visible.

In the maximal elongation in z-direction a discontinuity can clearly be seen at $b = 0.0$. In the range $-2000.0 \leq b \leq -100.0$ a decelerated increase occurs. After that the jump to the maximum elongation occurs at $b = 0.0$. Then in the range $0.0 \leq b \leq 2000.0$ a nearly linear decrease can be seen. This behavior is equivalent to the one in the x-y-plane.

Chapter 7

Runtime and memory consumption

In this chapter the runtime and the memory consumption of the two algorithms are examined. For the examination both algorithms were tested without any application to tree structure or ramification. That means just the plain functions developed in chapter 3 were run. The only dependency between the runtime or memory consumption and the function parameters was found for the discretization parameter $disc$ and the number of finite elements n_FE. It must be noted that the measurements for the runtime in every setting was repeated 200 times in order to achieve valid and reproducible results. The Java function *nanoTime()* was used for this purpose. Due to the reason that the language XL is Java-based there is no simple way for memory consumption measurements. The Java internal garbage collection also makes it nearly impossible to determine memory consumption on a native way as there is no possibility of turning off garbage collection and furthermore the time information of its occurrence is not obtainable. Because of that it was chosen to manually measure the consumption by counting all space reservations and supposing that the occupied space is not freed until the function fully terminates. All tests were run on an *Intel Core i7-3632QM 2.2 GHz 8.0 GB RAM*.

7.1 Runtime

7.1.1 Force-applying approximative method

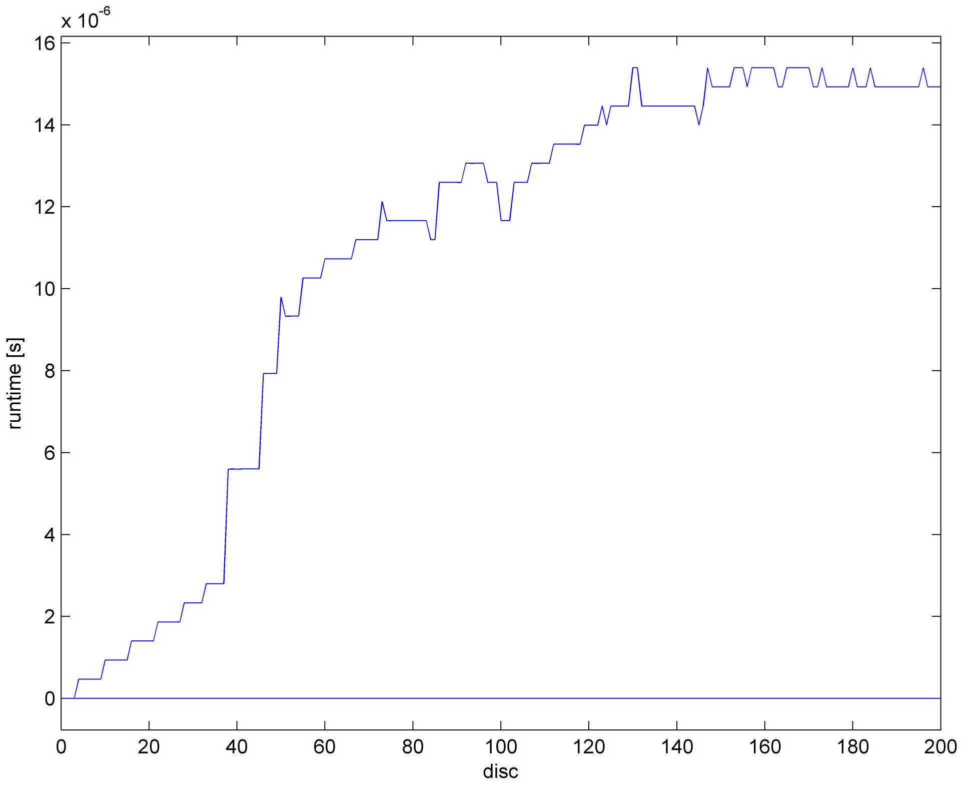

Figure 7.1: Runtime, Force-applying approximative method

The first thing visible is the increase in runtime with rising parameter $disc$. This is exactly what was expected. The discontinuous look of the curve comes from the resolution limits of the time measurements. The runtime itself is in the range $0.0 \ldots 0.000016$ seconds. This indicates that the approximative algorithm is very fast. It is difficult to give a general functional dependence, for example exact linearity can not fully be taken off the readings. Firstly for $0 \leq disc \leq 40$ a linear relation between the runtime and the discretization parameter $disc$ can be seen. In the mid section for $40 \leq disc \leq 140$ initially the gradient for the increase in runtime is higher than afterwards where it slightly decreases. After passing $disc \geq 140$ the runtime stays within a tight range and is likely to be constant.

7.1.2 Incremental finite element method

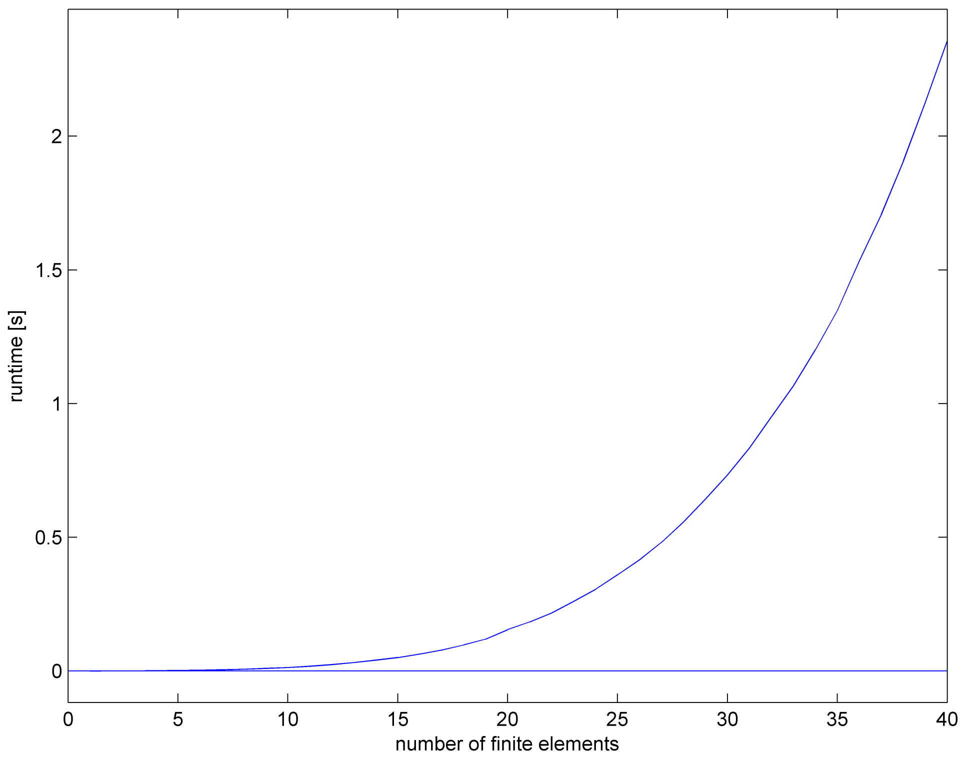

Figure 7.2: Runtime, Incremental finite element method

In contrast to the approximative algorithm's runtime the curve for the finite element algorithm is much smoother. The reason for this is that the algorithm is significantly slower. The runtime is in the range $0.0 \ldots 2.5$ seconds. Considering the functional dependence a quartic correlation between the number of finite elements and the runtime can be read off the measurements. The reason for this is the rising size of the linear equation system that has to be solved after every growth step leads to higher effort for solving it. Comparing both algorithms in the mid range for example at $disc = 100$ and $n\_FE = 20$ the finite element method is about 13250 times slower. Actually at the end of the examination range this difference once again rises about more than 10 times compared to the mid range. For $n\_FE \leq 30$ the runtime considerably remains below $\approx 0.75s$.

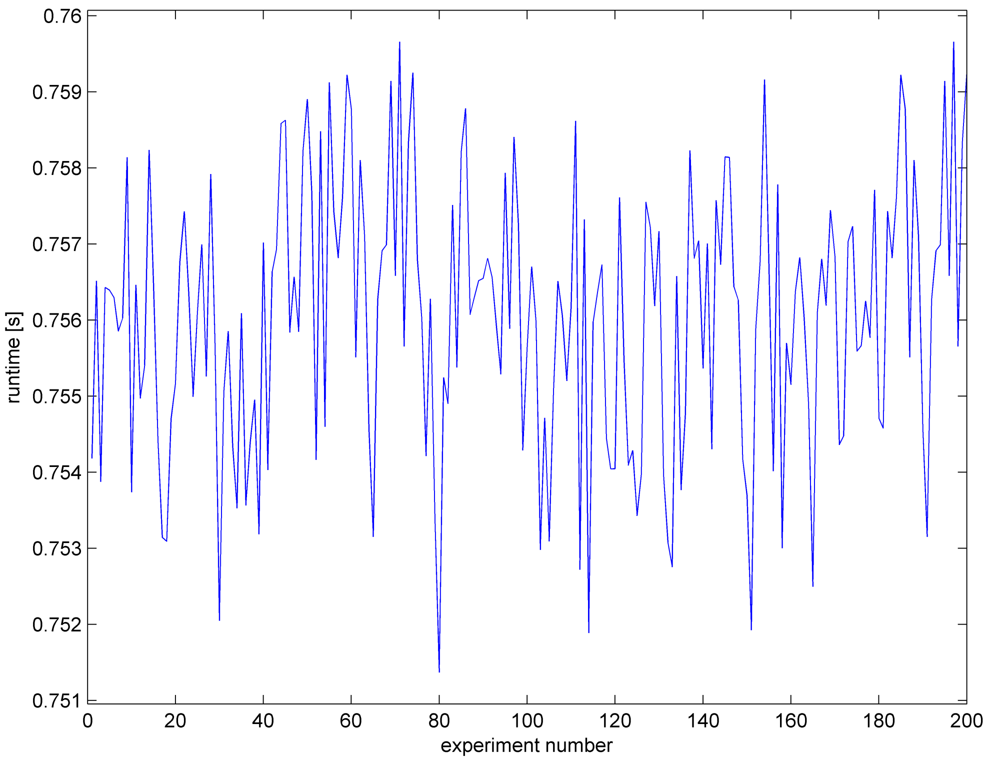

Figure 7.3: Runtime variation, Incremental finite element method, *n_FE*=30

The figure 7.3 shows the stochastic variation of the runtime for the repeatedly carried out experiments. The standard deviation is about $0.0018s$. Hence the runtime stays within a small corridor.

7.2 Memory consumption

7.2.1 Force-applying approximative method

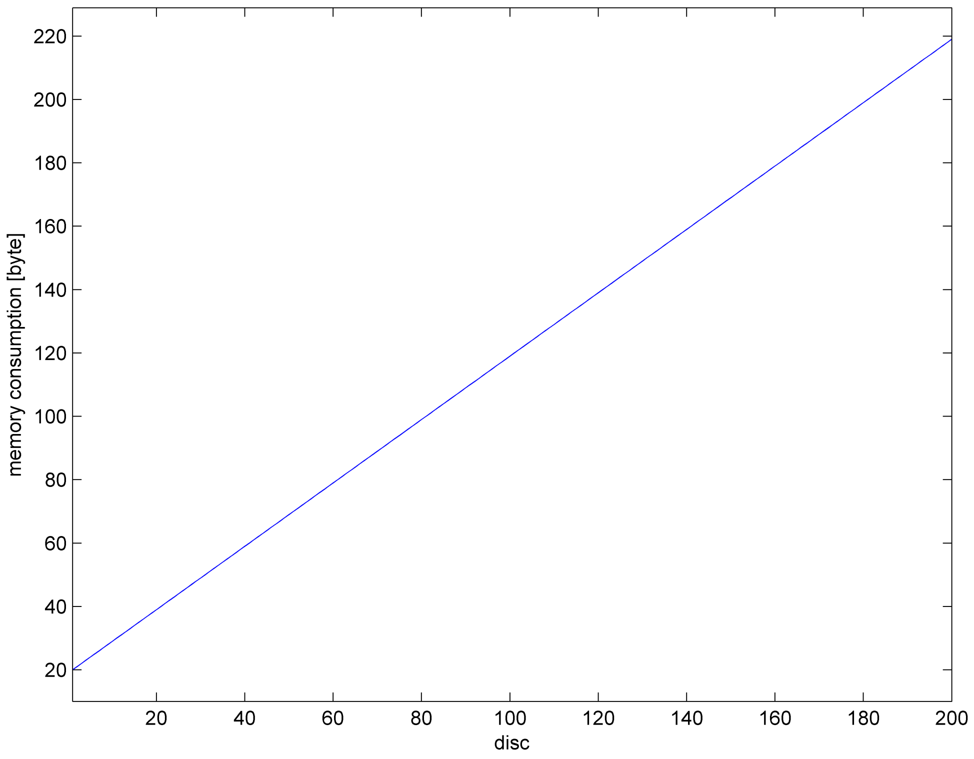

Figure 7.4: Memory consumption, Force-applying approximative method

Between the parameter $disc$ and the memory consumption a linear relation is clearly visible. The main reason for this behavior is that for the rising number of rotation angles space for the according number of array entries must be reserved. In the internal calculations the loop variables are reused and not all intermediate results are stored. The memory consumption is within the range $20 \ldots 220$ bytes.

7.2.2 Incremental finite element method

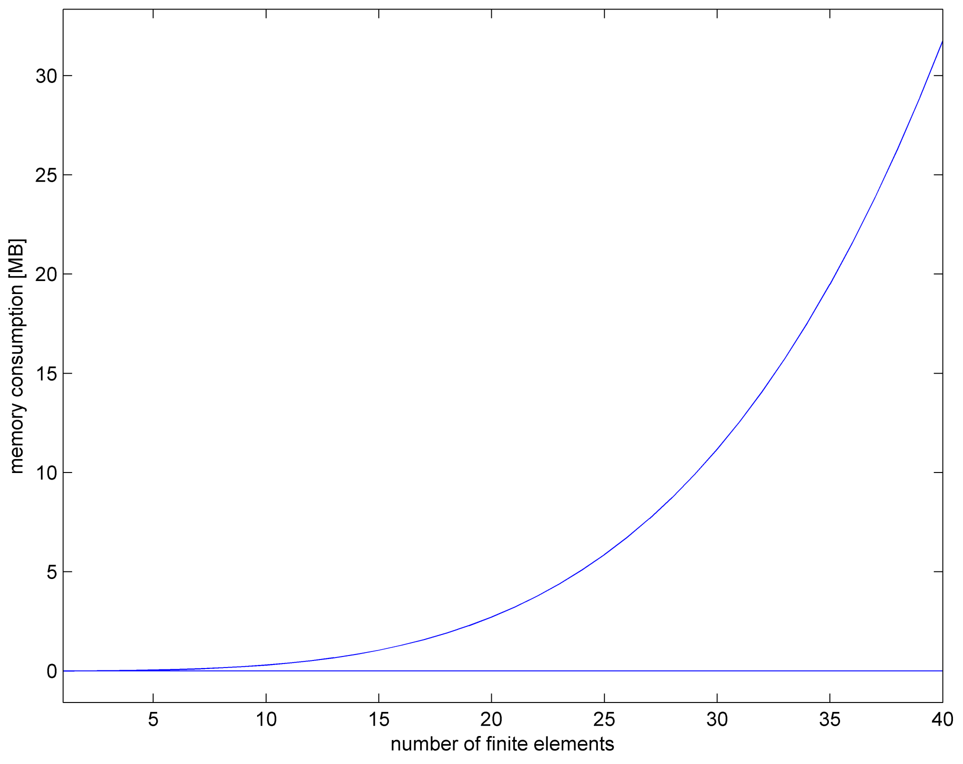

Figure 7.5: Memory consumption, Incremental finite element method

The curve is very similar to the runtime plot. A quartic dependence between the number of finite elements and the memory consumption is evident. This mainly comes from the size of the linear equation system as well as the array size for the rotation angles. The memory consumption is within the range $0 \ldots 32$ megabytes. This is a scale that resides 10^6 times higher than the one from the approximative algorithm.

Chapter 8

Conclusion and outlook

This work dealt with two methods for physics-based simulations of tree branches bending and growing under their self-weight. The first approach was a force-applying approximative method that utilizes a cylindrical axis profile and applies a force to its tip as counterpart for its self-weight. The curvature was obtained by the theory of elasticity. The second was an incremental finite element approach that divides the branch into beam elements where the tree ring structure as well as different material characteristics are also considered.

The physical and mathematical basis was precisely explained for both approaches. The same holds true for the fundamental assumptions that underlie every simulation. Resulting from the theoretical outline two algorithms were developed and finally implemented in the plant modeling language XL. Both deliver lists of rotation angles that specify the branch curvature. Both delivered algorithms comprise the concept of primary and secondary growth, the different tropisms, namely gravitropism and phototropism, and the finite element approach furthermore the tree reaction mechanism of reaction wood formation.

In order to show how to adapt the physics of branches to existing XL tree models three trees with different architecture were supplemented and modified with the developed algorithms. It was shown what is necessary to obtain the structural information from the tree models to get the right parameter setting for the application of the algorithms. As a feature it is also shown how to consider the successively occurring gain in branch mass as tree branches constitute ramified structures.

In terms of comparing the algorithms' output results a branch of order 1 was used to examine the phenotype differences for various parameter ranges. A mathematical way of describing the differences for a particular setting was developed ahead of the measurements. It was found out that for most parameter ranges the branches grow apart. Nevertheless for initial deflection angles over $90°$ the overall difference is acceptable. Thus for the correct choice of parameter setting the approximative algorithm can deliver reasonable results similar to the finite element method. Why this is so important and what else is the approximative algorithm's major advantage is shown in the later examinations.

The analysis of sensitivity for the algorithms' parameters then was carried out for two tree models. The primary insight is that the effect and its sensitivity devolution on the whole tree structure is different for every parameter. That is why it is shown that the value of some parameters must be chosen carefully. The secondary outcome is that the strength of the behaviors also depends on the tree construction itself as this leads to different bases of operations for the parameters.

At the end the runtime and memory consumption was examined. The approximative algorithm is massively faster than the finite element algorithm which can be considered as its main advantage. The same relation holds true for the memory consumption. The consumption of space for the approximative algorithm is linear to the amount of discretization and remains within low ranges whereby the finite element algorithm has a quadratic dependency on the number of finite elements and needs space especially for the effort to solve the linear equation system and for the elementary calculation variables.

Further adaptations

Considering the runtime and memory consumption advantage of the approximative algorithm it might be of interest to minimize the phenotype differences. A possibility for this could represent the development of correction factors for the parameter $K0$ as this parameter mainly is responsible for the strength of bending. These correction values then are obtained experimentally by applying a minimization method to the error function while examining the whole parameter range. Two different possibilities of characterization are imaginable. In one case a table for the various parameter ranges could contain the corresponding correction factors. In the other case, proceeding from the table a polynomial interpolation could give a functional dependence on the respective parameter.

The stem as a main part of a tree has not been considered yet. The whole work only considered the branches being subject to bending under self-weight. But tree stems also experience the influence of gravity. An imbalanced distribution of branch weight induces forces that may lead to stem bending to one side. Lateral forces can be induced by wind. What also can happen is the so called buckling. Additionally the same phenomenon of reaction wood formation that occurs in branches even more intensely occurs in tree stems to insure stability.

In the algorithm for the finite element method homogeneous material distribution is supposed. Through this fact the reality is not fully covered where densities, thicknesses and elasticities are not the same in all wood layers. However the construction of the class FE for a finite element holds all mechanisms for setting different densities, elasticities and ring thicknesses. Because of that it is easy to modify the algorithm by supplementing the parameter list of the function and then setting these properties for each finite element within the code.

When considering the location of a tree it mostly stands not only by itself. Forests represent competitive systems where trees fight for light, water and nutrients. As a result the growth of branches is not only influenced by self-weight but heavily by their location. This leads to different bending curvatures. This behavior could also be implemented by adding suitable parameters for the competitive situation as well as other concepts of reaction wood formation.

The displacement vector from the solution of the finite element approach can be used to calculate tensions and strains inside the wood. This was one major component of the work in [4] and [5]. This information can be used as a stimulation for the reaction wood formation. Furthermore based on these calculations the criteria of failure for branches can be implemented in the model. By that real tree copies in simulations can determine failing branches as a prevention for upcoming dangers.

The parameter lists for the algorithms represent static material properties and forces. That is why for simulations of tree dynamics that consider non-stationary forces, for example sudden winds the properties must evolve dynamically. Additionally every mechanical part owns eigenfrequencies and hence for dynamic simulations they should also be calculated for the stem and branches.

Bibliography

[1] O. Kniemeyer, "Design and Implementation of a Graph Grammar Based Language for Functional-Structural Plant Modelling," *PhD Thesis, Brandenburgische Technische Universität Cottbus*, 2008.

[2] P. de Reffye, "AMAPLESSON 7: Mechanics on Plants," *CIRAD Montpellier*, no year.

[3] J. Hua and M. Kang, "Optimize Tree Shape: Targeting for Best Light Interception," *Proceedings of the 7th International Conference on Functional-Structural Plant Models, Saariselkä, Finland*, 2013.

[4] T. Fourcaud and P. Lac, "Numerical modelling of shape regulation and growth stresses in trees: I. An incremental static finite element formulation," *Trees*, vol. 17, no. 1, pp. 23–30, 2003.

[5] T. Fourcaud, F. Blaise, P. Lac, P. Castéra, and P. de Reffye, "Numerical modelling of shape regulation and growth stresses in trees: II. Implementation in the AMAPpara software and simulation of tree growth," *Trees*, vol. 17, no. 1, pp. 31–39, 2003.

[6] P. de Reffye, C. Edelin, J. Françon, M. Jaeger, and C. Puech, "Plant Models Faithful to Botanical Structure and Development," *Computer Graphics*, vol. 22, no. 4, 1988.

[7] P. de Reffye, F. Houllier, F. Blaise, and T. Fourcaud, "Essai sur les relations entre l'architecture d'un arbre et la grosseur de ses axes végétatifs," in *Modélisation et simulation de l'architecture des végétaux, Institut national de la recherche agronomique*, J. Bouchon, P. de Reffye, and D. Barthélémy, Eds., 1997, pp. 255–382.

[8] T. Fourcaud, "Analyse du comportement mécanique d'une plante en croissance par la méthode des éléments finis," *PhD Thesis, L'Université Bordeaux I*, 1995.

[9] T. Aleméras, J. Gril, and E. Costes, "Bending of apricot tree branches under the weight of axillary growth: test of a mechanical model with experimental data," *Trees*, vol. 16, no. 1, pp. 5–15, 2002.

[10] D. Gaffrey and O. Kniemeyer, "The elasto-mechanical behaviour of Douglas fir, its sensitivity to tree-specific properties, wind and snow loads, and implications for stability - a simulation study," *Journal of Forest Science*, vol. 48, no. 2, pp. 49–69, 2002.

[11] R. Taylor and O. Zienkiewicz, “The finite element method: Solid and fluid mechanics dynamics and non-linearity,” *Mc-Graw-Hill, New York*, vol. 2, no. 4, 1998.

[12] M. Fournier, H. Baillères, and B. Chanson, “Tree biomechanics: growth, cumulative prestresses and reorientation,” *Biomechanics*, vol. 2, pp. 229–251, 1994.

[13] W. Kurth, “Fichtenmodell,” *http://www.uni-forst.gwdg.de/~wkurth/sm09_fichte.rgg*, 2009.

[14] O. Kniemeyer, R. Hemmerling, and W. Kurth, “GroIMP,” *http://www.grogra.de/*.

[15] L. Gürtler, “Internship report: Implementation and test of an approximative method for the bending of cylindrical objects under their own weight,” *not published*, 2017.