Extending the Programming Language XL to Combine Graph Structures with Ordinary Differential Equations

Doctoral Thesis
for Conferral of a Doctorate in Mathematics and Natural Sciences
"Doctor rerum naturalium"
at Georg-August University Göttingen

Submitted by
Reinhard Hemmerling
From Frankfurt(Oder)

Göttingen 2012

Referee:	Prof. Dr. Winfried Kurth
Co-referee:	Prof. Dr. Robert Schaback
Other referee:	Prof. Dr. Paul-Henry Cournède
Date of the oral examination:	13 April 2012

Acknowledgements

There are several people who supported me during the last years in preparation of this thesis and to which I owe my thank.

First of all, I thank my supervisor Prof. Dr. Winfried Kurth, who guided me throughout the whole time and always had an open ear for questions and good advice. His profound knowledge and sharp mind were of great help. Prof. Kurth gave me the opportunity to work at his Chair for Graphics Systems at the Brandenburg Technical University of Cottbus, where part of the implementation for this thesis was done, and he also enabled my switch to the Chair for Computer Graphics and Ecological Informatics at the Georg-August-University of Göttingen in 2008.

I also thank Prof. Dr. Robert Schaback for his interest in this topic and taking the position as my secondary supervisor. He gave me advice in numerical things and we also had some fruitful discussions.

My gratitude goes to Prof. Dr. Paul-Henry Cournède for inviting me to the École Centrale Paris, where I could present some of my work, with subsequent prolific discussions, and for becoming my third referee.

Much appreciation goes to Dr. Ole Kniemeyer, who created the programming language XL and the implementation of GroIMP. Without his prior work the proposed extension to the language would not have been possible.

Special thank also goes to my colleagues Prof. Dr. Gerhard Buck-Sorlin, Dr. Jochem B. Evers and Dr. Katarína Smoleňová, who as the main users of the present work were testing the software and provided useful feedback that helped to improve it.

Thank also goes to people that helped to improve the GroIMP software, especially (in alphabetical order) Octave Etard, Konni Hartmann, Michael Henke, Thomas Huwe, Uwe Mannl, Paul Masters, and Stephan Rogge.

Furthermore, I thank all my colleagues in Cottbus and Göttingen for the nice working atmosphere.

Contents

List of Tables

List of Figures

Listings

1. Introduction

Natural sciences try to discover laws that allow to explain and predict observations. Often, models are created with the purpose to verify or falsify proposed theories, which are based on these laws and certain other assumptions. This frequently involves the solution of differential equations, and in many cases this can only be done numerically.

In the context of plant simulation, functional-structural plant models (FSPMs) combine a simulated structure (3D geometry) with functional aspects (physiology of the plant) [GS05, VEBS$^+$10]. L-systems [PL90] are often used to describe the structural development of plants. The two most widely used platforms for L-system modelling of plants have been reported to be L-Studio and GroIMP [VEBS$^+$10].

XL [Kni08] is the underlying programming language of GroIMP and combines the benefits of L-systems and the Java programming language [GJSB05]. It allows to simulate plants in which the structure of the plant is described by L-system growth rules or, more generally, by graph grammars, and in which the functional aspects are described by ordinary Java code. Similarly, the language of L-Studio (called L+C) allows to use C++ constructs in the L-system rules [KP03].

Physiological processes such as assimilation and allocation of nutrients are usually described by ordinary differential equations (ODEs). Transport processes like diffusion are naturally described by partial differential equations (PDEs), but can be mapped to ODEs via the method of lines [Sch91]. Some other processes (like in [LlCC09]) depend on a former state and thus lead to delay differential equations (DDEs), that can be converted to ODEs via the method of steps [BBK65] under certain conditions. Therefore, the main task remains how to numerically integrate ODEs.

Numerical integration of the ODEs in FSPMs is often found to be performed by Euler integration, e.g. [EVY$^+$10, LKBS10, CSPH11]. In L-system-based models Euler integration is usually implemented as part of the rewriting step of rule applications. Using this integration method, the results from a numerical integration can differ substantially from the exact analytical solution, depending on the type of ODE and the integration step size chosen [LlCC09]. Additionally, Euler integration also suffers from stability problems if the step size chosen was too large.

If more advanced numerical methods should be used the question arises how to combine the discrete nature of rule applications with the continuous nature of differential equations. Differential L-systems (dL-systems) have been proposed by Prusinkiewicz et al. in [PHM93] to overcome this issue. As was pointed out by the authors themselves, "the user must explicitly specify the formulae for numerically solving the differential equations included in the models (the forward Euler method was used in all cases)", so that dL-systems are merely "a notation for expressing developmental models that include growing systems of ODEs" [FP04].

1.1. Motivation

As will be obvious later on, correctly implementing a numerical integration algorithm can become an arbitrarily complex task, and one should resort to existing libraries instead. Therefore, the aim of this thesis is to make those methods accessible to the average user (i.e. biologist, chemist, physicist), who in general is no expert in numerical mathematics. Two solutions will be presented.

The first solution is targeted at the easy specification of stoichiometric equations as they occur in chemical kinetics, and from which the differential equations can be derived according to the law of mass action (under certain conditions). For this purpose XL was extended by the feature of operator overloading and user-defined implicit conversion functions.

As will be shown, these features are general enough so that not only stoichiometric equations can be entered, but even the whole parsing of the productions (right-hand sides) of rules can be explained in terms of overloaded operators. Still there are cases where this solution encounters its limits, like when arbitrary ODEs should be entered. Also dynamical structures, which necessarily occur in growing plants, impose additional problems. Furthermore, those structures are in general not regular, but they can form an arbitrary network (in most observed cases a tree).

So a second solution addresses these issues by the introduction of a new operator symbol and some special handling for it by the compiler and runtime system. This way the user can specify any ODE in a direct way, while the framework performs the bookkeeping tasks (like memory management, etc.) in the background. An interesting aspect of this solution is, that from the user's point of view not much changed compared to how the users defined their model originally. This lowers the threshold for beginners and levels the learning curve.

1.2. Structure of the Thesis

A very short overview and motivation for this work has been given in this chapter. In short, we want to solve differential equations on network like structures with a strong focus towards modelling of plants and provide the user with a tool how to easily formulate such problems and solve them using advanced numerical algorithms.

In the next chapter (chapter 2) we will introduce numerical methods for the solution of ordinary differential equations. However, this is just the tip of the iceberg. In the end it should become clear that implementation of such numerical algorithms can be very challenging and therefore should not be left to the user. Still, the user must possess basic understanding about the problems associated with numerical integration.

Chapter 3 introduces formal languages, L-systems, and RGGs with their implementation in the language XL. L-systems allow to easily describe the topology of a plant. In combination with turtle graphics this also allows generation of a 3D representation. RGGs extend this idea further and generalize it to graph structures. Since the thesis is concerned about extending the language XL, a short summary about the features of the

language will be given. This is also helpful to understand some examples given later on.

Operator overloading and user-defined conversion functions will be presented in chapter 4. It will be discussed how operator overloading was implemented in some other programming languages to derive a design for implementing this feature as an extension to XL. A short overview about the implementation will also be given. Finally, some examples demonstrate the usefulness of the approach.

Chapter 5 adresses the specification and numerical solution of ODEs on graph structures. Some examples will be given and are used throughout the chapter to discuss problems that need to be solved and the derived solution. Important issues are memory management and the mapping between attributes of nodes and the elements of the state vector, as well as handling interactions with the integrator during the solution process via so-called monitor functions. Some available numerical libraries will be discussed and a wrapping interface to these will be suggested. How native libraries can be included will be demonstrated on the example of CVODE, which is part of SUNDIALS. Finally, an overview about the implementation of the framework will be given, and how the user can provide tolerances for attributes.

Results for both approaches will be provided and discussed in chapter 6. This includes the transformation of two dL-systems, a model of the dragon curve and a model of the cyanobacterium *Anabaena catenula* (from [PHM93]), into equivalent XL code. Another example shows how Turing instability can be used for pattern formation, and how the same set of ODEs can be applied to different structures without having to modify the rate function (only the definition of the structure has to be modified as a matter of course). Finally, the solution of PDEs with the method of lines will be demonstrated, and a recipe for discretization of a tree structure with the help of XL rules will be provided.

Chapter 7 gives a short summary of the present work and also provides some ideas for further extensions.

2. Numerical Methods

In this part we will present the theoretical foundations of numerical solution of differential equations to the extent needed for the other parts of this thesis. A wide range of literature is available for this topic, for instance [Ama95, HW02, QSS07, Wal00]. Proofs of the theorems can be found in [Ama95, Wal00]. Surveys/reviews about various techniques and codes (implementations) can be found in [GSDT85, Cas03, Deu85, SWD76, BH87]. Comparison of methods can be based on their performance on test problems. A program DETEST evaluates several methods by their ability to solve a set of 75 test problems [HEFS72]. Shampine, Watts and Davenport show that not only the method itself, but also its implementation is important [SWD76].

Some theoretical considerations use the *Euler method*, which is defined as

$$y_{n+1} = y_n + hf(t_n, y_n)$$

and consitutes the simplest of the standard methods. It belongs, at the same time, to the class of Runge-Kutta methods and linear multistep formulas, that will be defined below.

2.1. Terminology

The notation used in the following sections will be according to the following table:

$u^{(i)}(t)$	– function and its derivatives of the real solution
$u', u'', \ldots$	– same as $u^{(1)}, u^{(2)}, \ldots$
$f(t, u(t))$	– differential equation to integrate, same as u'
t_n, y_n	– approximations made by the integration method
(t_n, y_n)	– a point in the solution space, also called *node*
h	– step size used for integration
τ_n	– truncation error at step n
e_n	– global error at step n
RK4	– Runge-Kutta method of 4th order
RK5(4)	– embedded Runge-Kutta method of 5th order using 4th order for error estimation and step size control

2.2. To the Theory of Initial Value Problems

Many problems in science lead to the numerical integration of ordinary differential equations. In the most simple case the first derivative of a function $u(t)$ is given by

$u'(t) = f(t, u(t))$. Such a relationship is called *ordinary differential equation* and describes the behaviour of the system. Together with some *initial state* $u_0 = u(t_0)$ at some *initial time* t_0, also called *initial condition*, one obtains an *initial value problem*, or Cauchy problem [QSS07]. The result obtained by numerical integration is then the integral

$$u(t) = u_0 + \int_{t_0}^{t} f(s, u(s))ds.$$

An initial value problem is said to be *autonomous*, if u' does not explicitly depend on t, so that $u'(t) = f(u(t))$. This is valid for laws of nature, that remain the same no matter what the current time is.

Still the question remains whether the numerical solution is correct or not. For some simple examples, where an analytical solution is available, this can be checked easily. In general, an analytic solution is not known and this is, after all, the reason why numerical integration is needed. The correctness of the solution can then be judged by indicators like if an expected minimum/maximum occurs, appearance of unexpected oscillations or by the asymptotic behaviour of the solution.

2.2.1. Systems of Ordinary Differential Equations

A more general formulation are systems of ordinary differential equations:

$$\begin{aligned} u_1'(t) &= f_1(t, u_1(t), \ldots, u_n(t)) \\ u_2'(t) &= f_2(t, u_1(t), \ldots, u_n(t)) \\ &\vdots \\ u_n'(t) &= f_n(t, u_1(t), \ldots, u_n(t)) \end{aligned}$$

This can be written in vectorized form as:

$$u'(t) = f(t, u(t)),\ u'(t) = \begin{pmatrix} u_1'(t) \\ u_2'(t) \\ \vdots \\ u_n'(t) \end{pmatrix},\ f(t, u(t)) = \begin{pmatrix} f_1(t, u_1(t), \ldots, u_n(t)) \\ f_2(t, u_1(t), \ldots, u_n(t)) \\ \vdots \\ f_n(t, u_1(t), \ldots, u_n(t)) \end{pmatrix}$$

The initial condition becomes a vector as well:

$$u_0 = u(t_0) = \begin{pmatrix} u_1(t_0) \\ u_2(t_0) \\ \vdots \\ u_n(t_0) \end{pmatrix}$$

2.2.2. Higher Order Ordinary Differential Equations

Given an ordinary differential equation of higher order d

$$u^{(d)}(t) = f(t, u(t), u^{(1)}(t), \ldots, u^{(d-1)}(t))$$

this can be transformed into a system of first order ordinary differential equations by using the helper functions $u_i(t) = u^{(i-1)}(t), i = 1, \ldots, d$. The resulting system is:

$$\begin{aligned} u_1'(t) &= u_2(t) \\ &\vdots \\ u_{d-1}'(t) &= u_d(t) \\ u_d'(t) &= f(t, u_1(t), \ldots, u_d(t)) \end{aligned}$$

Because higher order ordinary differential equations can be mapped to systems of first order ordinary differential equations, only the latter ones will be discussed on the following pages (in their vectorized form).

2.2.3. Existence of Solutions

The *Peano existence theorem* guarantees the existence of local solutions of an initial value problem:

Theorem 1. *(Peano existence theorem) Let $f : D \to \mathbb{R}^d$ be a continuous function on the domain*

$$D = \{(t, x) \in \mathbb{R} \times \mathbb{R}^d | \ |t - t_0| \leq \alpha, ||x - u_0|| \leq \beta\}$$

with $\alpha, \beta > 0$. Let $M = max(||f(D)||)$ be the greatest value of f on D and let $T = min(\alpha, \frac{\beta}{M})$. Then there exists at least one solution to the initial value problem $u'(t) = f(t, u(t)), u_0 = u(t_0)$ on the interval $I = [t_0 - T, t_0 + T]$.

Figure 2.1 graphically illustrates this theorem. The shaded area is enclosed between lines with slope $+M$ and $-M$. Because this is the maximum slope of u in the domain D, all solutions (if any exists) must lie in this shaded area.

The proof of the theorem consists of the construction of a sequence of piecewise linear functions $u^h(t)$ using the *Euler method*, that approximate the solution. By proving the *equicontinuity* condition $||u^h(s) - u^h(t)|| \leq M|s - t|$ and using the *Arzelà-Ascoli theorem* it can be shown, that $u^h(t)$ converges to a continuous function $u(t)$ for $h \to 0$. Then it is shown that this function fulfils the integral equation $u(t) = u_0 + \int_{t_0}^{t} f(s, u(s))ds$. With the *fundamental theorem of calculus* it follows that this function is differentiable and is a solution to the initial value problem.

Theorem 2. *(Arzelà-Ascoli theorem) Let $F = \{f_k\}$ be a family of continuous functions $f_k : I \to \mathbb{R}^d$, that are uniformly bounded and equicontinuous:*

$$M = \sup_{f \in F} \max_{t \in I} ||f(t)|| < \infty$$

$$\forall \epsilon > 0 : \exists \delta > 0 : \forall f \in F : \forall s, t \in I : |s - t| < \delta \Rightarrow ||f(s) - f(t)|| \leq \epsilon$$

Then there exists a subsequence $\{f_n\}_{n \in \mathbb{N}} \subset F$, that converges uniformly towards a function $f : I \to \mathbb{R}^d$:

$$\lim_{n \to \infty} \max_{t \in I} ||f_n(t) - f(t)|| \to 0$$

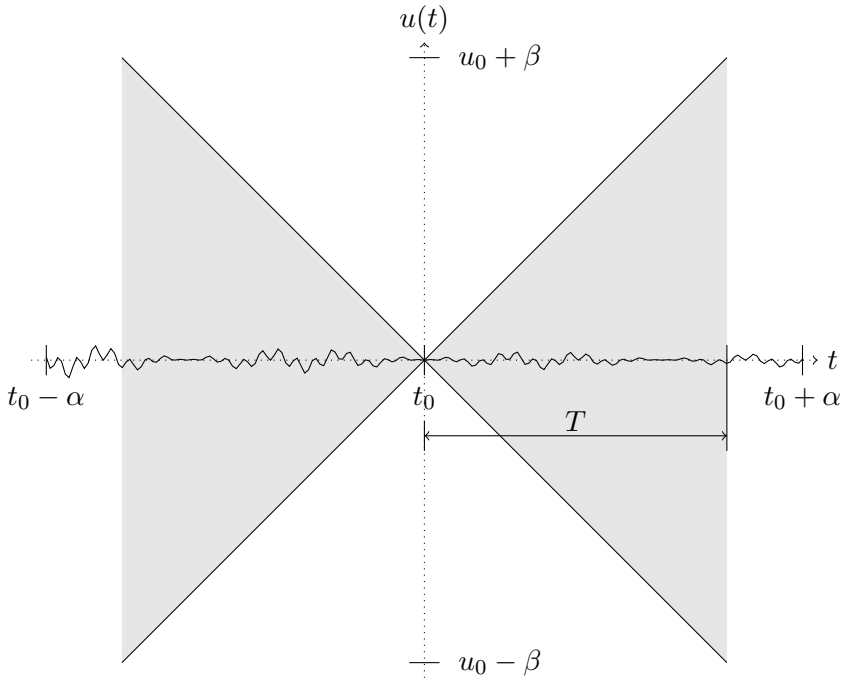


Figure 2.1.: Graphical illustration of the Peano existence theorem.

Although the *Peano existence theorem* can tell whether there are any solutions to the initial value problem, it does not tell anything about their uniqueness. For instance

$$u'(t) = u(t)^{\frac{2}{3}},\ u(0) = 0$$

has valid solutions with $c \geq 0$:

$$u(t) = \begin{cases} 0, & 0 \leq t \leq c \leq 1 \\ \frac{1}{27}(t-c)^3, & c < t \leq 1. \end{cases}$$

The *Euler method* used for proving the *Peano existence theorem* can only find the solution $u(t) = 0$, but not the infinitely many other solutions.

Definition 1. *(Lipschitz continuity) Let (X, d_X) and (Y, d_Y) be metric spaces. A function $f : X \to Y$ is Lipschitz continuous, if there exists a real constant $L \geq 0$ (Lipschitz constant) such that*

$$\forall x_1, x_2 \in X : d_Y(f(x_1), f(x_2)) \leq L \cdot d_X(x_1, x_2).$$

A function is locally Lipschitz continuous if for every $x \in X$ there exists a neighbourhood $U \subseteq X$ such that for f restricted to U the condition is fulfilled.

For the function $f(t, x)$ to be *Lipschitz continuous* in x this means, that

$$||f(t, x) - f(t, x')|| \leq L||x - x'||,\ (t, x), (t, x') \in D. \tag{2.1}$$

Theorem 3. *(Picard-Lindelöf theorem) Let $f(t,x)$ be a function that is continuous in t and Lipschitz continuous in x and consider the initial value problem*

$$u'(t) = f(t,u(t)),\ u(t_0) = u_0,\ t \in [t_0 - \alpha, t_0 + \alpha].$$

Then, for some value $\epsilon > 0$, there exists a unique solution to the initial value problem within the interval $[t_0 - \epsilon, t_0 + \epsilon]$.

The *Picard-Lindelöf theorem* has stronger requirements than the *Peano existence theorem*, but ensures that there is a unique solution to the intial value problem, if any exists.

A proof of the *Picard-Lindelöf theorem* can use the *Peano existence theorem* to show that local solutions exist, and then use *Gronwall's lemma* to show its uniqueness.

Theorem 4. *(Gronwall's lemma) Let $\omega(t)$ and $b(t)$ be continuous functions, $a(t)$ be integrable and non-decreasing, $\omega(t) \geq 0$, $a(t) \geq 0$, $b(t) \geq 0$ and*

$$\omega(t) \leq a(t) + \int_{t_0}^{t} b(s)\omega(s)ds,\ t \geq t_0,$$

then also

$$\omega(t) \leq a(t) \exp\left(\int_{t_0}^{t} b(s)ds\right),\ t \geq t_0.$$

For estimation of the global truncation error later on, a discrete version of the Gronwall lemma can be formulated as well:

Theorem 5. *(Discrete Gronwall lemma) Let ω_n, a_n, b_n be nonnegative sequences, a_n non-decreasing, such that*

$$\omega_n \leq a_n + \sum_{i=0}^{n-1} b_i\omega_i,\ n \geq 0,$$

then also

$$\omega_n \leq a_n \exp\left(\sum_{i=0}^{n-1} b_i\right).$$

2.2.4. Local and Global Error

Numerical integration can just approximate the real solution $u(t)$ of an initial value problem

$$u'(t) = f(t,u(t)),\ u(t_0) = u_0.$$

Most methods then compute a sequence $y_1, y_2, \ldots, y_n$ at $t_1, t_2, \ldots, t_n$ as result. To obtain y_{n+1}, the method solves

$$y'(t) = f(t,y(t)),\ y(t_n) = y_n.$$

Therefore the numerical method tries to approximate a different curve $u_n(t)$ than the curve of the real solution $u(t)$. The *local error* introduced in the calculation of y_{n+1} is then

$$l_{n+1} = u_n(t_{n+1}) - y_{n+1}.$$

The *global error* at t_{n+1} is

$$\begin{aligned} g_{n+1} &= u(t_{n+1}) - y_{n+1} \\ &= u(t_{n+1}) - u_n(t_{n+1}) + l_{n+1} \end{aligned}$$

and therefore depends on the local error at the current step and the local errors at all the previous steps.

Reasons for local error are *truncation error* and *round-off error*. Round-off error is caused by a limited precision when performing calculations with the computer. For instance, a number like π or a fraction like $\frac{1.0}{3.0}$ introduces round-off error when represented as floating-point number. Similarly, basic arithmetic operations can introduce errors. This problem is discussed in [FH07, Gol91]. Especially subtraction of two similar numbers is problematic, because it causes cancelation of the most significant digits. If the operands were subject to rounding errors, then *catastrophic cancelation* can cause the difference to have an error arbitrarily large and even the sign of the result can be undetermined.

For instance the solution to the quadratic equation $ax^2 + bx + c = 0$ can be calculated as

$$\begin{aligned} x_1 &= \frac{-b + \sqrt{b^2 - 4ac}}{2a} \\ x_2 &= \frac{-b - \sqrt{b^2 - 4ac}}{2a}. \end{aligned}$$

When $|b| \gg |ac|$, then $\sqrt{b^2 - 4ac} \approx |b|$ and so for calculating either x_1 or x_2 catastrophic cancelation occurs. The problem can be circumvented by calculating

$$q = \begin{cases} -(b - \sqrt{b^2 - 4ac})/2, & b < 0 \\ -(b + \sqrt{b^2 - 4ac})/2, & otherwise \end{cases}$$

$$\begin{aligned} x_1 &= \frac{q}{a} \\ x_2 &= \frac{c}{q} \end{aligned}$$

as suggested in [Gol91, PH04].

But even when calculations with unlimited precision would be possible, truncation errors would be produced by the numerical method. As an example, consider the *Euler method*, which is defined as:

$$y_{n+1} = y_n + hf(t_n, y_n). \tag{2.2}$$

The *local truncation error*[1] is the difference between the increment of the real solution and the increment of the numerical method in a step, so

$$\tau_n = \frac{u(t_{n+1}) - u(t_n)}{h} - f(t_n, u(t_n)) \tag{2.3}$$

is a measure for the quality of the numerical method.

A Taylor series expansion of the function $u(t)$ at t_{n+1} yields

$$u(t_{n+1}) = u(t_n) + hu'(t_n) + \frac{h^2}{2!}u''(t_n) + \cdots .$$

Inserting this expression for $u(t_{n+1})$ into the expression for τ_n and noticing that $u'(t_n) = f(t_n, u(t_n))$ gives

$$\tau_n = \frac{h}{2!}u''(t_n) + \cdots = \mathcal{O}(h).$$

For a reasonable numerical method the local truncation error goes to zero as $h \to 0$. If this is true for every τ_i, the method is called *consistent*. The *consistency order* (or *order of the method*) tells how fast the truncation error disappears. For a method of order p the local truncation error is $\mathcal{O}(h^p)$. As can be seen, the Euler method is of first order.

The *global truncation error* is $e_n = y_n - u(t_n)$. By using equations (2.2) and (2.3) it follows:

$$e_{n+1} = e_n + h\big(f(t_n, y_n) - f(t_n, u(t_n))\big) - h\tau_n.$$

Using the Lipschitz condition (2.1) this can be estimated as

$$||e_{n+1}|| \leq ||e_n|| + hL||e_n|| + h||\tau_n||.$$

Repeated application of this equation yields

$$||e_{n+1}|| \leq ||e_0|| + hL\sum_{i=0}^{n} ||e_i|| + h\sum_{i=0}^{n} ||\tau_i||.$$

Using the discrete Gronwall lemma (Lemma 5) it follows

$$||e_{n+1}|| \leq \left(||e_0|| + h\sum_{i=0}^{n} ||\tau_i|| \right) e^{L(t_{n+1}-t_0)}$$

as approximation of the global truncation error of the Euler method.

A numerical method is *convergent*, if its global truncation error goes to zero as the step size h goes to zero:

$$\lim_{h\to 0} ||e_n|| = 0.$$

[1] This definition is from [SB05], in [GSDT85] the local truncation error is defined as $\tau_n = u(t_{n+1}) - u(t_n) - hf(t_n, u(t_n))$.

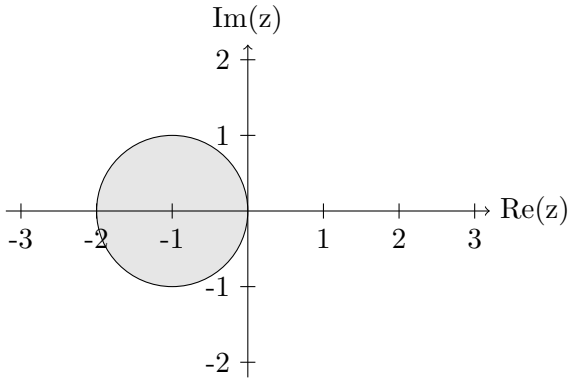


Figure 2.2.: Region of absolute stability for the explicit Euler method for $z = h\lambda$.

2.2.5. Stability and Stiffness

To successfully solve an integration problem the numerical method must be stable, therefore *stability* dictates the range of usable step sizes. If the step size is too big, the truncation error causes instability and prevents finding a solution, while for a step size too small, round-off errors dominate.

Different types of stability have been proposed, usually based on some test problem. For instance, consider

$$y' = \lambda y,\ y(0) = y_0$$

with exact solution

$$y(t) = y_0 e^{\lambda t}$$

with a complex-valued parameter λ. This problem represents exponential decay or growth. For linear systems of differential equations λ represents the eigenvalues of the system. If $Re(\lambda) < 0$ then $y(t) \to 0$ as $t \to \infty$. A numerical method that mimics this behaviour is said to be *A-stable*, as was defined by Dahlquist in [Dah63].

Now consider the explicit Euler method $y_{n+1} = y_n + hf(t_n, y_n)$ that was introduced in equation (2.2). Substituting the definition of $f(t, y) = \lambda y$ for the problem above into the numerical method yields

$$y_{n+1} = y_n + h\lambda y_n$$

and after transformation

$$y_{n+1} = (1 + h\lambda)y_n = (1 + h\lambda)^{n+1} y_0.$$

To be A-stable $Re(\lambda) < 0$ and thus $|1 + h\lambda| < 1$. Designating $z = h\lambda$ with $z \in \mathbb{C}$ the *region of absolute stability* as depicted in figure 2.2 is obtained for the explicit Euler method. Stability therefore seriously limits the usable range of step sizes h.

Contrast this with the use of the implicit Euler method $y_{n+1} = y_n + hf(t_{n+1}, y_{n+1})$ giving

$$y_{n+1} = y_n + h\lambda y_{n+1}$$

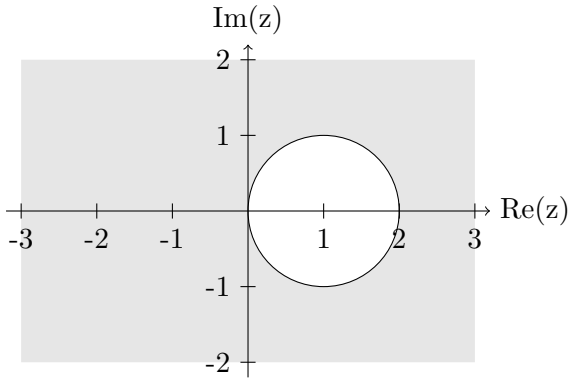


Figure 2.3.: Region of absolute stability for the implicit Euler method for $z = h\lambda$.

after substitution and

$$y_{n+1} = \frac{1}{1-h\lambda} y_n = \left(\frac{1}{1-h\lambda}\right)^{n+1} y_0$$

after transformation. The implicit Euler method is A-stable if $|1 - h\lambda| > 1$ with a region of absolute stability depicted in figure 2.3. As can be seen, the whole region left to the y-axis is stable making this method *unconditionally stable*. This feature of implicit methods makes them particularly attractive for this type of problems. The step size used for integration is not limited by stability, but by accuracy requirements.

This is true especially for *stiff* problems. Consider the problem

$$u' = z,\ z' = -(\lambda + 1)z - \lambda u,\ \lambda > 0$$

given by Cash in [Cas03], based on a damped harmonic oscillator $u'' + (\lambda+1)u' + \lambda u = 0$. The exact solution is

$$u(t) = Ae^{-t} + Be^{-\lambda t}$$

with A and B being some coefficients depending on the initial conditions. To obtain stability when using the explicit Euler method,

$$0 < h < 2 \ \text{ and } \ 0 < h < \frac{2}{\lambda}$$

must be fulfilled. For big values of λ the second part $Be^{-\lambda t}$ of the solution vanishes rapidly but then still requires to keep the step size h very low to remain stable. Compare this with the problem

$$u' = -u,$$

which resembles the first part Ae^{-t} of the previous problem. After a very short time the solutions for both problems are basically identical, but in this second case the step size is only limited by

$$0 < h < 2,$$

which is much less restrictive for large λ. For the implicit Euler method this restriction is removed and selection of a step size can be done according to the required accuracy.

For the definition of *stiffness* not only the problem itself must be accounted. For instance, if integration were limited to the interval $[0, 1/\lambda]$ then the restriction for the step size caused by λ would be no restriction at all.

Besides A-stability, also other notions of stability have been suggested. Instead of requiring the whole negative half-plane to be included in the region of absolute stability, *A(α)*-stability requires this only for a certain sector [Wid67]. Another stability property is *A_0-stable* [Cry73], not to be confused with $A(0)$-stable. *Strong A-stability* defined by Chipman [Chi71] is a more restrictive kind of A-stability for Runge-Kutta methods. *B-stability* "is a natural extension of the notion of A-stability for non-linear systems" for Runge-Kutta methods [But75]. A similar criterion for multistep methods is *G-stability* [Dah76], although in [Dah78] the same author has shown that G-stability is equivalent to A-stability. Similarly, Hairer and Türke have shown in [HT84] the equivalence of B-stability and A-stability. *S-stability* provides an extension of A-stability to stiff problems for implicit one-step methods [PR74]. If S-stability only holds for $|arg(-h\lambda)| < \alpha$ the method is S(α)-stable. Properties equivalent to A_0-stability for delay differential equations are DA_0-stability and GDA_0-stability [Cry74]. Extensions are *Q-stability* and *GQ-stability* as well as *P-stability* and *GP-stability* [Bar75], and *P[α,β]-stability* [Bic82].

2.3. Explicit Runge-Kutta Methods

The most widely used class of one-step methods are *Runge-Kutta formulas*, due to Runge [Run95] and Kutta [Kut01]. In each step y_{n+1} is computed with *stages*

$$k_i = f(t_n + hc_i, y_n + h\sum_{j=1}^{s} a_{ij}k_j)$$

as

$$y_{n+1} = y_n + h\sum_{i=1}^{s} b_i k_i.$$

According to [But64b] this can be represented in tabular form as:

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\ c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & \vdots & & \vdots \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array}$$

If $a_{ij} = 0$ for $j \geq i$ the method is *explicit* (ERK). In this case those coefficients may also be omitted from the tableau. If $a_{ij} = 0$ for $j > i$ and at least one of the $a_{ii} \neq 0$ the method is *diagonally implicit* (DIRK). If then also all entries on the diagonal have the same value, $a_{ii} = \lambda$, then the method is *singly diagonally implicit* (SDIRK). In the remaining cases it is just *implicit* (IRK).

i	number of conditions for order i	number of parameters for i stages	largest obtainable order by an i-stage method
1	1	1	1
2	2	3	2
3	4	6	3
4	8	10	4
5	17	15	4
6	37	21	5
7	85	28	6
8	200	36	6

Table 2.1.: Properties of some explicit Runge-Kutta methods.

Runge-Kutta methods can be developed by deriving the coefficients in the tableau [But63, Sti67, Jen76]. The order and stability properties of a method depend on the choice of these coefficients. Basically a Taylor series expansion with terms up to a certain order is used to obtain a set of nonlinear equations. Solving these equations then gives the coefficients for a method of that order, if such a method exists.

Table 2.1 lists the number of conditions (also called *elementary differentials*) for a given order, the number of parameters for an explicit i-stage Runge-Kutta method and the largest order obtainable by such a method [Jen76]. Note that there exists no 5-stage explicit Runge-Kutta method of 5th order [But64b]. Note further, that for implicit methods, orders higher than the number of stages can be obtained, for instance a 3-stage method of 5th order [But63], or in general an s-stage method of order $2s$ [But64a].

Euler's method

The most simple explicit Runge-Kutta method was presented by Leonard Euler in his *Institutionum calculi integralis volumen primum*[2] [Eul68] in paragraph 650. The integral is approximated by calculating line segments with the slope defined by the function f at the beginning of each segment. The algorithm to calculate the connecting points of the segments for a step size h is therefore:

$$y_{n+1} = y_n + hf(t_n, y_n). \tag{2.4}$$

The corresponding tableau is:

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array}$$

The *Euler method* is a 1-stage method of 1st order. By construction, this is the only such method.

[2] A translation into English can be found at `http://www.17centurymaths.com/contents/integralcalculus.html` (last access: April 29th 2011)

Runge's methods

Based on the ideas of Euler, Carl Runge developed methods of higher orders [Run95]. In the first method (also known as *midpoint method*) the steps are calculated as

$$y_{n+1} = y_n + hf(t_n + \frac{h}{2}, y_n + \frac{h}{2} f(t_n, y_n)) \tag{2.5}$$

with the corresponding tableau

$$\begin{array}{c|cc} 0 & & \\ \frac{1}{2} & \frac{1}{2} & \\ \hline & 0 & 1 \end{array}$$

The second method (also known as *Heun's method*) is defined as

$$y_{n+1} = y_n + \frac{h}{2}\Big(f(t_n, y_n) + f(t_n + h, y_n + hf(t_n, y_n))\Big) \tag{2.6}$$

with tableau

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

Both 2-stage methods are of 2nd order.

Heun's methods

In [Heu00] Karl Heun obtained the formulas of Runge and of some more of up to 4th order. Based on the idea of the Gaussian quadrature rule he starts with

$$\Delta y = \sum_{\nu=1}^{n} \Big(\alpha_\nu f(x + \epsilon_\nu \Delta x, y + \Delta'_\nu y)\Big) \cdot \Delta x$$

where $\Delta y = y_{n+1} - y_n$, $\Delta x = h$ and

$$\begin{aligned} \Delta'_\nu y &= \epsilon_\nu f(x + \epsilon'_\nu \cdot \Delta x, y + \Delta''_\nu y) \cdot \Delta x \\ \Delta''_\nu y &= \epsilon'_\nu f(x + \epsilon''_\nu \cdot \Delta x, y + \Delta'''_\nu y) \cdot \Delta x \\ &\vdots \\ \Delta^{(m)}_\nu y &= \sum_{\nu}^{(m-1)} f(x, y) \cdot \Delta x \end{aligned}$$

to obtain coefficients for a method by comparison of the terms with the Taylor expansion of Δy.

A 3-stage method of 3rd order provided by Heun is, for instance,

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + \frac{h}{3}, y_n + \frac{h}{3}k_1) \\ k_3 &= f(t_n + \frac{2h}{3}, y_n + \frac{2h}{3}k_2) \\ y_{n+1} &= y_n + \frac{h}{4}(k_1 + 3k_3) \end{aligned} \tag{2.7}$$

with tableau

$$\begin{array}{c|ccc} 0 & & & \\ \frac{1}{3} & \frac{1}{3} & & \\ \frac{2}{3} & 0 & \frac{2}{3} & \\ \hline & \frac{1}{4} & 0 & \frac{3}{4} \end{array}$$

Kutta's methods

Motivated by the work of Runge, Wilhelm Kutta tried to generalize the ideas of Runge and Heun [Kut01]. He searched for methods that achieve a high order while keeping the number of evaluations of f low and the coefficients rational.

Kutta observed, that Heun's approach requires computation of n series of function evaluations, where each evaluation just depends on the previous one. Kutta's approach is more flexible in that a function evaluation can depend on all previous evaluations, which leads to the formulation of Runge-Kutta methods used nowadays.

For instance, Kutta provides a 3-stage method of 3rd order similar to *Simpsons rule*:

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \\ k_3 &= f(t_n + h, y_n - hk_1 + 2hk_2) \\ y_{n+1} &= y_n + \frac{h}{6}(k_1 + 4k_2 + k_3) \end{aligned} \tag{2.8}$$

with tableau

$$\begin{array}{c|ccc} 0 & & & \\ \frac{1}{2} & \frac{1}{2} & & \\ 1 & -1 & 2 & \\ \hline & \frac{1}{6} & \frac{4}{6} & \frac{1}{6} \end{array}$$

Kutta also provides a 4-stage method of 4th order known today as *classical Runge-*

Kutta method:

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \\ k_3 &= f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2) \\ k_4 &= f(t_n + h, y_n + hk_3) \\ y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \tag{2.9}$$

with tableau

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{2}{6} & \frac{2}{6} & \frac{1}{6} \end{array}$$

In the case that $f(t, y)$ does not depend on y, both methods transform into Simpsons rule.

Another 4-stage method of 4th order by Kutta is (also known as *3/8 method*):

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + \frac{h}{3}, y_n + \frac{h}{3}k_1) \\ k_3 &= f(t_n + \frac{2h}{3}, y_n - \frac{h}{3}k_1 + hk_2) \\ k_4 &= f(t_n + h, y_n + hk_1 - hk_2 + hk_3) \\ y_{n+1} &= y_n + \frac{h}{8}(k_1 + 3k_2 + 3k_3 + k_4) \end{aligned} \tag{2.10}$$

with tableau

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{3} & \frac{1}{3} & & & \\ \frac{1}{3} & -\frac{2}{3} & 1 & & \\ 1 & 1 & -1 & 1 & \\ \hline & \frac{1}{8} & \frac{3}{8} & \frac{3}{8} & \frac{1}{8} \end{array}$$

Kutta also investigated methods of higher orders. He conjectured that no 5-stage method of 5th order exists, but it was more than 60 years later that Butcher proved this

[But64b]. Two 6-stage methods of 5th order provided by Kutta are:[3]

$$\begin{array}{c|cccccc}
0 & & & & & & \\
\frac{1}{5} & \frac{1}{5} & & & & & \\
\frac{2}{5} & 0 & \frac{2}{5} & & & & \\
1 & \frac{9}{4} & -\frac{20}{4} & \frac{15}{4} & & & \\
\frac{3}{5} & -\frac{63}{100} & \frac{180}{100} & -\frac{65}{100} & \frac{8}{100} & & \\
\frac{4}{5} & -\frac{18}{75} & \frac{60}{75} & \frac{10}{75} & \frac{8}{75} & 0 & \\
\hline
 & \frac{17}{144} & 0 & \frac{100}{144} & \frac{2}{144} & -\frac{50}{144} & \frac{75}{144}
\end{array}
\qquad
\begin{array}{c|cccccc}
0 & & & & & & \\
\frac{1}{3} & \frac{1}{3} & & & & & \\
\frac{2}{5} & \frac{4}{25} & \frac{6}{25} & & & & \\
1 & \frac{1}{4} & -\frac{12}{4} & \frac{15}{4} & & & \\
\frac{2}{3} & \frac{6}{81} & \frac{90}{81} & -\frac{50}{81} & \frac{8}{81} & & \\
\frac{4}{5} & \frac{6}{75} & \frac{36}{75} & \frac{10}{75} & \frac{8}{75} & 0 & \\
\hline
 & \frac{23}{192} & 0 & \frac{125}{192} & 0 & -\frac{81}{192} & \frac{125}{192}
\end{array}$$

2.4. Variable Step Size and Error Control

For some problems it might be desirable to not compute the solution with a fixed step size everywhere, but to modify the size of the steps according to some heuristic so that bigger steps are made where the solution does not change much.

Ideally the chosen step size should depend on some estimate of the error produced by the next step. For the one-step methods presented above, this error can be estimated by comparing the errors made by doing one step of size h and doing two steps of sizes $h/2$ each. This argument is given in [Nys25]. If the method is of order p, the error made in one step of size h is proportional to h^{p+1}, so $\epsilon_1 = Kh^{p+1}$. Making two steps of size $h/2$ each gives $\epsilon_2 = K(\frac{h}{2})^{p+1}$ for one step, and approximately twice that value for two steps. This gives $2\epsilon_2 = \frac{\epsilon_1 - 2\epsilon_2}{2^p - 1}$, meaning that the error made by two steps with size $h/2$ each is just $\frac{1}{2^p-1}$ times the difference of both results.

This was also shown in [SB05], but based on the global error. Also an algorithm for performing step size control is given there and will be shown below. Basically one wants to obtain the largest possible step size that keeps the global truncation error below some user defined threshold ϵ. Typically one sets $\epsilon = K \cdot \texttt{eps}$, where $K = max\{|u(t)|\}$ gives the size of the numbers in the integration interval and `eps` is the machine epsilon (for double precision arithmetic this is 2^{-53}).

For the intial value problem $u'(t) = f(t, u(t)); u(t_0) = u_0$ let $\eta(t; h)$ denote the approximation of $u(t)$ at t when integrating with a step size of h. Now given some step size H and two approximations $\eta(t_0 + H; H)$ and $\eta(t_0 + H; H/2)$ a choice of a good step size h respecting the error ϵ can be calculated as[4]

$$\frac{H}{h} \approx \left(\frac{2^p}{2^p - 1} \cdot \frac{|\eta(t_0 + H; H) - \eta(t_0 + H; H/2)|}{\epsilon} \right)^{\frac{1}{p+1}},$$

where p is the order of the Runge-Kutta method (for instance $p = 4$ for the classical Runge-Kutta method). In the optimal case $H \approx 2h$ and $\eta(t_0 + H; H/2)$ has error ϵ. If

[3] Both methods contained errors in the coefficients, as was observed by Fehlberg in [Feh58] for the first method and by Nyström in [Nys25] for the second method. The correct methods are presented here.

[4] In [Jus09] basically the same formula is derived, but with an exponent of $\frac{1}{p}$ instead of $\frac{1}{p+1}$.

$\frac{H}{h} \gg 2$, the error of $\eta(t_0 + H; H/2)$ is greater than ϵ. In this case H will be set to $2h$ and the calculation will be repeated. Otherwise accepting $\eta(t_0 + H; H/2)$ completes the step and the process can be repeated for the next step. A diagram showing the algorithm is shown in figure 2.4.

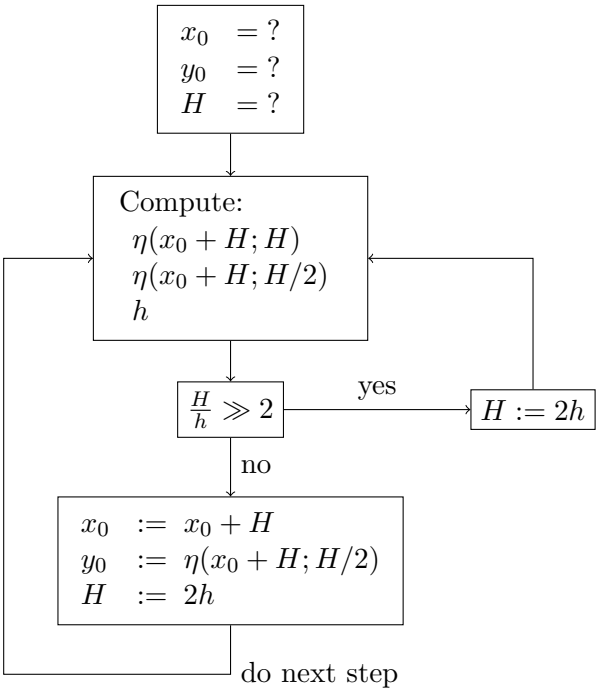


Figure 2.4.: Flowchart of variable step size algorithm (from [SB05]).

Instead of accepting $\eta(t_0 + H; H/2)$ as solution for the step, a better estimate can be found by using *Richardson's extrapolation to the limit*:

$$\eta = \frac{2^p \eta(t_0 + H; H/2) - \eta(t_0 + H; H)}{2^p - 1}$$

As an example consider the initial value problem (also from [SB05])

$$u' = -200tu^2,\ u(-3) = \frac{1}{901},$$

with exact solution $u(t) = 1/(1 + 100t^2)$. The graph of the function $u(t)$ is shown in figure 2.5. Problematic for the numerical method is the peak at $t = 0$. Close to that point the derivatives are becoming very high and the required step size must be rather small, while on the remaining interval the step size can be much bigger.

Computing with 12 digits precision and replacing $\frac{H}{h} \gg 2$ by $\frac{H}{h} \geq 3$ they obtained

error at $t = 0$	number of steps	smallest H used
$-0.13585 \cdot 10^{-6}$	1476	$0.1226 \cdot 10^{-2}$

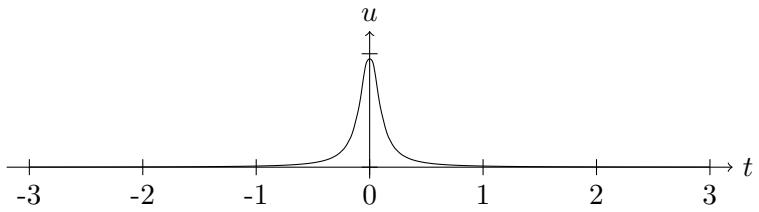


Figure 2.5.: Plot of the function $u(t) = 1/(1 + 100t^2)$.

for the method with variable step sizes and

step size h	error at $t = 0$	number of steps
$\frac{3}{1476} = 0.2032 \cdot 10^{-2}$	$-0.5594 \cdot 10^{-6}$	1476
$0.1226 \cdot 10^{-2}$	$-0.4052 \cdot 10^{-6}$	$\frac{3}{0.1226 \cdot 10^{-6}} = 2446$

when using the same number respective size of fixed steps. As can be seen, the variable step method is more accurate in both cases. One should notice, however, that three times more function evaluations of f are needed per step compared to the fixed step methods.

For a practical implementation one needs to handle the case when the two estimates $\eta(t_0 + H; H)$ and $\eta(t_0 + H; H/2)$ have the same value. This can happen, for instance, if H became too small so that no change in u can be detected, or if the function u is (at least locally) linear. In the former case the step size should be limited by a minimum step size h_{min}, so that the integrator does not get stuck and also roundoff errors do not accumulate too much, or the integration can just be halted with an error message. In the latter case the step size must be bounded by a maximum allowed step size h_{max} defined by the user, which also prevents skipping important parts of the solution. Also the choice of a good initial step size must be taken care of.

2.5. Embedded Runge-Kutta Methods

Instead of using the same method with different step sizes, like in the previous section, we can also use different methods of different order to estimate the error made in one step. According to [But96] this idea was first proposed by Merson [Mer57], who tried to construct a method with five stages for which the first four stages give a method of 4th order and all five stages a method of 5th order. As was shown by Butcher in [But64b], no explicit Runge-Kutta method of 5th order with just five stages can exist, and so the Merson method is appropriate only for problems which are approximately linear. However, the underlying idea has lead to a new class of Runge-Kutta methods named *Embedded Runge-Kutta* (ERK) methods or, due to of Fehlberg [Feh68, Feh69a], also called *Runge-Kutta-Fehlberg* (RKF) methods.

An explicit embedded method can be written with stages

$$k_i = f(t_n + hc_i, y_n + h\sum_{j=1}^{i-1} a_i j k_j)$$

and estimates

$$y_{n+1} = y_n + h\sum_{i=1}^{s} b_i k_i$$
$$\hat{y}_{n+1} = y_n + h\sum_{i=1}^{t} \hat{b}_i k_i$$

of different order that share stages. Using an extended Butcher tableau this can also be written as

$$\begin{array}{c|cccc} 0 & & & & \\ c_2 & a_{21} & & & \\ \vdots & \vdots & \ddots & & \\ c_s & a_{s1} & a_{s2} & a_{ss-1} & \\ \hline & b_1 & b_2 & \cdots & b_s \\ \hline & \hat{b}_1 & \hat{b}_2 & \cdots & \hat{b}_s \end{array}$$

A method of fifth and sixth order RK 5(6) is derived by Fehlberg in [Feh69a] as

$$\begin{array}{c|cccccccc} 0 & & & & & & & & \\ \frac{1}{6} & \frac{1}{6} & & & & & & & \\ \frac{4}{15} & \frac{4}{75} & \frac{16}{75} & & & & & & \\ \frac{2}{3} & \frac{5}{6} & -\frac{8}{3} & \frac{5}{2} & & & & & \\ \frac{4}{5} & -\frac{8}{5} & \frac{144}{25} & -4 & \frac{16}{25} & & & & \\ 1 & \frac{361}{320} & -\frac{18}{5} & \frac{407}{128} & -\frac{11}{80} & \frac{55}{128} & & & \\ 0 & -\frac{11}{640} & 0 & \frac{11}{256} & -\frac{11}{160} & \frac{11}{256} & 0 & & \\ 1 & \frac{93}{640} & -\frac{18}{5} & \frac{803}{256} & -\frac{11}{160} & \frac{99}{256} & 0 & 1 & \\ \hline & \frac{31}{384} & 0 & \frac{1125}{2816} & \frac{9}{32} & \frac{125}{768} & \frac{5}{66} & & \\ \hline & \frac{7}{1408} & 0 & \frac{1125}{2816} & \frac{9}{32} & \frac{125}{768} & 0 & \frac{5}{66} & \frac{5}{66} \end{array}$$

with an estimate of the local truncation error of

$$\tau = \frac{5}{66}(k_1 + k_6 - k_7 + k_8)h.$$

Here the 5th order method is used to compute the solution, while the 6th order method is only used to estimate the error made by the 5th order method. The coefficients of the

0													
$\frac{2}{27}$	$\frac{2}{27}$												
$\frac{1}{9}$	$\frac{1}{36}$	$\frac{1}{12}$											
$\frac{1}{6}$	$\frac{1}{24}$	0	$\frac{1}{8}$										
$\frac{5}{12}$	$\frac{5}{12}$	0	$-\frac{25}{16}$	$\frac{25}{16}$									
$\frac{1}{2}$	$\frac{1}{20}$	0	0	$\frac{1}{4}$	$\frac{1}{5}$								
$\frac{5}{6}$	$-\frac{25}{108}$	0	0	$\frac{125}{108}$	$-\frac{65}{27}$	$\frac{125}{54}$							
$\frac{1}{6}$	$\frac{31}{300}$	0	0	0	$\frac{61}{225}$	$-\frac{2}{9}$	$\frac{13}{900}$						
$\frac{2}{3}$	2	0	0	$-\frac{53}{6}$	$\frac{704}{45}$	$-\frac{107}{9}$	$\frac{67}{90}$	3					
$\frac{1}{3}$	$-\frac{91}{108}$	0	0	$\frac{23}{108}$	$-\frac{976}{135}$	$\frac{311}{54}$	$-\frac{19}{60}$	$\frac{17}{6}$	$-\frac{1}{12}$				
1	$\frac{2383}{4100}$	0	0	$-\frac{341}{164}$	$\frac{4496}{1025}$	$-\frac{301}{82}$	$\frac{2133}{4100}$	$\frac{45}{22}$	$\frac{45}{164}$	$\frac{18}{41}$			
0	$\frac{3}{205}$	0	0	0	0	$-\frac{6}{41}$	$-\frac{3}{205}$	$-\frac{3}{41}$	$\frac{3}{41}$	$\frac{6}{41}$	0		
1	$-\frac{1777}{4100}$	0	0	$-\frac{341}{164}$	$\frac{4494}{1025}$	$-\frac{289}{82}$	$\frac{2193}{4100}$	$\frac{51}{82}$	$\frac{33}{164}$	$\frac{12}{41}$	0	1	
	$\frac{41}{840}$	0	0	0	0	$\frac{34}{105}$	$\frac{9}{35}$	$\frac{9}{35}$	$\frac{9}{280}$	$\frac{9}{280}$	$\frac{41}{840}$		
	0	0	0	0	0	$\frac{34}{105}$	$\frac{9}{35}$	$\frac{9}{35}$	$\frac{9}{280}$	$\frac{9}{280}$	0	$\frac{41}{840}$	$\frac{41}{840}$

Table 2.2.: Coefficients of RK 7(8).

method are carefully chosen in a way such that the first element of the truncation error is small compared to other methods of the same order (for instance Kutta's methods) thus allowing for larger step sizes without loss in accuracy. Fehlberg also derived an RK 7(8) method which computes the solution with a 7th order method and estimates the error with an 8th order method (see table 2.2).

A method RK 8(9) is given by Fehlberg in [Feh68]. However, the gains compared to RK 7(8) are only small, so Fehlberg concludes, that the optimum with those methods is reached [Feh69a]. Methods of even higher order would require more function evaluations and the number of stages grows stronger than the order. After all, the methods given by Fehlberg provide savings of 40% to 60% computation time compared to previous methods of the same order by carefully chosen coefficients that reduce the local truncation error.

In a later work Fehlberg investigated the application of Runge-Kutta methods to heat transfer problems [Feh69b, Feh70]. For such problems choosing a small grid size for the discretization in space requires to use a small step size for integration over time. Runge-Kutta methods of high order, that become efficient for big step sizes only, thus are of limited use. Fehlberg provides several methods of lower orders that are beneficial in such cases.

Two such RK 2(3) methods are given by the tableaux

$$\begin{array}{c|ccc} 0 & & & \\ 1 & 1 & & \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{4} & \\ \hline & \frac{1}{2} & \frac{1}{2} & \\ \hline & \frac{1}{6} & \frac{1}{6} & \frac{2}{3} \end{array} \qquad \begin{array}{c|cccc} 0 & & & & \\ \frac{1}{4} & \frac{1}{4} & & & \\ \frac{27}{40} & -\frac{189}{800} & \frac{729}{800} & & \\ 1 & \frac{214}{891} & \frac{1}{33} & \frac{650}{891} & \\ \hline & \frac{214}{891} & \frac{1}{33} & \frac{650}{891} & \\ \hline & \frac{533}{2106} & 0 & \frac{800}{1053} & -\frac{1}{78} \end{array}$$

where the left method is an extension to the so called *improved Euler-Cauchy* method. The other method, while being RK 2(3), has an extra stage providing some freedom in the choice of coefficients. This allows to reduce the local truncation error and to make the last function evaluation of the current step equal to the first of the next step (here $a_{3i} = b_i$). Such methods are also called FSAL (first same as last). Effectively only three function evaluations are needed except for the first and unsuccessful steps, which are rare with a good step size control.

A Runge-Kutta pair often referred to by other publications as RKF4(5) is given by the tableau

$$\begin{array}{c|cccccc} 0 & & & & & & \\ \frac{1}{4} & \frac{1}{4} & & & & & \\ \frac{3}{8} & \frac{3}{32} & \frac{9}{32} & & & & \\ \frac{12}{13} & \frac{1932}{2197} & -\frac{7200}{2197} & \frac{7296}{2197} & & & \\ 1 & \frac{439}{216} & -8 & \frac{3680}{513} & -\frac{845}{4104} & & \\ \frac{1}{2} & -\frac{8}{27} & 2 & -\frac{3544}{2565} & \frac{1859}{4104} & -\frac{11}{40} & \\ \hline & \frac{25}{216} & 0 & \frac{1408}{2565} & \frac{2197}{4104} & -\frac{1}{5} & \\ \hline & \frac{16}{135} & 0 & \frac{6656}{12825} & \frac{28561}{56430} & -\frac{9}{50} & \frac{2}{55} \end{array} \tag{2.11}$$

and provided by Fehlberg in [Feh69b, Feh70].

Another method given by the tableau

$$\begin{array}{c|ccc} 0 & & & \\ \frac{1}{2} & \frac{1}{2} & & \\ 1 & \frac{1}{256} & \frac{255}{256} & \\ \hline & \frac{1}{256} & \frac{255}{256} & \\ \hline & \frac{1}{512} & \frac{255}{256} & \frac{1}{512} \end{array}$$

is RK 1(2) and requires one function evaluation per step like the Euler method (because it is FSAL), but with a much smaller error term allowing for greater step sizes and making it more efficient. Also interesting is, that the denominators of all coefficients are a power of two, which might be beneficial in some cases.

Fehlberg tried to minimize the leading terms of the truncation error for the lower order method, but that can result in underestimation of the actual truncation error [DP78].

This in turn can lead to poor step size control. Therefore the leading truncation terms for the higher order method should be minimized and this method should also be used to compute the solution.

Dormand and Prince presented in [DP80] a family of improved Runge-Kutta formulae, for instance RK5(4)7M is given by the tableau

$$
\begin{array}{c|ccccccc}
0 & & & & & & & \\
\frac{1}{5} & \frac{1}{5} & & & & & & \\
\frac{3}{10} & \frac{3}{40} & \frac{9}{40} & & & & & \\
\frac{4}{5} & \frac{44}{45} & -\frac{56}{15} & \frac{32}{9} & & & & \\
\frac{8}{9} & \frac{19372}{6561} & -\frac{25360}{2187} & \frac{64448}{6561} & -\frac{212}{729} & & & \\
1 & \frac{9017}{3168} & -\frac{355}{33} & \frac{46732}{5247} & \frac{49}{176} & -\frac{5103}{18656} & & \\
1 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & \\
\hline
 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & 0 \\
\hline
 & \frac{5179}{57600} & 0 & \frac{7571}{16695} & \frac{393}{640} & -\frac{92097}{339200} & \frac{187}{2100} & \frac{1}{40}
\end{array}
$$

and provides better accuracy and a slightly improved region of absolute stability (the RK5(4)7S provides an even larger stability region) compared to RK4(5) [Feh69b]. As can be seen, the method is also using the "Fehlberg trick" of reusing the last function evaluation of the current step as first function evaluation of the next step, resulting effectively in six evaluations per step.

For automatic step size control the new step size can be calculated from the estimate of the truncation error $\tau = y_{n+1} - \hat{y}_{n+1}$ and the old step size h as

$$h_{new} = 0.9h \left(\frac{\epsilon}{||\tau||_\infty} \right)^{\frac{1}{p+1}},$$

where p is the order of the lower order method and ϵ the allowed *error per step* and

$$h_{new} = 0.9h \left(\frac{\epsilon}{||\frac{\tau}{h}||_\infty} \right)^{\frac{1}{p}},$$

if ϵ is allowed *error per unit step* [HEFS72]. The value of 0.9 is a safety factor ensuring that taking a step too small is more likely than taking a step too big.

Another way to improve efficiency of Runge-Kutta methods is to group integration into a block of N steps. By this the evaluations computed for previous steps are available for later steps and fewer evaluations are needed in total. Still the benefits of one-step Runge-Kutta methods like being self-starting (unlike multistep methods, see section 2.8) are kept. Rosser mentions in [Ros67] three references [Col66, Mil53, Sar65] that make use of a block of steps, but in those cases this was used only to obtain starting values for a *predictor-corrector method* (section 2.8.3). The use of block embedded Runge-Kutta (BERK) methods as stand-alone numerical method is described in [Ros67] and [Cas85]. Also because multiple solution points and their derivatives are available in a

block, interpolation (see next section 2.6) for such methods is relatively easy and cheap. A more thorough overview over BERK methods and comparisons with standard Runge-Kutta methods is given in [Cas83a] and [Cas83b].

In case the solution contains discontinuities or sharp slopes, the derivatives have large values in such regions. This causes rejections of the step until an appropriate step size is found. The idea of Cash and Karp [CK90] was to detect such cases early by embedding formulas of orders 1 through 4 into a fifth-order formula. The computation of the step can then be aborted or a lower order solution can be accepted, whatever seems more appropriate.

The formulas were designed by considering several criteria. The coefficients were chosen in a way that every elementary differential contributes to the local error and its estimate, as suggested in [DP80]. Further, the relative size of local error in the imbedded formula compared to that of the higher order formula was concerned, as was investigated by Shampine in [Sha86]. Finally, the coefficients c_i were chosen in a way that they span the interval $[0, 1]$ approximately uniformly so as to detect discontinuous behaviour of f, if any occurs. The embedded method that was derived is given by the following tableau:

0							
$\frac{1}{5}$	$\frac{1}{5}$						
$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$					
$\frac{3}{5}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$				
1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$			
$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$		
	$\frac{37}{378}$	0	$\frac{250}{621}$	$\frac{125}{594}$	0	$\frac{512}{1771}$	Order 5
	$\frac{2825}{27648}$	0	$\frac{18575}{48384}$	$\frac{13525}{55296}$	$\frac{277}{14336}$	$\frac{1}{4}$	Order 4
	$\frac{19}{54}$	0	$-\frac{10}{27}$	$\frac{55}{54}$	0	0	Order 3
	$-\frac{3}{2}$	$\frac{5}{2}$	0	0	0	0	Order 2
	1	0	0	0	0	0	Order 1

It is assumed that all six function evaluations of the previous step are available. Denoting with $y_n^{(i)}$ the imbedded solution of order i at t_n

$$ERR(n, i) = ||y_n^{(i+1)} - y_n^{(i)}||^{\frac{1}{i+1}},\ i \in \{1, 2, 4\}$$

can be defined. Note that $ERR(n, 3)$ is not defined since all stages need to be computed to obtain $y_n^{(4)}$ and then quitting early would not be possible anymore. Given the current step size $h = t_n - t_{n-1}$, an optimized step size for the next step for the methods of different orders can be computed by

$$h_n^{(i)} = \frac{SF \times h}{E(n, i)} \text{ with } E(n, i) = \frac{ERR(n, i)}{\epsilon^{\frac{1}{i+1}}},\ i \in \{1, 2, 4\}$$

where ϵ is the desired accuracy and SF a safety factor (usually 0.9). Some quitting factors

$$QUIT(n,i) = \frac{h_n^{(4)}}{h_n^{(i)}} = \frac{E(n,i)}{E(n,4)},\ i \in \{1,2\}$$

are introduced. To compute $h_{n+1}^{(1)}$ the first two stages k_1 and k_2 must be computed together with first- and second-order solutions $y_{n+1}^{(1)}$ and $y_{n+1}^{(2)}$. Under the assumption $QUIT(n+1,1) \approx QUIT(n,1)$ the optimized step size for the fifth-order method can be estimated as

$$h_{n+1}^{(4)} = QUIT(n+1,1) \times h_{n+1}^{(1)} \approx QUIT(n,1) \times h_{n+1}^{(1)} = \frac{QUIT(n,1) \times SF \times h}{E(n+1,1)}.$$

If the step size for the fifth order in the next step would be reduced by too much, that is if $h_{n+1}^{(4)} < SF \times h$ or equivalently $QUIT(n,1) < E(n+1,1)$, then it is expected that the fifth-order solution will be rejected and so the current step should be aborted.

In case the current step might be successful, then the appropriate order solution is used. For $y_{n+1}^{(2)}$ only the interval $[t_n, t_n + h/5]$ was checked, so the integration will only be advanced up to $t_n + h/5$, if the error estimate for this range is small enough. Similarly, if $y_{n+1}^{(3)}$ is chosen, integration would only be advanced to $t_n + 3h/5$, if the error estimate allows this. A detailed algorithm for the whole method is given in [CK90].

2.6. Interpolation for Runge-Kutta Methods

The Runge-Kutta methods calculate as solution a set of nodes (t_n, y_n). Sometimes it might be desirable to obtain function values at intermediate points, for instance to perform root finding of a switching function for event based integration or for integrating delayed differential equations. As solution values y_i and derivatives y_i' are readily available, Hermite interpolation of these values would seem to be a natural choice, but has its issues [Sha85]. Instead interpolation should be local, so that it only depends on data of the current step.

One approach was given in [O'R70] for the integration of differential equations with discontinuities signalled by an implicit function $\phi(t,y)$. The calculations for the step are performed as usual, but when ϕ changes its sign during the step, a discontinuity is found and the exact fraction αh of the step is computed. The computed stages are then reused to interpolate to the discontinuity at $t_n + \alpha h$. This is obtained basically by deriving coefficients b_i for a new Runge-Kutta method. The b_i and α will usually be found by a Newton iteration over α. O'Regan demonstrates the approach on an example using the classical Runge-Kutta method for integration. Still the question remains if using stages located behind the discontinuity, that is $\alpha < c_i$ for such stages, is meaningful and which discontinuity will be found if more than one is located inside the step. Also this method suffers from the 4th order Runge-Kutta method dropping to a 3rd order interpolation.

Horn [Hor81, Hor82, Hor83] performs the interpolation only after a successful step. This way data at the end of the step is available in addition to the stages of the Runge-Kutta method for the current step.

A scaled Runge-Kutta method is proposed, for which

$$y^* = y(t_n + h^*) = y_n + h^* \sum_{i=1}^{s^*} b_i^* k_i^*$$

with a number s^* of stages, $s^* \geq s$,

$$k_i^* = f(t_n + c_i^* h^*, y_n + h^* \sum_{j=1}^{i-1} a_{ij}^* k_j^*)$$

and $h^* = \sigma h$ with $0 \leq \sigma \leq 1$. The coefficients c_i^* and a_{ij}^* are chosen so that $k_i = k_i^*$ by setting $c_i^* = c_i/\sigma$ and $a_{ij}^* = a_{ij}/\sigma, 1 \leq i \leq s$. Evaluation of additional stages $k_{s<i\leq s^*}^*$ may be needed to obtain a scaled method of proper order.

Horn states in [Hor81] that for given method of third order a scaled method of the same order can be obtained without further function evaluations, while for a fourth order method only a single additional evaluation is needed. For a fifth order method, per two additional evaluations a scaled solution point may be obtained, or using five additional evaluations a scaled method to compute as many intermediate points for dense output may be derived. Horn derived those for the RKF4(5) method in equation (2.11), for instance. A comparison of the scaled versions of the Fehlberg Pair, Dormand and Prince pair and Sarafyan pair can be found in [PST88].

As pointed out by Shampine [Sha86], a drawback of Horn's method is that the interpolants do not transition smoothly between the steps, that is y^* does not tend to y_{n+1} as $\sigma \to 1$. Shampine therefore proposes to include solution and derivative at beginning and end of the step, as interpolation is only performed after a successful step. Additionally some methods provide solution points for free. For instance, in RKF4(5) a 4th order solution is produced at $t_n + 0.6h$, as pointed out by Horn [Hor83].

Interpolation according to Shampine [Sha85] is performed the following way. A number of stages defines solution points and derivatives by

$$\begin{aligned} t_{n,i} &= t_n + c_i h, \\ y_{n,i} &= y_n + h \sum_{j=1}^{s} a_{ij} k_j, \\ k_i &= f(t_{n,i}, y_{n,i}) = y'_{n,i}, \end{aligned}$$

and

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i.$$

Additional stages might be added, for instance,

$$k_{s+1} = f(t_n + h, y_{n+1}),$$

which is right available after a successful step. A subset of those values is selected such that

$$\begin{aligned} \zeta_k &= t_{n,j(k)}, \quad k = 1, \ldots, m \\ u_k &= y_{n,j(k)}, \quad k = 1, \ldots, m \\ u'_k &= y'_{n,j(k)}, \quad k = 1, \ldots, r \end{aligned}$$

with $j(k)$ a suitable function and $r \leq m$. A hermite interpolant $p(t)$ is then defined according to the conditions

$$\begin{aligned} p(\zeta_k) &= u_k \\ p'(\zeta_k) &= u'_k \end{aligned}$$

as

$$p(t) = \sum_{i=1}^{m} A_i(t) u_i + \sum_{i=1}^{r} B_i(t) u'_i$$

where

$$\begin{aligned} A_i(t) &= \begin{cases} \{1 - (t - \zeta_i)[L'_{i,m}(\zeta_i) + L'_{i,r}(\zeta_i)]\} L_{i,m}(t) L_{i,r}(t), & i = 1, \ldots, r, \\ L_{i,m}(t) \prod_{k=1}^{r} \left(\frac{t-\zeta_k}{\zeta_i - \zeta_k} \right), & i = r+1, \ldots, m \end{cases} \\ B_i(t) &= (t - \zeta_i) L_{i,m}(t) L_{i,r}(t) \end{aligned}$$

and

$$L_{i,m}(t) = \prod_{\substack{k=1 \\ k \neq i}}^{m} \left(\frac{t - \zeta_k}{\zeta_i - \zeta_k} \right), \quad i = 1, \ldots, m.$$

2.7. Implicit Runge-Kutta Methods

For stiff ODEs the choice of the integration step size may not be determined by accuracy requirements anymore, but by stability instead. Explicit Runge-Kutta methods tend to decrease their step size dramatically while increasing their computational load at the same time just for the purpose of keeping the step size in their region of stability. In the worst case round-off errors may accumulate up to the point where no successful integration with such an explicit method is possible. A better alternative in such cases are implicit Runge-Kutta processes [But64a].

An implicit Runge-Kutta process is characterized by having non-zero coefficients a_{ij} for $i \leq j$. In this case, the result calculated for some stages depends on the result of those stages themselves. An example is the *implicit Euler method* (also called *backward Euler*, as opposed to *forward Euler* defined by equation (2.2))

$$y_{n+1} = y_n + h f(t_{n+1}, y_{n+1}).$$

Under certain conditions (the function $u(t)$ is Lipschitz continuous and h is sufficiently small) a unique solution can be found by iteration (e.g., fixed point iteration or Newton-Raphson). Butcher has shown in [But64a] that $|h|La < 1$ must be fulfilled for the simple iteration to converge, where L is the Lipschitz constant of f and $a = l + u$ with

$$l = \max_i \left\{ \sum_{j=1}^{i-1} |a_{ij}| \right\} \text{ and } u = \max_i \left\{ \sum_{j=i}^{s} |a_{ij}| \right\}.$$

The stages of the Runge-Kutta method can be rewritten into another form

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i f(t_n + c_i h, Y_i),$$

where the approximations Y_i are given by

$$Y_i = y_n + h \sum_{j=1}^{s} a_{ij} f(t_n + c_j h, Y_j).$$

For stiff ODEs, a modified Newton-Raphson method is used to solve the implicit equations [But76, GSDT85]. In each iteration the values $Y_i^{(k-1)}$ are replaced by $Y_i^{(k)} = Y_i^{(k-1)} + \omega_i^{(k)}$, where the $\omega_i^{(k)}$ are given by

$$\omega_i^{(k)} - h \sum_{j=1}^{s} a_{ij} J \omega_j^{(k)} - z_i^{(k)} = 0,$$

with J being the Jacobian matrix of f and z_i being the residuals defined by

$$z_i^{(k)} = -Y_i^{(k-1)} + y_n + h \sum_{j=1}^{s} a_{ij} f(t_n + c_j h, Y_j^{(k-1)}).$$

The Jacobian J can be approximated by finite differences as shown in [PTVF07].

Rewritten into matrix form one obtains a set of linear equations to solve:

$$\begin{pmatrix} I - ha_{11}J & -ha_{12}J & \cdots & -ha_{1s}J \\ -ha_{21}J & I - ha_{22}J & \cdots & -ha_{2s}J \\ \vdots & \vdots & & \vdots \\ -ha_{s1}J & -ha_{s2}J & \cdots & I - ha_{ss}J \end{pmatrix} \begin{pmatrix} w_1^{(k)} \\ w_2^{(k)} \\ \vdots \\ w_s^{(k)} \end{pmatrix} = \begin{pmatrix} z_1^{(k)} \\ z_2^{(k)} \\ \vdots \\ z_s^{(k)} \end{pmatrix}. \tag{2.12}$$

Solving such a system is costly and can be done with classical techniques like LU factorization. To reduce the amount of computation, Butcher suggests in [But76] to transform the inverse A^{-1} of the coefficient matrix $A = (a_{ij})$ into the Jordan canonical form

$$T^{-1}A^{-1}T = \begin{pmatrix} \lambda_1^{-1} & 0 & 0 & \dots & 0 \\ \mu_1 & \lambda_2^{-1} & 0 & \dots & 0 \\ 0 & \mu_2 & \lambda_3^{-1} & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & \lambda_s^{-1} \end{pmatrix},$$

where μ_i is zero if $\lambda_i \neq \lambda_{i+1}$, zero or an arbitrary non-zero number otherwise. If it is non-zero then $\mu_i = \lambda_i^{-1}$. Let D be the diagonal matrix with diagonal entries $\lambda_1, \lambda_2, \ldots, \lambda_s$. Then select two matrices $P = (p_{ij}) = DT^{-1}A^{-1}$ and $Q = (q_{ij}) = T$ so that

$$PQ = \begin{pmatrix} 1 & 0 & 0 & \ldots & 0 \\ \epsilon_1 & 1 & 0 & \ldots & 0 \\ 0 & \epsilon_2 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \ldots & 1 \end{pmatrix},$$

with the ϵ_i either zero or one. Application of the transformation

$$\tilde{\omega}_i^{(k)} = \sum_{j=1}^{s} q_{ij}\omega_i^{(k)} \text{ and } \tilde{z}_i^{(k)} = \sum_{j=1}^{s} q_{ij} z_i^{(k)}$$

reduces the system (2.12) to

$$\begin{pmatrix} I - h\lambda_1 J & 0 & \cdots & 0 \\ -h\mu_1 J & I - h\lambda_2 J & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & I - h\lambda_s J \end{pmatrix} \begin{pmatrix} \tilde{w}_1^{(k)} \\ \tilde{w}_2^{(k)} \\ \vdots \\ \tilde{w}_s^{(k)} \end{pmatrix} = \begin{pmatrix} \tilde{z}_1^{(k)} \\ \tilde{z}_2^{(k)} \\ \vdots \\ \tilde{z}_s^{(k)} \end{pmatrix}.$$

Instead of having to solve one $sN \times sN$ linear system we now only have to solve s different $N \times N$ systems sequentially.

Examples for implicit Runge-Kutta processes based on Gaussian quadrature formulas with a number $s = 1, 2, 3, 4, 5$ of stages and orders $2, 4, 6, 8, 10$ are given by Butcher [But64a]. For instance, the coefficients in the tableau

$$\begin{array}{c|cc} \frac{1}{2} - \frac{\sqrt{3}}{6} & \frac{1}{4} & \frac{1}{4} - \frac{\sqrt{3}}{6} \\ \frac{1}{2} + \frac{\sqrt{3}}{6} & \frac{1}{4} + \frac{\sqrt{3}}{6} & \frac{1}{4} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

define an implicit Runge-Kutta method with 2 stages and of 4th order, and was previously given by Hammer and Hollingsworth [HH55]. Other families of implicit Runge-Kutta processes exist, for instance the Radau family of order $2s - 1$ and the Lobatto family of order $2s - 2$.

2.8. Linear Multistep Methods

The methods previously covered, namely the one-step Runge-Kutta methods, only used information from the current step to calculate the next point of the solution. Multistep methods increase efficiency by using several previous solution points instead. A *linear*

multistep formula (LMF) approximates the next point of the solution by a linear combination of previous solution points and their derivatives. The general form of an LMF according to Dahlquist [Dah56] is

$$\sum_{i=0}^{k} \alpha_i y_{n+i} = h \sum_{i=0}^{k} \beta_i f(t_{n+i}, y_{n+i}), \tag{2.13}$$

where h is the step size and k is the number of previous points. The point (t_{n+k}, y_{n+k}) is the next to compute, $f(t_{n+k}, y_{n+k})$ its derivative. Normalization is obtained by setting $\alpha_k = 1$. If $\beta_k = 0$ the method is explicit (Adams-Bashforth type), otherwise it is implicit (Adams-Moulton type). In the latter case y_{n+k} can be found via simple iteration (in the non-stiff case) or Newton's method (in the stiff case).

To start with an LMF the k initial points need to be obtained. A Taylor series expansion of $y(t)$ around t_0 can be used if the solution is analytic at t_0 [Mil26]. Another possibility also suggested by Milne uses successive approximations by iterating a formula obtained from the polynomial $p(t)$. One could as well use Runge-Kutta methods, especially [Col66, Mil53, Sar65], to obtain starting values. Working with LMFs of variable orders also allows to find starting values successively.

2.8.1. Adams-Bashforth Methods

In [Bas83] Francis Bashforth describes an attempt to test the theories of capillary action by forms of fluid drops. The numerical methods for this were developed by John Couch Adams. The idea is to interpolate previous derivatives $f_{n-i} = f(t_{n-i}, y_{n-i}), 0 \leq i < k$ by a polynomial $p(t)$ and replace the integrand in the equation

$$y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} f(t, y(t))dt$$

by $p(t)$. Analytic integration of the polynomial then yields a numerical integration method. The methods obtained for different values of k are given in table 2.3. Note that for $k = 1$ the Euler method is obtained.

$$\begin{aligned}
k = 1: &\quad y_{n+1} = y_n + h f_n \\
k = 2: &\quad y_{n+1} = y_n + \tfrac{h}{2}(3f_n - f_{n-1}) \\
k = 3: &\quad y_{n+1} = y_n + \tfrac{h}{12}(23f_n - 16f_{n-1} + 5f_{n-2}) \\
k = 4: &\quad y_{n+1} = y_n + \tfrac{h}{24}(55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3})
\end{aligned}$$

Table 2.3.: Adams-Bashforth methods.

2.8.2. Adams-Moulton Methods

Implicit LMFs can be obtained by including f_{n+1} into the calculation of y_{n+1}. The polynomial $p(t)$ now interpolates $f_{n-i} = f(t_{n-i}, y_{n-i}), -1 \leq i < k$. Methods obtained

this way are shown in table 2.4 and have been derived in [Mou26]. Solution of the implicit equation can be performed by fixed-point iteration (in the non-stiff case) or by Newton iteration (in the stiff case). Fixed-point iteration only converges if $|h\beta_k L| < 1$ with L the Lipschitz constant of f.

$$\begin{aligned}
k = 0: &\quad y_{n+1} = y_n + h f_{n+1} \\
k = 1: &\quad y_{n+1} = y_n + \tfrac{h}{2}\left(f_{n+1} + f_n\right) \\
k = 2: &\quad y_{n+1} = y_n + \tfrac{h}{12}\left(5f_{n+1} + 8f_n - f_{n-1}\right) \\
k = 3: &\quad y_{n+1} = y_n + \tfrac{h}{24}\left(9f_{n+1} + 19f_n - 5f_{n-1} + f_{n-2}\right)
\end{aligned}$$

Table 2.4.: Adams-Moulton methods.

2.8.3. Predictor-Corrector Methods

Another way how to solve the implicit equations is to use those methods in combination with the explicit ones (both using the same k). The explicit method predicts the next solution $\hat{y}_{n+1}$ and the implicit method then uses this value to compute $f_{n+1} = f(t_{n+1}, \hat{y}_{n+1})$ and y_{n+1}. The amount of correction needed gives an estimate for the error made [Mil26].

Such a predictor-corrector scheme is also named PECE, where P denotes prediction, C denotes correction, and E refers to evaluation of f. If the correction step is performed several times, the method is P(EC)^{n}E.

2.8.4. Nordsieck Method

Better performance of LMFs can be achieved by using a variable step size. Compared to the one-step Runge-Kutta methods, this requires regeneration of the k previous values upon each change of step size. One way to obtain this is to ignore old values and restart with $k = 1$ (Euler method), then working up to higher order methods. A better alternative is to obtain new values by interpolation/extrapolation.

An even better way to store the values was suggested by Nordsieck [Nor62]. He suggested, instead of storing previous values of y or f, to store the $k+1$ scaled derivatives $\frac{h^i}{i!} y^{(i)}(t_n), 0 \leq i \leq k$ for the current step (*Nordsieck vector*). This way the stored values are independent of the current step size and thus allow easily to work with a variable step size. Changing step size from h_{old} to h_{new} with $\nu = h_{new}/h_{old}$ is a simple matter of scaling each row of N by ν^i, i.e. multiplication of N by a matrix $diag(1, \nu, \nu^2, \ldots, \nu^k)$.

The value of the next step can then be computed by a Taylor series as

$$y(t+h) = \sum_{i=0}^{k} \frac{h^i}{i!} y^{(i)}(t) + \mathcal{O}(h^{k+1}),$$

which is exact for $y(t)$ being a polynomial of degree up to k. Higher order derivatives can be calculated in a similar way. For a general formulation to advance the Nordsieck vector see [Gea67] and see also equation (2.5) in [CLM89]. Gear [Gea67] also showed how to transform between the Adams and the Nordsieck representation.

2.8.5. Backward Differentiation Formula

Unlike the Adams formulas, that interpolate the derivatives f_i, the backward differentiation formulas (BDF) interpolate the previous solution values y_i and y_{n+1} such that the derivative of this polynomial at t_{n+1} agrees with f_{n+1}. By construction these methods are implicit in nature and as such are useful to solve stiff problems (if Newton iteration is used).

For a k-step BDF a polynomial $q(t)$ is constructed through Newton interpolation of $y_{n+1}, y_n, \ldots, y_{n-k+1}$ and by requiring that $q'(t_{n+1}) = f(t_{n+1}, q(t_{n+1}))$ one obtains

$$\sum_{i=1}^{k} \frac{1}{i} \Delta^i y_{n+1} = h f_{n+1}$$

with $\Delta^{i+1} y_{n+1} = \Delta^i y_{n+1} - \Delta^i y_n$ and $\Delta^0 y_n = y_n$. Conversion into the form of equation (2.13) results in the BDFs shown in table 2.5. They have been introduced in [CH52] and have been also described in [Gea71]. Methods with $k \leq 2$ are A-stable, those with $k > 2$ are at least $A(\alpha)$-stable. Methods with $k \geq 7$ are of no use as they are not stable.

$$\begin{aligned}
k = 1: &\quad y_{n+1} - y_n = h f_{n+1} \\
k = 2: &\quad \tfrac{3}{2} y_{n+1} - 2 y_n + \tfrac{1}{2} y_{n-1} = h f_{n+1} \\
k = 3: &\quad \tfrac{11}{6} y_{n+1} - 3 y_n + \tfrac{3}{2} y_{n-1} - \tfrac{1}{3} y_{n-2} = h f_{n+1} \\
k = 4: &\quad \tfrac{25}{12} y_{n+1} - 4 y_n + 3 y_{n-1} - \tfrac{4}{3} y_{n-2} + \tfrac{1}{4} y_{n-3} = h f_{n+1} \\
k = 5: &\quad \tfrac{137}{60} y_{n+1} - 5 y_n + 5 y_{n-1} - \tfrac{10}{3} y_{n-2} + \tfrac{5}{4} y_{n-3} - \tfrac{1}{5} y_{n-4} = h f_{n+1} \\
k = 6: &\quad \tfrac{49}{20} y_{n+1} - 6 y_n + \tfrac{15}{2} y_{n-1} - \tfrac{20}{3} y_{n-2} + \tfrac{15}{4} y_{n-3} - \tfrac{6}{5} y_{n-4} + \tfrac{1}{6} y_{n-5} = h f_{n+1}
\end{aligned}$$

Table 2.5.: Backward differentiation formulas.

2.9. Extrapolation Methods

The idea of extrapolation goes back to Lewis Fry Richardson [Ric11, Ric27]. When using central differences [She99] the approximation $\phi(x, h)$ for a function $f(x)$ can be expressed by

$$\phi(x, h) = f(x) + h^2 f_2(x) + h^4 f_4(x) + \ldots$$

with h occurring only in even powers.[5] The functions $f_2, f_4, \ldots$ of the error terms are generally unknown. Richardson now observed that knowing $\phi(x, h)$ for two values h_1 and h_2 an estimate for $f(x)$ can be obtained by

$$f(x) = \frac{h_2^2 \phi(x, h_1) - h_1^2 \phi(x, h_2)}{h_2^2 - h_1^2}$$

[5] This can be checked easily by forming the Taylor series expansions of $f(x + h)$ and $f(x - h)$ around x and setting $\phi(x, h) = [f(x + h) + f(x - h)]/2$.

under the assumption that h_1 and h_2 are small enough so that $h^2 f_2(x)$ is the leading error term. Thus the combination of the two approximations removes the f_2-term and yields a more accurate approximate for $f(x)$. Similarly, using a third approximation $\phi(x, h_3)$ would allow to remove the f_4-term, and so on.

A more direct view of this for the solution of ODEs is to consider obtaining an approximation for y_{n+1} by integration from y_n over some *basic stepsize* H. For some decreasing sequence $h_0 > h_1 > \ldots > h_N > 0$ of *internal stepsizes* the function $a(h)$ gives an estimate for y_{n+1}. *Extrapolation to the limit* is then performed by fitting a polynomial through the $a(h_i)$ and evaluating $a(0)$.

Under the assumption that $a(h)$ allows an expansion of the form

$$a(h) = a(0) + a_1 h^\gamma + a_2 h^{2\gamma} + \ldots + a_N h^{N\gamma} + \mathcal{O}(h^{(N+1)\gamma})$$

the *Aitken-Neville algorithm* can be used to calculate the *extrapolation tableau*

$$\begin{array}{lllllll}
T_{00} & & & & & & \\
 & \searrow & & & & & \\
T_{10} & \rightarrow & T_{11} & & & & \\
 & \searrow & & \searrow & & & \\
T_{20} & \rightarrow & T_{21} & \rightarrow & T_{22} & & \\
\vdots & & \vdots & & & \ddots & \\
T_{N0} & \rightarrow & T_{N1} & \rightarrow & T_{N2} & \cdots & T_{NN}
\end{array}$$

with coefficients T_{ij} determined by

$$\begin{aligned}
T_{i,0} &= a(h_i) \\
T_{i,j} &= T_{i,j-1} + \frac{T_{i,j-1} - T_{i-1,j-1}}{(h_{i-j}/h_i)^\gamma - 1}.
\end{aligned}$$

Thus no explicit representation of the approximating polynomial needs to be created. The approximations $T_{i,0}$ are obtained by one of the standard methods, usually the Euler method ($\gamma = 1$) or central differences ($\gamma = 2$). It was shown that each column converges faster than the preceding one and that the diagonal converges faster than any column [Gra65].

The sequence of internal stepsizes h_i can be represented by an integer sequence

$$\mathfrak{F} = \{n_i\}$$

with $h_i = \frac{H}{n_i}$. Proposed sequences $\mathfrak{F}_R = \{2, 4, 8, 16, \ldots\}$, $\mathfrak{F}_B = \{2, 4, 6, 8, 12, 16, \ldots\}$ and $\mathfrak{F}_H = \{2, 4, 6, 8, 10, 12, 14, 16, \ldots\}$ are the Romberg [Rom55], Bulirsch [Bul64] and (double) Harmonic [Deu83] sequences. A comparison of the performance of the sequences can be found in [MN92]. Note that the *Toeplitz condition* (for a constant α)

$$\frac{n_i}{n_{i+1}} \leq \alpha < 1,$$

a necessary and sufficient condition for convergence of the $T_{i,j}$ [Gra65], is satisfied for $\mathfrak{F}_R$ and $\mathfrak{F}_B$, but not for $\mathfrak{F}_H$. But then, evaluation of the $T_{i,j}$ is not performed ad infinitum. A different approach to show convergence does not need the Toeplitz condition anymore [Deu83, Deu85].

Extrapolation methods allow to easily increase the order by just appending another row to the tableau, but by doing so the method becomes sensitive to round-off errors. Therefore Bulirsch and Stoer combined extrapolation with an automatic stepsize selection [BS66]. Basically their algorithm tries, for a given accuracy, to keep a fixed number of columns in the tableau by stepsize reduction ($\bar{H} = 0.9 \cdot H \cdot 0.6^{N-7}, N \geq 7$) and increase ($\bar{H} = 1.5 \cdot H, N < 7$). Improved algorithms, based on work per unit step, have been suggested by Stoer [Sto74] and Deuflhard [Deu83, Deu85].

For the integration of stiff systems, implicit discretizations can be used, like implicit Euler or trapezoidal and implicit mid-point rule [Deu85]. However, the fully implicit discretizations cannot compete with BDF integrators presented in section 2.8.5. Semi-implicit extrapolation methods [BD83] remedy this situation.

2.10. Symplectic Methods

Consider the following two-body problem of a function $y(t) = (y_1, y_2, y_3, y_4)^T$ defined by

$$\begin{aligned} y'(t) = f(t, y(t)) &= \begin{pmatrix} y_3 \\ y_4 \\ -y_1/r^3 \\ -y_2/r^3 \end{pmatrix}, \; r = \sqrt{y_1^2 + y_2^2}, \\ y_0 = y(t_0) &= \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \end{aligned} \tag{2.14}$$

with (y_1, y_2) describing the position and (y_3, y_4) the speed of an object orbiting another object (centered at the origin) in a 2D plane. The exact solution is a stable orbit with $r = 1$ around the object in the origin. Numerical integration using the forward Euler method causes the objects to depart from each other, while using the backward Euler method the objects approach each other (see Fig. 2.6). A mathematical proof for this and a detailed survey about symplectic integrators can be found in [Yos93].

Problems of celestial mechanics like the one above, springs or a pendulum can be described by *Hamilton equations*

$$\begin{aligned} \dot{p} &= -\frac{\partial H}{\partial q} \\ \dot{q} &= \frac{\partial H}{\partial p} \end{aligned}$$

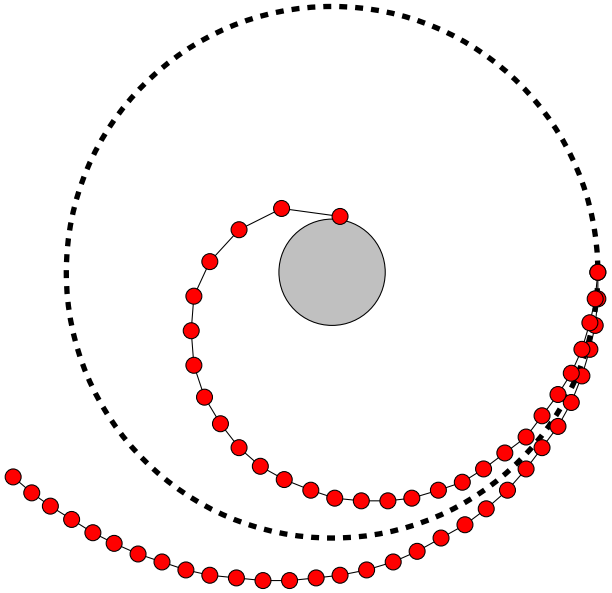


Figure 2.6.: Orbit obtained by numerical integration with forward and backward Euler method.

where H is the *Hamiltonian* and represents the energy of the system, p the *generalized momentum* and q the *generalized coordinates*. An energy conserving numerical integration scheme is called a *symplectic integrator*.

A well-known symplectic integrator is the *Verlet algorithm* [Ver67, Ver68]. The method can be derived by a Taylor series expansion of the position function $r(t)$ giving

$$r(t+h) = r(t) + hv(t) + \frac{h^2}{2}a(t) + \frac{h^3}{6}b(t) + \mathcal{O}(h^4)$$
$$r(t-h) = r(t) - hv(t) + \frac{h^2}{2}a(t) - \frac{h^3}{6}b(t) + \mathcal{O}(h^4)$$

with $v(t)$ the velocity, $a(t)$ the acceleration and $b(t)$ the jerk. Adding both equations yields the Verlet method

$$r(t+h) = 2r(t) - r(t-h) + h^2 a(t) + \mathcal{O}(h^4).$$

Only the current and previous position need to be stored and the acceleration must be computed for the current position. The method is accurate to the third order. Using this method on the problem (2.14) one obtains a trajectory as shown in figure 2.7.

For some problems knowledge of the velocity may be required as well, i.e. to compute kinetic energy. Verlet suggested to use $v(t) = [r(t+h) - r(t-h)]/2h$, which has an error of $\mathcal{O}(h^2)$. However, knowledge of the velocity always lags one step behind knowledge of

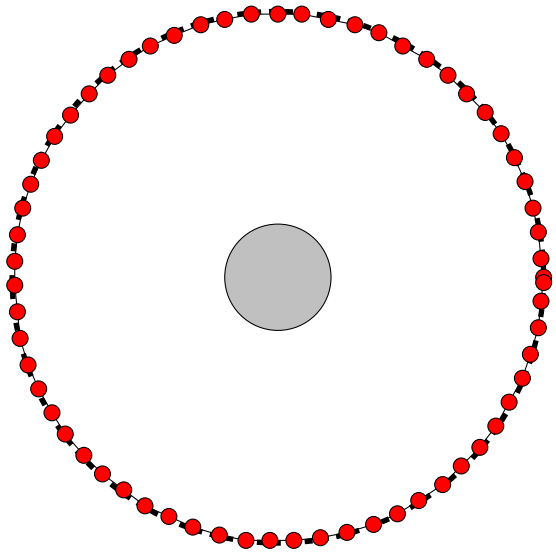


Figure 2.7.: Orbit obtained with Verlet method.

the positions. Therefore a better alternative is the *velocity Verlet algorithm*

$$r(t+h) = r(t) + hv(t) + \frac{h^2}{2}a(t)$$
$$v(t+h) = v(t) + \frac{h}{2}[a(t) + a(t+h)]$$

If acceleration $a(t+h)$ only depends on location $r(t+h)$ and not on velocity $v(t+h)$ the values can be directly computed by

$$\begin{aligned} r(t+h) &= r(t) + hv(t) + \frac{h^2}{2}a(t) \\ v(t+h/2) &= v(t) + \frac{h}{2}a(t) \\ a(t+h) &= \dots \\ v(t+h) &= v(t+h/2) + \frac{h}{2}a(t+h) \end{aligned}$$

in the given order without needing additional storage for intermediate values, as those can replace the previously stored ones.

2.11. DDEs and the Method of Steps

If in the description of a problem a time delay is involved, such problems are described by a *delay differential equation* (DDE), sometimes also called *retarded differential equation*.

Examples where DDEs are needed are population dynamics or chemical kinetics, for instance also in plant models [LlCC09]. A very simple form of such a DDE with a constant lag τ is

$$y'(t) = f(t, y(t), y(t-\tau)) \tag{2.15}$$

with an *initial function* $\Psi(t)$ for $t \in [t_0 - \tau, t_0]$. Another example with a variable lag is the *pantograph equation*

$$y'(t) = \alpha y(t) + \beta y(\nu t), \quad 0 < \nu < 1$$

with an initial value $y(0)$. More general forms of DDEs might depend on multiple previous points or have a variable lag, possibly state-dependent.

Several strategies to solve a DDE exist. For small τ it might be possible to ignore the delay at all, but counterexamples exist [Kua93]. Alternatively an ODE integrator with dense output might be used to track the solution and look up the delayed values, but then τ should not become bigger than the internal step size of the ODE integrator and the integrator must be able to detect and handle discontinuities in the solution, which usually appear.

Another way how to solve a DDE is *the method of steps*. The DDE given in equation (2.15) can be split into a set of equations

$$\begin{aligned}
y_0' &= f(t, y_0(t), \Psi(t-\tau)), && t_0 \le t \le t_1 \\
y_1' &= f(t, y_1(t), y_0(t-\tau)), && t_1 \le t \le t_2 \\
&\vdots \\
y_n' &= f(t, y_n(t), y_{n-1}(t-\tau)), && t_n \le t \le t_{n+1}
\end{aligned}$$

solved for distinct intervals $[t_i, t_{i+1}]$ of t with $t_i = t_0 + i\tau$. Integration is then performed in a step-wise fashion and gave the method its name. The solution $y(t)$ is then a concatenation of the pieces $y_0(t), y_1(t), \ldots, y_n(t)$.

In [BBK65], the variable-lag DDE

$$y'(t) = f(t, y(t), y(t - \tau(t)))$$

was solved with the method of steps by converting the DDE into a set of ODEs. The steps $[t_i, t_{i+1}]$ are defined by $t_{i+1} - \tau(t_{i+1}) = t_i$ and it is assumed that $\tau(t)$ is monotonously decreasing (therefore each step is shorter than the previous one). Definition of the lag-functions $L_0(t) = t, L_1(t) = t - \tau(t), L_k(t) = L(L_{k-1}(t))$ one obtains the set of ODEs

$$\begin{aligned}
y_0'(t) &= L_n'(t) f(L_n(t), y_0(t), \Psi(L_{n+1}(t))) \\
y_1'(t) &= L_{n-1}'(t) f(L_{n-1}(t), y_1(t), y_0(t)) \\
&\vdots \\
y_{n-1}'(t) &= L_1'(t) f(L_1(t), y_{n-1}(t), y_{n-2}(t)) \\
y_n'(t) &= L_0'(t) f(L_0(t), y_n(t), y_{n-1}(t))
\end{aligned}$$

for integrating the interval $[t_k, t_{k+1}]$ with initial conditions $y_i(t_k) = y(t_i)$ being the endpoints of the previous steps. However, for each successive step an additional ODE must be added to the system resulting in increasing work the further integration proceeds.

A better approach is to use an ODE integrator with dense output. Integration of a step results in a function that can then be sampled to integrate the next step, therefore trading memory for speed.

Now consider the simple DDE defined as

$$y'(t) = y(t - \tau)$$

with initial function $y(t) = 1$ for $t \leq 0$. Integration over the interval $[0, \tau]$ yields for the first step the function

$$y_0(t) = t + 1,$$

then for the interval $[\tau, 2\tau]$ in the second step

$$y_1(t) = \frac{t^2}{2} + t + 1 - \frac{\tau^2}{2}$$

and so on. The general formula to calculate $y(t)$ for $t \in [i\tau, (i+1)\tau]$ in this example is

$$y_i(t) = y_{i-1}(i\tau) + \int_{i\tau}^{t} y_{i-1}(s - \tau)ds.$$

As can be seen, the discontinuity at $t = 0$ propagates to $\tau, 2\tau, \ldots$, each time increasing the order by one, and thus smoothes out with increasing number of steps.

This effect on discontinuities is a general feature of DDEs and does also apply to discontinuities not at the end of the steps. Therefore and for other reasons, special DDE solvers have been developed. A more thourough discussion of the issues with numerical integration of DDEs can be found in [BPW94, BPW95]. An application of the method of steps to an optimization problem can be found in [Ros88].

2.12. PDEs and the Method of Lines

Many problems in natural sciences relate spatial quantities with temporal ones. A prominent example is the *heat-equation*

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2} \tag{2.16}$$

where $u(x, t)$ describes the evolution of heat distribution, but also diffusion processes. When applied to heat conduction it is *Fourier's second law*, where u is the temperature as a function of time t and location x and κ is the *thermal diffusivity*. When applied to diffusion processes, then it is *Fick's second law* with u describing the concentration and κ being the *diffusivity*.

An equation that contains partial derivatives of different independent variables is called *partial differential equation* (PDE). Often, the partial derivatives are written as an index, so that equation (2.16) could also be written as

$$u_t = \kappa u_{xx}$$

with $u_t = \frac{\partial u}{\partial t}$ and $u_{xx} = \frac{\partial^2 u}{\partial x^2}$.

The *order* of an independent variable is the highest derivative that occurs in the equation. For the heat-equation, t is therefore of first order and x is of second order. The order of the PDE is then the highest order among all independent variables. The *degree* of the PDE is the highest power of the terms containing the dependent variables. A first-degree equation is said to be *linear*.

2.12.1. Initial and Boundary Conditions

To complete the definition of the integration problem, *auxiliary conditions* have to be specified. For each independent variable as many conditions are needed as high the order of the variable is.

An *initial condition* (IC) for t can be given by $u(x, 0) = u_0(x)$, where $u_0(x)$ is an initial temperature distribution as a function of x. If more than one condition is needed, all of them are given *for the same value of the independent variable* (in this case t with $t_0 = 0$).

A *boundary condition* (BC) can be set *for more than one value of the independent variable*. For instance, a condition for u can be $u(0, t) = u_1(t)$, which is a *Dirichlet boundary condition* since the dependent variable u is specified. Another condition could be $u_x(L, t) = 0$, which is a *Neumann boundary condition* because the derivative of the dependent variable is specified. The first condition describes the temperature at $x = 0$ as a function of time, while the second condition describes the temperature gradient at $x = L$. If the dependent variable and its derivative occur in the condition it is called a *boundary condition of the third type*.

In most cases, the boundary conditions are given for the spatial variables and the initial conditions for the time. Care must be taken when defining the condition equations, as discontinuities (for instance by inconsistencies in the conditions for the same point) can cause problems for the solution of the PDE.

2.12.2. Solution of a PDE

Given the partial differential equation together with sufficient initial and boundary conditions, the solution of the PDE is a function $u(x, t)$ that describes the dependent variable u as a function of the independent variables x and t. The solution can either be an analytical function or a numerical approximation to such a function.

An analytic solution is useful as it is exact and thus allows to check numerical solutions for their correctness or derive asymptotic behaviour for stability analysis. However, an analytic solution can only be found in very simple cases.

For most problems, just a numerical solution can be computed. Different methods for this exist, and one of them is the *method of lines* (MOL) [Sch91]. The basic idea is to transform the PDE problem into a system of ODEs, that can then be solved by some known numerical integration method.

The first step is to discretize the PDE in all but one of the independent variables. Typically time is the variable for which the independent condition was given and which thus remains. The partial derivatives of the other variables are then replaced in the PDE by algebraic approximations. This then forms the basis for methods such as the finite element method (FEM), finite volume method (FVM) and finite difference method (FDM). We will now stick to the latter one.

2.12.3. Spatial Discretization using Finite Differences

As an example consider heat conduction through a one-dimensional metal stick with the ends located at $x = 0$ and $x = L$. A grid $x_i = i\Delta x, 0 \leq i \leq N$ with $\Delta x = L/(N+1)$ is created so that x_0 and x_N are now the endpoints of the stick, as is shown in figure 2.8.

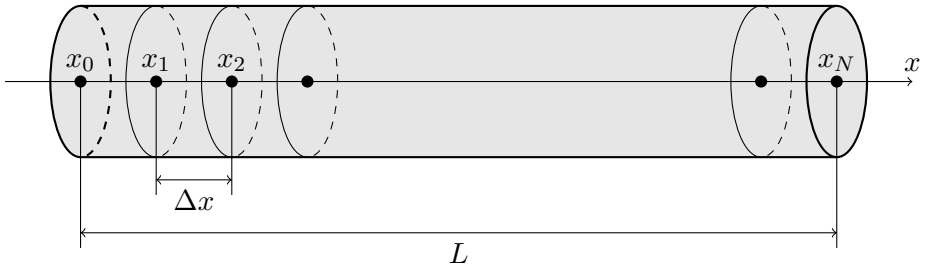


Figure 2.8.: Metal stick with spatial discretization.

To discretize a spatial derivative u_x it is approximated locally by a polynomial

$$u(x) = a_0 + a_1(x - x_i) + a_2(x - x_i)^2 + a_3(x - x_i)^3 + \cdots$$

for a point x_i on the grid. The coefficients $a_0, a_1, a_2, \ldots$ can be obtained by successive differentiation as

$$a_\nu = \frac{u^{(\nu)}(x_i)}{\nu!}$$

when setting $x = x_i$. Letting $n \to \infty$ results in the Taylor series expansion of $u(x)$.

With this, first-order derivatives with a first-order truncation error can be derived easily. The approximation of $u(x_{i+1})$ is

$$u(x_{i+1}) = u(x_i) + u'(x_i)\Delta x + \frac{1}{2}u''(x_i)\Delta x^2 + \frac{1}{6}u'''(x_i)\Delta x^3 + \mathcal{O}(\Delta x^4) \tag{2.17}$$

from which an approximation to $u'(x_i)$ follows as

$$u'(x_i) = \frac{u(x_{i+1}) - u(x_i)}{\Delta x} + \mathcal{O}(\Delta x).$$

Similarly, using

$$u(x_{i-1}) = u(x_i) - u'(x_i)\Delta x + \frac{1}{2}u''(x_i)\Delta x^2 - \frac{1}{6}u'''(x_i)\Delta x^3 + \mathcal{O}(\Delta x^4) \tag{2.18}$$

one obtains

$$u'(x_i) = \frac{u(x_i) - u(x_{i-1})}{\Delta x} + \mathcal{O}(\Delta x).$$

Subtracting equations (2.17) and (2.18) from each other gives the *second-order central finite difference* approximation

$$u'(x_i) = \frac{u(x_{i+1}) - u(x_{i-1})}{2\Delta x} + \mathcal{O}(\Delta x^2).$$

This works for the grid points $x_1, \ldots, x_{N-1}$, but a problem occurs for the points x_0 and x_N, as those would depend on points x_{-1} respectively x_{N+1} that are outside the grid. Instead, the points x_0, x_1, x_2 can be used to obtain a second-order approximation for $u'(x_0)$ and x_{N-2}, x_{N-1}, x_N for $u'(x_N)$.

Using a polynomial representation at x_0 allows to approximate $u(x_1)$ and $u(x_2)$ as

$$u(x_1) = u(x_0) + u'(x_0)\Delta x + \frac{1}{2}u''(x_0)\Delta x^2 + \mathcal{O}(\Delta x^3)$$
$$u(x_2) = u(x_0) + u'(x_0)(2\Delta x) + \frac{1}{2}u''(x_0)(2\Delta x)^2 + \mathcal{O}(\Delta x^3)$$

Elimination of $u''(x_0)$ results in

$$u'(x_0) = \frac{-3u(x_0) + 4u(x_1) - u(x_2)}{2\Delta x} + \mathcal{O}(\Delta x^2).$$

Similarly an approximation for $u'(x_N)$ can be obtained as

$$u'(x_N) = \frac{u(x_{N-2}) - 4u(x_{N-1}) + 3u(x_N)}{2\Delta x} + \mathcal{O}(\Delta x^2).$$

Adding the two equations (2.17) and (2.18) a central finite difference approximation

$$u''(x_i) = \frac{u(x_{i-1}) - 2u(x_i) + u(x_{i+1})}{\Delta x^2} + \mathcal{O}(\Delta x^2)$$

for u_{xx} in equation (2.16) can be derived. Replacing u_{xx} by the finite difference approximation in equation (2.16) reduces the PDE into a set of ODEs

$$\frac{du(x_i, t)}{dt} = \kappa \frac{u(x_{i-1}) - 2u(x_i) + u(x_{i+1})}{\Delta x^2}$$

for $1 \leq i \leq N-1$ and similar equations for the $i = 0$ and $i = N$.

Higher order approximations, also for higher-order derivatives, can be found in a similar way. In [Bic41] a formula to compute the coefficients together with tables for common cases is given. An algorithm to compute coefficients for arbitrarily spaced grids is given in [For88]. As can be seen, the approximations are exact for polynomials up to a certain degree and decrease when Δx decreases. Also the error in the approximations becomes larger the more uncentered the approximations are (especially at the borders).

2.12.4. CFL Condition

An interesting observation made by Richard Courant, Kurt Friedrichs and Hans Lewy is that when the length of the time step h is decreased then the space interval Δx must decrease at least proportionally to retain convergence [CFL28, CFL67]. A derivation of this *CFL condition* for the advection equation $u_t + \nu u_x = 0$ is given in [Sch91] as

$$|\nu| \frac{\Delta t}{\Delta x} \leq 1$$

using first-order finite differences for the spatial discretization and explicit Euler integration over time. A similar expression can be derived for other PDE problems.

Increasing the spatial resolution therefore increases the stiffness of the problem. Using implicit numerical methods (like BDF methods) can compensate this to some degree, but still care must be taken to stay in the stability region of the method. Also with increasing spatial resolution the amount of work increases, as with more grid points also more ODEs have to be integrated. Higher-order finite differences could be used to increase accuracy without increasing the resolution of the grid. But then these are based on polynomials and may show unexpected oscillations with increasing order.

2.12.5. An Example

Coming back to the metal stick from section 2.12.3, we can now investigate the solution of a problem given by Schiesser. The problem is defined by the heat equation (2.16)

$$u_t = u_{xx}$$

together with inital and boundary conditions

$$\begin{aligned} u(x,0) &= sin\left(\frac{\pi}{2L}x\right) \\ u(0,t) &= 0 \\ u_x(L,t) &= 0. \end{aligned}$$

The exact solution of the problem is

$$u(x,t) = e^{-t\left(\frac{\pi}{2L}\right)^2} sin\left(\frac{\pi}{2L}x\right),$$

as can be easily checked by computing the partial derivatives

$$\begin{aligned} u_t &= -\left(\frac{\pi}{2L}\right)^2 e^{-t\left(\frac{\pi}{2L}\right)^2} sin\left(\frac{\pi}{2L}x\right) \\ u_{xx} &= -\left(\frac{\pi}{2L}\right)^2 e^{-t\left(\frac{\pi}{2L}\right)^2} sin\left(\frac{\pi}{2L}x\right) \end{aligned}$$

and inserting them into the PDE and the initial and boundary conditions.

The numerical solution is shown in figure 2.9. As can be seen, for each ODE generated by the MOL a solution curve is produced, which is the origin of this method's name.

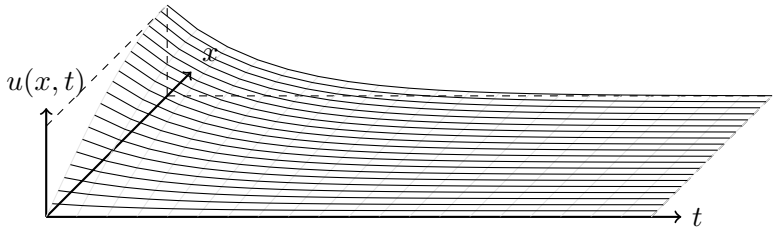


Figure 2.9.: Method of lines solution of heat conduction.

2.12.6. Geometric Classification

As there exist many different kinds of PDEs, some kind of categorization together with associated properties seems useful. Consider the general form of a second-order PDE with two independent variables:

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + \ldots = 0.$$

The *geometric classification* distinguishes three types of PDEs analogous to conic sections by the discriminant $D = B^2 - AC$:

- $D < 0$: elliptic
- $D = 0$: parabolic
- $D > 0$: hyperbolic

Elliptic PDEs describe stationary processes or minimum energy states. In other words, the PDE does not depend on partial derivatives of time. Examples are the *Laplace equation*

$$\Delta u = 0$$

with the *Laplace operator* $\Delta = \sum_i \frac{\partial^2}{\partial x_i^2}$, or the more general *Poisson equation*

$$\Delta u = f.$$

For the MOL this means that the set of ODEs can be extended to be time dependent and then be integrated to equilibrium (time derivatives vanish) to obtain a solution. This is also called *method of false transients*.

Parabolic PDEs usually depend on a first-order partial derivative in time and second-order derivatives in space, but not on first-order spatial derivatives. An example is the *heat equation* (2.16), in a more general form written as

$$u_t = D\Delta u,$$

which also describes diffusion processes. The solution smoothes out with increasing time, even if the initial or boundary conditions contain discontinuities.

The remaining cases are *hyperbolic PDEs*, like the first-order *advection equation*

$$u_t + \nu u_x = 0,$$

the second-order *wave equation*

$$u_{tt} = c^2 \Delta u,$$

or the hyperbolic-parabolic *Burger's equation*

$$u_t = -uu_x + \nu u_{xx},$$

where ν is the viscosity and controls how strong the hyperbolic character of the equation is. Contrary to the previous classes, solutions of hyperbolic PDEs do not smooth out, but instead shocks are propagated. Therefore they are the most difficult class of PDEs for numerical integration. The MOL solution for a hyperbolic PDE should consider the preferred spatial direction (i.e., in which waves propagate) when approximating the finite differences. More details on this can be found in [Sch91].

2.13. Summary

In natural sciences very often differential equations or systems thereof have to be integrated. In rare cases, an analytic solution can be found, but in the general case one has to resort back to numerical integration. Several methods have been proposed, of which the most important ones can be classified into three groups, namely the Runge-Kutta method, linear multi-step methods and extrapolation methods. Development of such methods concerns things like accuracy, stability and efficiency. Nevertheless, implementation of such methods can be quite challenging, especially if additional features like variable step, variable order, interpolation or handling of discontinuities are considered. Selection of an appropriate numerical method depends on the integration problem. Still, those methods share a common interface and can thus easily be exchanged if there is need.

3. Formal Languages, L-systems and RGGs

Language and automata theory forms the basis of many important application areas including, but not limited to, language understanding, text translation or compiler construction. Formally, a *language* is a set of *sentences* of finite length. The sentences (also named strings or words) are formed as a sequence of a set of symbols, the *alphabet*, usually designated by Σ. A *grammar* is a means to construct all sentences of that language by a finite number of rules. For natural languages the language would be considered a finite set, but for better theoretical analysis, formal languages are also allowed to be infinite sets.

A formal grammar is a tuple $\langle N, \Sigma, P, S \rangle$. Σ is a finite set of *terminal* symbols. N is a finite set of *nonterminal* symbols disjoint from Σ. Sometimes also a set V of all symbols (the *vocabulary*) is used instead of N, with $V = \Sigma \cup N$. The notation V^+ means a string of one ore more symbols of V, and V^* means zero or more, where * is the *Kleene star*. The empty string is designated by ϵ, thus $V^* \setminus \{\epsilon\} = V^+$. A finite irreflexive set $P \subset V^+ \times V^*$ of *production rules* defines how to replace symbol sequences. A production rule is often written as $\alpha \to \beta$ with $(\alpha, \beta) \in P$. A *derivation* $\varphi \rightsquigarrow \psi$ is the application of a rule $\alpha \to \beta$ with $\varphi = \gamma\alpha\delta$ and $\psi = \gamma\beta\delta$ for $\gamma, \delta \in V^*$. The derivation process starts with a start symbol S (which stands for sentence). The language generated by the grammar is the set of sentences consisting exclusively of terminal symbols. Two grammars are said to be *equivalent* if they generate the same language. By convention, nonterminal symbols are usually represented by uppercase letters, terminal symbols by lowercase letters and sequences of symbols by lowercase greek letters, but this is not mandatory.

Consider now a simple grammar (from [Cho56]) defined by the rules

$$\begin{aligned} S &\to NP\ VP \\ VP &\to V\ NP \\ V &\to are\ flying \\ V &\to are \\ NP &\to they \\ NP &\to planes \\ NP &\to flying\ planes \end{aligned}$$

where nonterminals are S, NP (noun phrase), VP (verb phrase) and V (verb). Beginning with S, sequential application of the rules can derive a sentence of terminal symbols only. Under appropriate restrictions the derivation can be visualized graphically with the benefit that the order of rule application becomes unimportant. For the sentence "they are flying planes", actually two derivations shown in figure 3.1 are possible.

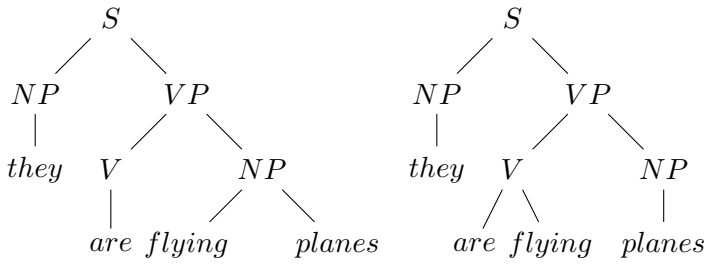


Figure 3.1.: Derivation trees for the sentence "they are flying planes".

Chomsky hierarchy	Grammar	Language	Automaton	Production rule
type 0	unrestricted	recursively enumerable	Turing machine	$\alpha \to \beta$
type 1	context-sensitive	context-sensitive	linear bounded	$\gamma A \delta \to \gamma \alpha \delta$
type 2	context-free	context-free	nondeterministic pushdown	$A \to \alpha$
type 3	regular	regular	finite state	$A \to aB$, $A \to a$

Table 3.1.: Chomsky hierarchy of grammars and their generated languages, automata that recognize them, and production rules that are characteristic for each type of grammar.

Chomsky called this ambiguity in the derivative structure *constructional homonymity*. There is also a semantic ambiguity, as the sentence could either describe objects on the sky (they – are – flying planes) or alternatively what pilots are doing with the planes (they – are flying – planes). If the grammar should serve the purpose of natural language understanding a semantic ambiguity should be also reflected in an ambiguity of the derivate structure. For compiler construction a semantic ambiguity would be problematic, as here a given program text must be understood in a unique way.

Grammars and their generated languages can be classified into different types. Chomsky defined type i grammars and languages, with $i \in 0, 1, 2, 3$ [Cho59]. Further types have been introduced as well, for instance LR(k), LL(k) and LF(k). For each type of language there exists also a formal automaton that recognizes it. A tabular overview is given in table 3.1. For reference see also [HU79, Sal73, AU68].

Type 0

Type 0 grammars are *unrestricted* in the sense that they can contain any kind of rule as was defined above. The set of type 0 languages $\mathcal{L}_0$ generated by those grammars are the *recursively enumerable languages* (RE). A *Turing machine* [Tur37] can recognize these languages. For any sentence in the language, the Turing machine will halt and return 1, but for sentences not in the language it may either halt and return 0 or loop forever. This is related to the *halting problem* [Tur37], where it cannot be decided if a program will

terminate or might run forever, which is also related to *Gödel's incompleteness theorem* [Göd31]. Thus a Turing machine can not *decide* if a sentence belongs to the language, but it can only *enumerate* the sentences. The class co-RE is set of the complements of all languages in RE, and for those a Turing machine can only disprove membership of a sentence. The union of RE and co-RE is the set of *recursive languages* (R). The class R is decidable, as two Turing machines can be run in parallel until either of them halts.

TYPE 1

Type 1 grammars are obtained by restricting the allowed types of rules to those of the form $\gamma A\delta \to \gamma\alpha\delta$, where $\gamma, \delta \in V^*$, $A \in N$ and $\alpha \in V^+$. They are also called *context-sensitive grammars* (CSG) and the set $\mathcal{L}_1$ of languages they generate are called *context-sensitive languages*. The strings γ and δ are called the *context*. An equivalent restriction is that the rules are non-shortening. Thus for a rule $\alpha \to \beta$ must hold $|\alpha| \leq |\beta|$. This yields monotone or noncontracting grammars, which are different from type 1 grammars, but generate the same context-sensitive languages. To include the empty sentence ϵ it is sometimes permitted to include a rule $S \to \epsilon$, provided that there is no rule that produces an S. It is always possible to introduce a new symbol $\bar{S}$ and rules $\bar{S} \to \epsilon, \bar{S} \to S$ and using $\bar{S}$ as start symbol. A context-free language can be recognized by a *linear bounded automaton* (LBA), which is a nondeterministic Turing machine for which its memory is limited linearly by the length of the input. More important, it is decidable if a given sentence belongs to the language or not.

TYPE 2

Further restricting the rules to $A \to \alpha$ with $A \in N$ and $\alpha \in V^*$ yields type 2 grammars, which are *context-free grammars* (CFG). In the original definition of type 2 grammars by Chomsky the rules $A \to \alpha$ required $A \in N$ and $\alpha \in V^+$. A grammar of this kind is said to be ϵ-free. Every context-free grammar can be transformed[1] into an ϵ-free grammar, unless the context-free grammar generated the empty sentence. In this case a single rule $S \to \epsilon$ might be allowed, like it was for type 1 grammars above. Context-free grammars can also be transformed into one of their normal forms. The *Chomsky normal form* (CNF) restricts the allowed types of rules to $A \to BC$, $A \to a$ and $S \to \epsilon$, where $A, B, C, S \in \mathbb{N}$, $a \in V$ and S the start symbol. The *Greibach normal form* (GNF) allows only rules of the form $A \to aA_1A_2 \ldots A_n$, and optionally $S \to \epsilon$ if S is not contained in any of the productions. The GNF generates exactly one terminal symbol with each application of a rule (disregarding the optional ϵ-rule). With restriction $n \in \{0, 1\}$ type 3 grammars are obtained, as explained below. All of the mentioned types of grammars generate the type 2 languages $\mathcal{L}_2$, the set of *context-free languages*. A non-deterministic pushdown automaton is a recognizer for context-free languages, and can be constructed from a context-free grammar using the GNF.

TYPE 3

If only rules of the forms $A \to a$ and $A \to aB$ with $A, B \in V$ and $a \in \Sigma$ are allowed, type 3 grammars or *regular grammars* are obtained. Optionally ϵ-rules $A \to \epsilon$ might be allowed, but not in the original definition given by Chomsky. In fact, the given rules define *right regular grammars*, where the generated string is followed by a single nonterminal symbol. A *left regular grammar* has only rules of the form $A \to a$ and $A \to Ba$ and the nonterminal precedes the generated string. These two kinds

[1] For each rule $A \to \epsilon$ replace A in all productions by iterating $P := P \cup \{(B, \gamma\delta) | (A, \epsilon), (B, \gamma A\delta) \in P\}$ until converged. Now remove all ϵ-productions by setting $P := P \setminus (N \times \epsilon)$.

of grammars are also called *strictly right/left regular grammars*, as extended versions allowing rules $A \to \alpha B$ respectively $A \to B\alpha$ with $\alpha \in \Sigma^*$ have also been defined. The set of languages $\mathcal{L}_3$ generated by regular grammars are called *regular languages*. These languages can also be described by *regular expressions*. Each regular grammar can be recognized by a nondeterministic finite automaton (NFA), but any NFA can be converted to a deterministic finite automaton (DFA) by the *powerset construction*[2].

The four types of languages $\mathcal{L}_i$ defined above form a proper inclusion relationship

$$\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0.$$

The inclusion is clear as the sets $\mathcal{L}_i$ were formed by increasing restrictions on the allowed rules. That the inclusions are proper can be shown using examples that belong to one type of language set, but not another. Consider the following languages

$$L_1 = \{a^n b^n c^n | n \geq 1\}$$
$$L_2 = \{a^n b^n | n \geq 1\}.$$

Pumping lemma for regular languages

The *pumping lemma for regular languages* is a necessary, but not sufficient, condition for a language being in $\mathcal{L}_3$. For every regular language L exists a number $n \in \mathbb{N}$, such that for all $\omega \in L$ with $|\omega| \geq n$ exist $\alpha, \varphi, \beta \in \Sigma^*$ for which

$$\omega = \alpha\varphi\beta \wedge |\varphi| > 0 \wedge |\alpha\varphi| \leq n \wedge \forall i \in \mathbb{N} : \alpha\varphi^i\beta \in L.$$

The proof of the lemma is based on the fact that for every regular language there exists a finite automaton that recognizes it. If the finite number of states of the automaton is n then to recognize a sentence of length n or more the automaton must enter a loop. The string φ recognized by the loop can thus be repeated any number of times.

L_2 does not fulfill the pumping lemma for regular languages. For some number n we can consider the word $\omega = a^n b^n$ and because of $|\varphi| > 0$ and $|\alpha\varphi| \leq n$ we know that φ must be made up of a only. Pumping up ω yields $\alpha\varphi^i\beta$ yields sentences with an unequal number of a and b, thus those sentences are not in L_2 from which follows that L_2 is not a regular language. In fact, L_2 is a context-free language which can be shown easily by providing a context-free grammar with rules $S \to ab$ and $S \to aSb$ that generates L_2.

Pumping lemma for context-free languages

The *pumping lemma for context-free languages* is a necessary, but not sufficient, condition for a language being in $\mathcal{L}_2$. For every context-free language L exists a number $n \in \mathbb{N}$, such that for all $\omega \in L$ with $|\omega| \geq n$ exist $\alpha, \beta, \gamma, \varphi, \psi \in \Sigma^*$ for which

$$\omega = \alpha\varphi\beta\psi\gamma \wedge |\varphi\psi| > 0 \wedge |\varphi\beta\psi| \leq n \wedge \forall i \in \mathbb{N} : \alpha\varphi^i\beta\psi^i\gamma \in L.$$

[2] Given an NFA (Q, Σ, T, q_0, F) with set of states Q, set of input symbols Σ, set of transitions $T : Q \times \Sigma \to \mathcal{P}(Q)$ with $\mathcal{P}(Q)$ denoting the power set of Q, initial state $q_0 \in Q$ and a set of final states $F \subseteq Q$. A state in the corresponding DFA is a subset of Q, indicating the possible states the NFA might be in after accepting a certain input. The initial state q_0 of the NFA is represented as the set $\{q_0\}$ for the DFA. The states and transitions of the DFA can be build incrementally. For some input symbol and the NFA states of a DFA state $\bar{q}_1$ generate the set of NFA states $\bar{q}_2$ reachable by transitions from T. For the DFA append $\bar{q}_2$ to the set of states, if not already done so, and add a transition from $\bar{q}_1$ to $\bar{q}_2$. If any of the NFA states in $\bar{q}_2$ is a final state append $\bar{q}_2$ to the DFA set of final states. Repeat this until no new DFA states or transitions can be found.

To proof this lemma consider a grammar in CNF that generates L and a sentence $\omega \in L$. By induction it can be shown that if the longest path in ω has length i then ω has a length of at most 2^{i-1}. If $i = 1$ then a rule of the form $S \to \epsilon$ or $S \to a$ must have generated the sentence, so its length is at most $2^0 = 1$. If $i > 1$ then a rule $S \to AB$ must have been used and under the assumption that A and B generate strings of at most 2^{i-2} symbols then the generated sentence has a length of at most 2^{i-1} symbols. If ω is sufficiently long, that is $|\omega| \geq 2^{|N|} = n$ with N the set of nonterminal symbols, then since $|\omega| > 2^{|N|-1}$ there must be a path of length at least $|N|+1$. This path has $|N|+2$ vertices, of which only one is a terminal symbol. The remaining symbols are nonterminal and at least one of them, let us call it A, must appear twice. The sequence of derivations $A \rightsquigarrow^* A$ can be repeated and generates the prefix φ and suffix ψ an arbitrary number of times.

L_1 does not fulfill the pumping lemma for context-free languages. For some number n we consider the word $\omega = a^n b^n c^n$. Because $|\varphi\beta\psi| \leq n$ the string $\varphi\beta\psi$ contains at most two kinds of symbols. If φ and ψ get pumped since $|\varphi\psi| > 0$ the third kind of symbol does not change its count while at least one of the other two does. Therefore L_1 is not context-free. In fact, it is context-sensitive, as a grammar that generates it is

$$\begin{aligned} S &\to aA & aB &\to ab \\ S &\to aSA & bB &\to bb \\ A &\to BC & bC &\to bc \\ CB &\to BC & cC &\to cc. \end{aligned}$$

and another with start symbol X_0 is (from [Sal78])

$$\begin{aligned} X_0 &\to X_0 Z_1 & X_0 &\to X_1 Z_1 & X_1 &\to aY_1 \\ X_1 &\to aX_1 Y_1 & Y_1 Z_1 &\to Y_1 Z_3 & Y_1 Z_3 &\to Y Z_3 \\ Y Z_3 &\to Y Z & Y_1 Y &\to Y_1 Y_2 & Y_1 Y_2 &\to Y Y_2 \\ Y Y_2 &\to Y Y_1 & Z Z_1 &\to Z_2 Z_1 & Z_2 Z_1 &\to Z_2 Z \\ Z_2 Z &\to Z_1 Z & Y &\to b & Z &\to c. \end{aligned}$$

The first grammar is non-shortening while the second one is context-sensitive.

3.1. L-systems

In 1968 Aristid Lindenmayer studied the development of multicellular structures and simple plants [Lin68a, Lin68b]. Symbols represented distinguishable states of a cell. A transition function maps the state of a cell (considering optionally also state of its neighbour cells) to a new state. Cell division is explained by replacing a symbol by a sequence of other symbols. This way a linear array of cells is connected to formal language theory. Formal definitions for L-systems have been given in [Lin71, vD71, RD71, PL90], for instance (from [RD71]):

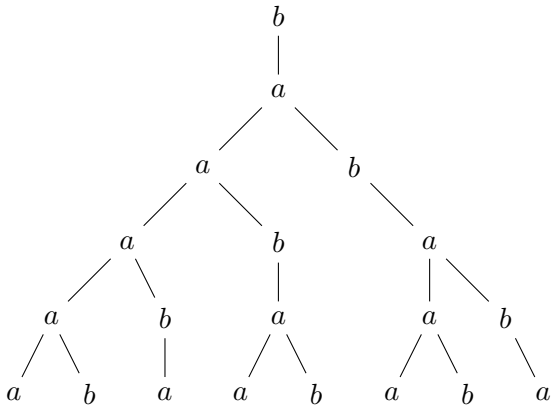


Figure 3.2.: Strings generated by a simple L-system.

Definition 2. *A* $0L$*-system is a system* $G = \langle \Sigma, P, \sigma \rangle$*, where* Σ *(the* alphabet*) is a finite, nonempty set,* σ *(the* axiom*) is an element of* Σ^+*, and* P *(the set of* productions*) is a finite subset of* $\Sigma \times \Sigma^*$*, such that* $\forall a \in \Sigma : \exists \alpha \in \Sigma^* : (a \to \alpha) \in P$*.*

Definition 3. *(parallel rule application) Let* $G = \langle \Sigma, P, \sigma \rangle$ *be a* $0L$*-system; let* $x \in \Sigma^+$*,* $x = a_1 \cdots a_m$ *with* $m \geq 1$ *and* $a_j \in \Sigma$ *for* $j = 1, \ldots, m$*; let* $y \in \Sigma^*$*. Then* $x \Rightarrow y$ *if and only if* $\exists p_1, \ldots, p_m \in P : \forall j \in \{1, \ldots, m\} : p_j = a_j \to \alpha_j \wedge y = \alpha_1 \cdots \alpha_m$*.*

Definition 4. *Let* $G = \langle \Sigma, P, \sigma \rangle$ *be a* $0L$*-system. The language generated by* G *then is* $L(G) = \{\omega | \sigma \Rightarrow^* \omega\}$*.*

An example of a simple L-system is shown in figure 3.2. Derivation starts with b and uses rules given in [PL90], which are

$$a \to ab$$
$$b \to a.$$

The difference of L-systems to the rewriting systems considered so far is

1. parallel rule application and
2. no distinction between terminal and non-terminal symbols.

The first difference, parallel application of production rules, can be justified biologically as development of each cell progresses simultaneously. The second difference concerns the fact that if a cell reaches a terminal state it can be considered dead or at least irreversibly differentiated. To simulate living organisms (as words in the language generated by the L-system) no distinction between terminal and non-terminal symbols must be made.

An iL-system [vD71] is an L-system where the transition function depends on the current symbol only ($i = 0$), also the state of its left neighbour ($i = 1$), or state of both left and right neighbour ($i = 2$). Often also the name OL-system is used instead of

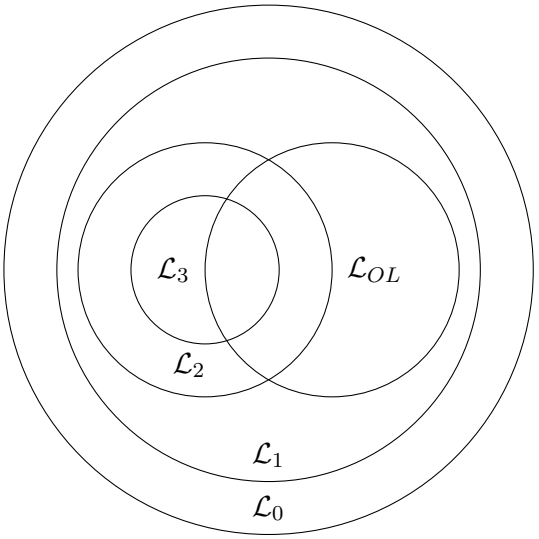


Figure 3.3.: Relation of OL-system languages to the Chomsky hierarchy.

0L-system. An L-system is *propagating* or a POL-system if the empty string ϵ cannot be substituted for any symbol. It is *determinate* or a DOL-system if there is at most one transition rule for every symbol. A DPOL-system is determinate and propagating.

Theorems for L-systems and their generated languages have been developed, e.g. [Lin68a, Lin68b, Lin71, RD71, Her69, vD71]. Interesting results consider the relation of languages generated by L-systems to the Chomsky hierarchy. It was shown [RD71] for $\mathcal{L}_{OL}$, the set of all OL-languages, that they share some regular ($\mathcal{L}_{OL} \cap \mathcal{L}_3 \neq \emptyset$, $\mathcal{L}_{OL} \nsubseteq \mathcal{L}_3$, $\mathcal{L}_3 \nsubseteq \mathcal{L}_{OL}$) and context-free ($\mathcal{L}_{OL} \cap \mathcal{L}_2 \neq \emptyset$, $\mathcal{L}_{OL} \nsubseteq \mathcal{L}_2$, $\mathcal{L}_2 \nsubseteq \mathcal{L}_{OL}$) languages and are context-sensitive ($\mathcal{L}_{OL} \subset \mathcal{L}_1$). This is visualized in figure 3.3. It was also shown that 1L-systems and 2L-systems are as strong as Turing machines [Her69, vD71]. An interesting consequence is that there are L-systems for which it cannot be decided if the modelled organism will eventually die or not.

Turtle graphics

In combination with Turtle graphics [Ad80] the power of L-systems to describe structures grows beyond simple topology [Smi84, Pru86]. The *turtle* can be imagined as a virtual pen with a position and movement direction (heading). Simple commands allow the turtle to move forward and rotate. While moving, the turtle can also draw a line. A graphical representation is obtained by *interpretation* of the generated string. The following turtle commands have been used for L-systems:

- **F** Move forward a distance d and draw a line.
- **f** Move forward a distance d, but don't draw a line.
- $+$ Turn left by an angle θ.
- $-$ Turn right by an angle θ.

These commands modify the *state* of the turtle (position and orientation). Other commands are ignored. The parameters d and θ have to be defined somewhere else.

Branching structures

To allow simulation of branching structures, Lindenmayer suggested to use symbols [

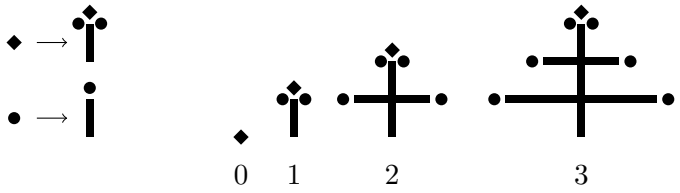


Figure 3.4.: A simple tree described by an L-system.

and] [Lin68b]. For the graphical representation this means to push the current state of the turtle onto a stack when [occurs, and to pop it from the stack when the matching] is found. Thus when a branch is complete the turtle jumps back to the position where the branch started. An example of a simple tree structure is shown in figure 3.4. The growth is governed by the rules:

$$A \to F[+B][-B]A$$
$$B \to FB$$

An extension of turtle graphics to three dimensions is straightforward [Pru87]. Three direction vectors $\vec{H}, \vec{L}, \vec{U}$ define the orientation of the turtle, where in addition to the *heading* $\vec{H}$ also *left* $\vec{L}$ and *up* $\vec{U}$ directions are given. The three vectors are perpendicular to each other and have unit length. It is assumed that $\vec{H} \times \vec{L} = \vec{U}$ holds. In addition to these directions the turtle state is extended to include also thickness and color. A thick line is represented in three dimensions by a cylinder.

To facilitate modelling of leaves additional symbols { and } have been introduced. When encountering a { subsequent positions of the turtle are recorded and upon } a polygon is created from these positions and filled. Instead of tracing the contour of the polygon, an alternate method is described in [PL90], where a tree structure spans over the leaf and polygon vertices are marked explicitly by a dot.

STOCHASTIC L-SYSTEM

Nondeterminism in L-systems can occur if multiple rules are applicable for the same symbol. It can be resolved by assigning probabilities to the rules. This can be used to control the speed of growth [Nis80], or to prevent repetetive structures when L-systems are used for image generation [Pru87]. A more general definition of *probabilistic L-systems* has been given earlier in [Jür76].

CONTEXT-SENSITIVE L-SYSTEM

Communication between entities in the string, for instance to simulate transport of hormones in a plant, can be supported by providing the context in which the entity develops. This has been used in one-sided and two-sided L-systems (1L-system respectively 2L-system). A more general view is a $\langle m, n \rangle$ L-system [Lin75], where m and n describe the number of symbols belonging to the left and right context. A $\langle 0, 0 \rangle$ L-system is the same as an OL-system, but for $m > 0$ or $n > 0$ it is called an IL-system (the I stands for interactive), or also *context-sensitive L-system*.

PSEUDO L-SYSTEM

Not only the left and right context can be extended to a sequence of symbols, but also the string on the left side of the rule that is to be replaced. *Pseudo-L-systems* (pL-systems) were introduced with this in mind [Pru86]. To be deterministic a precedence

order over the rules is established and the string to be rewritten is partitioned by scanning it from left to right for applicable rules.

In some cases it might be desirable to have multiple sets of rules. An example can be found in [SL69], where growth behaviour in light is different from the one in darkness. An extension to OL-systems proposed by Rozenberg are *table L-systems*, also TOL-systems [Roz73].

TOL-SYSTEM

Traditional L-systems describe development in discrete steps. However, some aspects of a model can be better described by continuous quantities, like concentrations of nutrients, or size and age of cells. Lindenmayer suggested to associate parameters to the symbols [Lin74]. A definition of *parametric L-systems* has been given in [PH90, Han92].

PARAMETRIC L-SYSTEM

Definition 5. *A parametric 0L-system is an ordered quadruplet* $G = \langle V, \Sigma, \omega, P \rangle$*, where*

- V *is the alphabet of the system,*
- Σ *is the set of formal parameters,*
- $\omega \in (V \times \mathbb{R}^*)^+$ *is a nonempty parametric word called the axiom,*
- $P \subset (V \times \Sigma^*) \times \mathcal{C}(\Sigma) \times (V \times \mathcal{E}(\Sigma)^*)^*$ *is a finite set of productions.*

The productions $(a, C, \chi) \in P$ are written as $a : C \to \chi$, where $a \in V \times \Sigma^*$ is the predecessor and $\chi \in (V \times \mathcal{E}(\Sigma)^*)^*$ the successor. When the production is matched, the actual parameters of the match are substituted for the formal parameters of the predecessor. The production is applicable if the condition, a logical expression $C \in \mathcal{C}(\Sigma)$, evaluates to true. The parameters of the successor are computed by arithmetic expressions from $\mathcal{E}(\Sigma)$. As an example of such a production rule consider

$$F(a) : t < 5 \to F(a + 0.1).$$

Here $F(a)$ repesents a cylinder of length a. By repeated application of this rule the F will grow 0.1 length units per step until a final length of 5 is reached.

The definition of a parametric L-system can be easily extended to become context-dependent. A definition has been given in [Han92]:

Definition 6. *A parametric IL-system is an ordered quadruplet* $G = \langle V, \Sigma, \omega, P \rangle$*, where*

- V *is a nonempty set of letters called the alphabet of the system,*
- Σ *is the set of formal parameters,*
- $\omega \in (V \times \mathbb{R}^*)^+$ *is a nonempty parametric word called the axiom,*
- $P \subset (V \times \Sigma^*)^* \times (V \times \Sigma^*) \times (V \times \Sigma^*)^* \times \mathcal{C}(\Sigma) \times (V \times \mathcal{E}(\Sigma)^*)^*$ *is a finite set of productions.*

The difference to parametric 0L-systems is that productions $(\eta_l, a, \eta_r, C, \chi)$, usually written $\eta_l < a > \eta_r : C \to \chi$, are now equipped with a left and right context η_l and η_r.

As an example consider the development of blue-green alga *Anabaena catenula*. A model for this has been presented in [BH72a, BH72b], and based on it an L-system model in [Lin74, KL87]. Cells are represented by symbols $C(t, c, a)$, where t is the type (v for vegetative, h for heterocyst), c the inhibitor concentration, and a the age of the cell. Diffusion of the inhibiting substrates is modelled by an ODE

$$\frac{dc}{dt} = D(c_l - c) + D(c_r - c) - \mu c,$$

where c is the concentration of the current cell, c_l and c_r concentrations of the left and right neighbours, D the diffusion constant, and μ a decay constant (diffusion to environment). After discretization and setting $D = \mu$ and $K = \mu\Delta t$ one obtains

$$\Delta c = K(c_l + c_r - 3c).$$

A vegetative cell transforms into a heterocyst if its concentration c drops below some threshold T_c. Heterocystic cells do not grow anymore and their inhibitor concentration takes a constant value c_h. A vegetative cell divides if its age reached a terminal age T_a. Thus the growth process is governed by the following rules

$$\begin{aligned}
&C(-, cl, -) < C(v, c, a) > C(-, cr, -) : c > T_c \wedge a < T_a \\
&\quad \to C(v, c + K(cl + cr - 3c), a + 1) \\
&C(-, cl, -) < C(v, c, a) > C(-, cr, -) : c \leq T_c \\
&\quad \to C(h, c_h, 0) \\
&C(-, cl, -) < C(v, c, a) > C(-, cr, -) : c > T_c \wedge a = T_a \\
&\quad \to C(v, c + K(cl + cr - 3c), 0)\ \ C(v, c + K(cl + cr - 3c), 0)
\end{aligned}$$

where "don't care" parameters are indicated by a minus in the formal parameters list.

A similar model is used in [PL90, Han92], where cells are represented by line segments F and their age corresponds to their length. For visualization of the cells the way the turtle interprets the string was extended. If the symbol F, f, etc., is followed by a parameter list, the first parameter is used instead of the one in the environment (for instance for F this defines the length of the line segment).

The original formulation of L-systems was discrete in time and space. The introduction of parametric L-systems allowed a continuous view on space. Still, time remained a discrete quantity as this is the nature of rule application. If time could be made continuous this would be beneficial when creating animations with L-systems or simulating differential equations.

tDOL-systems

Timed DOL-systems (tDOL-systems) achieve this by associating a local time to each symbol [PL90]. The pair $(a, \tau) \in V \times \mathbb{R}$ is referred to as the *timed letter* a with τ the *age* of a. A sequence of timed letters is called a *timed word*.

Definition 7. *A timed DOL-system is a triplet $G = \langle V, \omega, P \rangle$, where*

- V *is the alphabet of the L-system,*
- $\omega \in (V \times \mathbb{R})^+$ *is a nonempty timed word over* V*, called the initial word,*
- $P \subset (V \times \mathbb{R}) \times (V \times \mathbb{R})^*$ *is a finite set of productions.*

A rule can be written $(a, \beta) \to (b_1, \alpha_1) \dots (b_n, \alpha_n)$, where β is the *terminal age* of a and each α_i is the *initial age* of a_i. It is assumed that for each symbol a the terminal age is unique and the initial age is always smaller than its terminal age. For a global time t a timed word can be derived by $\mathcal{D}(\omega, t)$, where the *derivation function* $\mathcal{D} : ((V \times \mathbb{R})^+ \times \mathbb{R}) \to (V \times \mathbb{R})^*$ has the following properties:

P1 $\mathcal{D}(((a_1, \tau_1), \dots, (a_n, \tau_n)), t) = \mathcal{D}((a_1, \tau_1), t) \dots \mathcal{D}((a_n, \tau_n), t)$

P2 $\mathcal{D}((a, \tau), t) = (a, \tau + t),$ if $\tau + t \leq \beta$

P3 $\mathcal{D}((a, \tau), t) = \mathcal{D}((b_1, \alpha_1) \dots (b_n, \alpha_n), \tau + t - \beta),$ if $\tau + t > \beta$.

A derivation of a timed word is obtained by deriving each timed letter in it on its own (P1). Each timed letter grows until its terminal age is reached (P2). After reaching the terminal age, the timed letter is replaced by the corresponding timed word as is defined by the production and the remaining time $\tau + t - \beta$ is used to recursively perform derivation of the new symbols (P3).

If tDOL-systems are used to create animations, the age of a timed letter has to be mapped to its appearance. This was done in [PL90] on the example of *Anabaena catenula*, where the length of a cell was controlled by its age. *Continuity requirements* must be fulfilled to make the animation look smooth. If the length of a cell changes as a function of time, this *growth function* $g(a, \tau)$ must be continuous. Similarly, if a cell divides into other cells, the summed lengths of the new cells must be equal to the length of the original cell. So for a production $(a, \beta) \to (b_1, \alpha_1) \dots (b_n, \alpha_n)$ it follows that

$$g(a, \beta) = \sum_{i=1}^{n} g(b_i, \alpha_i).$$

A linear mapping from age to length fulfills both requirements. However, when a cell divides, the growth rate of the total cell array is discontinuous at this point (two cells grow faster than one cell). This can be prevented if the two requirements must hold also for higher orders of the growth function. Specifically $g^{(k)}(a, \tau)$ must be continuous and

$$g^{(k)}(a, \beta) = \sum_{i=1}^{n} g^{(k)}(b_i, \alpha_i) \text{ for } k = 0, 1, \dots, N$$

must hold. But then also growth functions of higher order should be used. An alternative is to use an exponential growth function. This can be justified as cellular cultures show exponential growth as a whole, so assuming this for an individual cell should be legitimate. In the example given in [PL90] this causes the continuity requirements to hold for every order.

An extension to parametric tDOL-systems with conditional and pseudo-stochastic productions is given in [NTT92]. The difference to the tDOL-systems above is that productions only apply if an associated condition is fulfilled (like in ordinary parametric L-systems), and in this case application of a production only happens with a certain probability. A difficulty with the stochastic part is that for similar values of time the generated models should be similar as well. This was solved by using a fixed table of random numbers and proper indexing.

dL-systems

In many cases time-dependent processes are described by differential equations. The intention of *dL-systems* [PHM93] is to combine these continuous differential equations with the discrete nature of production rules. It is assumed that parameters w of a module $A(w)$ may vary inside a domain D_A of legal values. If a boundary C_A of D_A is eventually reached a production will be triggered, which can cause a topological change of the structure and also a discontinuity of the parameter values.

Development of $A(w)$ is described by an ordinary differential equation

$$\frac{dw}{dt} = f_A(w_l, w, w_r),$$

where f_A is continuous in D_A, and w_l and w_r are the parameters of the left and right neighbours A_l and A_r of A. The boundary C_A is assumed to consist of nonintersecting segments C_{A_k}. A production

$$p_{A_k} : A_l(w_l) < A(w) > A_r(w_r) \rightarrow B_{k,1}(w_{k,1})B_{k,2}(w_{k,2}) \cdots B_{k,m_k}(w_{k,m_k})$$

is applied when $w \in C_{A_k}$. Modules $A_l(w_l)$ and $A_r(w_r)$ define the *context*, $A(w)$ is the *strict predecessor*, and $B_{k,1}(w_{k,1})B_{k,2}(w_{k,2}) \cdots B_{k,m_k}(w_{k,m_k})$ the *successor*, where the initial values $w_{k,i}$ may depend on w_l, w, and w_r. Different productions may apply depending on which part of the boundary was hit.

As an example consider the model of *Anabaena catenula* from above. The symbols F_v and F_h now represent vegetative cells and heterocysts. A symbol F is used to refer to both types. The initial string is set to

$$F_h(x_{max}, c_{max})F_v(x_{max}, c_{max})F_h(x_{max}, c_{max})$$

with initial length x_{max} and initial concentration of nitrogen compounds c_{max}. The growth process is then governed by the rules

$$\begin{aligned}
&F(x_l, c_l) < F_v(x, c) > F(x_r, c_r) : \\
&\quad \textbf{if } x < x_{max} \wedge c > c_{min} \\
&\qquad \textbf{solve } \tfrac{dx}{dt} = rx, \tfrac{dc}{dt} = D \cdot (c_l + c_r - 2c) - \mu c \\
&\quad \textbf{if } x = x_{max} \wedge c > c_{min} \\
&\qquad \textbf{produce } F_v(kx_{max}, c)F_v((1-k)x_{max}, c) \\
&\quad \textbf{if } c = c_{min} \\
&\qquad \textbf{produce } F_h(x, c) \\
&F_h(x, c) : \\
&\qquad \textbf{solve } \tfrac{dx}{dt} = r_x(x_{max} - x), \tfrac{dc}{dt} = r_c(c_{max} - c)
\end{aligned}$$

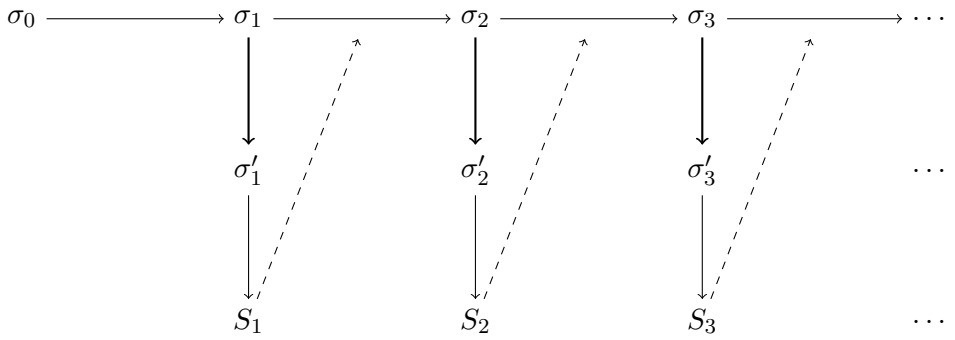


Figure 3.5.: Structure generation by a sensitive, two-phase growth grammar.

As long as a vegetative cell F_v did not reach its final length x_{max} and has a concentration of nitrogen compounds of more than c_{min} it elongates exponentially. In parallel, diffusion between the cell and its neighbour cells as well as the environment is applied. If the cell reached its final length while its concentration c is still above the threshold c_{min}, it divides into two new vegetative cells with initial lengths distributed according to k. If instead the concentration c dropped below the threshold c_{min} before the final length was reached, the cell differentiates into a heterocyst. Heterocystic cells grow to the final length x_{max} and increase the concentration of nitrogen compounds to c_{max} exponentially.[3]

3.2. Growth Grammars

Growth Grammars [Kur94, Kur99] are an extension of parametric, stochastic 0L-systems. A difference to the latter is, that rewriting rules are distinguished semantically into *generative* and *interpretative rules*, in analogy to the "two-level grammars" of van Wijngaarden [Kup80]. This is shown in figure 3.5. In each step, the generative rules rewrite a string σ_i into a new string σ_{i+1}. Thereafter, interpretative rules transform σ_{i+1} into σ'_{i+1}, from which then a geometrical representation S_{i+1} is derived by means of turtle graphics. This comes in handy if, for instance, some object is represented by a single symbol in σ_i, but its graphical representation by a multitude of turtle-commands in σ'_i.

Another extension is *sensitivity*, where generative rules are affected by the geometrical representation of the string (i.e. S_i affects rules that transform σ_i into σ_{i+1}). This is indicated in the figure by a dashed line. An example would be tree growth depending on local light conditions (light cone). A rich set of turtle commands is provided.

Special commands are the *repetition* and *expansion operators*. The repetition operator allows to produce a symbol sequence a given number of times. The expansion operator

[3]Remark: In the original model of *Anabaena catenula* the concentration of nitrogen compounds was assumed to be constant for the heterocystic cells. However, in the dL-system model of *Anabaena catenula*, the concentration in such cells may vary as described by a differential equation. Consequently one should include diffusion also into the ODE for the heterocystic cells.

instead takes a sequence of symbols and derives it a certain number of times. The result of either computation is then inserted into the output string in place of the operator. A possible application would be a random number of branches generated in a tree model.

For symbols produced by the right hand side of a rule parameter values can be calculated by arithmetic expressions, as is common for parametric L-systems. Relational growth grammars include in addition *arithmetic-structural operators*, that allow to query certain features of the generated structure. For instance, one could sum the total leaf area of a tree to calculate a photosynthesis rate and control growth of the structure by this value.

3.3. Relational Growth Grammars

Although many things can be done with L-systems, there are still limitations. First, and most important, the generated structures are exclusively trees (in the computer science meaning). This might not be problematic if real trees are to be modelled, but might impose severe restrictions if regulatory networks should be included. Second, the global interaction between geometric entities generated from the string is complicated. A more direct approach would consider symbols as geometry, not visualize them by it. Third, symbols in L-systems can only have one of two kinds of relations (successor and branch), whereas for some applications like multi-scale modelling additional relations might prove to be useful. These problems have been addressed partially already by several extensions to L-systems. Still a new approach was needed to fuse together those extensions and go beyond.

Relational growth grammars [KBSK03, Kni04, Kni08] use a typed attributed graph with inheritance for the structure. The former symbols of the string become nodes in that graph, and relations (like successor or branch) become edges that connect the nodes. This allows to define any kind of relation and between any two nodes any combination of relations can be established. For more flexibility nodes become instances of classes in an object-oriented fashion, with parameters being their attributes. Turtle-commands are part of that class hierarchy, and the geometric structure is obtained by traversing the graph only by successor and branch type edges (the induced structure by turtle-interpretation must still be a tree for obvious reasons).

Such flexibility introduces a complication. Using a graph, it follows that rules now do not replace symbols by a sequence of other symbols anymore, but that a pattern of nodes and edges in the graph is replaced by some other pattern. The complication that arises in this case is the question of *embedding*. It must be defined how outgoing and incoming edges from and to the original pattern should be connected to the newly generated production. Also conflicts between deletion and preservation of a node must be properly handled (for instance, when one node in the graph identifies as two such nodes in the pattern). RGGs use SPO (single pushout) productions, where deletion always takes precedence over preservation.

An example of a graph replacement rule and its application to a graph is shown in figure 3.6. The pattern on the left hand side of the rule is searched for in the graph. For

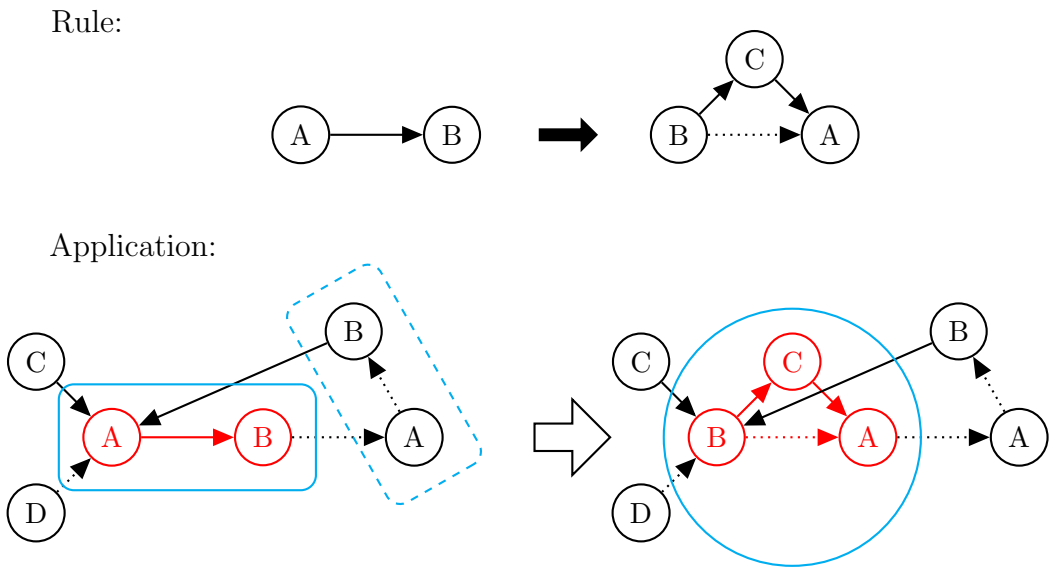


Figure 3.6.: Graph rewriting rule and its application (from [Kur07]).

the pattern to match, not only the types of nodes (A and B) must coincide, but also the configuration and type of edges (solid and dotted). Therefore a replacement only takes place in the region marked by a solid rectangle, but not in the other region marked by a dashed rectangle.

L-system rules have been shown to be a versatile tool for plant modelling. Therefore it is favourable to keep L-system style rules as a subset of graph replacement rules for backward-compatibility. RGGs achieve this by defining a special kind of *embedding mechanism*, where for a rule (given in textual representation) all incoming edges to the first node in the pattern on the left hand side are redirected to the first node of the production on the right hand side, and outgoing edges from the last node of the pattern to the last node of the production.

RGGs combine the rule-based approach, like in ordinary L-systems, with imperative programming. Productions are allowed to contain code fragments, e.g., to compute intermediate values or node attributes. Control flow statements allow to dynamically create productions, for instance a number of side branches depending on a vitality attribute of the main branch. *Aggregate functions* can be used to gather data from the whole graph, for instance to get the total leaf area of a modelled tree. A *query language* defines how to specify the values to aggregate, and is also used to define patterns on the left hand side of rules. A set of rules does not form the main program anymore, but rules are merely embedded into it. Flow control in the (object-oriented) main program decides which rules to execute and in which order, therefore generalizing table L-systems.

Type	Arrow symbol
execution rule	`::>`
SPO rule with implicit connections	`==>`
SPO rule without implicit connections	`==>>`

Table 3.2.: Types of rules in XL.

3.4. XL

The concepts of relational growth grammars have been implemented in the *XL programming language* [Kni08]. XL is based on the second edition of the Java programming language [GJSB00], with some extensions from the third edition [GJSB05]. This means, that every Java program (according to the 2nd edition) is also a valid XL program, but not necessarily the other way around. In the following, we assume familiarity with Java.

The *XL compiler* transforms an XL program into Java bytecode. The bytecode is then run by a *Java Virtual Machine* (JVM) [LY99]. The JVM executes bytecode by interpretation or compilation to native machine code, with prior verification of bytecode at runtime, performs memory management with garbage collection, and provides security by sandboxing (for instance, when executed in a web-browser). Although the JVM was originally aimed as platform for running Java programs, many other languages now compile to Java bytecode to be run on a JVM, for instance, Scala[4], Groovy[5], Clojure[6], Fortress[7], Jelly[8], and many more.

Rule-based extensions to the Java language can be added at places where they do not interfere with normal Java syntax. Most important, Java defines blocks of statements by enclosing them in braces { and }. A logical choice in the design of XL therefore was to allow blocks of rules, enclosed in brackets [and], wherever statement blocks are allowed. By nesting these different types of blocks inside each other it is possible to switch from Java to rule mode and vice versa. Three types of rules are supported in rule blocks and are shown in table 3.2.

XL does not prescribe the semantics of `==>` and `==>>` rules, but rather this is up to the implementation of the used producer. In GroIMP (see section 3.5), the `==>` rules are used to emulate L-system rules with implicit connections, while `==>>` gives full control over the rewriting process. Execution rules do not modify the topology of the graph, but are handy if attributes of nodes need to be modified.

Rules inside a rule-block are executed sequentially. Parallel rule application is achieved by storing all changes to be made to the graph into a modification queue. Therefore all rules search the original graph for possible places where they apply, and all changes become visible simultaneously when the queue is flushed.

QUERIES Left hand sides of rules are described by a *query*. It can contain node patterns

[4] http://www.scala-lang.org/
[5] http://groovy.codehaus.org/
[6] http://clojure.org/
[7] http://projectfortress.java.net/
[8] http://commons.apache.org/jelly

`>`	`<`	`---`	`<->`	successor
`+>`	`<+`	`-+-`	`<+>`	branch
`/>`	`</`	`-/-`	`</>`	refinement
`-->`	`<--`	`--`	`<-->`	any type

Table 3.3.: Standard edge patterns in XL.

(possibly named, e.g., for later use on the right hand side of the rule), path patterns (to describe relations between nodes), context graphs (that are not replaced but restrict where the rule matches), and application conditions (that must evaluate to true for the pattern to match).

Node patterns are usually used to search for nodes of a certain type or one derived thereof. If the type is preceded by a name separated by a colon (as in `x:X`), the name behaves like a local variable and allows to refer to the currently bound node of the match. The type may be followed by a parenthesized list of parameters, like in parametric L-systems, to obtain variables set to the actual values of the node's attributes (in the order given in the module definition). *Expression patterns* are functions that generate a sequence of nodes. *Unary predicates* are boolean functions that tell if a certain node should be considered.

Path patterns allow to search for graph structures instead of individual nodes by providing relations between the nodes. *Explicit path patterns* are of one of the forms

`-`*r*`->` `<-`*r*`-` `-`*r*`-` `<-`*r*`->`

with *r* the relation to use. The first two are used for directed relations (forward and backward), the third one for undirected relations, and the last one for bidirectional relations. Standard edge patterns have been defined as shorthand notations and are shown in table 3.3. If *r* is the name of a boolean function, this function acts as *binary predicate*. If *r* is a generator function, it generates a sequence of target nodes for a given source node.

Patterns can be combined. If multiple patterns are listed in sequence, they define a *connected pattern graph*. In case a node pattern is followed by another node pattern, a successor edge is implicitly assumed (to provide compatibility to L-systems). Path patterns must always be surrounded by other patterns. Two path patterns given in sequence implicitly assume a node pattern of any type in between.

Compound patterns allow to search for multiple unconnected structures. These are separated by a comma. In addition, boolean-valued expressions in parantheses can be listed, which are considered as *application conditions*.

Transitive closures of patterns can be designated by appending a quantifier to a pattern. To match, the pattern must then appear the requested number of times. The syntax of the predicates was taken over from regular expressions. Table 3.4 lists the supported quantifiers and the associated number of repetitions.

Certain parts of the structure in the query can be marked as *context* by surrounding them with `(*` and `*)`. They are not modified by rule applications, but nevertheless must be present for the rule to match.

Quantifier	Repetitions
`+`	1 to n
`*`	0 to n
`?`	0 or 1
`{n}`	exactly n
`{n, m}`	n to m
`{n,}`	at least n

Table 3.4.: Pattern quantifiers and associated number of repetitions.

Generator Expressions

A query can also appear in Java code as *query expression* and has to be surrounded by `(*` and `*)`. The query then is a *generator expression*, producing a sequence of nodes for each match (the right-most non-bracketed node of the pattern is used).

Another way to generate a sequence of values are *generator methods*. These are designated in the code by appending an asterisk `*` to the return type of a method. A `yield` statement takes the place of the `return` statement of ordinary functions to return a value, with the difference that execution continues with the next statement after `yield` instead of returning control flow to the caller.

A sequence of values can also be generated by the *range operator*, written as `a:b`. It yields all values from `a` to `b`, or no values at all if $a > b$. The *array generator* `a[:]` yields all values of the array `a`. The *guard operator* `a::b` yields only values `a`, for which `b` evaluates to `true`. The sequence of values `a` is thus filtered by `b`.

Filter Methods

Filter methods are a generalization of the guard operator. They take as input a sequence of values and produce as output another sequence. The types of the values in input and output sequence may be different. Two standard filter methods `first(a, n)` and `slice(a, m, n)` are provided. The former yields the first `n` elements of the input sequence, while the latter yields `n-m` elements starting with element `m` of the sequence.

Aggregate Expressions

The counterpart of generator expressions are *aggregate expressions*. They take as input a sequence of values and produce a single value as output. The *containment operator* `a in b` evaluates to `true`, if any value from the sequence `b` is equal to *a*. *Aggregate methods* are user-defined functions that are successively called with the values from the sequence together with a state object to produce the desired output value. A set of standard aggregate methods is shown in table 3.5.

Production Statements

The right hand side of rules consists of *production statements* and is thus fully dynamic. A *current producer* is responsible for construction of the new structure. *Node expressions* specify nodes to be created (the preceeding `new` and succeeding parentheses may be omitted) or reinserted into the graph (if they occurred in a query and were named), and may be connected by edges of various types. Multiple independent structures can be created by a production if they are separated by a comma.

Blocks of ordinary Java code can be embedded into productions by enclosing it in braces { and }. To access variables declared in such a block later on outside of this block, the code block does not introduce its own scope, but instead behaves as if inserted into its enclosing scope.

`array`	convert sequence to array
`count`	number of elements in sequence
`empty`	test if sequence contains no elements
`exist, forall`	logical or/and of all elements of sequence
`first, last`	first/last element of sequence
`max, min`	maximum/minimum element of sequence
`mean`	arithmetical mean
`prod, sum`	product/sum
`string`	convert to string containing comma-separated list of values enclosed in brackets
`selectRandomly`	randomly select one value, either with uniform probability or by a provided relative probabilities
`selectWhere`	select first value for which an additional boolean parameter evaluates to `true`
`selectWhereMax, selectWhereMin`	select the value for which an additional parameter becomes maximal/minimal

Table 3.5.: Standard aggregate methods in XL.

Control flow statements in rule mode allow dynamically created productions. For instance, loop instructions (`for`, `do`, `while`) can be used to repeat creation of a certain structure, thereby extending the repetition operator of growth grammars. The body of the control flow statement has to be enclosed in parentheses (to remain in rule mode) or braces (to switch to Java mode).

PROPERTIES

Properties are similar to instance fields, but values are read and written using special methods. They are accessed by `e[n]`, where `e` is an expression of reference type `T` and `n` is the name of a property declared in `T`.

In addition to the usual operators `=`, `+=`, etc., properties can also be modified using *deferred assignments* (see table 3.6). These operators are prefixed by a colon and were introduced to support parallel rule application by delaying attribute modifications to a later point (again XL does not dictate this semantic, but this is how it is used in GroIMP).

MODULES

As for parametric L-systems, XL allows to define *modules*. Nodes that are module instances provide attribute access by positional parameters in addition to named attributes. Modules are defined like ordinary Java classes, but instead of `class` the keyword `module` has to be used, with the name of the module followed by a parenthesized list of parameters. The XL compiler will then generate a node pattern so that the module can be searched for in queries, and a constructor for occurrences in productions.

In the list of module parameters, attributes of superclasses can be listed by preceeding their name with the keyword `super` and a period. This allows specialized modules that reorder the parameter list or extend it.

Another feature of module definitions are *instantiation rules*. The module definition is followed by an arrow `==>`, which in turn is followed by a production that creates

`:=`	assign
`:+=`	increment
`:-=`	decrement
`:*=`	multiply
`:/=`	divide
`:%=`	modulo
`:**=`	raise to power
`:<<=`	signed/unsigned shift left
`:>>=`	signed shift right
`:>>>=`	unsigned shift right
`:&=`	and
`:|=`	inclusive or
`:^=`	exclusive or

Table 3.6.: Deferred assignment operators in XL.

geometry that should be drawn in addition to the module. This can be used to create geometry algorithmically "on the fly" (trading memory for computation time), or to place geometry defined elsewhere to the location of the module instance.

ANONYMOUS FUNCTION EXPRESSIONS

Functors (function objects) are functions with associated state, and are typically used as callback functions. In Java functors can be defined using anonymous inner classes. However, for simple functions such a definition becomes very verbose. XL provides *anonymous function expressions*

```
X x => Y e
X x => Y* e
```

to generate inner classes (their name depends on the types `X` and `Y`) with an evaluate function, that computes the expression `e` based on the input parameter `x`. The first form returns a single value, while the second form yields a sequence of values.

Examples for using anonymous function expressions have been given in [Kni08]. For the single-value form, one can write

```
DoubleToDouble f = double x => double x * Math.sin(x);
```

to define a functor for the function $f(x) = x \cdot sin(x)$. A sequence of values can be generated by

```
ObjectToObjectGenerator<Node,Shoot> children =
  Node parent => Shoot (* parent --> Shoot *)
```

containing all nodes of type `Shoot` connected to some `parent` node.

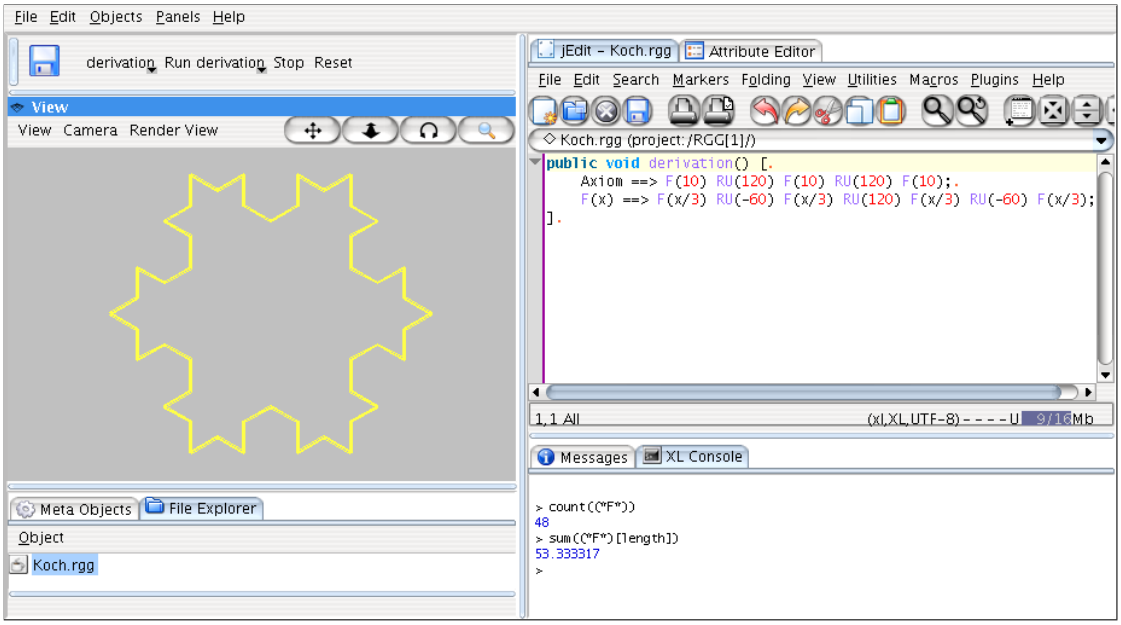


Figure 3.7.: Screenshot of the graphical user interface of GroIMP.

3.5. GroIMP

GroIMP[9] (growth-grammar related interactive modelling platform) is an open-source software implemented in Java and licensed under the GNU General Public License (GPL), version 3. Its main components are

- the XL compiler and runtime environment, integrated with a text editor (jEdit[10]) and a message panel (showing compilation errors with clickable links to their source code location),
- classes for geometric primitives like sphere, cone, cylinder, box, etc., for modelling and visualization,
- a shader system for texturing geometric objects,
- a graphical user interface (shown in figure 3.7) with 3D view for interactive visualization and manipulation of the model,
- a raytracer *Twilight* for rendering the 3D view,
- and a 2D view of the graph.

[9] `http://sf.net/projects/groimp`, accessed 18 October 2011
[10] `http://www.jedit.org`, accessed 18 October 2011

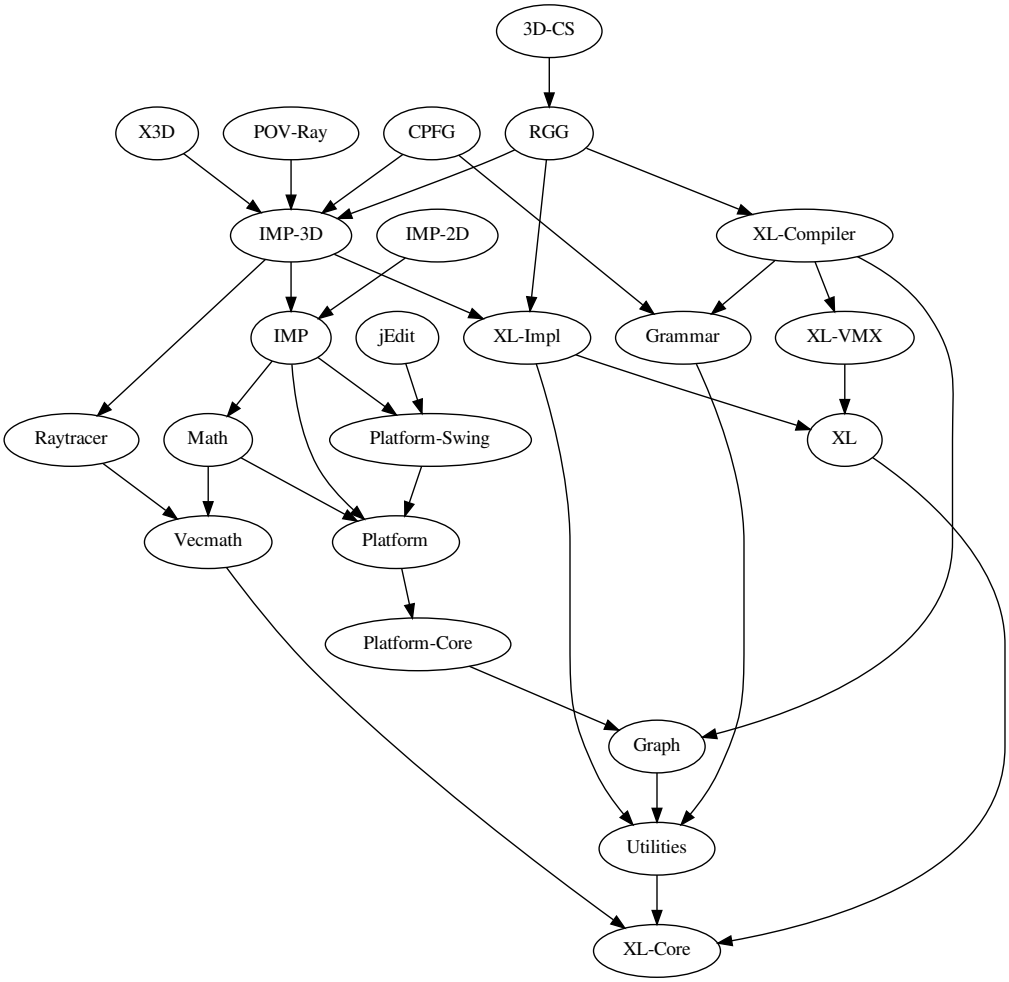


Figure 3.8.: Schematic view of GroIMP plug-ins and their dependencies.

Being based on Java, the software runs on any platform that is supported by a JVM. GroIMP ships with a battery of examples, that demonstrate some of its features. The user-interface is fully dynamic and consists of panels, that can be moved and (un)docked. A layout of the GUI can be saved for later use. This is used also, for instance, to provide two initial workspace configurations, one for interactive modelling and the other for RGG development.

Several import and export filters allow to share files with other programs, for instance GROGRA[Kur94] (`dtd`, `dtg`), external libraries (`jar`), or images (`png`, `jpg`, ...) for use as textures. Export of images rendered by the built-in raytracer to such file formats is also possible. Images can be managed by a panel called image explorer. Panels for other types of resources like shaders, functions, curves, and datasets exist as well.

GroIMP has a plug-in architecture, similar to the Eclipse platform[11]. Each plug-in is described by an XML file `plugin.xml` which defines the plug-in's capabilities, a list of prerequisite plug-ins, and libraries the plug-in depends on. Capabilities are represented as a tree structure and include menu entries, input/output filters, etc., and are combined for all plug-ins into a single registry. The list of prerequisite plug-ins is used to derive the initialization order of the plug-ins and prevents cyclic dependencies between the plug-ins. A schematic view of the plug-ins and their dependencies can be found in figure 3.8, and a short description in [Kni08].

The RGG plug-in connects GroIMP to the XL programming language. It provides compilation filters for different source file formats (`rgg`, `xl`, `java`, `lsy`, `ssy`), and calls the XL compiler for all such files contained in a project. The plug-in also provides a class `de.grogra.rgg.RGG`, that serves as base class for relational growth grammar development and manages the life cycle for such RGGs. The package `de.grogra.turtle` contains scene graph nodes for turtle commands, to provide compatibility with the GROGRA software. Finally, the class `de.grogra.rgg.Library` provides utility functions that ease model development.

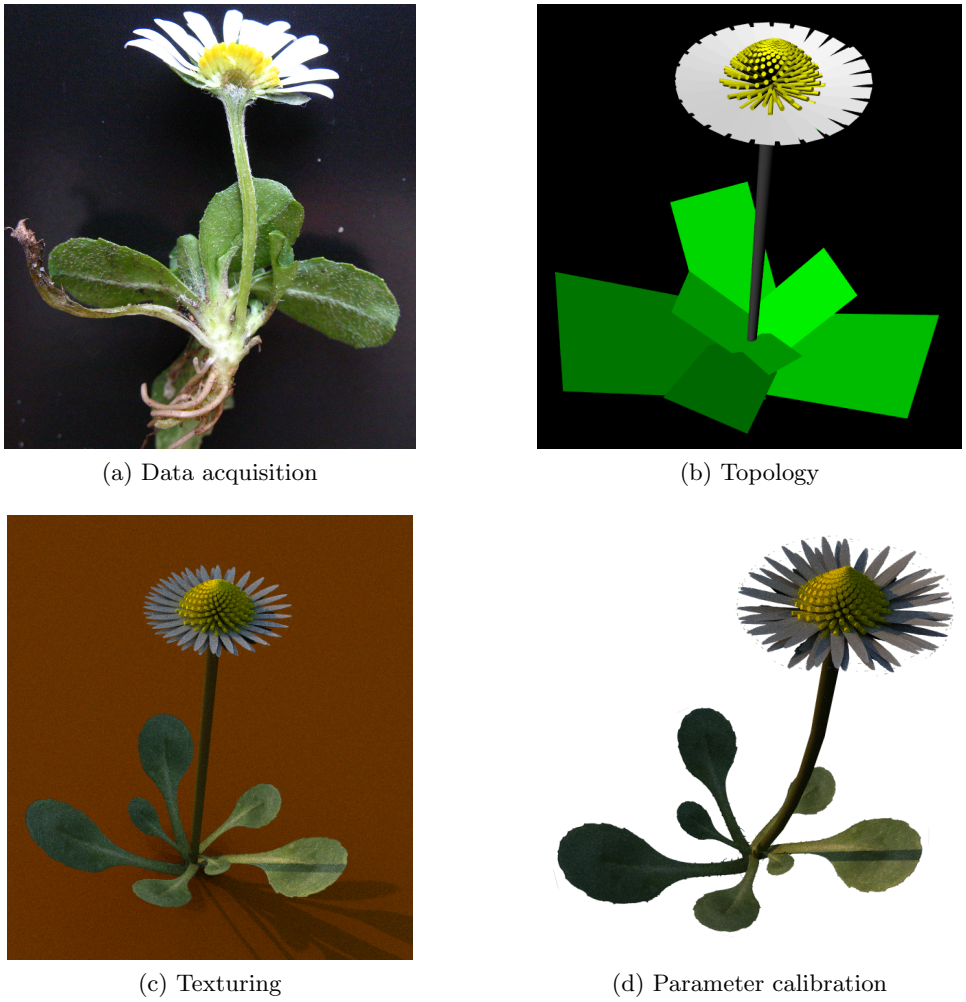
As many parts of GroIMP, and especially the RGG plug-in, are repeatedly involved when creating a new model, a new file format (`rgg`) has been designed that simplifies this action. Files with such an extension are modified before they are passed to the XL compiler. The *RGG dialect* of XL implicitly surrounds the code by a declaration of a subclass of `RGG` and the addition of several import statements. It also defines a static field `INSTANCE` that allows to refer to the instance, for instance, if a project consists of multiple files.

3.6. Structural Modelling with GroIMP/XL

Model development with GroIMP and XL usually starts by creating a simple structural model. This can then be extended by inclusion of physiological processes to an FSPM (functional-structural plant model), or used on its own in areas like computer graphics. How to create a simple structural model of a daisy has been shown in [SH10].

The steps involved in creating a structural plant model are (see figure 3.9)

[11] `http://www.eclipse.org`, accessed 18 October 2011



(a) Data acquisition (b) Topology

(c) Texturing (d) Parameter calibration

Figure 3.9.: Steps to create a structural model of a daisy.

(a) Data acquisition,

(b) Creation of topology,

(c) Texturing,

(d) Parameter calibration.

Data Acquisition Data about the plant can be obtained by collecting real plants and comparing their structure, measuring lengths (of steam, branches, etc.), count parts of them (like leaves, blades, etc.), and observing their color or texture. Also books can serve as reference for such data.

Topology Topology in the model defines what parts the plant is made of, how many of each occur, and how they are connected. Figure 3.10 shows a schematic view of the daisy. In

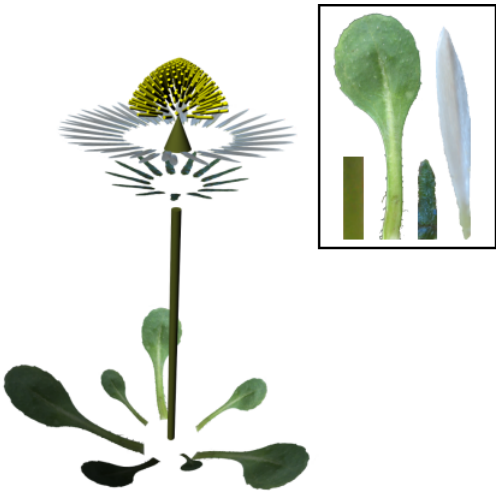


Figure 3.10.: Schematic view of the daisy.

the model, for each part of the plant a corresponding module is created. Each module also possesses a list of parameters relevant for it. Modules are also equipped with a graphical representation, either by deriving them from some geometric primitive, or by using an instantiation rule. For instance, the stem and flower petals were defined as

```
module Stem(float length, float diameter)
    extends Cylinder(length, diameter/2);
module Flower(float length, float diameter, int color)
==> if (color == YELLOW)
        (Cylinder(length, diameter/2))
    else
        (leaf(length, diameter));
```

and the other modules in a similar way. The instantiation rule for `Flower` dynamically chooses a geometric primitive (cylinder or parallelogram) based on the parameter `color`.

Use of these parameters is made in a rule, that replaces the initial `Axiom`:

```
Axiom ==>
    // create rosette of 7 leaves, diameter half of length
    for (int i : 1:7)
    ( [
        RH(i * 137.5)   // rotate by Fibonacci angle
        { double r = 50 - i * 5; }
        Leaf(r, r/2)
    ] )
    // create stem, length 70mm, diameter 2mm
    Stem(70, 2)
    ...
```

Leaves (and also flowers) are created in a loop by successively rotating around the *Fibonacci angle* [PL90]. The remaining parts are created in a similar way by that rule. Additional rotation and translation commands were inserted to control orientation of the parts.

TEXTURING

Using textures allows to give the model a more realistic look. The importance of this can be seen by comparing figures 3.9b and 3.9c. For texturing, GroIMP provides a material system, which makes it possible to map images onto surfaces, or define textures procedurally. In case of the daisy model, image data was obtained by making photographs, and then using an image editing tool like the GIMP[12] to cut out the textures. Then, these image files were imported interactively into the daisy project using GroIMP so that they appear in the *Image Explorer* and *Shader Explorer*.

A material defined in the shader explorer can be used in the model by obtaining a reference to it, and then applying the material to the module. For instance, the leaves can be assigned a texture by

```
ShaderRef leafShader = shader("leafShader");
...
module Leaf(float length, float diameter)
    ==> leaf(length, diameter).(setShader(leafShader));
```

PARAMETER CALIBRATION

Finally, to further improve the visual appearance, parameters like lengths and angles can be calibrated. Also, randomness can be introduced by replacing a fixed parameter value by a normally distributed value (where mean and variance depend on measurements). Parameter calibration can also be reversed, by first creating the structure and collecting statistical data from it [DKH+07], then adjusting parameters so that the values derived from the model correspond to the measured data.

Other plants like fern (figure 3.11a) and horsetail (figure 3.11b) were modelled in a similar way. Figure 3.12 shows a scene that combines daisy, fern and horsetail, together with an alder tree [Rog08], grass leaves, an ivy plant, and some spruce trees [HKL+08]. It is also possible to distribute the plants according to a function. For instance, in figure 3.14 the daisy flowers were arranged according to a black/white image containing the text SCCG08.

The structure of trees (and many other plants) follows one of the architectural models proposed by [HOT78], which are shown in figure 3.13. Templates of such architectural models in form of GroIMP projects have been developed by Smolenova [SH10], and can serve as a starting point when developing new tree models.

To obtain an FSPM (functional-structural plant model), these templates can then be extended to include physiological processes. An often used approach is the *pipe model theory* [SYHK64a, SYHK64b]. It considers a so-called *unit pipe* (figure 3.15a), a collection of leaves associated with a pipe of constant cross-sectional area, corresponding to water-conducting vessels in branches and stem. A community of plants can then be thought of as an assemblage of unit pipes (figure 3.15b), but also an individual plant (figure 3.15c).

[12] `http://www.gimp.org`, accessed 20 October 2011

(a) Fern (*Dryopteris filis-mas*) (b) Horsetail (*Equisetum arvense*)

Figure 3.11.: Virtual fern and horsetail modelled with GroIMP/XL.

Figure 3.12.: A virtual scene combining different plants.

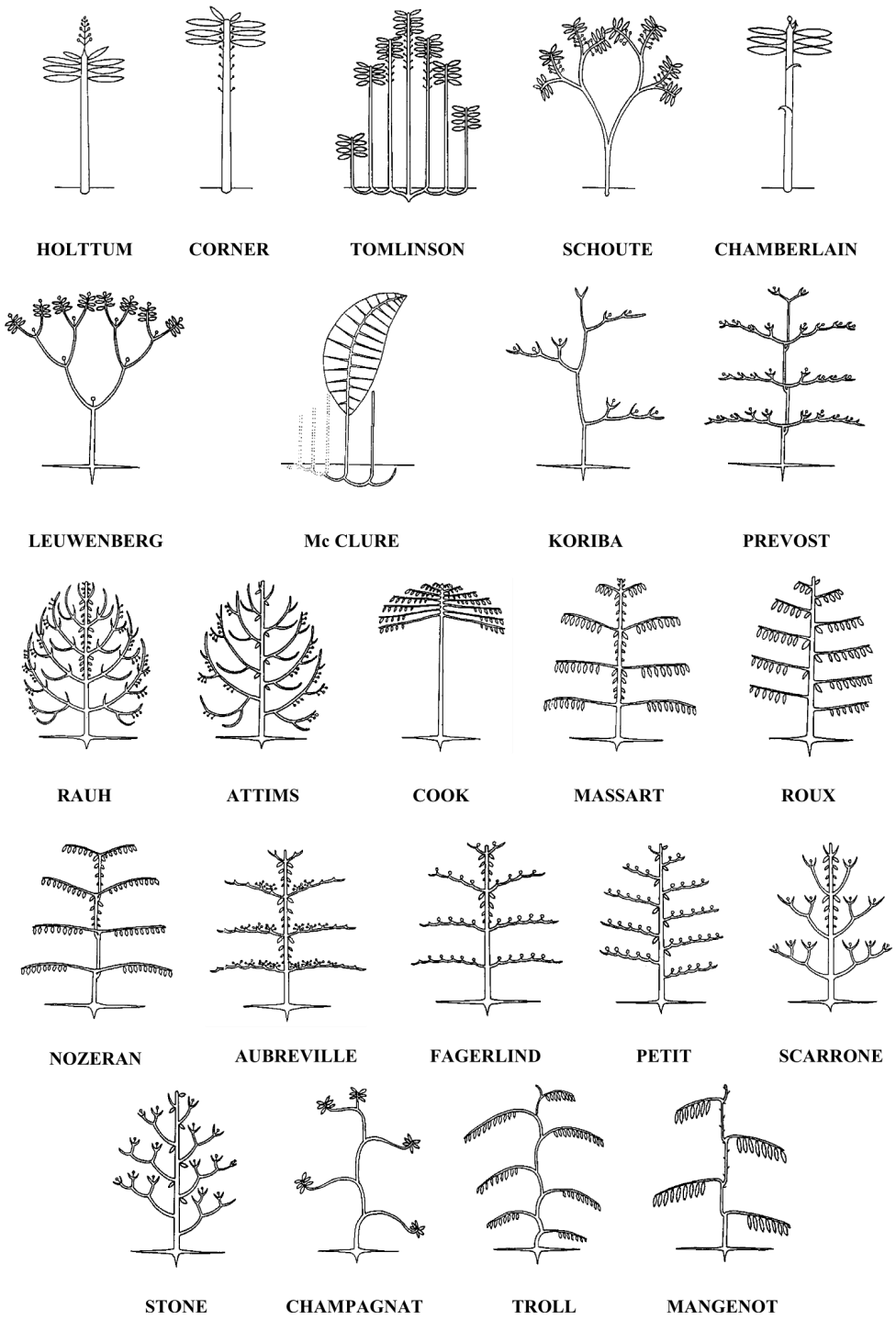


Figure 3.13.: Architectural models of trees (image from [Fer00]).

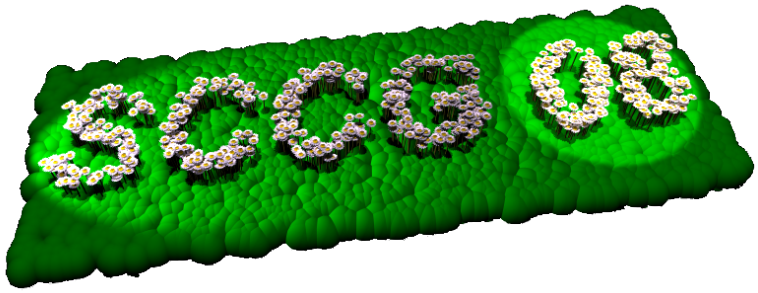
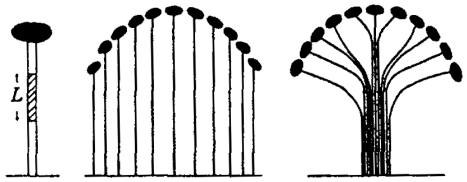


Figure 3.14.: Daisy flowers distributed procedurally.

The pipe model can also explain the development of tree growth, where when lower branches are shed parts of their pipes remain in the trunk giving them a conical shape (figure 3.16). In GroIMP this shape can be modelled by a frustum.

Pipe model theory also conforms to *Leonardos rule*, that states that "all the branches of a tree at every stage of its height when put together are equal in thickness to the trunk" [Ric70]. This means that the total cross-sectional area remains constant at every branching point (disregarding shed branches).



(a) Unit pipe (b) Stand (c) Plant

Figure 3.15.: Diagrammatic representation of the simple pipe model (from [SYHK64a]).

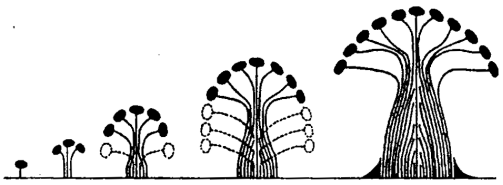


Figure 3.16.: Diagrammatic representation of the pipe model of tree form, showing the successive accumulation of disused pipes in the trunk associated with the progress of tree growth (from [SYHK64a]).

4. Operator Overloading

In this chapter we will introduce *operator overloading* into the XL language. We will start with analyzing existing implementations of operator overloading in other programming languages. Then we will define how operators should be overloaded in XL. This was strongly influenced by operator overloading in C++. Finally we will discuss how this was implemented and in which way the XL compiler had to be extended and modified.

4.1. Operator Overloading in Existing Programming Languages

Programming languages generally provide a set of operator symbols that allow to specify numerical expressions. For instance, the operator `+` usually represents addition of numbers, and the expression `a + b` means to calculate the sum of expressions `a` and `b`.

Operator overloading allows the programmer to give an operator another semantics. This is supported by many programming languages (like C++ [ISO11], D [Ale10], Groovy[1]). Even Java overloads operators, namely the operator `+`, which can be used to concatenate strings, besides its use as addition operator for numerical expressions ([GJSB00, section 15.18.1]). However, in Java it is not possible to provide user-defined operator functions, while other languages like Prolog and Lisp even allow the user to introduce new operator symbols.

One might argue if operator overloading is a good thing or not. As with any tool, it should be used wisely and where appropriate. As an example of bad usage, consider a 3D math library, where the cross product is performed by an operator `*`. Then the question arises what operator should be used for the dot product. Using another operator (like `/` or `%`) for the cross product is even less intuitive. However, these issues are not due to operator overloading, but have to be attributed to bad design decisions (like if a function is called `print`, but adds two numbers instead). So in the example, free functions `dot` and `cross` would be a better choice.

Proper usage of operator overloading would refrain from operator abuse. Common guidelines[2] for overloading operators suggest to use "common sense", that is the operators should behave as expected and similarly to how the numerical operators work. For instance, it is perfectly fine to extend semantics of the addition operator `+`, etc., to operate on vectors, matrices, and complex numbers. The aim is to make working with these objects easier for the user, albeit it might need some experience from the developer to do things right.

[1] `http://groovy.codehaus.org`, accessed 21 October 2011

[2] `http://www.parashift.com/c++-faq-lite/operator-overloading.html#faq-13.9`, accessed 21 October 2011

Operators should be *symmetric*, like that when an increment operator `+=` is defined, also a decrement operator `-=` should be defined, with oppositional semantics, and they should be *consistent*, like that `a += b` has the same semantics as `a = a + b`.

4.1.1. Operator Overloading in C++

The programming language C++ provides operators in the form of symbols (like `+`, `-`, `*`, `/`, etc.) and keywords (like `new`, `delete`, etc.). The available set of operators is fixed, and each operator has a corresponding precedence and associativity. Almost all of them can be overloaded. Note that some operators are considered distinct, even though they are using the same operator symbol (like unary/binary `+` and `-`, or prefix/postfix `++` and `--`). At least one of the operands of any overloaded operator must be a user-defined type. This prevents redefinition of any built-in operator.

To overload an operator, a function must be defined and its name must be the operator symbol prefixed by the word `operator`. For instance, a common practice is to overload the assigment operator of a class:

```
class Complex
{

private:
  double real, imag;

public:
  Complex& operator= (const Complex& other)
  {
    real = other.real;
    imag = other.imag;
    return *this;
  }

  ...
};
```

This allows to assign an instance of class `Complex` to another instance. The implementation of the assignment operator ensures that the data is copied in the correct way. In this example this is trivial, but in more complex cases (like std::string) the operator implementation must also allocate/release internal buffers and check for self-assigments. Returning a reference to the operand on the left hand side of the assignment allows *invocation chaining*, so that the overloaded operator behaves for instances of class `Complex` like the ordinary operator for built-in datatypes (note that assignments are right-associative):

```
Complex a, b, c;
a = b = c;
```

Operators can be implemented as methods of a class (like done above). Then the operand on the left hand side is implicitly represented by the `this` pointer. Another option is to implement them as global functions, which is often used for binary operators.

C++ provides the keyword `friend` to give a function access to private data of a class, although the function itself is visible globally. For instance, the addition operator of class `Complex` could be implemented as:

```
class Complex
{
  ...

public:
  friend Complex operator+ (const Complex& a, const Complex& b)
  {
    Complex result;
    result.real = a.real + b.real;
    result.imag = a.imag + b.imag;
    return result;
  }

};
```

This becomes powerful if combined with C++'s implicit conversion mechanism by using *conversion constructors* and *conversion operators*. Like *standard conversions* exist for primitive types (for instance, from `int` to `float`), conversions to and from a user-defined type can be defined. A conversion constructor takes a single parameter of some other type, and uses this value to initialize the instance. To suppress unwanted implicit conversions, the function can be qualified with the keyword `explicit`. Then manually casting to another type invokes the conversion function.

In the example above, an instance of class `Complex` can be initialized from a `double`, if an appropriate constructor is implemented:

```
class Complex
{
  ...

public:
  Complex(double d)
  {
    real = d;
    imag = 0;
  }

};
```

For `a`, `b`, and `c` instances of class `Complex`, all code above together allows to write statements like the following:

```
c = a + b;
c = a + 3;
c = 3 + b;
```

The first line simply calls `operator+` with two complex numbers as arguments. The second line implicitly calls the conversion constructor to convert the integer 3 into an

instance of `Complex`, then calls `operator+` to perform the addition. The third line does the same, but note that this only works because the `operator+` was defined as `friend` function. If it were instead defined as member function, the first operand must already be an instance of `Complex` for the `operator+` to be called.

A subtle difficulty arises when overloading the increment and decrement operators `++` and `--`. Two forms exists, a prefix and a postfix version, like in `++i` and `i++`. Both of them increment/decrement the variable, but the former returns the original value, while the latter returns the modified value as result of the expression. As for both forms the name of the operator function is the same (`operator++` respectively `operator--`), an additional feature is needed to distinguish both of them. In C++ this is solved by requiring the prefix operator to accept no parameters, while the postfix operator is required to accept an `int` (a dummy parameter, whose name doesn't matter).

4.1.2. Operator Overloading in D

The D programming language [Ale10] also supports operator overloading. How this is done differs between version 1.0 and version 2.0 of the language. Common to both approaches is that the overloaded operators are specially named member functions of a class or struct.

Operator Overloading in D 1.0

A unary operator `op` applied to a class or struct (like in `-a`) is interpreted as if its corresponding member function `opfunc` was called (like `a.opNeg()`). The name of the member function always consists of a prefix `op` followed by a string corresponding to the operator to overload. The prefix operators `++` and `--` cannot be overloaded directly, but are instead rewritten using `+=` and `-=` (so `++e` becomes `(e += 1)`).

Binary operators can be overloaded in an analogous way. An expression containing a binary operator `op` (like `a + b`) is rewritten by calling a member function `opfunc` (like `a.opAdd(b)`). To enable also expressions like `3 + a`, an additional (reversed) member function `opfunc_r` may be provided (like `b.opAdd_r(a)`). If no match is found and the operator is commutative, also operator functions for both operands exchanged are searched for (like `b.opAdd(a)` and `a.opAdd_r(b)`).

Comparison operators `==` and `!=` both use the same operator function `opEquals`. So `a == b` is rewritten as `a.opEquals(b)`, and `a != b` is rewritten as `!a.opEquals(b)`. The other comparison operators `<`, `<=`, `>`, `>=` use the operator function `opCmp`. The function is used to compute a value, that is then compared against zero (for instance, `a < b` becomes `a.opCmp(b) < 0`).

The function call operator can be overloaded by providing member functions `opCall`. The object can then be called as if it were a function, easing its use as functor.

Other operators that can be overloaded are the array index operator `opIndex`, the array assignment operator `opIndexAssign`, and the array slice operators `opSlice` and `opSliceAssign`.

Operator Overloading in D 2.0

All unary operators can be overloaded through a single template function `opUnary`. The template has an additional string parameter, that can be used to deduce or restrict the operator to be overloaded. While for D 1.0 the prefix operators `++` and `--` were handled automatically, now this accounts to the postfix forms thereof. An expression `e++` is rewritten as `(auto t = e, ++e, t)`.

Binary operators are rewritten to template functions `opBinary` and `opBinaryRight`, with a template parameter of type string to recognize the operator to overload. The one that better matches is selected. Exchanging the operands for commutative operators, like in D 1.0, was abandoned. Compound assignment operators (like `+=`) were handled as binary operators in D 1.0, but in D 2.0 they are represented by a templated function `opOpAssign`.

Handling of comparison operators is similar to its handling in D 1.0, but for relational operators `<`, `<=`, `>`, `>=` also a reversed form is tried. For instance, for `a < b` both `a.opCmp(b) < 0` and `b.opCmp(a) > 0` are checked.

Index and slice operators work like in D 1.0, but index assignment and slice assignment operators now are represented by template functions `opIndexOpAssign` and `opSliceOpAssign`.

4.1.3. Operator Overloading in Groovy

Groovy is a programming language in its own right, but is compiled to Java bytecode, which makes it possible to easily combine both, Groovy and Java programs. However, there are some differences. For instance, in Java the operator `==` checks for equality for primitive types, but for identity for objects, while in Groovy the operator checks for both of them for equality (objects are compared by means of the `equals` function). Reasoning for this is that equality checks should also work as expected when combined with autoboxing. Comparison for identity can be done with function `is` in Groovy.

Overloaded operators are implemented as ordinary member functions. For instance, `a + b` is rewritten as `a.plus(b)`. The other operators are named similarly. Increment and decrement operators `++` and `--` are overloaded by functions `next` and `prev` for both forms, prefix and suffix. This makes them compatible to Java's iterators (in `java.util.Iterator`) so that they can be used like iterators in C++.

Comparison operators `<`, `<=`, `>`, `>=` are overloaded by providing a method `compareTo` whose result is compared against zero (e.g., `a < b` is rewritten as `a.compareTo(b) < 0`). An operator `<=>` exists, that is simply rewritten as `a.compareTo(b)`, and usually returns -1 if the left operand is smaller, 0 if both are equal, and 1 if the left operand is greater.

4.1.4. Conclusions for Operator Overloading in XL

In all three languages operators are overloaded by providing specially named functions. These functions can then be called like any regular function, or in form of an operator applied to some expression.

The naming scheme of operator functions is different between the three languages. While D and Groovy use ordinary names for operator functions, C++ uses special names containing the operator symbol, which are not allowed for ordinary functions. This naming for operator functions in C++ effectively suppresses name collisions with user-defined functions. In D such conflicts are at least very improbable because of the used naming scheme. In Groovy, chances are high that the user unintentionally overloads an operator, which might be even intended in case of the increment and decrement operators.

The way C++ handles naming of operator functions seems to be most suitable. So in XL operators should be overloaded in a similar way, by naming them `operator+` and the like, and transforming these names to an internal representation that prevents name collisions.

To make operator overloading more useful for binary operators, it should be allowed to combine two operands of different type, especially one of the operands should also be allowed to be of primitive type (like `int` or `double`). The primitive type may also appear as left operand, like in `3 + c` for a complex number `c`, or `1.2 * v` for a geometrical vector `v`. D solves this by allowing the programmer to provide two operator overloads, a normal form, and a reversed form with both operands exchanged. C++ instead relies on user-defined conversion functions, that allow to convert the operands into the correct form to apply the operator.

The C++ solution with user-defined implicit conversion functions seems to be more flexible, and also extends the idea of automatic conversions introduced as *auto(un)boxing* with version three of the Java language [GJSB05], as will be seen later on in section 4.3. Without implicit conversions, many versions of the operator would have to be implemented for sake of matching the types of the operands.

The increment and decrement operators `++` and `--` need special handling as well. While it might seem like a good idea to automatically rewrite those operators in terms of another (like was done in D), this takes away some flexibility in expressiveness. Also it is trivially possible to rewrite those operators manually in terms of another, so there is no need to enforce this by the language.

Therefore, in XL increment and decrement operators should be defined like it is done in C++. A dummy parameter of type `int` distinguishes the postfix form from the prefix form.

Care must be taken when overloading comparison operators `==` and `!=`. As already mentioned above, in Java these operators applied to primitive types check for *equality*, but applied to objects compare for *identity*. In Groovy this is solved by introduction of a new operator `is` to check for identity, and use of `==` and `!=` to check for equality instead. But this raises the question whether the `is` operator should be overloadable as well.

The intention for overloading operators usually is to make instances of classes behave as if they were primitive types. For instance, the same operations that can be performed with an `int` should also be possible with an instance of a class `Complex`. Overloading comparison operators is part of this process. Therefore, the intention when writing an expression like `a == b` is to check for equality of two objects, and not if `a` and `b` reference

the object.

In XL, if operators `==` and `!=` are overloaded they will check for equality. A check for identity is still possible by means of the `equals` method of class `java.lang.Object`, which is the (direct or indirect) superclass of any class.

C++ allows to overload the assignment operator. One should discuss if allowing this for XL is a good idea. Consider the following lines of code:

```
1 Complex a = new Complex(1, 2);
2 Complex b = new Complex(3, 4);
3 b = a;
4 a++;
```

If overloading the assignment operator was not possible, so assignments behave like in Java and copy just the reference and not the referenced object, then the third line would cause both references `a` and `b` to refer to the same object. The fourth line would modify the object `a` refers to, and by this also the object `b` refers to. This would be unexpected behaviour, supposed that `Complex` should behave like a primitive type.

Assuming the assignment operator was overloadable and would, in the third line, cause a copy of the referred data instead of just the reference. Then a modification like in the fourth line would only affect the object referenced by `a`, and not the one referenced by `b`. This is the expected behaviour. But what about the first line then ?

If it were also considered an assignment, then the assignment operator (if defined as a member function of class `Complex`) would be called on an uninitialized object (actually `a` does not refer to any object that the values could be stored in). The compiler could generate code that automatically allocates a new object in this case, but this would place restrictions on how the class has to be defined (it then must have an accessible default constructor) and could cause unexpected side-effects.

Alternatively, the statement in the first line could be explained as definition and initialization of a variable, so no assignment operator would be called. But if the user then wrote

```
5 Complex c = new Complex(1, 2);
6 Complex d = c;
7 c++;
```

then `c` and `d` would again refer to the same object and the increment in the seventh line would affect both, `c` and `d`, which is unexpected. Also, a reference in Java (and XL) is similar to a pointer in C++, and C++ does not allow overloading the assignment operator for pointers.

In conclusion, XL does not allow to overload the assignment operator.

4.2. Operator Overloading in XL

Operator overloading in XL is very similar to how it is done in C++. However, many complications that arise in C++ because of its memory management (objects can be allocated on the heap and locally on the stack) withdraw in XL (as there is only a

garbage collected heap). Accordingly, operator overloading becomes simpler and less error prone.

When overloading an operator, the name of the operator and number of parameters it accepts must match. For example the name `operator-` could refer to the binary or the unary minus operator. The compiler counts the number of parameters, including the implicit `this`-reference if available, then decides which operator function was overloaded.

In the example above, the unary minus operator may either be a static function with one parameter, or a member function with no parameter (besides the implicit `this`-reference every member function has).

The return type and parameter types of an operator function may be freely chosen, i.e. the `operator<` does not necessarily need to return a `boolean`. How this can become beneficial will be shown later on in section 4.4.4.

The user is not required to overload every operator, but only those that make sense. For instance, overloading `operator<` for class `Complex` does not make much sense (one could sort complex numbers in many ways). If the user tries to call a non-overloaded operator, a compile-time error is generated.

So picking up the example of a class for complex numbers from before, one could implement addition of two such numbers in the following way:

```
class Complex {
  private double real, imag;

  public Complex(double real, double imag) {
    this.real = real;
    this.imag = imag;
  }

  public static Complex operator+ (Complex a, Complex b) {
    return new Complex(a.real + b.real, a.imag + b.imag);
  }

  ...
}
```

Code that performs computations on such complex numbers can statically import the operator function, so that it becomes globally visible (in analogy to `friend` functions in C++). For instance, this way one could write:

```
import static Complex.*;
...
Complex a = new Complex(1, 2);
Complex b = new Complex(3, 4);
Complex c = a + b;              // c.real == 4 and c.imag == 6
```

The remaining operators can be implemented in a similar way. For the increment and decrement operators `++` and `--`, the prefix and postfix forms must be distinguished. In the complex number example, prefix and postfix form of the increment operator could be implemented in this way:

```
class Complex {
  ...

  // prefix increment
  public Complex operator++ () {
    real++;
    return this;
  }

  // postfix increment
  public Complex operator++ (int _i) {
    Complex result = new Complex(real, imag);
    real++;
    return result;
  }
}
```

The postfix variant expects an additional paramter of type `int`, its name can be chosen freely. As can be seen, the postfix form is required to create a temporary copy, so `i++` performs slightly less efficient than `++i`.

Comparison operators `==` and `!=` can be overloaded as well. Care must be taken to not recursively call the operator itself, as this will result in a stack overflow at runtime. For the complex number example, an implementation of the operator `==` might look like this:

```
class Complex {
  ...

  public static boolean operator== (Complex a, Complex b) {
    return a.real == b.real && a.imag == b.imag;
  }
}
```

One might argue if it is necessary to handle the case if either (or both) reference is `null`. The implementation, as defined above, would throw a `NullPointerException` in this case. Considering that overloading operators should help to substitute a class for a primitive type, passing in a `null`-reference indicates an error in the code that uses the class. Therefore, throwing an exception seems to be appropriate.

4.2.1. Implementation

The (internal) name of an operator function is restricted by the Java Virtual Machine specification to be a valid name in the Java programming language ([LY99, §2.7.1 and §4.6]). As stated in [GJSB00, section 3.8 on page 19], *identifiers* (of which method names are one kind) may consist of *Java letters* and *Java digits*, but must not start with the latter. Furthermore, "the Java letters include uppercase and lowercase ASCII Latin letters A–Z (\u0041–\u005a), and a–z (\u0061–\u007a), and, for historical reasons, the ASCII underscore (_, or \u005f) and dollar sign ($, or \u0024). The $ character should be

used only in mechanically generated source code or, rarely, to access preexisting names on legacy systems." Java digits are the normal ASCII digits 0–9 (\u0030–\u0039).

Naming a method `operator+` is not allowed, but the special handling of names containing a dollar sign $ opens up another possibility. Names of operator functions are mapped to legal names for Java functions by prefixing an operator dependent name with the string `operator$`. For instance, the internal name for the unary `operator+` becomes `operator$pos`, and for the binary `operator+` it becomes `operator$add`. This also prevents collisions with names of user-defined functions.

Table 4.1 on page 99 lists all operators available in XL, sorted by precedence. Overloadable operators are listed together with their internal name suffix, all other operators cannot be overloaded. Additional operators to support graph replacement rules are included in the table. The modifications to the XL compiler so that it supports operator overloading are outlined below.

The implementation of the XL compiler is described in [Kni08, chapter 8]. At first, the input file is decomposed into tokens by the lexical analyzer (`XLTokenizer`). Then tokens are combined into abstract syntax trees by the parser (`XLParser`). A semantic analyzer (`Compiler`) converts these into expression trees. Finally, a code generator (`BytecodeWriter`) produces bytecode that can be executed by the JVM. To facilitate implementation, the parser `XLParser` (generated from `XL.g`) extends class `Parser`, and the semantic analyzer `Compiler` (generated from `Compiler.tree.g`) extends class `CompilerBase`.

To make the compiler understand operator overloading, `CompilerBase` was retrofitted with a method `checkOperatorFunction`. This determines the number of operands (counting `this` for member functions as additional operand), then decides if such an operator is overloadable. In the parser grammar (`XL.g`) in rule `typeMember` (members of a type are fields, methods, etc.), a call to `checkOperatorFunction` ensures that only valid combinations of operator symbol and number of parameters are overloaded. The rule `methodIdent`, which parses the name of a method, is used to distinguish between normal and operator methods, and to check that for methods starting with the prefix `operator` only valid operator symbols are used (rule `overloadableOperator`). Also tokens of non-Java operators are included in the list.

Now that the parser accepts definitions of operator functions, the semantic analyzer must be instructed to call those methods in place of an operator. The grammar of the semantic analyzer (`Compiler.tree.g`) contains rules `unaryExpr`, `binaryOp`, etc., that in turn call the function `compileOperator` in `CompilerBase` to generate an expression tree from an abstract syntax tree. Application of an overloaded operator is done by generating a call to its operator function. If no matching operator function can be found, an error message is generated. An error is also generated, if multiple operator functions match (e.g., binary addition defined as static function with two parameters and as member function with one parameter).

4.3. Implicit Conversions

Operator overloading alone is already useful, but becomes powerful in combination with implicit conversions. Java already defines a number of conversions, namely

- Identity conversions,
- Widening primitive conversions,
- Narrowing primitive conversions,
- Widening reference conversions,
- Narrowing reference conversions,
- Boxing conversions,
- Unboxing conversions,
- Unchecked conversions,
- Capture conversions,
- String conversions,
- Value set conversions.

Different conversions apply in different contexts (assignment, method invocation, etc.). For reference see [GJSB05, chapter 5].

Identity conversion just states that conversion of a type to the same type is always permitted. This includes redundant application of cast operators.

A widening primitive conversion allows a type `byte`, `short`, `int`, `long`, `float`, `double` to be cast to any other type right to it in this list, and to cast `char` to `int` and types right to it in the list. Widening primitive conversions extend the magnitude of a numeric value, but might loose precision (for instance, when converting `int` to `float`).

A narrowing primitive conversion allows a type `byte`, `short`, `int`, `long`, `float`, `double` to be cast to any other type left to it in this list or to `char`, and `char` to `byte` or `short`. Note that conversion from `byte` to `char` first performs a widening primitive conversion to `int`, then a narrowing primitive conversion to `char`. Narrowing primitive conversions might loose information about magnitude and precision of a numeric value.

A widening reference conversion simply allows to convert a reference type to any of its supertypes.

A narrowing reference conversion allows, among others, to convert from a type to one of its derived types, and requires a test at run time if the conversion is really possible.

Boxing conversions convert a primitive type `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double` to a value of corresponding reference type `Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float`, `Double`. For a value v of primitive type p conversion to a

reference r of corresponding reference type R is performed such that r.p`Value() == `v. Conversion can be performed by calling the static function R`.valueOf(`v`)`.

Unboxing conversions convert from a reference type `Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float`, `Double` to primitive type `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`. A reference r of one of the reference types R can be converted to a value of its corresponding primitive type p by r`.`p`Value()`.

Unchecked and capture conversion are related to generics and will not be discussed here.

String conversion converts any type to type `String`. Primitive types can be converted to strings by static functions `valueOf` in `String`. Reference types can be converted to `String` by the method `toString`, that `Object` and so also every class derived from it possesses. String conversion applies, when one of the operands of the binary `+` operator is a `String` (string concatenation).

Value set conversion is related to FP-strict expressions. If FP-strict, values of type `float` (respective `double`), that are not an element of the float (double) value set, must always be mapped to the nearest element of the float (double) value set.

As can be seen, some patterns for conversion emerge. In fact, many other methods in the Java standard library follow these patterns. Constructors taking one argument can be also be seen as conversion functions (like conversion constructors in C++). Therefore, the following conversion functions will be considered in XL:

```
class C {
  C(S source);
  static C valueOf(S source);
  T toT_s();
  t tValue();
}
```

where C and T are reference types (class or interface), with T_s the *simple name*[3] of T, S is a primitive or reference type, and t a primitive type.

For already existing types, conversions can also be defined non-intrusively. Instead of an implicit `this` reference the `to`T_s method can be declared `static` and with an explicit reference to the conversion source, like in

```
class C {
  static T toT_s(S source);
}
```

and then may be statically imported into the current scope. In principle, the `valueOf` method could have been used for this purpose as well, as it contains the same signature. The intention for supporting both forms is to give better control about which conversions are visible in a certain scope. The `valueOf` methods must be declared in the reference type C to be applicable, while the static `to`T_s methods must be statically imported for applicability.

To keep compatibility to legacy code, applicability of conversion constructors must be restricted. While in C++ the `explicit` keyword disables a constructor for implicit

[3]For instance, if T is `java.lang.String`, then T_s is `String`. See also [GJSB05, section 6.2].

conversions, in XL the annotation `@ConversionConstructor` enables a constructor for such purpose. So only constructors that are explicitly marked with the annotation take part in implicit conversions. This prevents unexpected conversions from occurring.

The allowed conversions may also be controlled with an annotation `@UseConversions`. The enumeration `ConversionType` provides conversion types that can be enabled:

`VALUE_OF`	enables use of static `valueOf` methods
`TO_TYPE_IN_SCOPE`	enables `to`T_s methods
`CONVERSION_CONSTRUCTOR`	constructors marked with `@ConversionConstructor` are used for conversion
`CONSTRUCTOR`	all constructors are used for conversion (this supersedes `CONVERSION_CONSTRUCTOR`)

Initially, also implicit conversions over multiple conversion steps were considered. But this causes more problems than it solves. For instance, every object has a `toString` method and boxing classes like `Integer`, `Float`, etc. would thus allow any object to be converted to `int`, `float`, etc. For this reason, at most one autoconversion step is used, which also conforms to how implicit conversions are handled in C++.

In conclusion, *autoconversions* can be seen as a natural extension to auto(un)boxing.

4.3.1. Implementation

The XL compiler performs standard conversions as defined in [GJSB05, chapter 5] in the method `standardImplicitConversion` of class `CompilerBase`. To extend the compiler to support autoconversions, a method `implicitConversion` is used in appropriate conversion contexts (assignment, method invokation, etc.) instead of standard conversions. The method `implicitConversion` first checks if standard conversions apply by calling `standardImplicitConversion`. If this is not the case, conversion functions provided by the user are searched for.

The class `Scope` manages the enabled types of conversion functions, which can be queried by its method `isEnabledConversion`. If a candidate method has one of the valid patterns for autoconversions and its use is enabled, a function `checkCvCandidate` is called to perform additional checks (accessibility) and then enters it into a list of conversion functions.

Special treatment must be performed for `to`T_s and t`Value` conversion functions, as the target type is part of its name. The function `getToTypeMethodName` takes the target type of the conversion and generates a string that must match with the name of a conversion function in the current scope. Precisely, primitive types t generate the string t`Value`, array types T generate the string `to`T_c`Array`, where T_c is the array's component type, otherwise it is just `to`T_s with T_s the simple name of reference type T. Examples are:

```
int          → intValue
int[]        → toIntArray
int[][]      → toIntArrayArray
...
String       → toString
String[]     → toStringArray
```

```
String[][] → toStringArrayArray
...
```

Having obtained a list of candidate functions for conversion, `implicitConversion` checks if there is a best match. If there is no such match or more than one, the compiler generates an error. Otherwise, a call to the conversion function is generated.

4.4. Applications

The combination of operator overloading in combination with user-defined implicit conversion functions allows some interesting applications. Examples thereof will be presented in the following sections.

4.4.1. Vector and Matrix Computations

Probably the standard example for using operator overloading are vector and matrix calculations. GroIMP contains classes for this purpose in the package `javax.vecmath`, which is a modified version[4] of the original vecmath package[5]. The package contains classes `Vector{2,3,4}{d,f}` for vectors and `Point{2,3,4}{d,f}` for points, both of them derived from classes `Tuple{2,3,4}{d,f}`, classes for matrices, quaternions, rotations and colors.

To facilitate working with these classes, some operator overloads are provided in the class `VecmathOperators`. In addition, the class `Library` (containing many helper functions for working with RGGs) provides conversion functions to transform a node into a `Point3d` of its location. Using these functions, for three nodes `na`, `nb` and `nc` the user can simply write:

```
Point3d a = na;
Point3d b = nb;
Vector3d v = b - a;
Point3d c = nc + 3 * v;
```

The operators are defined as static functions and are statically imported into the current scope. This way, no modification of the vector classes was necessary. A complete example making use of these operators is the *Boids* model in GroIMP, which is based on [Rey87].

4.4.2. Arbitrary-Precision Arithmetic

The Java standard library provides in the package `java.math` classes *BigInteger* and *BigDecimal* for arbitrary-precision arithmetic. Usage of these classes is performed by calling methods on such objects to perform arithmetic operations, for instance a function `add` to calculate the sum, which is cumbersome. Furthermore, more complex arithmetic expressions must be written in prefix-form, which is hard to read and so prone to errors.

[4] `http://objectclub.jp/download/vecmath_e`, accessed 8. November 2011
[5] `http://java.net/projects/vecmath`, accessed 8. November 2011

By making use of operator overloading and autoconversions those data types can be made to work like any of the built-in types. For instance, one could provide a conversion function to transform an `int` to a `BigInteger`:

```
public static BigInteger toBigInteger(int i)
{
  return new BigInteger(i);
}
```

Then a `BigInteger` can be initialized from an integer literal, like in:

```
BigInteger a = 3;
```

The operations on `BigInteger` can also be overloaded. For instance, an overload for the function `add` can be provided as:

```
public static BigInteger operator+ (BigInteger a, BigInteger b)
{
  return a.add(b);
}
```

Then it is possible to write:

```
println(2 + a);
```

A nice feature of how XL implements operator overloading is that no modification of the original classes is necessary to retrofit them with operators. This wouldn't even be possible in this case, as those classes belong to the Java standard library.

4.4.3. Stream IO

Programming input/output operations is a common task in a programmers life. Most common is to print something to the screen, like in `println("Hello world!")`. In Java, basic input and output is performed by classes defined in the package `java.io`, and the class `System` provides a field `out` of type `PrintStream` to perform formatted output to the console.

Different functions in `PrintStream` can print data of different types, but to write something as simple as a number with a label, either multiple calls have to be made, like in

```
System.out.print("i = ");
System.out.println(i);
```

or label and number have to be concatenated to a string as in

```
System.out.println("i = " + i);
```

To fix this, `printf`-style functions that take a variable number of arguments have been introduced. Using these functions, the example can be written as:

```
System.out.printf("i = %d", i);
```

Unlike the counterpart in C, the `printf`-functions in Java provide type safety. However, interpreting the format string and checking argument types decreases performance, as does boxing of primitive values (like `i` above). A better approach to formatted output can be found in the `iostream` library of C++. Input and output of data can be performed by calling overloaded shift operators `>>` and `<<`. So to print the value of a variable `i` with a label, like in the examples above, in C++ one would write:

```
std::cout << "i = " << i << "\n";
```

The shift operators are assumed to expect the stream as first parameter and return it as result of the shift expression to allow invocation chaining. In comparison with `printf`-functions, the operator approach is more flexible, as formatted output of user-defined types can be added later on (by providing appropriate overloads for the operators).

In XL, the stream operator can be implemented by functions like this:

```
static PrintWriter operator<< (PrintWriter p, int i) {
  p.print(i);
  return p;
}
static PrintWriter operator<< (PrintWriter p, String s) {
  p.print(s);
  return p;
}
```

Then formatted output like in the examples above can be performed in GroIMP/XL by:

```
out << "i = " << i << "\n";
```

Also, fine-tuning of the output (number of digits, time and date format, etc.) can be controlled by inserting manipulators into the stream, instead of putting hard-to-remember flags into the format string. For instance, the C++ `iostreams` library provides standard manipulators like `endl` (insert end of line and flush output) or `hex` (all subsequent integral values will be written in hexadecimal format).

In C++, manipulators are implemented as functions and stream operators are provided to insert the manipulators into the stream. In XL, it is not possible to directly insert a function call into the stream, instead a functor must be used. Stream manipulators must implement an interface `Manipulator`:

```
interface Manipulator
{
  void apply(PrintWriter p);
}
```

An implementation of the `endl` manipulator then might look like this:

```
final static Manipulator endl = new Manipulator()
{
  public void apply(PrintWriter p)
  {
    p.print('\n');
```

```
    p.flush();
  }
};
```

To insert manipulators into the stream, an appropriate overload of the shift operator must be provided:

```
static PrintWriter operator<< (PrintWriter p, Manipulator m)
{
  m.apply(p);
  return p;
}
```

The manipulator can then be used like in this example:

```
out << "i = " << i << endl;
```

Some manipulators, like `hex`, might need to store an additional state. This can be done by working on an extended stream class with fields to store such a state, and letting the manipulator check if the stream object is really of this type (if not, the state change might be silently ignored) and then setting the state appropriately. Other stream operators, for instance those that output numbers, refer to the state and behave accordingly.

4.4.4. Expression Templates and Parsing of Chemical Reactions in XL

In C++, expression templates [Vel95, Vel98] provide a technique that allows to drastically improve the performance of applications. Consider a library that provides overloaded operators for vector calculations and the following simple computation:

```
DoubleVec w(1000), x(1000), y(1000), z(1000);
w = x + y * z;
```

The type `DoubleVec` represents a variable-length vector of `double` values and operators `+` and `*` are assumed to perform component-wise addition respectively multiplication. With traditional use of operator overloading, this code produces for `y * z` an intermediate vector `t`, which is then added to `x` to obtain the final result.This is cache-unfriendly and degrades performance.

Using expression templates, Veldhuizen reports to obtain more than 95% efficiency compared to a hand-coded C version. The trick is, not to evalute the operator in-place, but to return some object that knows how to perform the computation. The assignment then executes a loop over all target indices, and evaluates the vector line by line. This is equivalent to the following code:

```
for (int i = 0; i < 1000; i++)
  w[i] = x[i] + y[i] * z[i];
```

The technique relies heavily on the compiler to perform *partial evaluation* of expressions, and the ability of the compiler to construct syntax trees for expressions by matching types at compilation time of the program. It has been reported that this technique can also be applied to Java programs [Vel00].

Although XL does not support templates, some of the ideas from expression templates can be transferred to XL. The aim is to be able to parse chemical reactions like

$$2\,H_2 + O_2 \underset{k_b}{\overset{k_f}{\rightleftharpoons}} 2\,H_2O$$

and integrate them numerically. We make use of overloaded operators and autoconversions to let the compiler deduce the structure of the chemical reaction equation.

We start by defining a class that represents individual species that participate in a reaction:

```
class Molecule
{
  public static final Molecule H2  = new Molecule("H2");
  public static final Molecule H2O = new Molecule("H2O");
  public static final Molecule O2  = new Molecule("O2");

  String name;

  public Molecule(String name) {
    this.name = name;
  }

  public String toString() {
    return name;
  }
}
```

Each `Molecule` has a `name`. Some predefined species types have been defined (`H2`, `O2`, `H2O`), that can be statically imported later on into the scope.

To enter the chemical reaction, we want to use overloaded operators to capture the structure of the reaction, and then store this into a variable. So the resulting code in the application will look like this:

```
ChemicalReaction r = 2*H2 + O2 <=> 2*H2O;
```

The addition operator `+` must be overloaded, so that reactants can be "summed", and the comparison operator `<=>` is overloaded to serve as reaction arrow, indicating a bidirectional reaction is possible. Optionally, the operators `-->` and `<--` could be overloaded to indicate unidirectional reactions. The multiplication operator `*` allows to provide the *stoichiometric coefficient* of the species. The produced `ChemicalReaction` collects all information about the structure of the reaction, and can be called later on to evalute reaction speeds for given concentrations of the reactants. A parse tree of the oxyhydrogen reaction is shown in figure 4.1.

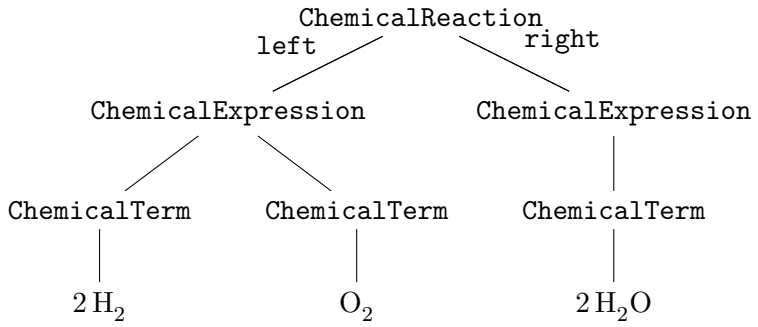


Figure 4.1.: Parse tree generated for chemical reaction $2\,H_2 + O_2 \underset{k_b}{\overset{k_f}{\rightleftharpoons}} 2\,H_2O$.

The mentioned operators are implemented in a class `ChemicalOperators`, together with some conversion functions. The overload of the reaction arrow is as follows:

```
public static ChemicalReaction operator<=> (
  ChemicalExpression lhs, ChemicalExpression rhs)
{
  ChemicalReaction result = new ChemicalReaction ();
  result.left = lhs;
  result.right = rhs;
  return result;
}
```

At compilation time, the compiler tries to match the types of the left and right hand side of operator `<=>` to the type `ChemicalExpression`, defined as:

```
class ChemicalExpression
{
  final ArrayList<ChemicalTerm> terms =
    new ArrayList<ChemicalTerm>();

  public void add (ChemicalTerm term)
  {
    terms.add (term);
  }
}
```

As can be seen, a `ChemicalExpression` just stores a list of `ChemicalTerms`, and provides a function to add new terms to the list. This is used by an overloaded operator `+` to form an expression from a sequence of terms:

```
public static ChemicalExpression operator+ (
  ChemicalExpression lhs, ChemicalTerm rhs)
{
  lhs.add (rhs);
  return lhs;
}
```

A `ChemicalTerm` stores the used `Molecule` together with its stoichiometric coefficient. Its definition is:

```
class ChemicalTerm
{
  double factor;
  Molecule m;

  public ChemicalTerm (Molecule m)
  {
    this.factor = 1;
    this.m = m;
  }

  public ChemicalTerm (double factor, Molecule m)
  {
    this.factor = factor;
    this.m = m;
  }
}
```

Finally, the operator `*` is overloaded to assign stoichiometric coefficients to a term:

```
public static ChemicalTerm operator* (
  double factor, ChemicalTerm term)
{
  term.factor *= factor;
  return term;
}
```

Some conversion functions are needed to handle special cases, like when the chemical expression consists of a single term. The following conversions are needed:

1) `Molecule` → `ChemicalTerm`
2) `Molecule` → `ChemicalExpression`
3) `ChemicalTerm` → `ChemicalExpression`

The first conversion is needed when a molecule is to be combined with a stoichiometric coefficient, so that the operator `*` can be called, and also to provide subsequent terms of a chemical expression if they are molecules without stoichiometric coefficient. In the example, the former condition applies to $2\,H_2$, while latter condition applies to O_2.

The second conversion helps if a chemical expression consists solely of a molecule, without any stoichiometric coefficient, or to provide the first term in the expression. In the example this would apply if the coefficient 2 was missing in front of H_2.

The third conversion allows an expression to consist of a single molecule with a stoichiometric coefficient. In the example, this applies to $2\,H_2O$.

The interesting result is that by adding overloading and autoconversions to the XL language, the XL compiler is able to analyze the structure of the chemical reaction and produces an error message at compilation time if its syntax was violated, provided the operators were overloaded as shown above.

An extension of this approach could also not just check the syntax of the reaction equation, but also its semantics. For instance, one could provide a class `Atom`, that captures properties of atoms, and make `Molecule` understand from which and how many atoms it is made of. Then the balance of atoms on the left and right side of the reaction could be ensured. In C++, it would be possible to perform this check at compilation time with the help of templates, but as XL does not support templates this must be done at runtime (when the reaction is constructed).

Not yet discussed was the structure of the `ChemicalReaction`. As this is used to numerically integrate the process, a differential equation must be obtained from the reaction formula. For *elementary reactions*, the *law of mass action* [WG64][6] allows to derive an equation. For instance, the dynamics of a reaction

$$\mathrm{aA} + \mathrm{bB} \underset{k_b}{\overset{k_f}{\rightleftharpoons}} \mathrm{cC} + \mathrm{dD}$$

with chemical species A, B, C, D with associated stoichiometric coefficients a, b, c, d can be described by differential equations

$$-\frac{1}{a}\frac{d[\mathrm{A}]}{dt} = -\frac{1}{b}\frac{d[\mathrm{B}]}{dt} = \frac{1}{c}\frac{d[\mathrm{C}]}{dt} = \frac{1}{d}\frac{d[\mathrm{D}]}{dt} = k_f[\mathrm{A}]^a[\mathrm{B}]^b - k_b[\mathrm{C}]^c[\mathrm{D}]^d,$$

where the *reaction rate* $v_f = k_f[\mathrm{A}]^a[\mathrm{B}]^b$ of the *forward reaction* depends on the concentrations of A and B and a *rate constant* k_f, and likewise for the *backward reaction* on concentrations of C and D, and a constant k_b. The coefficients a, b, c, d indicate the *order* of the chemical species, and their sum $a + b$ respectively $c + d$ indicates the order of the forward respectively backward reaction.

Note that elementary reactions with more than two molecules involved are very rare (the molecules must be at the same location to interact), and that the rate constants are temperature dependent (as described by the *Arrhenius equation* [Moo86, section 9.28]).

As the result of parsing the chemical formula an instance of `ChemicalReaction` is obtained, and additional parameters like k_f and k_b can be set on this object. To permit the numerical integrator evaluation of the rate equation, the class contains a function `eval` that calculates the rates for each reactant and accumulates them in a rate vector.

Usually more than one reaction is simulated at the same time, especially considering that many observed reactions consist of a network of elementary reactions. Therefore, a class `Model` is used to manage a collection of chemical reactions. In addition, other types of time dependent behaviour could be considered, like diffusion processes. By providing overloaded operators analogously to those for chemical reactions shown above, these diffusion processes could use their own syntax.

If multiple compartments are to be simulated, like in a cellular filament or tissue, multiple instances of the same `Molecule` have to be created. Each of them then represents the concentration of a species in a certain compartment. To support this, the class `Model` contains a method `assignIndices` that prior to integration automatically indexes all used `Molecule`s starting with zero. The indices refer to an entry in a array of `double`, the state vector, which stores the concentrations of the molecules.

[6] An English translation can be found in [Abr86] and a German translation in [Abe99].

4.4.5. Production Statements in XL

After the introduction of operator overloading in XL, parsing of production statements was rewritten to work in a similar way like the stream output facility. The parser then maps the syntax of productions to calls of operator functions. The following example

```
x [-e-> Y] < Z, x W
```

was given in [Kni08], and results in a sequence of calls

```
tmp1 = producer.producer$begin().operator$space(x);
tmp2 = tmp1.producer$push().producer$begin()
  .operator$arrow(new Y(), e).producer$end();
tmp1.producer$pop(tmp2).operator$lt(new Z()).producer$separate()
  .operator$space(x).operator$space(new W()).producer$end();
```

Besides `operator` functions there exist also `producer` functions, for instance the push-operator `[` is mapped to the function `producer$push`. The return value of these functions is another producer, so that invocation chaining is possible.

Usually, the same producer on which the function was called is returned. But by returning some other type of producer it is possible to activate another set of parsing rules. This is used when switching from the RGG producer (class `RGGProducer`) to vertex-vertex algebra rules (class `VVProducer`) [Kni08, section 10.6], and also within the vv system to only allow certain sequences of operations. The latter works by providing a producer with only the allowed set of operators defined. If the user specified some forbidden sequence of operations, this can be detected already at compilation time and the compiler can issue an error.

4.5. Conclusion

Operator overloading and user-defined implicit conversion functions are features known from many programming languages, among them C++. By introducing these features into the XL language, mathematical computations can be written in a user-friendly way, thus becoming more clear and less prone to errors.

Advanced usage includes parsing of chemical expressions. By providing appropriate operator overloads and conversion functions, the syntax of the language can be extended by the user to understand such chemical formulae. Also errors made in the specification of the formula can be detected at compilation time.

The possibility to also explain production rules for relational growth grammars and vertex-vertex algebras demonstrates that operator overloading is general purpose and thus very flexible.

Operator	Description	Name suffix
`` `a` ``	Quote	`quote`
`a[b]`	Array/Property access	`index`
`a(b)`	Invocation	`invoke`
`a[:]`	Array generator	`generator`
`a -> b`	Arrow	`arrow`
`a <- b`	Leaf arrow	`leafArrow`
`a++`	Postfix increment	`postInc`
`a--`	Postfix decrement	`postDec`
`a ** b`	Exponentiation	`pow`
`++a`	Prefix increment	`inc`
`--a`	Prefix decrement	`dec`
`+a`	Unary plus	`pos`
`-a`	Negation	`neg`
`~a`	Bitwise complement	`com`
`!a`	Logical complement	`not`
`a * b`	Multiplication	`mul`
`a / b`	Division	`div`
`a % b`	Remainder	`rem`
`a + b`	Addition	`add`
`a - b`	Subtraction	`sub`
`a << b`	Shift left	`shl`
`a >> b`	Shift right	`shr`
`a >>> b`	Unsigned shift right	`ushr`
`a instanceof T`	Type comparison	
`a < b`	Less than	`lt`
`a > b`	Greater than	`gt`
`a <= b`	Less than or equal	`le`
`a >= b`	Greater than or equal	`ge`
`a <=> b`	Comparison	`cmp`
`a in b`	Containment	`in`
`a <-> b`	Left-right arrow	`leftRightArrow`
`a --> b`	Long arrow	`longArrow`
`a <-- b`	Long left arrow	`longLeftArrow`
`a <--> b`	Long left-right arrow	`longLeftRightArrow`
`a --- b`	Line	`line`
`a +> b`	Plus arrow	`plusArrow`
`a <+ b`	Plus left arrow	`plusLeftArrow`
`a <+> b`	Plus left-right arrow	`plusLeftRightArrow`
`a -+- b`	Plus line	`plusLine`
`a /> b`	Slash arrow	`slashArrow`
`a </ b`	Slash left arrow	`slashLeftArrow`
`a </> b`	Slash left-right arrow	`slashLeftRightArrow`
`a -/- b`	Slash line	`slashLine`
`a == b`	Equality	`eq`
`a != b`	Inequality	`neq`
`a & b`	Bitwise and	`and`
`a ^ b`	Bitwise exclusive or	`xor`
`a | b`	Bitwise inclusive or	`or`
`a && b`	Logical and	`cand`
`a || b`	Logical or	`cor`
`a :: b`	Guard	`guard`
`a ? b : c`	Conditional	
`a : b`	Range	`range`
`a = b`	Assignment	
`a := b`	Deferred assignment	`defAssign`
`a` *op*`= b`	Compound assignment	*s*`Assign`
`a :`*op*`= b`	Compound deferred assignment	`def`*S*`Assign`
op ∈ {`**`, `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `>>>`, `&`, `^`, `|`}		*s* ∈ {`pow`, `mul`, `div`, ...} *S* ∈ {`Pow`, `Mul`, `Div`, ...}

Table 4.1.: XL operators, sorted by precedence.

5. Ordinary Differential Equations on Graphs

In this chapter we investigate how to formulate differential equations on graphs or networks and how to solve them with the help of XL programs. Core of this chapter is the introduction of a new operator into the language XL.

We will start with some examples that we want to integrate numerically. Then we show how integration was performed previously in GroIMP/XL using Euler integration. In a next step, we will apply numerical standard methods (like Runge-Kutta) to solve the same problems, we show how the examples need to be modified accordingly, and what difficulties arise when doing this.

As solution to automatically handle these problems, a new operator is introduced into the language XL, which provides the needed information. The examples will then be rewritten in terms of this operator to show how things simplify.

There will also be a short evaluation of some numerical libraries, and reasoning for inclusion of one of these into GroIMP will be described.

Finally, the considerations made during implementation of the system will be presented.

5.1. Introduction

There are many simulation problems that need numerical integration. We will start out by investigating which type of problems might be interesting to solve.

Already discussed in the last chapter in section 4.4.4 was the solution of chemical kinetics. It turns out that the approach using operator overloading, despite being interesting, is not sufficiently flexible to formulate all possible kinds of reactions that might occur. Also, for the average user it might be not straightforward to formulate his problem this way. Therefore, some issues that might arise when formulating chemical reaction networks will be discussed.

Many simulation models also combine processes with a geometrical structure. A common type are transport processes, as they are often used in functional-structural plant models (FSPMs). So this will be discussed as well.

5.1.1. Chemical Kinetics

In biological organisms *chemical reactions* play an important role. *Chemical kinetics* investigates what influences the speed of a chemical reaction, as well as its *reaction mechanism*. For reference see [Moo86].

$2\,N_2O_5 \longrightarrow 4\,NO_2 + O_2$	$-\frac{d[N_2O_5]}{dt} = k_1[N_2O_5]$
$2\,NO_2 \longrightarrow 2\,NO + O_2$	$-\frac{d[NO_2]}{dt} = k_2[NO_2]^2$
$(C_2H_5)_3N + C_2H_5Br \longrightarrow (C_2H_5)_4N^+Br^-$	$-\frac{d[C_2H_5Br]}{dt} = k_2[C_2H_5Br][(C_2H_5)_3N]$
$CH_3CHO \longrightarrow CH_4 + CO$	$-\frac{d[CH_3CHO]}{dt} = k_{1.5}[CH_3CHO]^{3/2}$

Table 5.1.: Examples of chemical reactions and their rate laws (from [Moo86]).

For a general equation

$$\nu_1 A + \nu_2 B + \ldots \longrightarrow \nu_1' A' + \nu_2' B' + \ldots$$

one obtains the *reaction rate*

$$r = -\frac{1}{\nu_1}\frac{d[A]}{dt} = -\frac{1}{\nu_2}\frac{d[B]}{dt} = \frac{1}{\nu_1'}\frac{d[A']}{dt} = \frac{1}{\nu_2'}\frac{d[B']}{dt} = \ldots$$

where ν_1, ν_2, ν_1', ν_2', ... are the *stoichiometric coefficients* in the equation.

A *rate law* describes how the reaction rate depends on concentrations of the involved species, and perhaps some other factors. They often are found to be of the form

$$r = k[A]^a[B]^b \ldots,$$

where k is the *rate constant* or *rate coefficient*. The exponents a, b, ... are the *order* with respect to A, B, ..., and their sum is the *reaction order*.

Table 5.1 shows some examples of reactions together with their rate laws. As can be seen, reactions do not necessarily have to be of an integer order. It was found that for *elementary reactions* the order of the involved species corresponds to its stoichiometric coefficient. However, most observed reactions follow a *reaction mechanism* consisting of a sequence of *elementary steps* that cannot be split up further, and their rate law cannot be determined directly from the stoichiometric equation. Also, for reactions taking place in a dimensionally-restricted environment such as biological cells, the traditional *law of mass action* is not valid anymore and *fractional kinetics* [Sav98] have to be considered.

Often the temperature has a strong influence on the reaction rate. The *Arrhenius equation*

$$k(T) = A \cdot e^{\frac{-E_a}{RT}}$$

with A the pre-exponential factor, E_a the activation energy, R the universal gas constant and T the temperature approximates this dependency of the rate coefficient.

In biology, many reactions are catalyzed by enzymes. *Michaelis-Menten kinetics* [MM13] is used to describe such processes, which are constituted by formation of an enzyme-substrate complex followed by a reaction to the final products:

$$E + S \underset{k_2}{\overset{k_1}{\rightleftharpoons}} ES \xrightarrow{k_3} E + P$$

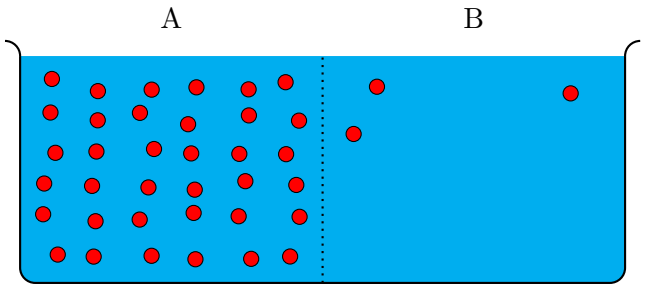


Figure 5.1.: Diffusion through membrane in aqueous solution.

For the observed reaction S $\longrightarrow$ P the corresponding rate law is

$$-\frac{d[\mathrm{S}]}{dt} = \frac{d[\mathrm{P}]}{dt} = v_{max}\frac{[\mathrm{S}]}{K_m + [\mathrm{S}]}.$$

A very high substrate concentration [S] results in saturation with a maximal reaction rate $v_{max} = k_3[\mathrm{E}]_0$, where $[\mathrm{E}]_0 = [\mathrm{E}] + [\mathrm{ES}]$ is the total enzyme concentration. The *Michaelis constant* $K_m = \frac{k_2+k_3}{k_1}$ is the substrate constant for which the reaction rate is half of the maximal rate v_{max}.

In conclusion, the rate law may not be directly derived from the stoichiometric equation. So being able to enter those equations (as was presented in section 4.4.4) is only of partial use if the reaction mechanisms are unknown. Also, additional effects (like dependency on temperature, pH, water, etc.) may need to be described. Therefore, the user should be able to easily specify (ordinary) differential equations.

5.1.2. Transport Processes

Simulation of transport processes plays an important role in many models, especially biological ones. An example of this is diffusion of a substance through a membrane, as is shown in figure 5.1. A difference between concentrations c_A and c_B causes a flux from higher to lower concentration according to the law:

$$-\frac{dc_A}{dt} = \frac{dc_B}{dt} = P \cdot A \cdot (c_A - c_B), \tag{5.1}$$

where P is the *permeability* of the membrane and A its area.

A more general formulation is

$$\frac{\partial c}{\partial t} = D\frac{\partial^2 c}{\partial x^2},$$

which is known as *Fick's second law* [Fic55], or when applied to heat conduction as *Fourier's second law* (see equation 2.16 on page 40).

Reaction-diffusion systems [Tur52] combine diffusion of chemical substances with reactions between them. They are described by the PDE

$$u_t = D \cdot \Delta u + f(u),$$

where $u(x,t)$ is a vector-valued function representing the concentrations of the chemical substances at location x for time t, u_t its time-derivative, D a diagonal matrix with diffusion coefficients, Δu the Laplacian of u, and $f : \mathbb{R}^n \to \mathbb{R}^n$ an arbitrary function accounting for the chemical reactions. The interesting aspect about these systems is that, although the original state may be quite homogeneous, they may later on develop a pattern or structure due to small random disturbances. *Activator-inhibitor systems* [You84] may be seen as a special case of reaction-diffusion systems.

Instead of on a grid, transport processes may also take place on graph structures or networks. An example are real trees, where the *phloem* transports carbon assimilates downward from the leaves, and the *xylem* transports water and nutrients upward from the roots. Water transport in trees has been investigated in [Frü95, FK99].

5.2. Examples for Integration on Graphs

Below we will present some examples that define processes on networks respectively graphs. Both examples are artificial in the sense that they do not claim to simulate any problem from the real world. Nevertheless, they are representative for real world problems, like reaction networks or transport of carbon and nitrogen in a plant. The examples have been described previously in [HSK10].

5.2.1. Circular Transport with Inhibition

Consider the following (artificial) example of transport with inhibition, as is shown in figure 5.2. A substrate at $S_{i\oplus 1}$ is transported to $S_{i\oplus 2}$, if the concentration of the substrate at S_i is below a certain threshold. Here, the operator $\oplus$ indicates addition in the ring, so $i \oplus j = mod(i+j, n)$ with n the number of nodes in the ring (in the example $n = 5$).

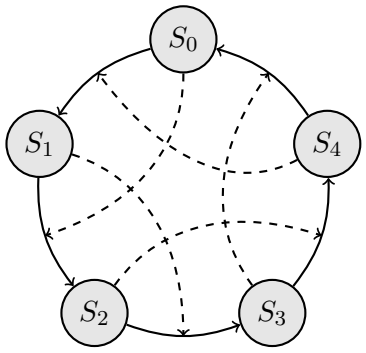


Figure 5.2.: Circular transport with inhibition (schematic).

The process is governed by the differential equations

$$\frac{d[S_{i\oplus 2}]}{dt} = -\frac{d[S_{i\oplus 1}]}{dt} = \begin{cases} \mu \cdot [S_{i\oplus 1}] & \text{if } [S_i] \leq T \\ 0 & \text{otherwise.} \end{cases}$$

In XL, this can be modelled by introducing a module for the substrate nodes S_i:

```
module S(double c);
```

The solid edges in figure 5.2 can be represented in the GroIMP graph by user-defined edges, such that there is an `EDGE_0` from S_i to $S_{i\oplus 1}$ for each $i \in \{0, 1, 2, 3, 4\}$. Simulation of transport can then be done in XL by the following rule:

```
x:S -EDGE_0-> y:S -EDGE_0-> z:S ::> {
  double rate = x[c] > T ? 0 : μ * y[c];
  y[c] :-= rate;
  z[c] :+= rate;
}
```

Starting with an initial substrate concentration of 1 at S_0, transport of a fraction μ of the substrate to S_1 will be performed. There will be no transport from S_1 to S_2 until the concentration at S_0 drops below the threshold value `T`. The effect is, that almost all of the substrate will be transported from S_0 to S_1, before it is transported further from S_1 to S_2. The resulting sequence of development of the system is shown in figure 5.3.

5.2.2. Transport in a Tree

As a more complicated example, consider transport of carbon assimilates through a tree. Transport is assumed to be caused by diffusion, which might not correspond to reality. But the purpose of the model is not to explain real trees, rather it should serve as an example that shows what complications arise if differential equations were to be solved on a dynamical structure.

For each connected pair A and B of nodes the differential equations

$$-\frac{d[A_c]}{dt} = \frac{d[B_c]}{dt} = D \cdot ([A_c] - [B_c])$$

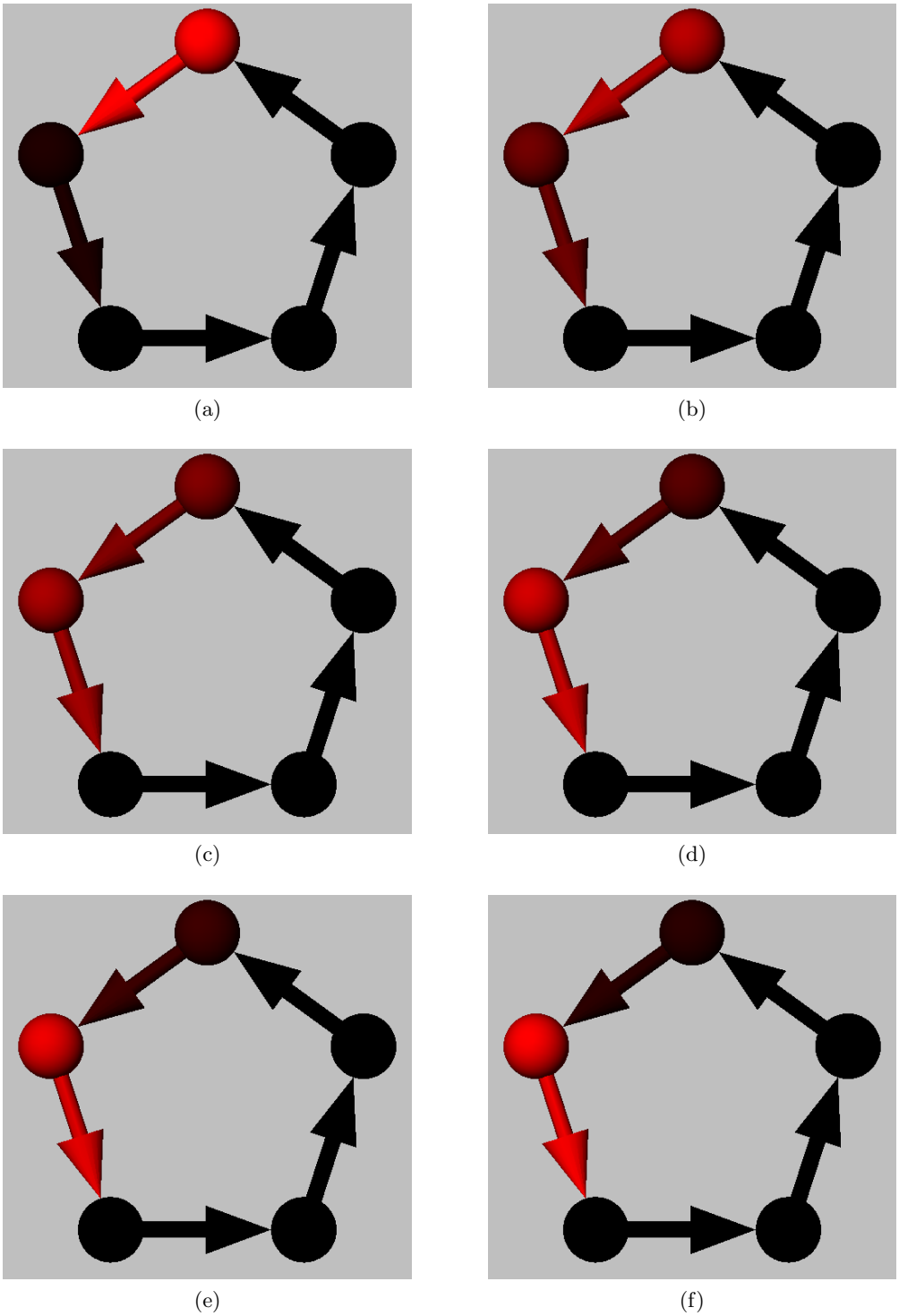
describe exchange of carbon assimilates between these nodes.

Besides transport, also production and consumption of the substrates is simulated. Production occurs in leaves, represented by green spheres. Consumption is performed in the branches, represented by cylinders, and causes growth of them. This is shown in figure 5.4, where the concentration of carbon assimilates is visualized by the radius of the corresponding objects.

Differential equations describing production and consumption of carbon and growth in length for each node A are

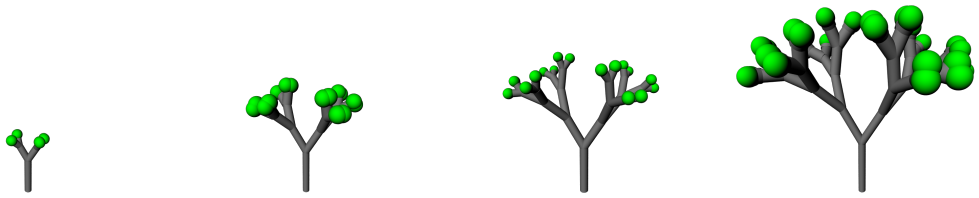
$$\frac{d[A_c]}{dt} = P_A - C_A \cdot [A_c]$$
$$\frac{d[A_l]}{dt} = \gamma \cdot C_A \cdot [A_c].$$

These apply simultaneously to the diffusion equations. Here, for each node A, P_A is a production coefficient ($P_A = 0$ if A is a branch) and C_A is a consumption coefficient ($C_A = 0$ if A is a leaf).



(a) (b) (c) (d) (e) (f)

Figure 5.3.: Circular transport with inhibition (result).



(a) (b) (c) (d)

Figure 5.4.: Growth of artificial tree by means of simulated carbon assimilation.

The rules in XL to model these processes are shown in listing 5.1. Type `A` represents a leaf and type `B` a branch, and both are derived from type `C`. The variable `h` provides the integration step size, γ is a conversion factor from consumed carbon to length increment, and `D` is the diffusivity. `PROD` and `CONS` are fixed coefficients for production and consumption rate computation. The rather complicated looking syntax for the pattern in the diffusion rule ensures, that only pairs of nodes of type `C` (or types derived thereof) are considered, for which there is a path from the first node `ca` to the second node `cb` without any other node of type `C` in between. This is necessary to skip rotation nodes.

Listing 5.1: XL rules that model processes for carbon assimilates in a tree.

```
1  // apply production to nodes
2  a:A ::> a[carbon] :+= h * PROD;
3
4  // apply consumption and convert to growth
5  b:B ::> {
6    double rate = CONS * b[carbon];
7    b[carbon] :-= h * rate;
8    b[length] :+= h * γ * rate;
9  }
10
11 // perform diffusion between nodes
12 ca:C (-->)+ : (cb:C) ::> {
13   double rate = D * (ca[carbon] - cb[carbon]);
14   ca[carbon] :-= h * rate;
15   cb[carbon] :+= h * rate;
16 }
```

The initial configuration of the tree is a single branch with a leaf on top. New branches are produced by a rule, that triggers when the carbon concentration in a leaf rises above a threshold `T`. The leaf is then replaced by two branches, each of them with a leaf on top, and the carbon assimilates of the original leaf are distributed among the new leaves. The rule is shown in listing 5.2.

Listing 5.2: Branching rule that triggers when assimilate concentration rises above a threshold `T`.

```
1 a:A, (a[carbon] > T) ==>
2   [RU( 30) RH(75) B(0) A(a[carbon]/2)]
3   [RU(-30) RH(75) B(0) A(a[carbon]/2)]
4 ;
```

5.3. Use of Advanced Numerical Methods

A closer inspection of the examples from the previous section reveals that Euler integration is used. Numerical issues with this method are known from literature and have also been discussed in chapter 2. Therefore, one of the advanced methods should be used instead.

To do this, the integration problem must be expressed as initial value problem. Repeating its definition from section 2.2 on page 5 we have to formulate and solve

$$u'(t) = f(t, u(t)), \quad u(t_0) = u_0,$$

where $f(t, u)$ is determined by the problem being integrated together with initial condition u_0 for some time t_0. The result is an approximation to the scalar or vector-valued function $u(t)$.

In an implementation of a numerical method, rate and state vectors are usually represented as an array of `double`, and the *rate function* $f(t, u(t))$ is provided in form of a method with the following signature:

```
void getRate(double[] rate, double time, double[] state);
```

The parameters `time` and `state` are input parameters, whereas `rate` is an output parameter providing just the memory to store the result. This prevents memory allocations each time the rate function is called.

5.3.1. Circular Transport with Inhibition

So to numerically integrate the example of circular transport with a method other than Euler, we need to provide such a rate function. Due to the specific structure of this problem, this can be expressed rather easily. Each concentration in a substrate node becomes an element of the state vector. For each triple of nodes, the corresponding indices into the state vector can be calculated with modulus arithmetic. The resulting function is shown in listing 5.3.

5.3.2. Transport in a Tree

A bit more elaborate is the simulation of transport in a tree structure. As this structure is dynamic, no fixed mapping between the attributes of nodes in the graph to elements of

Listing 5.3: Rate function for circular transport with inhibition.

```
1  void getRate(double[] rate, double time, double[] state) {
2    // zero output array
3    java.util.Arrays.fill(rate, 0);
4    // calculate transport rate
5    for (int i = 0; i < rate.length; i++) {
6      int x = i;
7      int y = (i + 1) % rate.length;
8      int z = (i + 2) % rate.length;
9      double r = state[x] > T ? 0 : μ * state[y];
10     rate[y] -= r;
11     rate[z] += r;
12   }
13 }
```

the rate/state vector can be used. Instead, this mapping has to be created dynamically when the integration is started (see figure 5.5).

The steps that must be performed are as follows:

1) Calculate length of state vector
2) Allocate state vector
3) Create mapping between attributes of nodes and entries in state vector
4) Copy state from graph to state vector
5) Perform integration
6) Copy state from state vector to graph

In this example, calculating the length of the state vector is rather simple. As there is just one type of node to consider (only type `C` stores data) it is sufficient to just count the number of instances of type `C`. Creation of the mapping can then be performed by enumerating the instances of type `C` starting from zero. The assigned number is then the index into the state vector.

If only the `carbon` attribute of every `C` node had to be integrated we were done. But carbon consumption results in branch growth, so the `length` attribute must be integrated as well. Here, we can just allocate two elements in the state vector per node by enumerating with an increment of two in step number three. The fact that this actually wastes a bit of memory and computation time, because the length growth only accounts for the branches, is outweighted by the fact that indexing becomes a lot simpler. In the more general setting when integration is performed on different types with a differing number of attributes, and these types are perhaps even organized in a class hierarchy, things become complicated and doing this manually is error-prone. Therefore, another solution will be developed later on.

In step number five, the integrator is called. This in turn calls back the rate function to obtain derivatives for different states at different times. One should be aware that the queried states do not necessarily lie on the solution curve. The steps that must be performed in the rate function are:

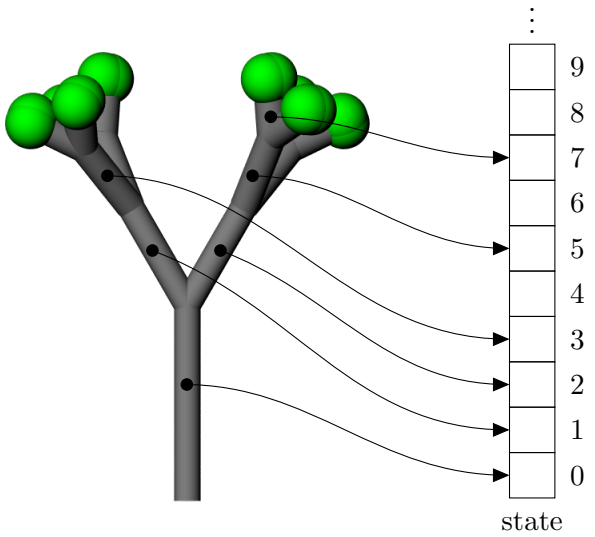


Figure 5.5.: Mapping from nodes in the graph to elements of the state vector.

1) Initialize rate vector with zero
2) Copy state from state vector to graph
3) Calculate rates using XL rules and accumulate them in the rate vector

Instead of copying the state to the graph each time the rate function is called, one might opt to look up the values directly from within the rate function. But this introduces another level of indirection for those values and they need to be read at least once anyways to participate in integration. So the suggested procedure is to just copy those values into the attributes of the nodes once. This way the rate calculations can make use of XL rules and therefore specification of the ODEs on a graph structure is simplified.

The resulting main function for the model is shown in listing 5.4. The factor two in line 2 when calculating the length of the state vector and the increment by two in line 11 is due to integrating two attributes (`carbon` and `length`) per node. For the same reason, the offsets (`+0` and `+1`) in lines 14,15 and 22,23 are used when copying between graph and state vector. The actual integration is initiated in line 18, where `DT` is a constant duration, `y0` the initial state, and `y` the final state.

The integrator calls back the rate function, which is given in listing 5.5. Again, the rate vector is initialized to zero. Then, the state vector is copied into the graph. This simplifies calculation of the rates in the third step (lines 11 to 26). One must be careful when writing to the rate vector to use the correct indexing and not to accidentially write to the attribute instead. An integration step size h does not appear anymore, as its choice is now up to the actual integration method.

Listing 5.4: Main function for diffusion in a tree using advanced integration methods.

```
1  // calculate length of state vector
2  final int N = 2 * (int) count((* C *));
3
4  // allocate state vector
5  final double[] y0 = new double[N];
6  final double[] y  = new double[N];
7
8  // create mapping between attributes of nodes
9  // and entries in state vector
10 int index = 0;
11 [ c:C ::> { c[index] = index; index += 2; } ]
12
13 // copy state from graph to y0
14 [ c:C ::> y0[c[index]+0] = c[carbon]; ]
15 [ c:C ::> y0[c[index]+1] = c[length]; ]
16
17 // integrate over time
18 integrate(time, y0, time + DT, y);
19 time += DT;
20
21 // copy state from y to graph
22 [ c:C ::> c[carbon] = y[c[index]+0]; ]
23 [ c:C ::> c[length] = y[c[index]+1]; ]
24
25 // split leaf if concentration over threshold
26 [
27   a:A, (a[carbon] > T) ==>
28     [RU(30) RH(75) B(0) A(a[carbon] / 2)]
29     [RU(-30) RH(75) B(0) A(a[carbon] / 2)]
30   ;
31 ]
```

5.4. Rate Assignment Operator

We have seen in the previous section 5.3.2 that even in a simple model the creation of the mapping between node attributes and the elements of the rate/state vector can become complex swiftly. In a more general setting, this can be even quite challenging, especially if class hierarchies are involved. Therefore, the system should assist the user by creating this mapping automatically.

Memory Allocation

Consider the class diagram shown in figure 5.6a. It is assumed that all attributes except `x` participate in integration. So for all of these attributes a location in the rate and state vector must be assigned and corresponding memory must be allocated. Let n_A, n_B, n_C, n_D designate the number of instances of the respective types `A`, `B`, `C`, and `D`. Let further $\nu_A = 1, \nu_B = 0, \nu_C = 2, \nu_D = 1$ indicate the number of attributes of these

Listing 5.5: Rate function for diffusion in a tree using advanced integration methods.

```
1  void getRate(double[] rate, double time, double[] state) {
2    // zero output array
3    java.util.Arrays.fill(rate, 0);
4
5    // copy state y to graph
6    [ c:C ::> c[carbon] = y[c[index]+0]; ]
7    [ c:C ::> c[length] = y[c[index]+1]; ]
8
9    // calculate rate vector
10   [
11     // apply production to A nodes
12     c:A ::> rate[c[index]] += PROD;
13
14     // apply consumption and convert to growth
15     c:B ::> {
16       double r = CONS * c[carbon];
17       rate[c[index]+0] -= r;
18       rate[c[index]+1] += γ * r;
19     }
20
21     // perform diffusion between nodes
22     ca:C (-->)+ : (cb:C) ::> {
23       double r = D * (ca[carbon] - cb[carbon]);
24       rate[ca[index]] -= r;
25       rate[cb[index]] += r;
26     }
27   ]
28 }
```

types. Then the total length n of the state vector can be computed as:

$$n = n_A \cdot \nu_A + n_B \cdot (\nu_A + \nu_B) + n_C \cdot (\nu_A + \nu_B + \nu_C) + n_D \cdot (\nu_A + \nu_D).$$

It follows that for each type the number of attributes it declares together with those inherited from its supertypes must be counted. With respect to integration, a *reduced* class hierarchy is obtained by omitting attributes and classes that do not participate in integration (see figure 5.6b).

An alternative way to compute n can be realized if $t_A = n_A + n_B + n_C + n_D, t_B = n_B + n_C, t_C = n_C, t_D = n_D$ denote the total number of instances that can be cast to a certain type. Then the size of the rate and state vector is evaluated as:

$$n = t_A \cdot \nu_A + t_B \cdot \nu_B + t_C \cdot \nu_C + t_D \cdot \nu_D. \tag{5.2}$$

This latter variant is beneficial as the values for ν_* can be precomputed. Also we will see later on that the values for t_* are directly provided by the GroIMP implementation at runtime. This makes calculation of the size of the rate and state vector straightforward.

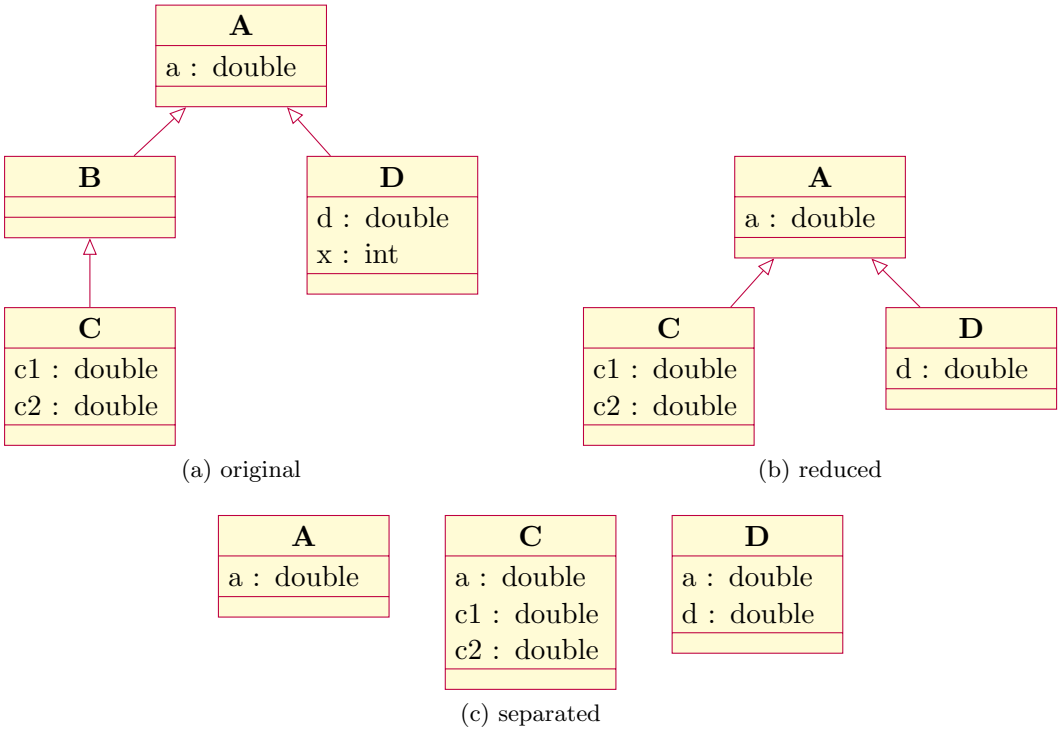


Figure 5.6.: Class diagram of module hierarchy used in an XL model.

Mapping

Once memory for rate and state vector has been allocated, for each attribute that participates in integration an index into these vectors must be assigned. A *separated* view on the class hierarchy is created (see figure 5.6c), where attributes have been merged for each class in the order they appear in the derivation chain. The attributes in each separated class can then be assigned offsets starting from zero, and the size of each class can be determined (in the example sizes are $\sigma_A = 1, \sigma_C = 3, \sigma_D = 2$). For each node, a base index is assigned, so that consecutive elements in rate and state vector belong to this node. For a node n with base index n_{base} and a property p of this node with offset p_{offset}, the index can be calculated as:

$$index = n_{base} + p_{offset}.$$

Rate Assignment Operator

What remains to do is to provide the user with a tool to indicate which attributes participate in integration and to assign rates to these attributes. Therefore, a new operator `:'=` was introduced into the language XL, called the *deferred rate assignment operator*. For a node `n` with a property `p` a rate `r` can be assigned by calling

```
n[p] :'= r;
```

The operator does not modify the attribute itself, but rather writes to a hidden rate vector. Multiple calls to this operator result in the rates being accumulated. There must be no step size h anymore, as managing this is up to the integrator.

When working in RGG mode (see [Kni08, appendix B.4]), the user benefits from the fact that the model class then implicitly extends a class `RGG`, which in turn is derived from `Node`. This allows to write rate assignments as

```
p :'= r;
```

for a (global) property `p` of the model and the current instance of `RGG` is used as node.

At compile time, rate assignments are written to a table as pairs of node type and property. At run time, these data are analyzed to calculate offsets for the attributes and sizes of the node types, as was described above. Then the system performs the steps that were described in section 5.3.2. These are now automatic and don't need any user assistance, as all necessary data is already available.

For the integrator to execute the rate assignments, the user is responsible to group them into a function `getRate`. The user must be aware that the proposed state does not necessarily belong to the solution curve, but is solely provided to compute rates. Enforcing to group all rate assignments into a single function helps the user to stay alert for this.

5.4.1. Circular Transport with Inhibition

The rule for circular transport from sections 5.2.1 and 5.3.1 can now be written as:

```
x:S -EDGE_0-> y:S -EDGE_0-> z:S ::> {
  double rate = x[c] > T ? 0 : μ * y[c];
  y[c] :'= -rate;
  z[c] :'= +rate;
}
```

It attains the simplicity of the Euler version, while it maintains support for advanced numerical methods. By using a rule, the differential equation can be directly applied to a graph.

5.4.2. Transport in a Tree

Similarly, we can reformulate the tree model in terms of rate assignments (see listing 5.6). Again, the simplicity of the Euler version from section 5.2.2 is attained (listing 5.1), while keeping the support for better integration methods.

One modification that is necessary is the introduction of a new property `len` for nodes of type `B`. This is due to the fact that the rate assignment operator is only defined on properties of type `double`, which reflects the observation that all investigated libraries for numerical integration work on `double` as well. However, the inherited attribute `length` of a cylinder is defined as single precision `float`.

Technically it would be possible to also define the rate assignment operator for attributes of type `float`. When passing values to the integrator, these would have to be converted implicitly to type `double`, and when such attributes are copied into the graph they would be converted back to `float`. However, this rounding to a lower precision might cause numerical issues and instability in integration, which may result in subtle

Listing 5.6: XL rules using rate assignments that model processes for carbon assimilates in a tree.

```
1  // apply production to nodes
2  a:A ::> a[carbon] :'= PROD;
3
4  // apply consumption and convert to growth
5  b:B ::> {
6    double rate = CONS * b[carbon];
7    b[carbon] :'= -rate;
8    b[len   ] :'= γ * rate;
9  }
10
11 // perform diffusion between nodes
12 ca:C (-->)+ : (cb:C) ::> {
13   double rate = D * (ca[carbon] - cb[carbon]);
14   ca[carbon] :'= -rate;
15   cb[carbon] :'= +rate;
16 }
```

bugs later one. So a design-decision was not to support rate assignments on single precision floats. This imposes no real restriction, as visualization is not performed in the rate function anyway.

5.5. Monitor Functions

The example from the previous section 5.4.2 is not complete yet. What is missing is a way to trigger the branching rule when the concentration in a node of type `A` rises above some threshold (listing 5.2). This cannot be done in the rate function, as this is only supposed to calculate rates, and the provided state might (and usually will) not lie on the solution curve. Instead, integration must be stopped, the modification of the structure then performed, and integration is restarted with the new structure.

If the time when the event occurs is not known *a priori*, some way is needed to tell the integrator when to stop. The solution lies in so called *monitor functions*, sometimes also called *trigger functions* or *switching functions*, and the technique is generally referred to as *G-stop facility*. A function $g(t, y)$ is provided by the user to compute a single value for a given point on the solution curve. If the sign of the computed values changes along this curve, an event is generated. Root-finding methods are used to determine the exact event location. The function g must be continuous for this sake, but not necessarily smooth. If more than one monitor function was provided, the earliest event is used.

In the branching rule the event should trigger when a threshold concentration is reached. A g-function for this can be derived easily by subtracting the threshold from the concentration, so:

$$g(t, y) = y - T.$$

To express this in XL, we can make use of anonymous function expressions (see section 3.4 on page 66). The current state y is provided implicitly as the attribute values of the nodes in the graph. This hides the mapping between such attributes and the state vector from the user and is thus easier to use. The g-function is then specified as

```
void=>double c - T
```

where `c` is the current concentration and `T` is the threshold. A function `monitor` allows to register this monitor for the integrator.

In the tree model we want to install such a g-function for every node of type `A`. So to stop integration when the threshold concentration was reached for any node of type `A` we can write a rule:

```
a:A ::> monitor(void=>double a[carbon] - T);
```

Although integration now stops at the correct time, we lack knowledge about which node actually exceeded the threshold concentration. An overloaded variant of the `monitor` function allows to supply an event handler along with the g-function. The event handler must implement the interface `java.lang.Runnable` and will be executed after integration was halted. Setting the monitor in conjunction with the branching rule looks like this:

```
a:A ::> monitor(void=>double a[carbon] - T, new Runnable() {
  public void run() {
    a ==>
      [RU( 30) RH(75) B(0) A(a[carbon]/2)]
      [RU(-30) RH(75) B(0) A(a[carbon]/2)]
    ;
  }
});
```

Another need for a g-function prevails when the user wants to plot or visualize some attributes. For instance, the attribute `len` was introduced to capture the length of the branches, while their attribute `length` is used for their visual representation as a cylinder. During integration their length should be updated in regular intervals so that the user can see the tree growing. The corresponding g-function must be continuous and change sign in regular intervals. Examples for such a function are a *triangle wave* or a *sine wave*. A *rectangle wave* or *sawtooth wave* should be avoided as they have a very large (perhaps even infinite) slope at the point where the sign changes.

A function `monitorPeriodic` is provided to trigger such periodic events. It uses a sine wave for the g-function with the specified period. Contrary to `monitor`, integration does not stop when the event triggers. Therefore, no modifications to the structure must be made, but changing the values of attributes is possible. The corresponding code to setup visualization is shown in listing 5.7.

5.6. Numerical Codes

It was shown in chapter 2 that there is a wide choice of numerical integration methods. Considering various features like adaptive step sizes with error control, interpolation and

Listing 5.7: Code to setup visualization in periodic intervals.

```
1 monitorPeriodic(PERIOD, new Runnable() {
2   public void run() [
3     b:B ::> b[length] = b[len];
4     c:C ::> c[radius] = c[carbon];
5     { derive(); }
6   ]
7 });
```

event handling, their implementation can be quite tricky. It is therefore recommended not to reimplement such methods, but to resort to existing libraries.

Several libraries for numerical integration of ODEs have been investigated, among them:

1. Apache Commons Math[1]
2. Open Source Physics[2]
3. SUNDIALS[3] [HBG+05]

We will shortly present these below and then discuss which one was chosen for inclusion in GroIMP.

5.6.1. Apache Commons Math

The *Apache Commons* project is a collection of reusable Java components. All source and documentation of the Commons project is available under the Apache License, version 2. The project is composed of three parts:

- The Commons Proper – A repository of reusable Java components.
- The Commons Sandbox – A workspace for Java component development.
- The Commons Dormant – A repository of Sandbox components that are currently inactive.

In the *Commons Proper* the *Apache Commons Math* project can be found. It contains packages for numerical analysis (including root finding and function interpolation), solvers for ODEs, statistics and much more.

[1] `http://commons.apache.org/math/index.html`, accessed 8. December 2011
[2] `http://www.opensourcephysics.org`, accessed 8. December 2011
[3] `https://computation.llnl.gov/casc/sundials/main.html`, accessed 8. December 2011

Specification of the ODE

The package `org.apache.commons.math.ode` and its subpackages contain all classes and interfaces related to integration of ordinary differential equations. In particular, the interface `FirstOrderDifferentialEquations` allows to specify the problem to simulate:

```
1 interface FirstOrderDifferentialEquations {
2   void computeDerivatives(
3     double t, double[] y, double[] yDot);
4   int getDimension();
5 }
```

The number of elements to be expected in the state and rate vector can be queried by calling the method `getDimension()`. The rate function $f(t, y)$ is evaluated by calling `computeDerivatives()`. The parameter `yDot` is used to pass in memory for the resulting rates.

Solver interface

Classes which are devoted to solve first order differential equations implement the interface `FirstOrderIntegrator`. The interface defines a single method `integrate` that is passed in the ODE to solve together with initial state `y0` at initial time `t0`. The ODEs will then be integrated up to time `t` and the result for this time will be written into the memory provided by `y`.

```
1 interface FirstOrderIntegrator extends ODEIntegrator {
2   void integrate(FirstOrderDifferentialEquations equations,
3     double t0, double[] y0, double t, double[] y);
4 }
```

Besides explicit Runge-Kutta methods (for instance, classical Runge-Kutta) using a fixed step size for integration also many adaptive step size integrators are provided. A complete list is shown in table 5.2. All methods are for non-stiff problems, and the Adams methods use the Nordsieck representation and therefore allow for efficient change of step size.

The interface implemented by the integrators also inherits from another interface `ODEIntegrator` that defines methods to install step and event handlers.

An `EventHandler` provides a g-function to monitor integration for possible events and an action to handle them. Event handling can be used with all integrators, even the ones using a fixed step size.

A `StepHandler` is called after each successful step and is used to provide continuous output. This can be used for plotting the solution. Interpolation is used to obtain intermediate points.

Monitor functions

The already mentioned `EventHandler` takes on the same role as monitor functions do in GroIMP. The interface is defined as follows:

Name	Type	Order	
`EulerIntegrator`	RK	1	fixed step size
`MidpointIntegrator`	RK	2	
`ClassicalRungeKuttaIntegrator`	RK	4	
`GillIntegrator`	RK	4	
`ThreeEightesIntegrator`	RK	4	
`AdamsBashforthIntegrator`	LMF	variable	adaptive step size
`AdamsMoultonIntegrator`	LMF	variable	
`GraggBulirschStoerIntegrator`	extrapolation	variable	
`DormandPrince54Integrator`	ERK	5(4)	
`DormandPrince853Integrator`	ERK	8(5,3)	
`HighamHall54Integrator`	ERK	5(4)	

Table 5.2.: Integration methods provided by Apache Commons Math library.

```
1 interface EventHandler {
2   double g(double t, double[] y);
3   int eventOccurred(double t, double[] y, boolean increasing);
4   void resetState(double t, double[] y);
5 }
```

Besides a function `g` to monitor state, also a handler is provided that is called once an event triggers. The handler returns are flag indicating if integration should proceed or has to stop, or if state or derivatives are discontinuous at this point.

Search for the exact time when an event triggers is performed using *Brent's method*[4], which is a combination of root bracketing, bisection, and inverse quadratic interpolation.

5.6.2. Open Source Physics

The *Open Source Physics* is a library for numerical simulation and visualization and a collection of examples for application in physics. It is released under GNU General Public License (GPL), version 2. A set of numerical integrators is included in a package `org.opensourcephysics.numerics`. Table 5.3 lists the implemented methods.

Specification of the ODE

Ordinary differential equations are provided to the integrators by implementing the interface `ODE`:

```
1 interface ODE {
2     void getRate(double[] state, double[] rate);
3     double[] getState();
4 }
```

[4] `http://mathworld.wolfram.com/BrentsMethod.html`, accessed on 14. January 2012

Name	Type	Order
`Adams4`	LMF	4
`Adams5`	LMF	5
`Adams6`	LMF	6
`Euler`	RK	1
`EulerRichardson`	RK	2
`Ralston2`	RK	2
`Heun3`	RK	3
`RK4`	RK	4
`Butcher5`	RK	5
`Fehlberg8`	RK	8
`CashKarp45`	ERK	5(4)
`DormandPrince45`	ERK	5(4)
`LeapFrog`	symplectic	3
`Verlet`	symplectic	n/a

Table 5.3.: Integration methods provided by Open Source Physics library.

The first derivatives are computed by the method `getRate`, where `state` is the current state as provided by the integrator, and `rate` is memory to store the resulting rates. No parameter for the time is mentioned explicitly, since laws of physics do not depend on the current time. If the user wishes to have a local time value accessible, it can be added as another element of the state with a rate of one.

The initial state is returned by the method `getState` and the length of the array is the dimension of the problem.

Solver interface

All numerical integrators implement an interface `ODESolver`:

```
1 interface ODESolver {
2     void   initialize(double stepSize);
3     double step();
4     void   setStepSize(double stepSize);
5     double getStepSize();
6 }
```

It is responsible for managing the size of the step. Calling the method `step` then integrates over the previously set duration. Subinterfaces `ODEAdaptiveSolver` for a variable (internal) step size and `ODEEventSolver` for providing a g-function exist.

With the exception of `CashCarp45` and `DormandPrince45`, all solvers use a fixed integration step size. The special handling of discontinuities with early exit, that characterizes the Cash-Karp method, is missing in the implementation. Symplectic integrators `LeapFrog` and `Verlet` are provided, but they need a special layout of the rate and state vector by design, making them not interchangeable with other methods. The `Fehlberg8`

is an implementation of the method shown in table 2.2 on page 23, but error estimation and adaptive step size control is missing. The Adams methods work in a PECE mode, and initial steps are computed by a Runge-Kutta method.

The implementation of the methods lacks any kind of interpolation (although an interface `ODEInterpolationSolver` exists, but it is never implemented). Event handling is supported by an `ODEBisectionSolver`, which implements `ODEEventSolver`. It uses another solver for integration (by default `RK4`), but since no interpolation is performed the achieved efficiency is suboptimal.

5.6.3. SUNDIALS

SUNDIALS is a *SU*ite of *N*onlinear and *DI*fferential/*AL*gebraic equation *S*olvers written in C/C++. As the name suggestes, besides methods for numerical integration of ODEs it also includes some for DAEs (Differential Algebraic Equations). SUNDIALS is distributed under a BSD license (BSD-3 with an additional notice). However, as of now (13. December 2011) the last release dates back to May 2009.

The library consists of five solvers:

CVODE – solves stiff/nonstiff ODE systems in the form $y' = f(t, y)$
CVODES – solves stiff/nonstiff ODE systems with sensitivity analysis in the form $y' = f(t, y, p)$
IDA – solves DAE systems in the form $F(t, y, y') = 0$
IDAS – solves DAE systems with sensitivity analysis in the form $F(t, y, y', p) = 0$
KINSOL – solves nonlinear algebraic systems

Interesting in our case are CVODE and CVODES. Both of them are using the same methods, with the latter one being a superset of the former one. For nonstiff problems, Adams-Moulton formulae are used with a variable order between 1 and 12. For stiff problems, it uses Gear's Backward Differentiation Formulae with a variable order between 1 and 5. The implicit equations are either solved by simple iteration or using Newton's method. The solvers work either serially or in parallel.

Specification of the ODE

The rate function must be provided as a callback, which is typical for C applications. The type of the callback function is defined as

```
typedef int (*CVRhsFn)(realtype t, N_Vector y, N_Vector ydot,
    void *user_data);
```

Here, `realtype` is a configurable data type for floating point numbers (usually `double`), and `N_Vector` is a type provided by SUNDIALS for representing a vector. The return value of the callback can be used to indicate errors to the solver.

Solver interface

Access to the solver is granted by a set of global functions. The most important ones and how they are typically used when defining an IVP are

```
void *cvode_mem = CVodeCreate(lmm, iter);
CVodeInit(cvode_mem, f, t0, y0);
CVodeSStolerances(cvode_mem, reltol, abstol);
CVDense(cvode_mem, N);
CVodeRootInit(cvode_mem, nrtfn, g);
CVode(cvode_mem, tout, yout, &tret, itask);
CVodeFree(&cvode_mem);
```

The memory referred to by `cvode_mem` is used by the library to hold the internal state of the solver. For stiff problems, `CV_BDF` and `CV_NEWTON` should be passed for `lmm` respectively `iter`.

The IVP is passed to the integrator with the second call. Initial time and state are given by `t0` and `y0`, and the signature of the rate function `f` must match `CVRhsFn` from above.

If Newton iteration is used to solve the implicit equations, a linear solver must be installed. Here, a dense linear solver is used, but the library also provides other kinds.

Rootfinding functions may also be installed, and will be described below. The actual integration is started by a call to `CVode`.

Monitor functions

To monitor progress of the integration, user-defined functions may be provided as a callback. The type of such callback functions is defined as

```
typedef int (*CVRootFn)(realtype t, N_Vector y, realtype *gout,
    void *user_data);
```

`N_Vector` and `realtype` are as explained above. The parameter `gout` provides access to memory for returning the values of all g-functions at once. The size of `gout` must be set prior integration by calling `CVodeRootInit` (parameter `nrtfn`).

5.6.4. Discussion

Libraries for numerical integration of ODEs have been developed in many programming languages, mainly Fortran and C/C++, but fortunately also for Java. As GroIMP/XL was implemented in Java and thus gains platform independence, one should favour libraries that have been implemented in Java as well, unless there is a good reason not to do so. As the basic interface between numerical integrator and problem definition is the same — an initial value problem expressed by a rate function and initial state — regardless which library was used, switching to another library later on should in principle be possible.

From the libraries presented above, Apache Commons Math looks more sophisticated than Open Source Physics, and should be used for non-stiff problems. For stiff problems, often arising for instance in the simulation of chemical kinetics, the SUNDIALS library should be used instead (BDF with Newton).

Other libraries have been also investigated, but they were either non-Java or lacked some features compared to the presented libraries. Therefore, the choice was to include

Apache Commons Math into GroIMP, but also a wrapper to SUNDIALS is provided. By default a Dormand-Prince 5(4) method with variable step size is used.

5.7. Wrapping Access to Numerical Libraries

Access to numerical integration algorithms is provided by a package `de.grogra.numeric` in the plugin `Numeric`. The bottom part of the UML-Diagram in figure 5.7 presents the types exposed by the plugin. The upper part in the diagram builds on top of this in the RGG plugin and is described later on in section 5.8.2.

An interface `Solver` has been introduced to facilitate support for different numerical libraries, where the implementation `FirstOrderIntegratorAdapter` provides access to those solvers provided by Apache Commons Math, and the implementation `CVodeAdapter` respectively to CVODE.

A compromise is made between the user-interfaces of the libraries presented above. As a result, the `Solver` interface allows to set monitor functions and tolerances. Additional options may be passed in a `Map` as well, but interpretation and availability of such options is up to the implementation of the actual integration method that is being used.

Monitor functions are provided, like for CVODE, all at once. This simplifies things if a special setup has to be performed before actually evaluating the g-functions, like having to copy the state into the graph (see section 5.8.2 below).

Actual integration is performed by calling the function `integrate`, to which initial conditions `t0` and `y0` are passed along with an implementation of the `ODE` interface for the rate function f.

5.7.1. Wrapping Access to Apache Commons Math

In Apache Commons Math the interface `FirstOrderDifferentialEquations` is used to describe the rate function of the ODE, while the initial state is passed directly to the solver when its `integrate` method is called. Besides a function `computeDerivatives` to evalute the rate function, another function `getDimension` is used by the solver to query the size of the state vector, so that memory for internal data structures may be allocated.

Since in Java it is possible to obtain this information by retrieving the length of the array for the state vector, in the design of the wrapper interfaces it was decided to omit such a function like `getDimension`.

Another complication when wrapping Apache Commons Math is that monitor functions are assumed to be registered individually, rather than all at once. Since during integration it is not determined in which order monitor functions will be called, a caching strategy has been implemented that will ensure that the monitor will only be evaluted if time or state does not match to the cached values.

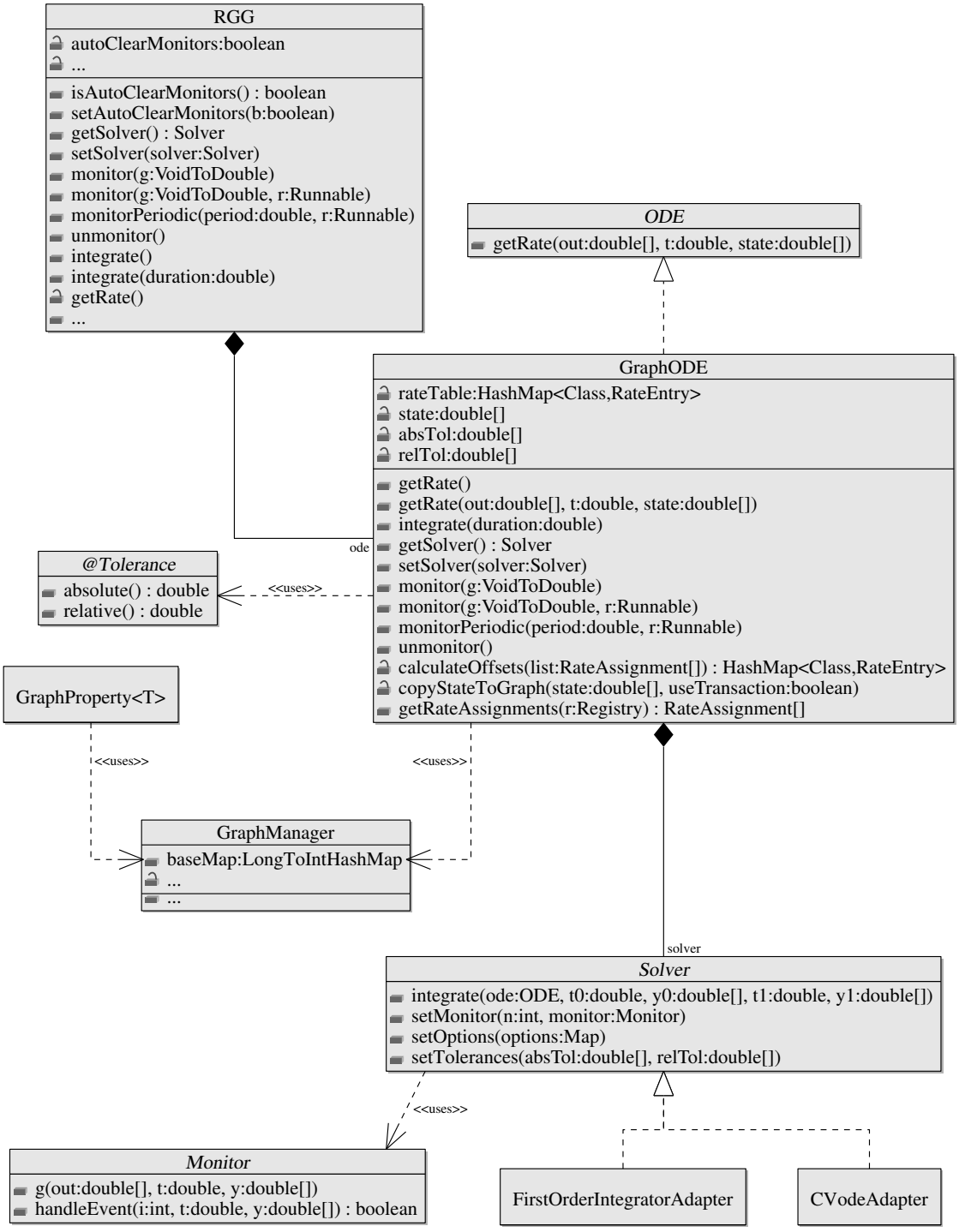


Figure 5.7.: Classes used in the ODE framework.

5.7.2. Wrapping Access to CVODE

CVODE is a C library, and as such access to it requires to think about how to transfer execution from Java to native code. One way to achieve this is to use the *Java Native Interface* (JNI). Here, the developer has to implement a wrapper class (in our case one that implements the `Solver` interface) marking all functions as `native` for which execution is to be forwarded to a native context.

Then, a tool `javah` (part of the JDK) is used to generate a C header file for the wrapper class. Functions that have been marked as `native` will appear as global C functions in the generated header file, using special name mangling to uniquely identify them. The task of the developer then is to implement those functions using ordinary C code, to finally produce a shared library. This will be loaded and linked by the wrapper class at runtime.

While this sounds simple at a first glance, implementation of the wrapper can become quite challenging, especially if the native code calls back to Java (which is the case for monitor and rate functions).

Java Native Access (JNA)[5] provides an alternative, where the wrapper can be implemented completely using just Java code. This simplifies development and also makes debugging much easier for the developer.

The classes in the package `de.grogra.numeric.cvode` provide access to CVODE to the degree needed to implement the `Solver` interface. There, the class `CVODE` contains wrappers for the functions in the library, and some constants that have been defined. It is also responsible for loading the shared library. The class `N_Vector` represents the analogously named structure for representation of an n-dimensional vector. It has been retrofitted with additional methods to more easily access the data stored in the vector. The classes `CVRhsFn` and `CVRootFn` represent callbacks for rate and monitor functions as they are used by CVODE. `CVodeAdapter` in the package `de.grogra.numeric` implements the `Solver` interface and resorts to those wrapper classes.

The native library itself must be created independently from the wrapper code. For CVODE, this can be done by downloading and then extracting the SUNDIALS source code to some directory. In a console the developer has to change to this directory and execute the common commands

```
./configure CFLAGS="-fPIC"
make
```

The `configure` script creates a suitable `Makefile` that is used to compile SUNDIALS. Setting the variable `CFLAGS` instructs `configure` to generate position-independent code and is needed when a shared library ought to be created. Calling `make` will initiate the compilation process and produce object files. These must be linked to obtain a shared library file.

The command used to create the shared library with GCC for Linux is

```
gcc -shared -Wl,-soname=libcvode.so -o libcvode.so -static-libgcc *.o
```

[5] `https://github.com/twall/jna`, accessed 11 Januar 2012

The option `-shared` instructs GCC to produce a shared library. The `-soname` linker option will cause the name of the library to be stored in the file itself.

Naming of shared library files is platform dependent. For instance, under Windows the library would be named `cvode.dll` instead of `libcvode.so`.

5.8. Implementation of the Rate Assignment Operator

Although introduction of the rate assignment operator looks very easy to the user, it requires several changes to be made in the implementation of the XL compiler. These changes can be separated into two groups, one for compile time and the other one for run time.

At compile time, all occurrences of the rate assignment operator `:'=` in a class are collected as pairs of property and the class it was accessed with. This table is written into a static field of a hidden class, the name of the class being derived from the name of the class that contained the rate assignments (using a specially generated name containing $ to prevent name collisions, as was already used for operator methods).

At run time, all classes belonging to an RGG project are searched for those tables to generate a combined list of $(class, property)$ pairs. Using so-called extents it is possible to obtain lists of node instances that are stored in the graph, one list per class. The extent also replicates the class hierarchy, so it is possible to extract the reduced and separated view on the class hierarchy that only considers those attributes that are being integrated.

5.8.1. Compile Time

To support a quasi-parallel modification of node attributes, the XL languages contains deferred versions of the traditional operators (see table 3.6 on page 66). The XL compiler does not associate any semantics to these operators, rather this is up to the run time system.

The rate assignment operator `:'=` inserts into this list of operators. Regarding the XL compiler, it does not associate any semantics to this operator as well. However, to support ODE integration as was explained above, the compiler will generate a table of all occurrences of rate assignments, one table per class.

To make the compiler aware of the existence of the new operator, the first step is to announce a new token `:'=` as `DEFERRED_RATE_ASSIGN` in the parser grammar (file `XL.g`).

Then we have to add a reference to this token to the places where the other deferred operators are allowed. In the parser grammar this is the case in the preamble (in an array `NON_JAVA_TOKENS`), and in rules `overloadableOperator`, `assignmentExpressionRest`, and `singleExpressionNoRange`. This ensures that rate assignments will be properly parsed and nodes in an abstract syntax tree (AST) will be created.

The tree parser (in file `Compiler.tree.g`) is responsible to convert the AST into expression trees [Kni08, section 8.3]. For rate assignments, this must be added to the rule `blockExpr`, so that these get converted as well. Compilation of deferred operators is performed by a helper function `compileDeferredAssignment` in the class `CompilerBase`,

Listing 5.8: Example `Model.rgg` that uses rate assignments.

```
1 module A(double n);
2
3 protected void getRate() [
4   a:A ::> a[n] :'= 1;
5   ...
6 ]
```

Listing 5.9: Generated helper class containing a table of rate assignments.

```
1 class Model$ODEHelper {
2   public static final RateAssignment[] TABLE;
3   static { TABLE = new RateAssignment[] {
4     RateAssignment.create(p_{A,n}, A.class),
5     ...
6   } }
7 }
```

from which the generated tree parser is derived from. Special handling for rate assignments was added to this function, so that it generates a table of all property and node type pairs the rate assignments were used with.

Considering the RGG model given in listing 5.8, then a class equivalent to the Java code in listing 5.9 will be generated, where $p_{A,n}$ is the property `n` of module `A`. The class `RateAssignment` just holds a reference to the property and class (listing 5.10).

In GroIMP/XL, properties are managed by the classes shown in the class diagram in figure 5.8. Each model interface/class defines an inner property interface/class. Also, for each compile time class, there exists a run time counterpart. The property referred to within `CompilerBase` during compilation is an instance of the interface `Property` defined within the interface `de.grogra.xl.property.CompiletimeModel`. When the expression tree is compiled to bytecode, this is replaced by its corresponding run time equivalent, which is an instance of `de.grogra.xl.property.RuntimeModel.Property` (actually an instance of `de.grogra.rgg.model.PropertyRuntime.GraphProperty<T>`). The little trick with the generated class thus helps us with the transfer of information about rate assignments from compile time to run time.

This is all the compiler is concerned about rate assignments. It does not prescribe any semantics, rather this is up to the run time system and will be described below.

5.8.2. Run Time

To the user, the ODE framework is exposed directly as a class `GraphODE`, or indirectly as a set of helper functions in the class `RGG` that forward calls to a `GraphODE` managed internally. Both classes are part of the RGG plugin, with `GraphODE` and some helper classes located in the package `de.grogra.numeric`. `GraphODE` automatizes the tasks to

Listing 5.10: Definition of the class `RateAssignment`.

```
1  class RateAssignment {
2    public RuntimeModel.Property property;
3    public Class cls;
4
5    public static RateAssignment create(
6      RuntimeModel.Property p, Class c)
7    {
8      return new RateAssignment(p, c);
9    }
10
11   ...
12 }
```

perform when numerically integrating ODEs on a graph structure.

List of Rate Assignments

The first action in using rate assignments is to find all helper classes that have been generated and extract a list of instances of `RateAssignment` from them. GroIMP manages a registry, which is mentioned in appendix A.2 in [Kni08, page 377]. An undocumented feature of this registry is that it contains information about all classes belonging to an opened project. So one just has to query the registry for all project classes and process those that end in `$ODEHelper`. For each such class one checks if it contains a static field `TABLE` (via Java's reflection capabilities) and if so appends all entries it contained in this field to the list.

Assign Offsets

Once all occurrences of rate assignments have been collected, the memory layout (separated view) of the classes can be computed and offsets to the attributes can be assigned. The result is a mapping from `Class` to `RateEntry`, referred to as `rateTable`. The type `RateEntry` collects all properties of a class that are being integrated (see figure 5.9).

A new integer field `offset` has been introduced into the class `GraphProperty<T>` together with an operator function for rate assignments of `double` values. The rate assignment operator is also included in the interface `DoubleProperty`, which is used by the compiler to automatically recognize which operations are allowed on a property for an attribute of type `double`.

Calculation of the offsets begins with an iteration over the list of all `RateAssignments`. In this pass, the `offset` fields are invalidated (set to -1) and instances of `ClsEntry` are created (see figure 5.9). For each class used in a rate assignment, there is one instance of `ClsEntry`, and its field `props` stores all properties that have been accessed in conjunction with this class (without duplicates).

In a second pass, a loop over all instances of `ClsEntry` builds up a hierarchical structure by setting fields `parent`, `child`, and `next` appropriately. An entry p is considered parent of an entry e if the associated class of p is a (direct or indirect) superclass of e. An entry for the class `java.lang.Object` was added to form the root of the hierarchy. This is no problem, as `Object` is predefined by Java and has no attributes that could

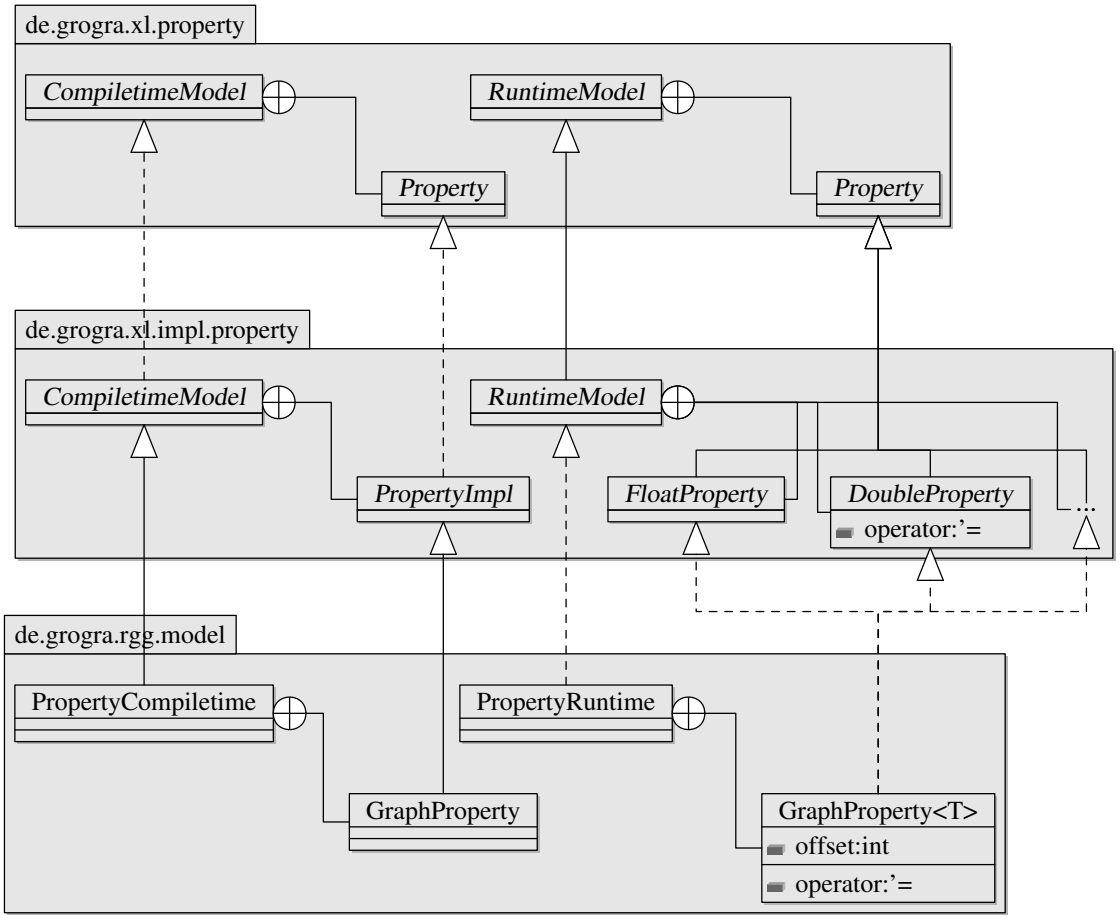


Figure 5.8.: Compile time and run time classes for managing properties.

be integrated. If p is considered parent of e, then e is inserted into the list of children, defined by first child (field `child` of p) and its siblings (field `next` of the last child). The generated tree structure is the reduced view on the class hierarchy of classes participating in integration.

In a third pass, the tree of `ClsEntry` is traversed in depth-first order starting from the root to calculate the number of attributes for each class, as is needed to obtain the separated view on the class hierarchy. For each entry, each property it contains is assigned an offset. This is complicated by the fact that there may be more than one `GraphProperty` per `PersistenceField`, the latter one being the superclass of all fields that hold the data for a property. A mapping from `PersistenceField` to `GraphProperty` helps to assign the same offset to all properties, and one `GraphProperty` is remembered as proxy for all the others that refer to the same field.

Offsets are numbered incrementally starting with the `size` of the parent. The size of the root entry is zero. When all properties have been processed, the `size` of the entry

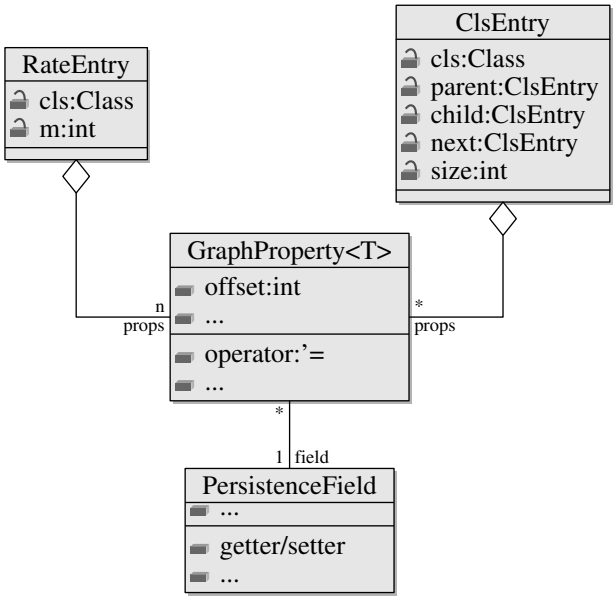


Figure 5.9.: Helper classes used for offset calculation.

is set to the next free number (sum of parent size and number of assigned offsets).

In a final fourth pass, the `rateTable` is extracted from the entries in the tree. Each `Class` is mapped to a `RateEntry`, where the same class is also stored in the field `cls` of the entry, the number of attributes to allocate in the field `m`, and a list of all properties that are accessed in rate assignments in conjunction with this class in the field `props`.

Length of State Vector

With the `rateTable` at hand, one can now continue with the steps that have been outlined in section 5.3.2. At first, we need to calculate how many values should be stored in the rate and state vector.

GroIMP helps us by providing extents, which are normally used in pattern matching to find all nodes of a given type. The extents are formed by instances of the class `Extent`, one instance per node type in the graph. The extents replicate the class hierarchy, and each extent also stores the number of nodes in the graph that exactly match (field `size`) and that can be converted (field `totalSize`) to the node type represented by the extent.

The latter one corresponds to the values t_* described in section 5.4. The length of the rate and state vector can then be calculated analogously to equation (5.2) on page 112, by iterating over every `RateEntry` of the `rateTable` and summing each product of the number of properties in the entry times the `totalSize` of the extent associated to the class from the same entry.

Memory Allocation

Once the size of rate and state vector is known, allocating memory for them is trivial. However, some of the provided integration methods seemed to not properly handle the case when no attributes were being integrated and so the length was zero. In this case one could either skip the remaining steps of the integration completely, or just allocate some non-zero number of elements for the vectors (we used one element).

Copy Graph to State Vector

Since the integrators work on arrays, prior to integration we have to copy the state from the graph into the state vector. This step includes assignment of memory locations in the state vector to each node in the graph. For calculation of the index for a property of a certain node later on, a mapping from the node ID to the base index into the state vector for this node is created in `baseMap` (see figure 5.7). This allows for the addressing that was described earlier in section 5.4.

For each `RateEntry`, a loop over all nodes in the associated extent and its sub-extents ensures that each node was assigned a base index. This is the case if the node was already put into `baseMap`. If not, a new base index is assigned directly behind the already allocated region, and the node is put into `baseMap`. The field `m` of `RateEntry` then indicates the size to allocate for the node, and the properties `props` in the entry are then used to copy all attributes.

One must be careful not to use `m` of the `RateEntry` that constitutes the outer loop, since the associated node type represents how the property was accessed, not by which node type it was defined. Node types that are removed in the reduced module hierarchy, because they do not provide any additional attributes that should be integrated, may still be found instantiated in the graph. To ensure that they get a base index assigned, one has to handle nodes in all sub-extents. However, these may include also derived node types that do provide additional attributes for integration, and so have a different size for allocation. Therefore, a `RateEntry` is obtained from `rateTable` for the actual type of the node to allocate space for, to obtain the correct `m`. If the node type is not contained in `rateTable`, the search is repeated with the supertype, until a `RateEntry` is found.

Integration

With all things prepared, it is now time to perform the actual integration. The user can select a solver implementation by calling the method `setSolver` on `GraphODE` or `RGG`, by default the Dormand-Prince 5(4) implementation of Apache Commons Math is used. When the integrator is called, it in turn calls back the rate function that has been provided by the user. The flow of execution is shown in figure 5.10.

Copy State Vector to Graph

Once integration is complete, the new state must become visible in the graph. A similar loop to the one before performs copying of the state vector into the node attributes. It is important to make use of the persistence mechanism described in [Kni08, appendix A.3.2], so that GroIMP is aware of these changes. Practically this means that the values have to be written to the nodes over the `GraphProperty`, not over the `PersistenceField` the property refers to.

Modifications to the attributes will then be recorded in a transaction, which is necessary to correctly update the graph in case it is shared by multiple GroIMP instances. However, this also introduces additional overhead and is thus only applicable after integration is done to commit the results.

The second option, which uses `PersistenceField`, turns out useful when preparing the graph for evaluation of the rates. In this case, all attributes must be written from the state vector to the graph as well, but it is not needed to distribute this information via the persistence mechanism. Strictly speaking, the additional overhead introduced by the transaction mechanism would cause allocation of a vast amount of memory, which would drastically degrade performance and is thus undesirable.

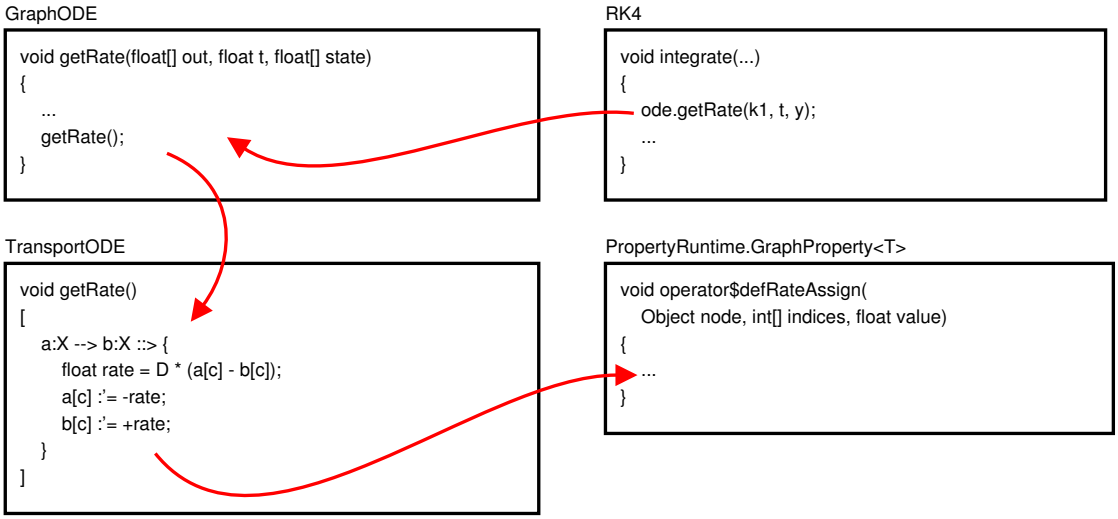


Figure 5.10.: Control flow when integrator evalutes rate equation.

5.9. Specification of Tolerances

The solvers implemented in numerical libraries usually allow the user to provide absolute and relative tolerances for the integrated values. Such tolerances can be either set as a scalar quantity and then apply equally to all elements in the state vector, or a tolerance vector can be provided to adjust the tolerance for each element individually.

The `Solver` interface contains a method `setTolerances` to provide vector-valued tolerances. Scalar tolerances are assumed to be set using the `setOptions` method.

When working on a graph structure, the `GraphODE` manages the mapping between node attributes and elements in the state vector. This mapping is not exposed to the user by purpose. Therefore, for the user it is not possible to set per-element tolerances using the `Solver` interface.

Instead, an annotation `@Tolerance` is provided so that the user can directly assign absolute and relative tolerances to node attributes. Its use in an RGG file is demonstrated in listing 5.11. Specification of absolute and relative tolerance is optional. If unspecified, a default value provided by the solver is used.

Listing 5.11: Example demonstrating use of `@Tolerance` annotation.

```
1 @Tolerance(absolute=1e-6, relative=1e-4)
2 double t;
3
4 module A(@Tolerance(absolute=1e-4) double n);
```

6. Results and Discussion

6.1. Chemical Kinetics with Operator Overloading

In chapter 4 an extension to the programming language XL, and in this sense also to Java, has been proposed that allows the user to redefine the meaning of operator symbols. The versatility of this extension has been demonstrated on some examples.

One of these examples, in section 4.4.4, shows how ODEs can be formulated using the law of mass action by properly overloading operator functions. This can be applied to the following conversion of a reactant A over an intermediate B to some product C:

$$\mathrm{A} \xrightarrow{k_1} \mathrm{B} \xrightarrow{k_2} \mathrm{C}$$

For demonstration purpose the rate constants have been set to $k_1 = 2$ and $k_2 = 1$. Specification of the two elementary reactions $\mathrm{A} \longrightarrow \mathrm{B}$ and $\mathrm{B} \longrightarrow \mathrm{C}$ of this reaction mechanism can then be written in XL as shown in listing 6.1 (for the complete model code see appendix A.1.1). The resulting time series plot is shown in figure 6.1. The initial concentration for A is 10 and for the other species it is zero. A classical Runge-Kutta method of 4th order with fixed step-size was used for numerical integration.

Similarly, Michaelis-Menten kinetics, which was discussed shortly in section 5.1.1, can be expressed using the overloaded operator symbols (listing 6.2, complete model code in appendix A.1.2). The implemented reaction mechanism is

$$\mathrm{E} + \mathrm{S} \underset{k_r}{\overset{k_f}{\rightleftharpoons}} \mathrm{ES} \xrightarrow{k_{cat}} \mathrm{E} + \mathrm{P},$$

with rate constants arbitrarily set to $k_f = 3$, $k_r = 0.1$ and $k_{cat} = 2$. Initial concentration for S was set to 10 and for the enzyme E set to (an unrealistic high value of) 3 results in the time series plot shown in figure 6.2. Again, a classical Runge-Kutta method of 4th order with fixed step-size was used for numerical integration.

6.2. dL-systems

In [PHM93] the formalism of dL-systems was presented. As was stated in that article "the simulations [...] were carried out using a programming language for parametric L-systems", where "the user must explicitly specify the formulae for numerically solving the differential equations included in the models (the forward Euler method was used in all cases)". This basically means that dL-systems are merely "a notation for expressing developmental models that include growing systems of ODEs" [FP04].

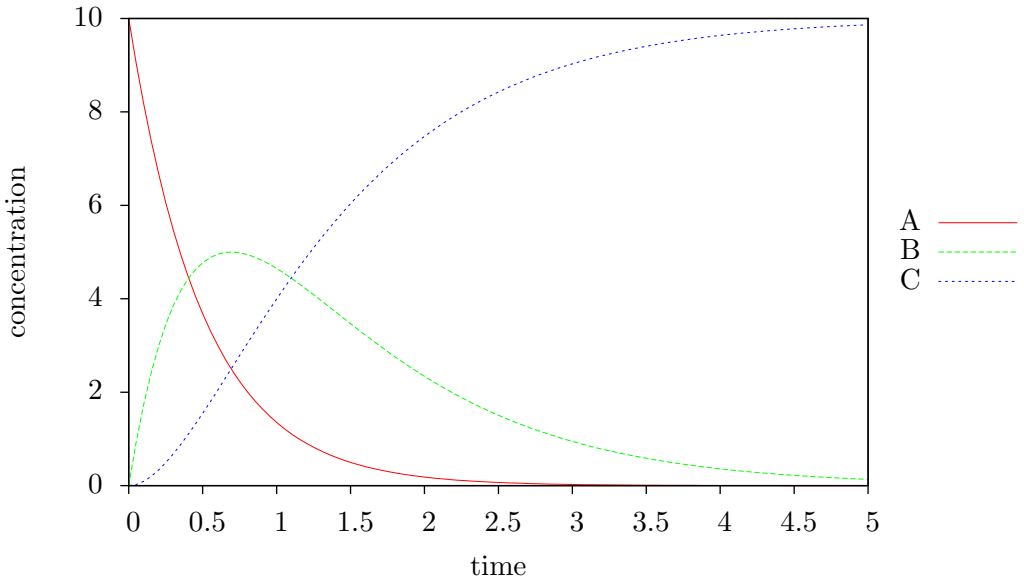


Figure 6.1.: Time series plot of the reaction mechanism A $\xrightarrow{k_1}$ B $\xrightarrow{k_2}$ C.

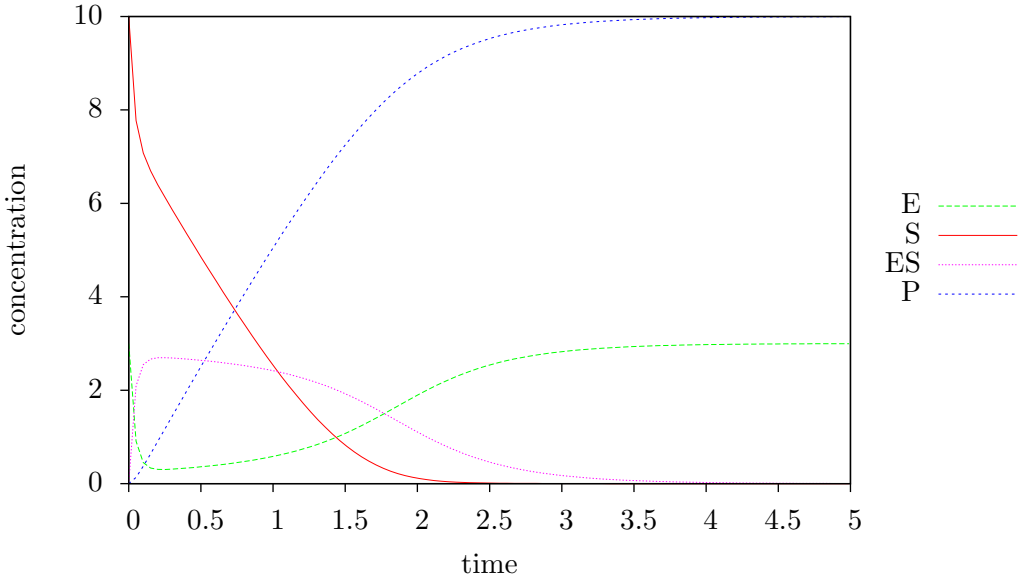


Figure 6.2.: Time series plot of Michaelis-Menten kinetics.

Listing 6.1: Specification of A $\xrightarrow{k_1}$ B $\xrightarrow{k_2}$ C in XL using operator overloading.

```
1 Model model = new Model();
2 model.add(A <=> B, k1);
3 model.add(B <=> C, k2);
```

Listing 6.2: Michaelis-Menten kinetics expressed using operator overloading.

```
1 Model model = new Model();
2 model.add(E + S <=> ES, kf, kr);
3 model.add(ES <=> E + P, kcat);
```

[PHM93] also provides some examples of dL-systems, for instance a model of the dragon curve or a model of the cyanobacterium *Anabaena catenula*. The latter was integrated numerically in [FP04] by means of the implicit Crank-Nicolson method [CN47, CN96]. However, it was entirely left to the user to manually transform the dL-system to an equivalent L-system and implement the numerical method in terms of L-system rules.

We can solve dL-systems more easily in GroIMP/XL by using the rate assignment operator. In the dL-system of the dragon curve given in [PHM93] the rules consist of two parts responsible for integration and for production, for instance:

$$
\begin{aligned}
&F_r(x,s):\\
&\quad \textbf{if } x < s \textbf{ solve } \tfrac{dx}{dt} = \tfrac{s}{T}, \tfrac{ds}{dt} = 0\\
&\quad \textbf{if } x = s \textbf{ produce } -F_r(0, s\tfrac{\sqrt{2}}{2}) + F_h(s,s) + F_l(0, s\tfrac{\sqrt{2}}{2}) -
\end{aligned}
$$

The first part of the rule describing the ODEs can be written in XL simply as:

```
fr:FR ::> fr[x] :'= fr[s] / T;
```

Since the rate of change for `fr[s]` is constant zero, the second ODE can be omitted. The second part containing the production can be defined using a monitor:

```
fr:FR ::> monitor(void=>double fr[x] - fr[s], new Runnable() {
  public void run() [
    fr ==> RU( A) FR(0, fr[s]*sqrt(2)/2) RU(-A) FH(fr[s], fr[s])
           RU(-A) FL(0, fr[s]*sqrt(2)/2) RU( A);
  ]
});
```

The `RU` denotes a rotation around the local up-axis, where parameter `A` provides an angle of 45°. The modules `FR`, `FL` and `FH` correspond to F_r, F_l and F_h, respectively. For complete model code see appendix A.2.1. The resulting developmental stages of the dragon curve are shown in figure 6.3.

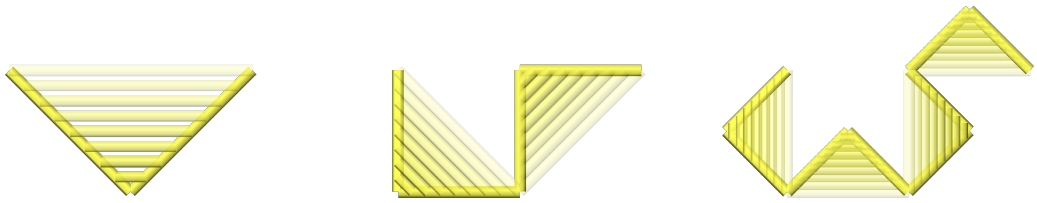


Figure 6.3.: Development of the dragon curve simulated in GroIMP with time steps of 0.125 T. Successive animation frames have been blended together (left 0–8, middle 8–16, right 16–24).

In a similar way, the model of *Anabaena catenula* (see figure 6.4) can be defined in XL (for complete model code see appendix A.2.2). For instance, the ODEs for the vegetative cells can be written as:

```
F(xl, cl) fv:FV(x, c) F(xr, cr) ::> {
  fv[x] :'= R * x;
  fv[c] :'= D * (cl - 2*c + cr) - MU * c;
}
```

Note that the actual parameters (when the rule is executed) of the modules can be used in the rule as well, saving some typing and thus making the representation of the ODEs more clear.

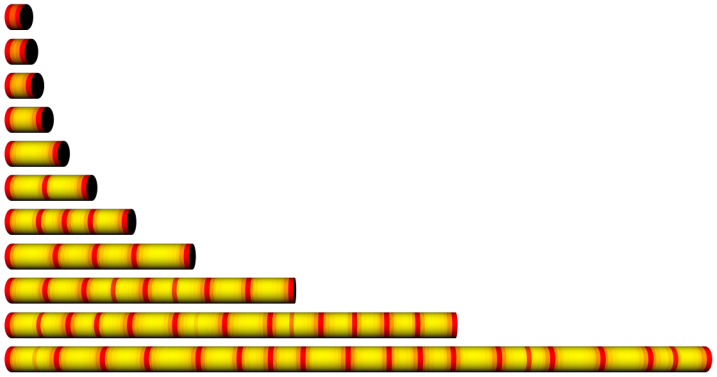


Figure 6.4.: *Anabaena catenula* simulation after $50n+20$ steps, $n \in \mathbb{N}, 0 \leq n \leq 10$. Color encodes nitrogen compound concentration (yellow – low, red – high).

In [PHM93] the diffusion process, described by the PDE $\frac{\partial c}{\partial t} = D\frac{\partial^2 c}{\partial x^2}$, was solved using a second order central difference approximation [Bic41] for $\frac{\partial^2 c}{\partial x^2} \approx D(c_l - 2c + c_r)$. As the concentration of the nitrogen compounds in the heterocysts is not assumed to be constant, but varies according to an ODE, it follows that diffusion between vegetative

and heterocystic cells should be included as well. Considering pairs of nodes allows to provide an equivalent, but easier, formulation of the diffusion process:

```
a:F b:F ::> {
  double r = D * (a[c] - b[c]);
  a[c] :'= -r;
  b[c] :'=  r;
}
```

This way diffusion is performed between both cell types (vegetative and heterocyst) and mass balance is ensured. In addition, by separation of the different processes the model becomes easier to understand as well. However, this technique cannot be used with dL-systems in the narrower sense, as for these in each rule there can be only one active symbol for which the ODE can be applied, not two as in the diffusion exchange.

6.3. Auxin Dynamics in Plants

Plant growth is believed to be controlled by the hormone auxin. A mechanistic model of auxin transport was developed by [PCS+09], based on the feedback between auxin flux from one metamer[1] to the next, and concentration of auxin transport proteins in the direction of the flux. For details on model equations and biological principles see [PCS+09].

The original model was created using the L+C modelling language [KP03]. The L+C implementation uses Euler integration with a step size of 0.05. The model was reimplemented in XL making use of the rate assignment operator to easily specify the differential equations. For reference, a version using Euler integration like in the original model was also implemented. The modelled structure was a simple static combination of 4 metamers comprising the main shoot, and two lateral metamers (Figure 6.5).

Simulations were performed measuring the wall-clock time between the moment the simulation was started, and the moment the auxin level in the bottom metamer exceeded a certain threshold (1.5). For the model using Euler integration this took approximately 5 seconds with an integration time step of 0.05, and 25 seconds with an integration time step of 0.01. Attempts to speed up Euler integration by increasing the integration time step from 0.05 to 0.1 resulted in chaotic model behaviour. By optimizing the code (e.g., taking out any other time-consuming part of the simulation such as visualization and chart plotting) simulation time could be decreased, but this situation was not representative for common modelling practice.

Using the new ODE solver system with Dormand-Prince 5(4) integration, the simulation time dropped to about 0.5 seconds. Although not precisely estimated, the simulation times of the L+C model were comparable to the Euler version of the XL model for both time step sizes of 0.05 and 0.01.

Figure 6.6 shows the results of the calculations. Euler integration with an integration time step of 0.01 and the Dormand-Prince integration showed comparable accuracy;

[1] A metamer consists of an internode (part of the stem/branch), a bud, and a leaf.

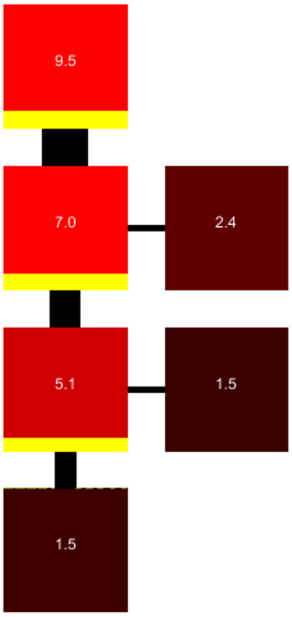


Figure 6.5.: Static plant structure used to model auxin transport. A square represents a metamer (the degree of red and the value in the square both represent the auxin level, the width of the yellow bar is the concentration of auxin transporter proteins). The connectors between the squares represent auxin flux from one metamer to the other, the width representing the value of the flux. Auxin transport is occurring basipetally (from the top metamer towards the bottom metamer). Auxin is only produced in the top metamer.

Euler integration with an integration time step of 0.05 showed an underestimation of auxin level. In other words, to get a comparable level of accuracy, Euler integration needs ca. 47 times more simulation time than Dormand-Prince integration. This disadvantage of Euler integration was already known; we have shown here that this downside can be avoided with minimal effort, merely by using a different operator and by placing all ODEs in one method.

6.4. Model of Gibberellic Acid Biosynthesis

Another hormone important for plant growth and development is gibberellic acid (GA). Its bioactive form, GA1, participates in the regulation of a number of physiological processes, such as seed germination and organ extension. [BSKK05] devised a simple model of GA biosynthesis involving the three last steps from GA19 over GA20 to the final product, GA1, and its catabolite, GA8. This simple model was extended considerably [BSHK$^+$08] by adding a signal transduction network with three further elements (essentially transcription factors), thereby linking the spatiotemporal dynamics of bioactive

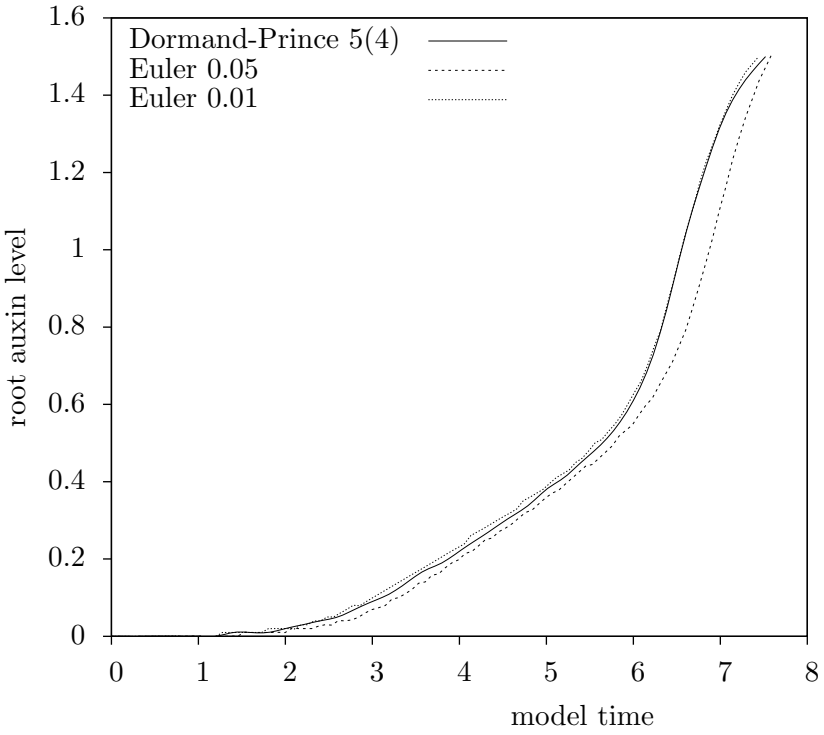


Figure 6.6.: Time series plot of auxin level in the bottom (root) metamer.

GA1 to internode extension in barley.

A simplified model consisting of 10 internodes was obtained by decoupling the network model from the morphogenetic model of barley. Initial concentrations of substances and parameters for the Michaelis-Menten equations were taken from [BSHK$^+$08]. To show how a more advanced numerical method can be used, in this case the classical Runge-Kutta method of 4th order, the model was modified accordingly. Furthermore, the model was reimplemented using the new ODE framework without and with monitor functions to show how this simplifies the use of advanced integration methods and to evaluate how different methods perform on this problem.

Table 6.1 shows the obtained accuracy for several numerical methods. A reference solution was obtained with Dormand-Prince (DP5(4)) using an absolute and relative error tolerance of 10^{-4}, and was checked against Gragg-Bulirsch-Stoer (GBS). As can be seen, Euler integration produces large errors, while Runge-Kutta (RK4) comes close to the reference solution. However, for a step size of 1, both methods become unstable. Also, the effort for RK4 to obtain the requested accuracy is slightly higher than for DP5(4), and also *a priori* it is not clear what is the correct step size for the problem. The Adams methods (AB and AM) perform efficiently and with good accuracy, but for the AB method one should not set the order too high to obtain a stable solution for this

problem.

Method	h	Calls	Deviation
Euler	1	280	387
	1/2	560	0.4015
	1/4	1120	0.1968
	1/8	2240	0.0975
	1/16	4480	0.0485
RK4	1	1120	364
	1/2	2240	0.0005
	1/4	4480	0.0003
	1/8	8960	0.0003
	1/16	17920	0.0003
DP5(4)	var.	3386	-
GBS	var.	10380	0.0003
AB	var.	1660	0.0089
AM	var.	3167	0.0007

Table 6.1.: Evaluation of GA network simulation results for GID2 obtained over 280 time steps. Simulations were done with use of selected integration methods with variable or fixed step size h as indicated (RK4 – classical Runge-Kutta, DP5(4) – Dormand-Prince 5(4), GBS – Gragg-Bulirsch-Stoer, AB – Adams-Bashforth, AM – Adams-Moulton). Calls give the number of evaluations of the rate function. Deviation is maximal distance to reference curve obtained with DP5(4). Requested accuracy of variable step size methods was 10^{-4}. AB and AM methods are of order 2.

A diagram showing deviations of the Euler method from the reference solution can be found in figure 6.7. One can see that the Euler method overestimates respectively underestimates the solution. Halving the step size also halves the error of the Euler solution, as is expected.

6.5. Radiation Modelling

In [HKL+08] a light model based on GroIMP's path tracer was presented. Light, as an important factor for modelling plant growth, controls how much photosynthesis a plant can perform and subsequently the amount of growth and the shape of the plant. This was applied to a mixed-species tree stand consisting of young specimens of beech (*Fagus sylvatica* L.) and spruce (*Picea abies* (L.) Karst.) trees. A virtual forest consisting of 700 trees is shown in figure 6.8.

The spruce trees were based on a model presented in [Kur99, chapter 5], but were extended to include dependency on light as a limiting factor. The model of young beech is a relatively complex functional-structural model that includes carbon assimilation via photosynthesis and distribution. The effect of competition for light can be easily seen

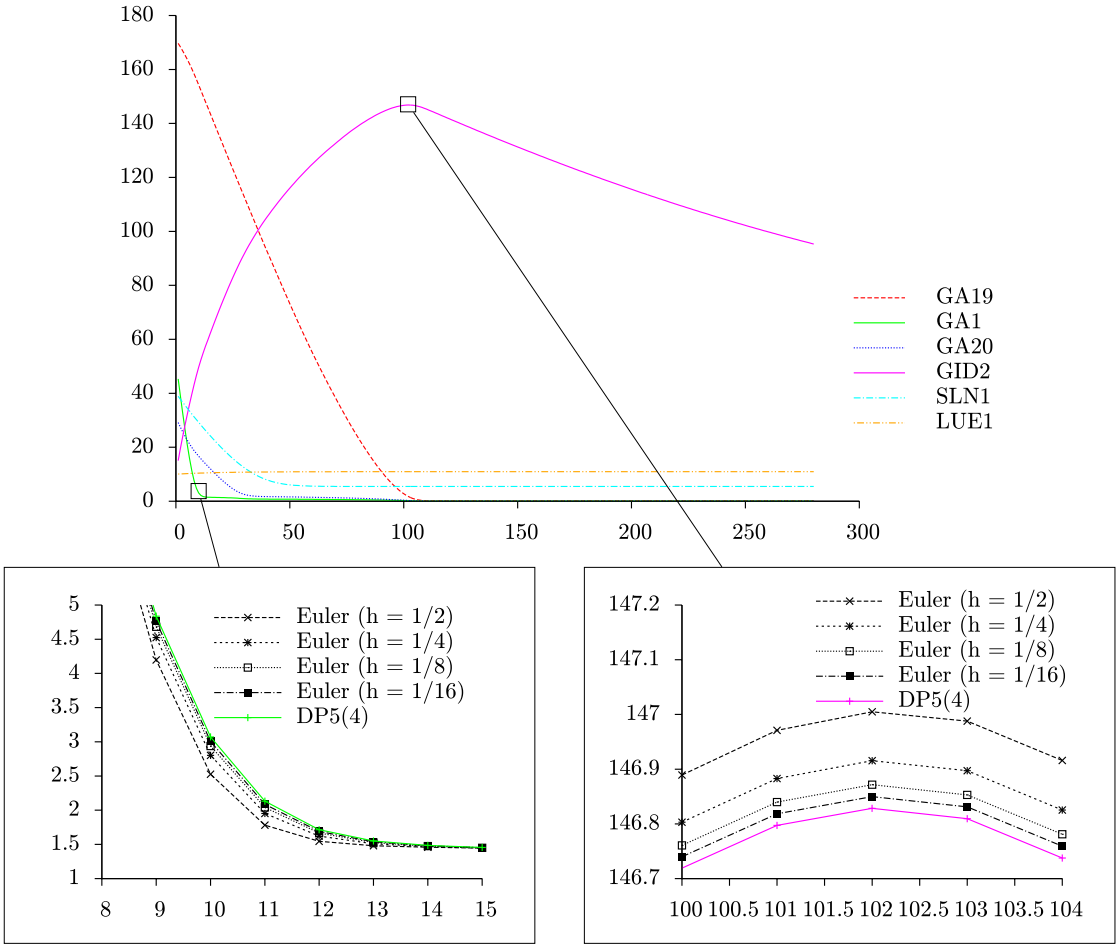


Figure 6.7.: Time series plot of GA network. Concentration of substances versus model time of 280 time units computed using Dormand-Prince method with variable step size is compared against results of Euler integration with step size h. Curves show summed concentration of hormones in the whole plant.

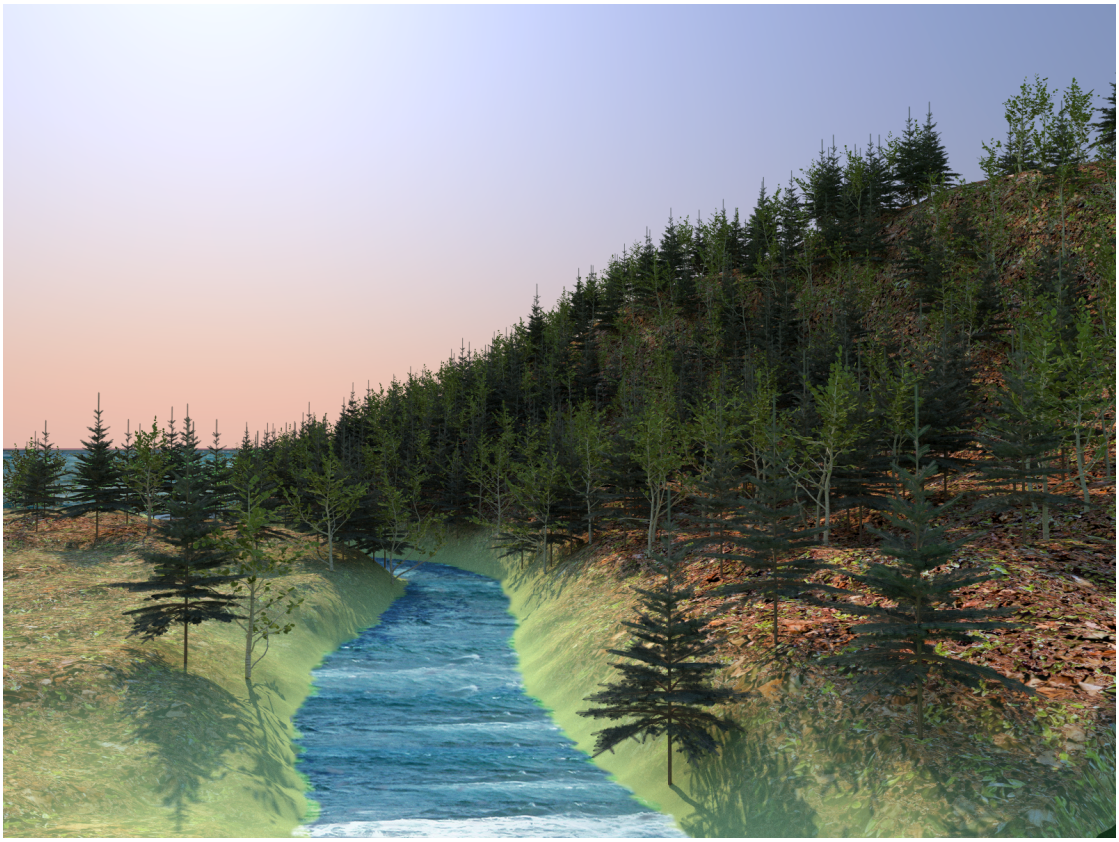
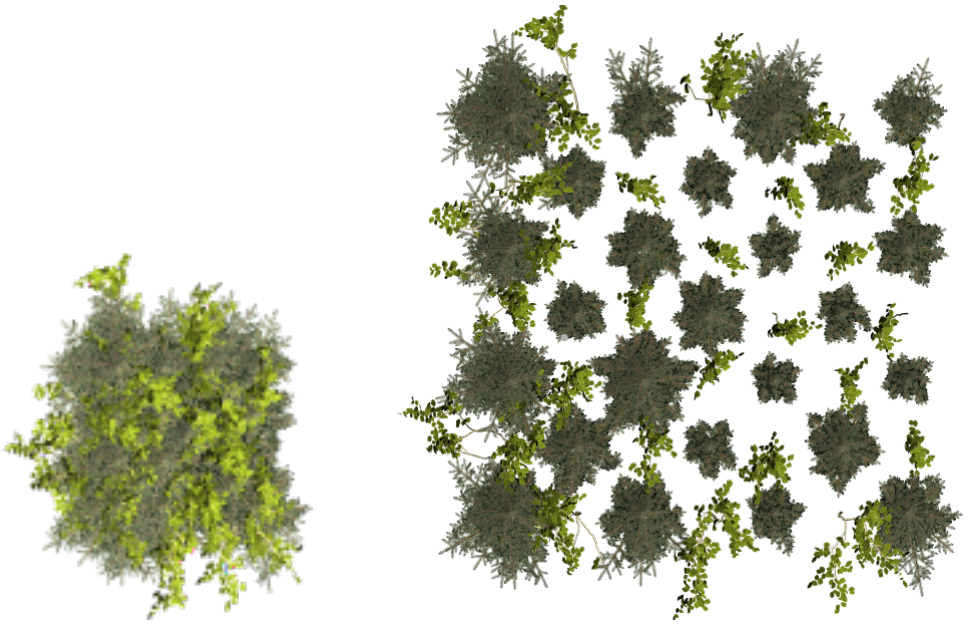


Figure 6.8.: Virtual landscape with 700 beech and spruce trees.

in figure 6.9. Light was incident mainly from the right, but also from other directions. Therefore, trees placed at the border of the stand were growing bigger than those that were placed inside. Note that the influence of light on growth was severely exaggerated to make the effect more clear.



(a) (b)

Figure 6.9.: Close tree stand after 10 years: (a) as simulated and (a) moved further apart afterwards.

Although the models of the trees did not make use of the ODE framework presented in chapter 5 (because it didn't exist yet at this time), it is interesting to think about what would happen if the model was integrated with advanced numerical algorithms.

The problematic aspect in the solution of the differential equations in this model is the stochasticity of the light model. Typically Monte Carlo integration is used to solve the *measurement equation* (see [Vea97, section 3.7.1]). This introduces noise into the solution, which in this case is light distribution in the scene. For the classical Runge-Kutta method, Runge investigated the error propagation behaviour of this method [Run05]. This way, if the noise introduced by the light model is known, the noise in the result of the integration process can be estimated. Further continuing the idea leads to *stochastic differential equations* (SDEs). For the solution of such equations we refer to [KP92].

6.6. Turing Patterns

The reaction-diffusion systems that have been mentioned in section 5.1.2 can be solved using the ODE framework to obtain so-called *Turing Patterns*. An example, provided by Turing himself in [Tur52, §10] is a linear ring of twenty cells. Two morphogens X and Y are considered, and diffusion of these between neighbouring cells. In each cell, a set of reactions is assumed to take place simultaneously, namely:

$$\begin{array}{rl} Y + X \rightarrow W & \text{at the rate } \frac{25}{16} YX, \\ W + A \rightarrow 2Y + B & \text{instantly}, \\ 2X \rightarrow W & \text{at the rate } \frac{7}{64} X^2, \\ A \rightarrow X & \text{at the rate } \frac{1}{16} \times 10^{-3} A, \\ Y \rightarrow B & \text{at the rate } \frac{1}{16} Y, \\ Y + C \rightarrow C' & \text{instantly}, \\ C' \rightarrow X + C & \text{at the rate } \frac{55}{32} \times 10^{+3} C'. \end{array}$$

The rate laws are based on a period of 1000s as units of time, and 10^{-11} mol/cm^3 as concentration unit. The equations above basically describe conversion from a substrate A, initially existing at a very high concentration of 1000 units, over X and Y to some final form B. The conversion is catalyzed by C and its combined form C', with a concentration of $10^{-3}(1+\gamma)$, where γ ranges from -0.5 to 0.5.

Effectively, a net rate of $\frac{1}{32}[50XY + 7X^2 - 55(1+\gamma)]$ for the conversion from X to Y is obtained, while at the same time X is produced at a fixed rate of $\frac{1}{16}$ and Y is consumed at a rate of $\frac{1}{16}Y$. In case that the conversion rate from X to Y is negative (so basically a conversion from Y to X takes place) and the concentration of Y is already zero, the rate is properly limited to ensure that the concentration of Y does not become negative. The diffusion coefficients, using the same units, are $D_X = \frac{1}{2}$ and $D_Y = \frac{1}{4}$.

Although the same equations apply for each cell, random fluctuations in the initial conditions may move the system out from an initial metastable equilibrium to some more stable state, in this case stationary waves of final wave-length. The resulting pattern for morphogen Y is shown in figure 6.10. The initial concentrations were varying around the dashed line.

The XL program used to produce this pattern creates a ring structure in the graph, like the one shown in figure 6.11, by making use of bidirectional user-defined edges (of type `E`). Each node is a module with two attributes `x` and `y` to store the concentrations of the morphogens. The XL-rules that have been used to describe the reaction-diffusion system are shown in listing 6.3. As can be seen, formulation of the rate-law in XL is straight-forward.

Using the same reaction-diffusion system on a two-dimensional grid of 50 $\times$ 50 cells results in the pattern shown in figure 6.12. The connection between the cells forms a torus world. Orange cells show a dominance of morphogen X, while greenish ones a dominance of Y. The interesting aspect of the ODE framework when going from 1D to 2D is that only the initial setup of the cells must be changed (which is needed anyway), but the part of the program related to numerical integration did not change at all.

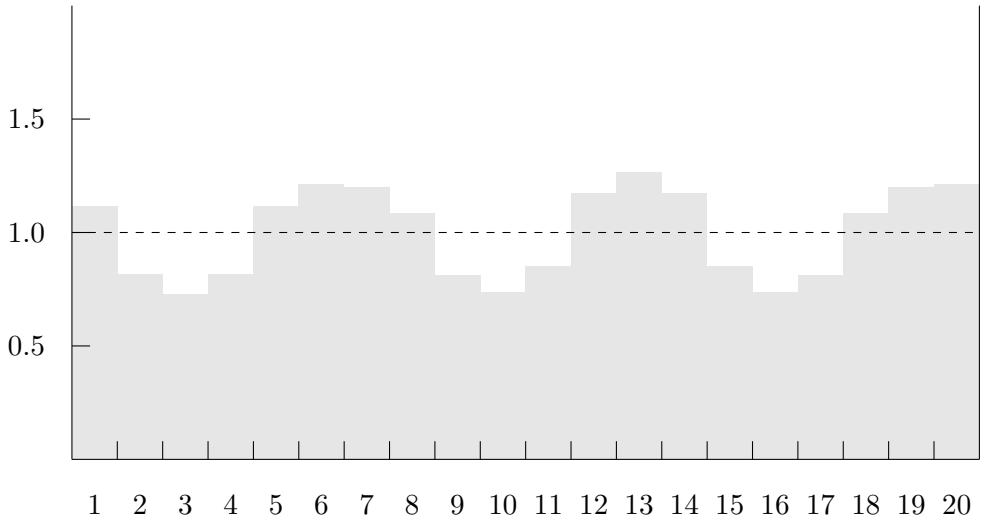


Figure 6.10.: Turing pattern produced by a reaction-diffusion system.

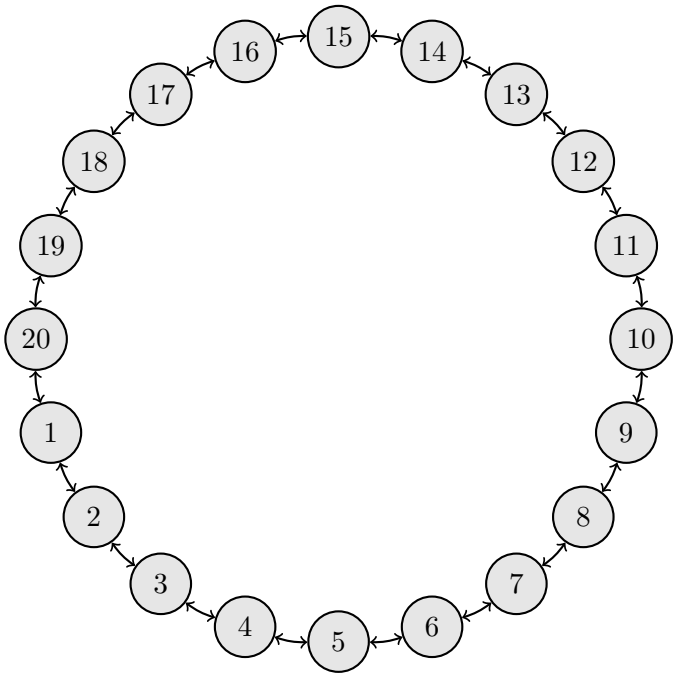


Figure 6.11.: Ring structure used to produce Turing pattern.

Listing 6.3: XL rules used to form Turing patterns.

```
1  // reaction
2  a:A(x,y) ::> {
3    double r = max((50*x*y + 7*x*x - 55*(1+gamma)) / 32, 0);
4    a[x] :'= 1.0/16.0 - r;
5    a[y] :'=            r - y / 16.0;
6  }
7  // diffusion
8  a:A <-E-> b:A, (a < b) ::> {
9    double rx = D_X * (a[x] - b[x]);
10   a[x] :'= -rx;
11   b[x] :'= +rx;
12   double ry = D_Y * (a[y] - b[y]);
13   a[y] :'= -ry;
14   b[y] :'= +ry;
15 }
```

Instead of a regular grid, one could also investigate how Turing patterns emerge on an arbitrary network. This idea has been pursued in [NM10].

6.7. PDE Solution

Let us recall the example of heat conduction in a metal stick from section 2.12.5. The PDE $u_t = u_{xx}$ models heat transport with an initial condition $u(x,0) = sin(x \cdot \pi/2)$ and boundary conditions $u(0,t) = 0$ and $u_x(L,t) = 0$.

The method of lines is applied to the stick with a spatial discretization into 21 points. Each point is represented in XL by a module `A` defined as

```
module A(double u) extends Box ==> setColor(u, 0, 0);
```

The attribute `u` stores the current value of the temperature for this segment, which is visualized by a shade of red set using an instantiation rule. For neighbouring segments the corresponding nodes are connected by successor edges.

Using finite differences and taking into account the boundary conditions, the following set of ODEs must be integrated:

$$\begin{aligned} \frac{du_0}{dt} &= 0, \\ \frac{du_i}{dt} &= \frac{u_{i-1} - 2u_i + u_{i+1}}{\Delta x^2}, \quad 0 < i < N \\ \frac{du_N}{dt} &= \frac{2 \cdot (u_{N-1} - u_N)}{\Delta x^2}, \end{aligned}$$

where N is the number of points, $\Delta x = L/N$ the spacing between the points, and u_i the temperature at point number i. A central finite difference approximation of second order has been used for all points, except those at the borders. For u_0 the first boundary

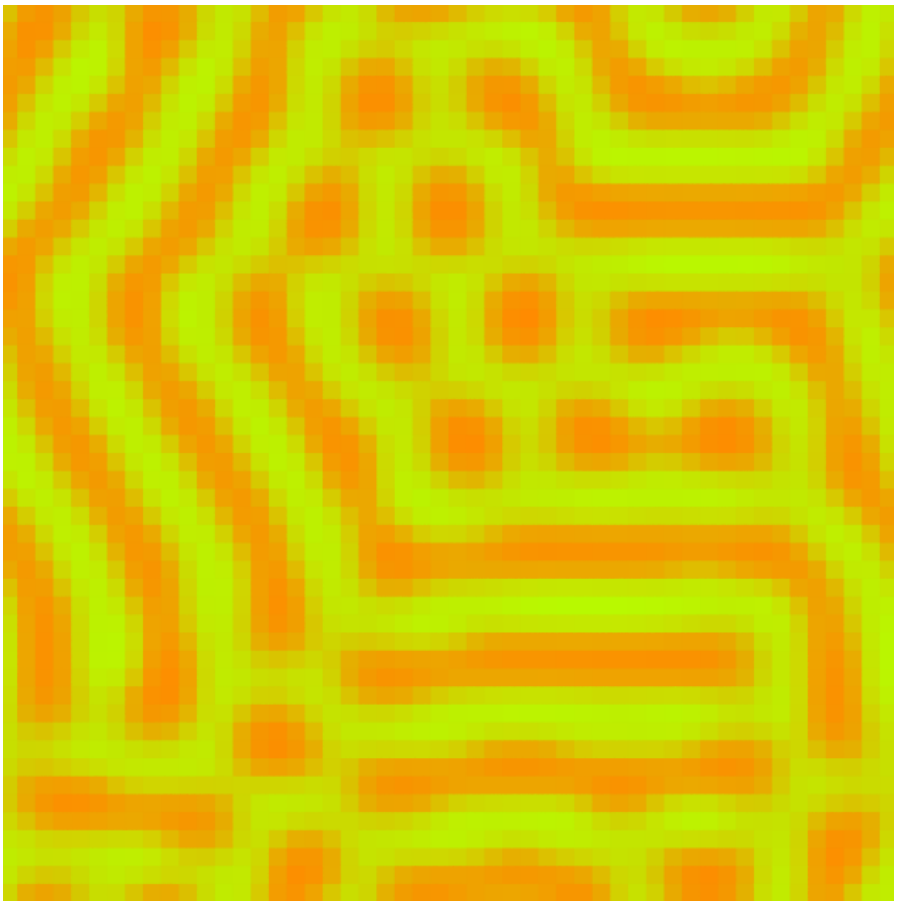


Figure 6.12.: Reaction-diffusion system in two dimensions. Amount of morphogen in each cell is indicated by RGB colors (red component for morphogen X, green component for morphogen Y).

Listing 6.4: XL rules used to apply method of lines to heat conduction.

```
a:A ::> {
  A l = first((* a < A *));
  A r = first((* a > A *));
  if (l == null) {
    // no change, keep at zero
  } else if (r == null) {
    a[u] :'= 2 * (l[u] - a[u]) / Δx²;
  } else {
    a[u] :'= (l[u] - 2*a[u] + r[u]) / Δx²;
  }
}
```

condition requires the value to remain zero, which is also true for its time-derivative. For the other end u_N of the stick we need the non-existent value u_{N+1}, which can be derived from the finite difference approximation of the second boundary condition

$$\frac{\partial u_N}{\partial x} \approx \frac{u_{N+1} - u_{N-1}}{2\Delta x} = 0$$

as $u_{N+1} = u_{N-1}$.

Listing 6.4 shows the equivalent formulation of the ODEs in XL. For each node `a` the left and right neighbour is searched in the graph. If the node was on one of the ends of the metal stick, such a neighbour is missing and this is handled accordingly.

Numerical integration of the model was done with an implicit Adams-Moulton integrator. The resulting time series is shown in figure 6.13. Note that explicit methods might have difficulties according to the CFL condition, especially if a finer discretization is used.

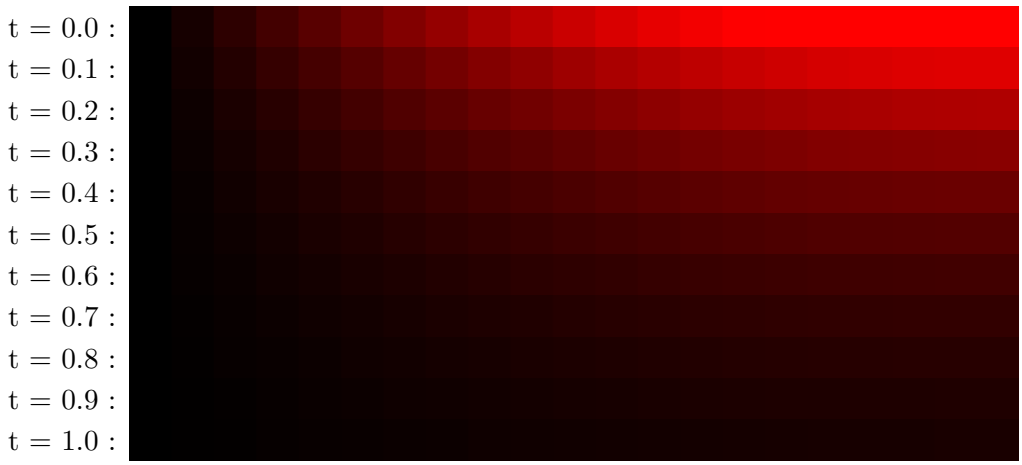


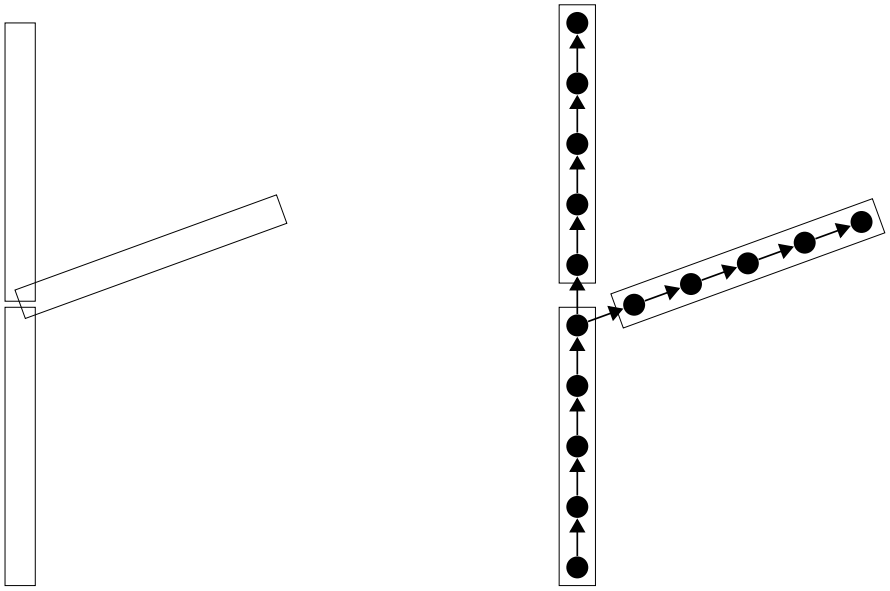
Figure 6.13.: Method of lines solution for a metal stick.

PDEs on Tree Structures

In the modelling of trees, which currently is one of the main application fields of

GroIMP/XL, segments of the trunk or the branches are often represented by cylinders as nodes in the graph. When accurate information about the spatial distribution of chemical substrates within the segments is needed, these segments must be discretized. This raises the question whether there is an automatic way to achieve this.

If we take a look at figure 6.14a we can see that the structure, as it is usually modelled, causes segments to overlap spatially. This causes problems if the discretization is one-dimensional along the main axis of each segment (see figure 6.14b), since in the overlapping region it must be decided for each point in space to which segment it should be assigned. A three-dimensional discretization would perhaps solve this issue, but now we have way more ODEs to integrate. Also it is not clear how the inner workings of the transport mechanism should be accounted for (i.e. transport pipes within branches).

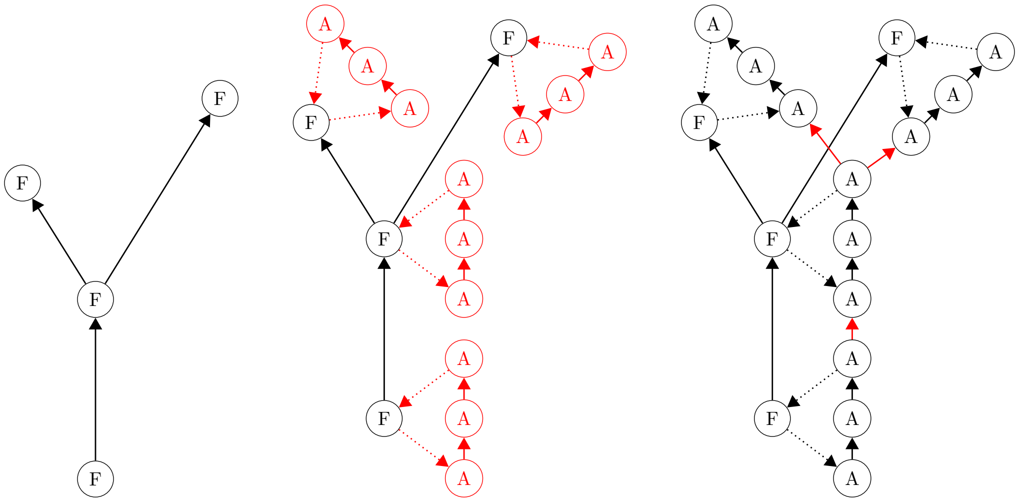


(a) overlapping segments (b) discretization

Figure 6.14.: Original tree structure (a) and a one-dimensional discretization (b) of it.

For these and other reasons, an automatic discretization of 3D structures is not feasible, thus the user should be responsible to provide a proper one. However, there should be some way to support the user in doing this. Since the topology is already available, the user can attach to it the discretized structure that is created by ordinary XL rules. This also gives freedom in interpretation of the topology, since this is fully under the user's control.

Each segment of the tree structure (cf. figure 6.15a) can be decomposed like it was done for the metal stick before, by creating a chain of nodes connected by unidirectional edges. These chains can then be linked to the original segment using some user-defined edge type, thus allowing to handle each segment separately in a first pass (see figure 6.15b. Then, in a second pass, these chains can be linked together according to the topology



(a) topology (b) discretized segments (c) discretizations connected

Figure 6.15.: Tree structure with attached spatial discretization.

(see figure 6.15c).

The program shown in listing 6.5 demonstrates how this idea can be implemented in XL. In `init` the structure from figure 6.15a is created. Then, in `discretize`, the first rule attaches a chain of user-defined nodes of type `A` to each segment `F`. The number of `A` nodes created could optionally depend on the length of the `F` segment to obtain a more regular discretization, but for sake of simplicity a fixed number of three `A`s was used here. Calling `derive` makes the modifications to the graph visible, so that the second rule can finally connect the chains according to the topology.

Usually, the structure created to represent the tree also contains other commands like rotations, translations, etc. and branches are connected by branch edges and not successor ones. So if the structure that ought to be discretized is created by this rule

```
Axiom ==> F F [RU(35) F(1.5)] RU(-30) F;
```

we have to slightly modify our discretization function to take care of this. The second rule would then be replaced by

```
(* a1:A -E-> f1:F (-(successor|branch)->)+ : (f2:F) -E-> a2:A *)
  ==>> a1 a2;
```

to ensure that `successor` and `branch` edges are properly skipped.

6.8. Performance Considerations

A discussion of the results would not be complete without a detailed look at how much overhead is introduced into the computation by using the ODE framework. Such an abstraction penalty may be viable if it is small compared to the total time needed.

Listing 6.5: Spatial discretization of a tree structure using XL rules.

```
1  const int E = EDGE_0;
2
3  module A(double u);
4
5  protected void init()
6  [
7    Axiom ==> F f:F F, f F;
8  ]
9
10 public void discretize()
11 [
12   // discretize each element individually
13   (* f:F *) ==>> f -E-> A A A -E-> f;
14   { derive(); }
15   // connect segments according to topology
16   (* a1:A -E-> f1:F f2:F -E-> a2:A *) ==>> a1 a2;
17 ]
```

Listing 6.6: Code to measure accuracy of `System.nanoTime()`.

```
1  int a = Integer.MAX_VALUE;
2  int b = Integer.MIN_VALUE;
3  for (int i = 0; i < 100000000; i++) {
4    long t0 = System.nanoTime();
5    long t1 = System.nanoTime();
6    int dt = (int)(t1 - t0);
7    a = min(a, dt);
8    b = max(b, dt);
9  }
```

For measuring execution time the Java standard library provides in the class `System` a command `currentTimeMillis`, and from version 1.5 for the language also a command `nanoTime`. The latter one returns the difference in nanoseconds between the current time and some arbitrary other time, perhaps even in the future. Although nanosecond precision is provided, this does not necessarily infer nanosecond accuracy, meaning that the same value might be returned by multiple successive calls until the returned value jumps to some higher value.

To get an idea about the time it takes to execute `nanoTime` and the accuracy obtained for the timing measurements the code in listing 6.6 was executed. In a loop many calls to the function are performed in succession. The value `a` captures the smallest time delta, while `b` the greatest one. On the system used to perform timing measurements both values were 26, indicating that a call to `nanoTime` takes 26ns to execute.

To measure the impact of the overhead on some example, the model of *Anabaena*

Listing 6.7: Code to measure execution time for a piece of code.

```
1 long s = 0;
2 ...
3 // start measurement
4 long t = System.nanoTime();
5   // execute some code here
6   ...
7 // stop measurement
8 s += System.nanoTime() - t;
```

Counter	Value	Description
time	600.069	Simulated time (variable `time`)
s0	3645.439ms	total time spent in `integrate`
s1	1697.772ms	total time spent in `getRate`
s2	1406.324ms	total time for first rule (outer)
s3	61.553ms	total time for first rule (inner)
s4	285.431ms	total time for second rule (outer)
s5	19.926ms	total time for second rule (inner)

Table 6.2.: Final simulated time and total execution times for different parts of the *Anabaena catenula* model. The rate function was called 10854 times, and `run` was called 663 times.

catenula from section 6.2 was used, with its complete code in appendix A.2.2. Timing instructions (see listing 6.7) bracket the code in question to find out how long its execution takes. The variable `t` stores the starting time, which is needed to compute a time delta later on. All time deltas are summed over multiple iterations to get a more accurate estimate of the time needed.

In the example, timings were taken for the call to `integrate(1)`, the rate function, and for each of the two rules. To distinguish between the time needed to find matches for the given patterns and actual computation and assignment of the rates times were taken outside the rule and inside the code block on the right-hand side of the rule. The obtained measurements as time series plot (over developmental age) are shown in figure 6.16, and the final values of the measurements in table 6.2.

As can be seen in figure 6.16, the accumulated times increase exponentially as is expected, since the number of simulated cells increases exponentially and execution time directly depends on this. It can also be seen that for the first (curves s2/s3) most time is consumed to search for the queried patterns, while only a negligible time is needed to actually compute and apply the rates. The situation is different for the second rule (curves s4/s5), as here the search is optimized by iterating over the corresponding extent. Still, for both rules the time needed to find a match is much higher than the time needed to compute and apply the rate.

A similar conclusion can be drawn from the timings presented in table 6.2. Still, there

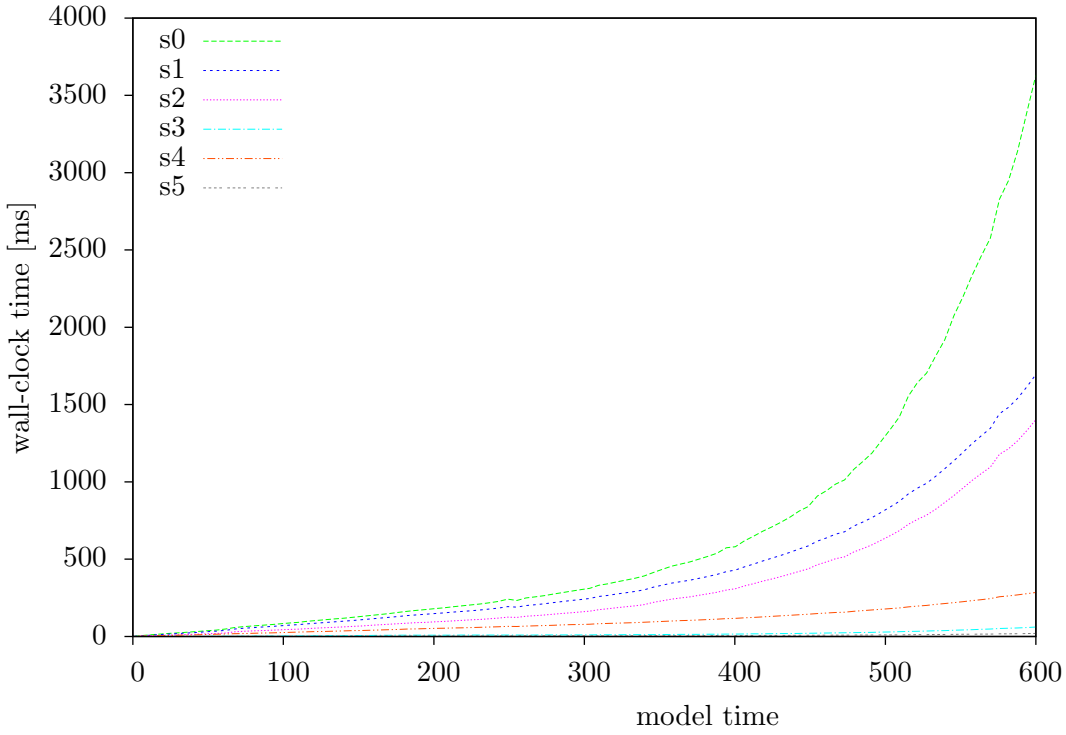


Figure 6.16.: Execution times (in ms) measured for *Anabaena catenula* model, plotted against model time. Curve s0 is the total time spent in `integrate`, s1 total time spent in `getRate`, s2/s3 total times for first rule (outer/inner), and analogously s4/s5 for second rule.

is a big discrepancy between the total time spent in the `integrate` function and the rate function, so an explanation is needed where the other half of the time went to, especially how much time can be attributed to the handling of monitor functions.

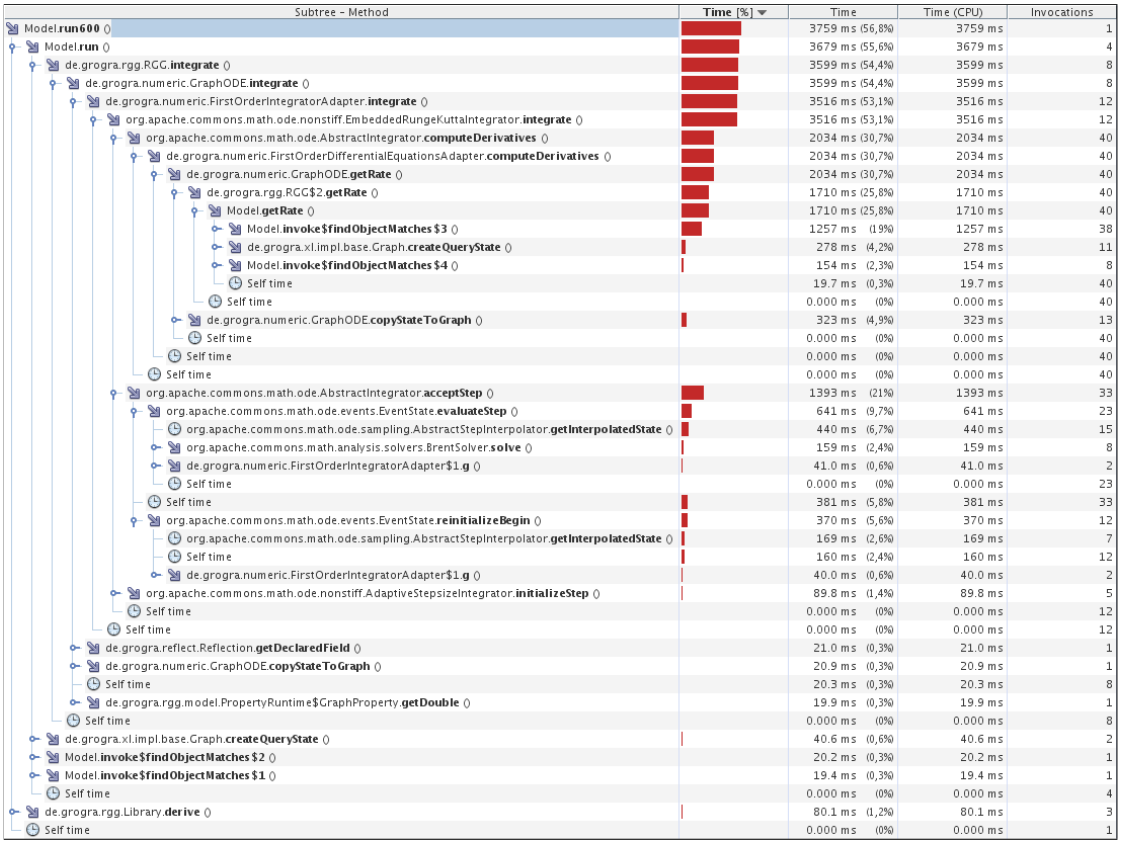
The JDK ships with a nice tool called *Java VisualVM*, which allows to monitor the execution of a running Java application. There are two modes of operation, a sampler and a profiler. The sampler peeks in regular intervals which method is currently being executed, thus long running methods or those that are called frequently will be sampled often, and thus the sample count is proportional to the execution time for this method. Fortunately, the running time of the application is almost unaffected, but it may take many samples to get accurate results. The profiler instead introduces timing instructions into the application code for every method, so execution speed of the application is slowed down drastically, but more accurate measurements are obtained in exchange.

Application of the sampler to the original model code (without `nanoTime` instructions and visualization) over a model time of 600 time units yields the measurements shown in figure 6.17. The sampling period was set to the lowest selectable value of 20ms (corresponds to 50Hz). Since for a wall-clock time of 3.6s there are just 180 samples, a second measurement over a model time of 800 time units (corresponds to 72s wall-clock time, or 3600 samples) was taken (figure 6.18).

In both cases, the time that was not spent in evaluation of rate assignments can be attributed to step handling, and there to step interpolation (`getInterpolatedState`). This is more pronounced for the second measurement over 800 time units of model time. The time needed to evaluate the monitor functions (method `g` and also part of the time needed to execute `BrentSolver.solve`), as well as the time needed for event handling (method `handleEvent`) is only a fraction of the time the integrator needs for step interpolation.

All in all, this is a good result, since it indicates that the additional overhead introduced by the ODE framework is only very little. The main contribution to the execution time of the rate function is the time needed to search patterns in the graph, which is needed anyways and thus cannot be counted. But by using the ODE framework it is possible to select more advanced integration methods that obtain the same accuracy with much fewer calls to the rate function, thus amortizing the overhead of copying the state to the graph and the indirection of the rate assignment operator. Similarly, the overhead introduced by monitor functions is small compared to the remaining computations the integration method performs.

Thus the versatility of working on a graph structure instead of an array more than outweighs the additional overhead introduced by the framework.



Subtree – Method	Time [%]	Time	Time (CPU)	Invocations
Model.**run600** ()		3759 ms (56,8%)	3759 ms	1
Model.**run** ()		3679 ms (55,6%)	3679 ms	4
de.grogra.rgg.RGG.**integrate** ()		3599 ms (54,4%)	3599 ms	8
de.grogra.numeric.GraphODE.**integrate** ()		3599 ms (54,4%)	3599 ms	8
de.grogra.numeric.FirstOrderIntegratorAdapter.**integrate** ()		3516 ms (53,1%)	3516 ms	12
org.apache.commons.math.ode.nonstiff.EmbeddedRungeKuttaIntegrator.**integrate** ()		3516 ms (53,1%)	3516 ms	12
org.apache.commons.math.ode.AbstractIntegrator.**computeDerivatives** ()		2034 ms (30,7%)	2034 ms	40
de.grogra.numeric.FirstOrderDifferentialEquationsAdapter.**computeDerivatives** ()		2034 ms (30,7%)	2034 ms	40
de.grogra.numeric.GraphODE.**getRate** ()		2034 ms (30,7%)	2034 ms	40
de.grogra.rgg.RGG$2.**getRate** ()		1710 ms (25,8%)	1710 ms	40
Model.**getRate** ()		1710 ms (25,8%)	1710 ms	40
Model.**invoke$findObjectMatches$3** ()		1257 ms (19%)	1257 ms	38
de.grogra.xl.impl.base.Graph.**createQueryState** ()		278 ms (4,2%)	278 ms	11
Model.**invoke$findObjectMatches$4** ()		154 ms (2,3%)	154 ms	8
Self time		19.7 ms (0,3%)	19.7 ms	40
Self time		0.000 ms (0%)	0.000 ms	40
de.grogra.numeric.GraphODE.**copyStateToGraph** ()		323 ms (4,9%)	323 ms	13
Self time		0.000 ms (0%)	0.000 ms	40
Self time		0.000 ms (0%)	0.000 ms	40
Self time		0.000 ms (0%)	0.000 ms	40
org.apache.commons.math.ode.AbstractIntegrator.**acceptStep** ()		1393 ms (21%)	1393 ms	33
org.apache.commons.math.ode.events.EventState.**evaluateStep** ()		641 ms (9,7%)	641 ms	23
org.apache.commons.math.ode.sampling.AbstractStepInterpolator.**getInterpolatedState** ()		440 ms (6,7%)	440 ms	15
org.apache.commons.math.analysis.solvers.BrentSolver.**solve** ()		159 ms (2,4%)	159 ms	8
de.grogra.numeric.FirstOrderIntegratorAdapter$1.**g** ()		41.0 ms (0,6%)	41.0 ms	2
Self time		0.000 ms (0%)	0.000 ms	23
Self time		381 ms (5,8%)	381 ms	33
org.apache.commons.math.ode.events.EventState.**reinitializeBegin** ()		370 ms (5,6%)	370 ms	12
org.apache.commons.math.ode.sampling.AbstractStepInterpolator.**getInterpolatedState** ()		169 ms (2,6%)	169 ms	7
Self time		160 ms (2,4%)	160 ms	12
de.grogra.numeric.FirstOrderIntegratorAdapter$1.**g** ()		40.0 ms (0,6%)	40.0 ms	2
org.apache.commons.math.ode.nonstiff.AdaptiveStepsizeIntegrator.**initializeStep** ()		89.8 ms (1,4%)	89.8 ms	5
Self time		0.000 ms (0%)	0.000 ms	12
Self time		0.000 ms (0%)	0.000 ms	12
de.grogra.reflect.Reflection.**getDeclaredField** ()		21.0 ms (0,3%)	21.0 ms	1
de.grogra.numeric.GraphODE.**copyStateToGraph** ()		20.9 ms (0,3%)	20.9 ms	1
Self time		20.3 ms (0,3%)	20.3 ms	8
de.grogra.rgg.model.PropertyRuntime$GraphProperty.**getDouble** ()		19.9 ms (0,3%)	19.9 ms	1
Self time		0.000 ms (0%)	0.000 ms	8
de.grogra.xl.impl.base.Graph.**createQueryState** ()		40.6 ms (0,6%)	40.6 ms	2
Model.**invoke$findObjectMatches$2** ()		20.2 ms (0,3%)	20.2 ms	1
Model.**invoke$findObjectMatches$1** ()		19.4 ms (0,3%)	19.4 ms	1
Self time		0.000 ms (0%)	0.000 ms	4
de.grogra.rgg.Library.**derive** ()		80.1 ms (1,2%)	80.1 ms	3
Self time		0.000 ms (0%)	0.000 ms	1

Figure 6.17.: Sampling results of the *Anabaena catenula* model for a model time of 600 time units.

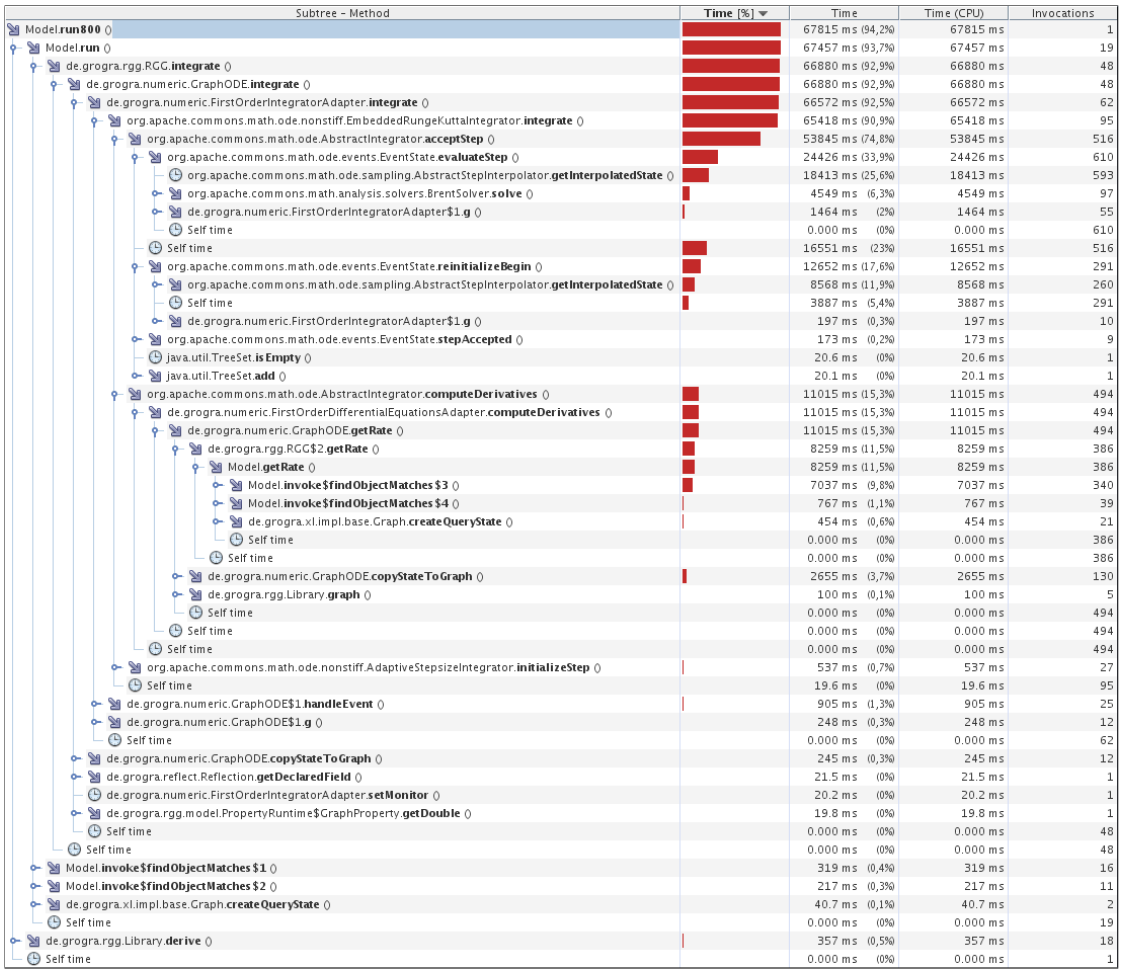


Figure 6.18.: Sampling results of the *Anabaena catenula* model for a model time of 800 time units.

7. Summary

The software GroIMP is an interesting tool for the creation of functional-structural models, especially of plants, and its programming language XL provides a powerful combination of the imperative and rule-based paradigm. However, rule-application is discrete by definition, while the differential equations used to describe functional aspects of a model are continuous.

The thesis proposes a way how to bridge this gap between the discrete and continuous world. A first solution to overcome this discrepancy was the extension of the language by operator overloading in combination with user-defined implicit conversion functions (autoconversions, which naturally extend Java's autoboxing). As was shown in chapter 4 this provides for some useful applications. Based on ideas learned from expression templates, properly overloaded operators allow chemical reactions to be entered as ordinary XL code, which can be analyzed to derive differential equations for numerical integration. The operator overloading approach is so flexible that even productions of a rule can be explained by it.

For dynamically growing structures, memory management becomes an important issue. As existing numerical libraries work on arrays, a mapping between the node attributes in the graph and the elements of these arrays must be created. Doing this manually is tedious and error-prone, so a second solution introduces a new operator symbol that allows the user to indicate which attributes should be integrated and how to compute their slope. This simplifies the specification of differential equations for the user and is even close to what users unintentionally wrote before, but with the added benefit that better numerical solvers can be used and also can be easily exchanged.

Some examples demonstrate the usage of the ODE framework and how the numerical solution improves. It was also shown how dL-systems can be directly transformed to equivalent XL code and then be solved on a graph. Turing patterns, which frequently occur in nature in many animals and plants, can also easily be described in XL and solved with help of the new framework. Examples of integration on a one-dimensional and a two-dimensional lattice are provided.

PDEs frequently occur when simulating natural phenomena. The method of lines is a common approach used to solve such PDEs numerically, but requires a spatial discretization of the geometry. How this can be done easily with the help of rules in XL has been shown.

7.1. Outlook

There are still many things that could be done to further enhance the framework. For instance, in physics often second-order ODEs arise in the description of motion. Taking

into account the second derivative along with the first one allows to develop methods with reduced computational effort while maintaining accuracy, as was already observed by Nyström in [Nys25]. In principle, a new operator `:''=` could be introduced that allows specification of the second derivative, but it must be investigated how this fits into the existing framework.

Implicit methods that perform Newton iterations for root finding need the Jacobian, which describes how changes in the input parameters affect the output parameters. The coefficients can be derived automatically by a finite difference approximation, but it can be much more efficient if the user can provide this matrix. Thus adding such a feature to the framework is beneficial.

Often, biological models combine processes that occur on different spatial and temporal scales. For instance, leaf formation is a matter of days for some trees, while for the same trees growth is modelled on a per-year basis instead. Numerical integration of such combined models can be problematic, as fast processes need a sufficiently small time step for the numerical algorithm to remain appropriately stable and accurate, but which for the slow processes results in a waste of computation at the same time. It should be researched how this issue can be solved robustly, for instance, by separately integrating slow and fast processes. This idea has been addressed previously in the DEVS framework [CJC10], but it would be interesting to also investigate how this can be combined with the ODE framework presented in this thesis.

With the upcoming emergence and availability of multi-core CPUs a natural question is how the framework can make use of this. Splitting a model into independent submodels, as was already mentioned, would be one way to do this. Another way to increase parallelism would be to distribute (parallel) rule applications to multiple cores, or even computers, as was already suggested in [Kni08]. Also, implicit integration methods that need to solve a system of linear equations can benefit when this task is performed in a distributed fashion (e.g., SUNDIALS allows for this).

Going one step further one might ask if GPUs can accelerate the computation as well. Since GroIMP is inherently dependent on Java, large parts of the runtime system probably need to be reimplemented in terms of a GPU language (CUDA or OpenCL) or a transformation of the Java bytecode to GPU instructions has to be performed. Still a challenge remains of how to map the graph structure to an appropriate data structure for the GPU and how this might look like, since this is critical to obtain an efficient GPU implementation.

A. Example Models with XL-Code

A.1. Chemical Kinetics with Operator Overloading

A.1.1. Model Code for the Reaction A $\xrightarrow{k_1}$ B $\xrightarrow{k_2}$ C

```
1  import de.grogra.chem.*;
2  import de.grogra.numeric.*;
3  import java.util.*;
4  import org.apache.commons.math.ode.nonstiff.*;
5  import static de.grogra.chem.ChemicalOperators.*;
6  import static java.lang.Math.*;
7
8  const Molecule A = new Molecule("A");
9  const Molecule B = new Molecule("B");
10 const Molecule C = new Molecule("C");
11
12 const DatasetRef fnc = dataset("Function");
13
14 protected void init()
15 {
16   fnc.clear().setTitle("")
17     .setColumnKey(0,"A")
18     .setColumnKey(1,"B")
19     .setColumnKey(2,"C");
20   chart(fnc, XY_PLOT);
21 }
22
23 public void run()
24 {
25   ChemicalReaction r1 = A <=> B;
26   r1.setForwardRateConstant(2);
27
28   final Model model = new Model();
29   model.addSlope(r1);
30   model.add(B <=> C, 1);
31
32   final HashMap m = new HashMap();
33   int count = model.assignIndices(0, m);
34
35   double[] y0 = new double[count];
36   double[] y = new double[count];
37
```

```
38    // set initial conditions
39    setValue(m, y0, A, 10);
40    plot(0, y0[idx(m, A)], y0[idx(m, B)], y0[idx(m, C)]);
41
42    // prepare differential equations
43    ODE ode = new ODE()
44    {
45      public void getRate(double[] out, double t, double[] y)
46      {
47        Arrays.fill(out, 0);
48        model.eval(out, t, y);
49      }
50    };
51
52    // perform numerical integration and plot data
53    Solver solver = new FirstOrderIntegratorAdapter(
54      new ClassicalRungeKuttaIntegrator(0.001));
55    solver.setMonitor(1, new Monitor()
56    {
57      public void g(double[] out, double t, double[] y)
58      {
59        out[0] = sin(PI * t * 20);
60      }
61      public boolean handleEvent(int i, double t, double[] y)
62      {
63        plot(t, y[idx(m, A)], y[idx(m, B)], y[idx(m, C)]);
64        return false;
65      }
66    });
67    solver.integrate(ode, 0, y0, 5, y);
68  }
69
70  void plot(double t, double a, double b, double c)
71  {
72    fnc.addRow()
73      .setX(0, t).setY(0, a)
74      .setX(1, t).setY(1, b)
75      .setX(2, t).setY(2, c);
76  }
77
78  int idx(Map m, Object obj)
79  {
80    Integer index = (Integer)m.get(obj);
81    return index != null ? index.intValue() : -1;
82  }
83
84  void setValue(Map m, double[] array, Object obj,
85    double value)
86  {
```

```
87    Integer index = (Integer)m.get(obj);
88    if (index != null)
89    {
90      array[index] = value;
91    }
92  }
```

A.1.2. Model Code for Michaelis-Menten Kinetics

```
1   import de.grogra.chem.*;
2   import de.grogra.numeric.*;
3   import java.util.*;
4   import org.apache.commons.math.ode.nonstiff.*;
5   import static de.grogra.chem.ChemicalOperators.*;
6   import static java.lang.Math.*;
7
8   const Molecule E = new Molecule("E");
9   const Molecule S = new Molecule("S");
10  const Molecule ES = new Molecule("ES");
11  const Molecule P = new Molecule("P");
12
13  const DatasetRef fnc = dataset("Function");
14
15  protected void init()
16  {
17    fnc.clear().setTitle("")
18          .setColumnKey(0,"E")
19          .setColumnKey(1,"S")
20          .setColumnKey(2,"ES")
21          .setColumnKey(3,"P");
22    chart(fnc, XY_PLOT);
23  }
24
25  public void run()
26  {
27    final Model model = new Model();
28    model.add(E + S <=> ES, 3, 0.1);
29    model.add(ES <=> E + P, 2);
30
31    final HashMap m = new HashMap();
32    int count = model.assignIndices(0, m);
33
34    double[] y0 = new double[count];
35    double[] y = new double[count];
36
37    // set initial conditions
38    setValue(m, y0, E, 3);
39    setValue(m, y0, S, 10);
40    plot(0, y0[idx(m, E)], y0[idx(m, S)],
41      y0[idx(m, ES)], y0[idx(m, P)]);
```

```
42
43    // prepare differential equations
44    ODE ode = new ODE()
45    {
46      public void getRate(double[] out, double t, double[] y)
47      {
48        Arrays.fill(out, 0);
49        model.eval(out, t, y);
50      }
51    };
52
53    // perform numerical integration and plot data
54    Solver solver = new FirstOrderIntegratorAdapter(
55      new ClassicalRungeKuttaIntegrator(0.001));
56    solver.setMonitor(1, new Monitor()
57    {
58      public void g(double[] out, double t, double[] y)
59      {
60        out[0] = sin(PI * t * 20);
61      }
62      public boolean handleEvent(int i, double t, double[] y)
63      {
64        plot(t, y[idx(m, E)], y[idx(m, S)],
65          y[idx(m, ES)], y[idx(m, P)]);
66        return false;
67      }
68    });
69    solver.integrate(ode, 0, y0, 5, y);
70  }
71
72  void plot(double t, double e, double s, double es, double p)
73  {
74    fnc.addRow()
75      .setX(0, t).setY(0, e)
76      .setX(1, t).setY(1, s)
77      .setX(2, t).setY(2, es)
78      .setX(3, t).setY(3, p);
79  }
80
81  int idx(Map m, Object obj)
82  {
83    Integer index = (Integer)m.get(obj);
84    return index != null ? index.intValue() : -1;
85  }
86
87  void setValue(Map m, double[] array, Object obj,
88    double value)
89  {
90    Integer index = (Integer)m.get(obj);
```

```
91    if (index != null)
92    {
93      array[index] = value;
94    }
95  }
```

A.2. dL-systems with GroIMP/XL

A.2.1. Model Code for Dragon Curve

```
1  import static java.lang.Math.sqrt;
2
3  module FR(double x, double s) extends F(x, 0.05);
4  module FL(double x, double s) extends F(x, 0.05);
5  module FH(double x, double s) extends F(x, 0.05);
6
7  const double ANGLE = 45;
8  const double T = 1;
9
10 protected void init()
11 [
12   Axiom ==> RU(2*ANGLE) FR(0, 1);
13 ]
14
15 public void run()
16 [
17   fr:FR ::> monitor(void=>double fr[x] - fr[s],
18     new Runnable() {
19       public void run() [
20         fr ==>
21           RU(ANGLE) FR(0, fr[s] * sqrt(2)/2)
22           RU(-ANGLE) FH(fr[s], fr[s])
23           RU(-ANGLE) FL(0, fr[s] * sqrt(2)/2)
24           RU(ANGLE);
25       ]
26     });
27
28   fl:FL ::> monitor(void=>double fl[x] - fl[s],
29     new Runnable() {
30       public void run() [
31         fl ==>
32           RU(-ANGLE) FR(0, fl[s] * sqrt(2)/2)
33           RU(ANGLE) FH(fl[s], fl[s])
34           RU(ANGLE) FL(0, fl[s] * sqrt(2)/2)
35           RU(-ANGLE);
36       ]
37     });
38
39   fh:FH ::> monitor(void=>double fh[x], new Runnable() {
```

```
40      public void run() [
41        fh ==> ;  // delete node
42      ]
43    });
44
45    { integrate(0.125); }
46
47    // visualize curve
48    fr:FR ::> fr[length] = fr[x];
49    fl:FL ::> fl[length] = fl[x];
50    fh:FH ::> fh[length] = fh[x];
51  ]
52
53  protected void getRate()
54  [
55    fr:FR ::> fr[x] :'=  fr[s]/T;
56    fl:FL ::> fl[x] :'=  fl[s]/T;
57    fh:FH ::> fh[x] :'= -fh[s]/T;
58  ]
```

A.2.2. Model Code for Anabaena Catenula

```
1  import static java.lang.Math.*;
2
3  module F(double x, double c) extends Cylinder(x, 2.5);
4  module FH(super.x, super.c) extends F(x, c);  // heterocyst
5  module FV(super.x, super.c) extends F(x, c);  // vegetative
6
7  // parameters from:
8  // P. Federl, P. Prusinkiewicz (2004) Solving differential
9  // equations in developmental models of multicellular
10 // structures expressed using L-systems.
11 // ICCS 2004, Springer, LNCS 3037, pp. 65--72
12 const double X_MAX = 1;
13 const double C_MAX = 255;
14 const double C_MIN = 5;
15 const double R_X = 0.1;
16 const double R_C = 0.15;
17 const double R = 0.01;
18 const double D = 0.03;
19 const double K = 0.37;
20 const double MU = 0.03;
21
22 protected void init ()
23 [
24   Axiom ==> RU(90)
25     FH(0, C_MAX) FV(0.9*X_MAX, C_MAX) FH(0, C_MAX);
26 ]
27
28 public void run ()
```

```
29  [
30    // set monitor for reaching maximum length
31    fv:FV ::> monitor(void=>double fv[x] - X_MAX,
32      new Runnable() {
33        public void run() [
34          fv ==> FV(K*X_MAX, fv[c]) FV((1-K)*X_MAX, fv[c]);
35        ]
36      });
37    // set monitor for reaching minimum concentration
38    fv:FV ::> monitor(void=>double fv[c] - C_MIN,
39      new Runnable() {
40        public void run() [
41          fv ==> FH(fv[x], fv[c]);
42        ]
43      });
44
45    // perform integration over at most one time unit
46    // integration may halt earlier if one of the set
47    // monitors triggers
48    { integrate(1); }
49
50    // perform visualization
51    f:F(x, c) ::> {
52      f[length] = f[x];
53      f.setColor(
54        java.awt.Color.HSBtoRGB((1 - c / C_MAX) / 6, 1, 1));
55    }
56  ]
57
58  protected void getRate()
59  [
60    F(xl, cl) fv:FV(x, c) F(xr, cr) ::> {
61      fv[x] :'= R * x;
62      fv[c] :'= D * (cl - 2*c + cr) - MU * c;
63    }
64    fh:FH(x, c) ::> {
65      fh[x] :'= R_X * (X_MAX - x);
66      fh[c] :'= R_C * (C_MAX - c);
67    }
68  ]
```

B. Other Contributions to GroIMP

B.1. Improved 3D-View for GroIMP using OpenGL

The time when the work on this thesis started, GroIMP's visualization capabilities consisted of an interactive software renderer based on AWT (wireframe canvas) and an export filter to render the scene with POV-Ray[1]. At that time, also a path tracer was in development, which is now known as "Twilight".

While the interactive view using a software renderer is beneficial, especially if GroIMP is to be run on a CPU cluster where this usually is the only means of interactive preview capabilities, it does not make use of any hardware acceleration provided by GPUs. Therefore, the idea was born to add an additional OpenGL-based 3D view with a more convincing representation of the current scene.

A strict requirement for this was that it should be usable in combination with any OpenGL-compatible graphics card, even on older systems. Also all graphical primitives supported by GroIMP (like box, sphere, NURBS, etc.) should be supported by the new view. Shading of the objects according to the light sources in the scene should improve realism and help the user to setup the scene for rendering with "Twilight" more easily.

For compatibility reasons the choice was made to only use features provided by OpenGL 1.1. This version is supported by Windows systems out of the box and meets the requirements mentioned above. To access OpenGL from Java the JOGL[2] library is used.

The resulting 3D view handles as many lights as are supported by the current OpenGL implementation (at least 8). For some objects (like sphere or cylinder) a level of detail (LOD) was implemented, so that a triangle of the object covers roughly the same number of pixels independently from the object's distance. So if an object is farther away fewer triangles will be used to approximate its shape.

Another improvement over the wireframe canvas is that surfaces can be textured. This allows to represent the leaves of a tree simply by using an image of a real leaf as texture for a rectangle. The transparency in this image is then used to approximate the shape of the leaf, which is also respected by GroIMP's raytracers (especially for the shadow created by a leaf).

With the advent of freely programmable GPUs and its wide availability in consumer graphics cards the wish for an even more realistic interactive visualization of the scene emerged. The OpenGL Shading Language (GLSL), originally introduced as an extension to OpenGL 1.4 and since OpenGL 2.0 part of the core specification, promised to allow

[1] `http://www.povray.org`, accessed 2nd February 2012

[2] formerly `http://java.net/projects/jogl`, currently `http://jogamp.org/jogl/www`, both accessed 2nd February 2012

the interactive evaluation of GroIMP's procedural shaders. Before, these had to be statically rendered into a texture for preview.

Other improvements over the previous OpenGL view are pixel-accurate lighting computations (Phong shading instead of Gouraud shading), deferred shading (so each pixel is shaded only once), shadows, transparency (via depth peeling), and high dynamic range (HDR) rendering with proper subsequent tone mapping. The advanced OpenGL view was implemented by Konni Hartmann as part of his bachelor's thesis [Har10]. A comparison of the three interactive views is shown in figure B.1.

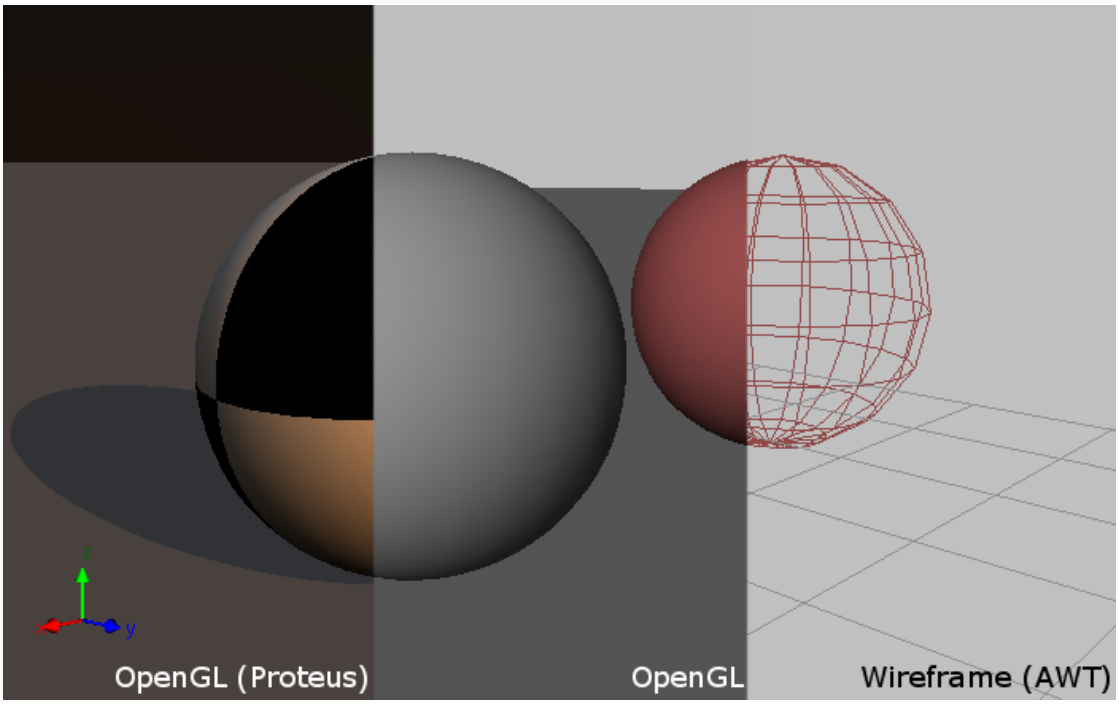


Figure B.1.: Comparison of the different 3D views in GroIMP. From left to right: advanced OpenGL view, normal OpenGL view, software renderer.

B.2. PDB Import Filter

The PDB file format allows to describe molecules, even complex ones like proteins. A collection of many such molecules can be found in the protein data bank[3] and their PDB files can be downloaded freely. The PDB plugin allows to import such files into GroIMP and visualize them. Examples are a space filled visualization of a cyclohexane molecule (figure B.2) and a ball stick representation of some protein.

The PDB plugin is based on Java code written by Fabian Dill and Matthias Dube

[3] `http://www.pdb.org`, accessed 8 Februrary 2012

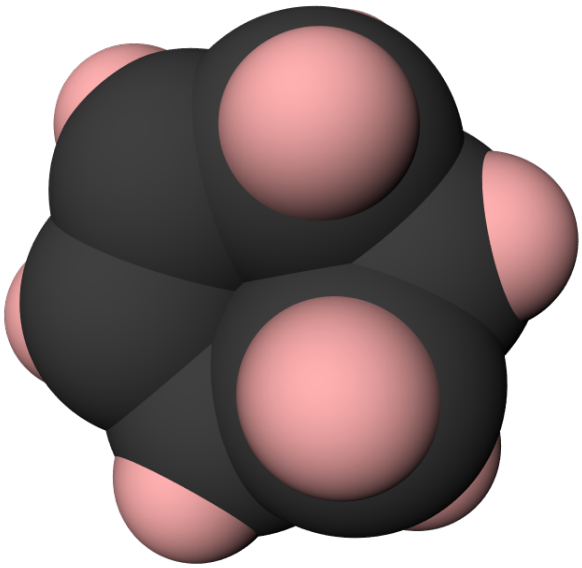


Figure B.2.: Space filled visualization of cyclohexane (imported from PDB file).

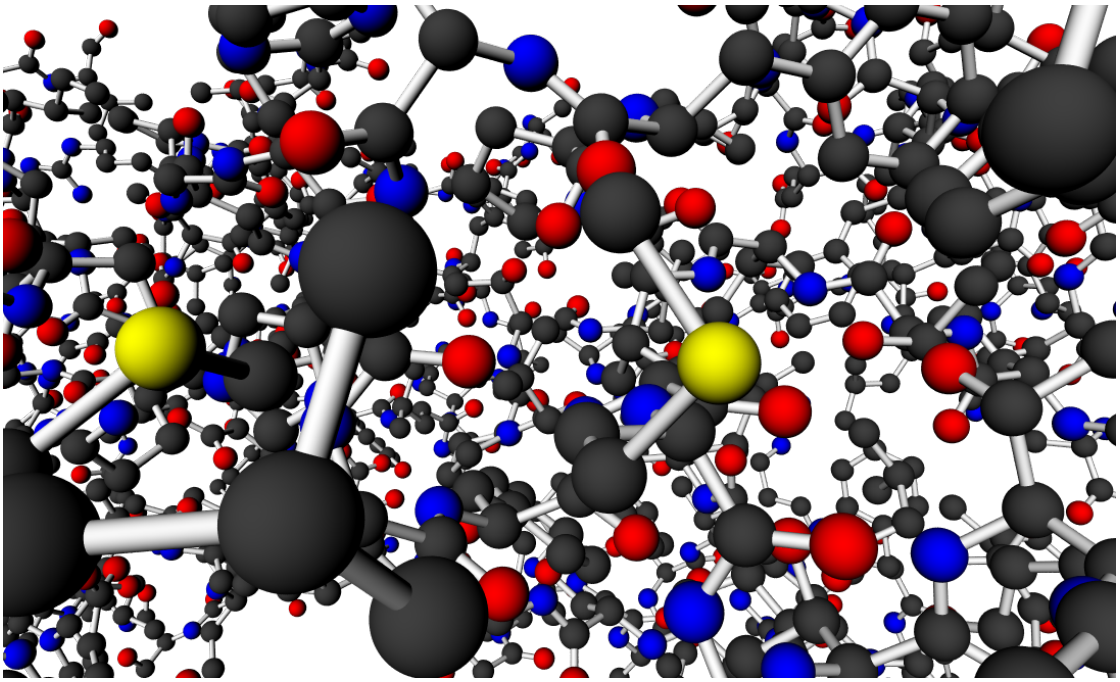


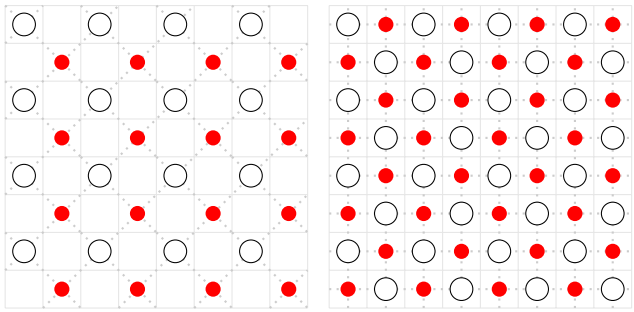
Figure B.3.: Ball stick visualization of some protein (imported from PDB file).

during the course "Datenanalyse und Visualisierung in der Bioinformatik", subtopic "Visualisierung von Proteinstrukturen" in summer semester 2004 for reading PDB files.

B.3. Synthetic Texture Generation

For the creation of virtual plant models it is sometimes useful to also create procedurally generated landscapes. An algorithm that was devised produce artificial landscapes is the so-called Carpenter's algorithm [FFC82], often also called diamond-square algorithm. The latter name also very tellingly describes its operation.

The goal is to create an image that represents the landscape by storing one height value per pixel. Initially, only the four corners are set. The algorithm subdivides the grid by alternately creating a center point in each square of already computed points (diamond step, figure B.4a), then creating a center point in each diamond of the resulting points (square step, figure B.4b). In both cases the value for the new point is the average of the four generating points plus some random offset. A landscape that was generated with this algorithm is shown in figure B.5.



(a) diamond step (b) square step

Figure B.4.: Graphical illustration of the diamond-square algorithm.

Autoregressive models provide another means to create procedural textures [PdSPK01]. A new pixel is created from the previously generated neighbourhood (figure B.6) according to the formula

$$x_{t+1} = \sum_{i=0}^{3} a_i x_{t-i} + \nu_{t+1}$$

with ν_t some additional white noise for pixel $t+1$. A texture generated by this method is shown in figure B.7.

B.4. Import/Export Filters for DXF and OBJ

During the seminar "Artifizielle Wachstumsprozesse" in the winter semester 2006/07 at the BTU Cottbus, students of architecture investigated how GroIMP can be used

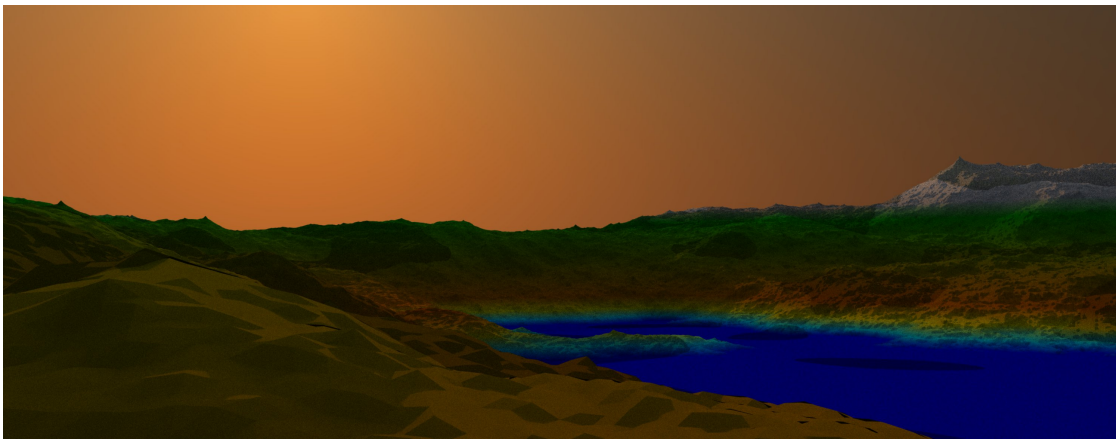


Figure B.5.: Procedural landscape generated with Carpenter's algorithm.

a_0	a_1	a_2
a_3		

Figure B.6.: Neighbourhood used for autoregressive texture generation.

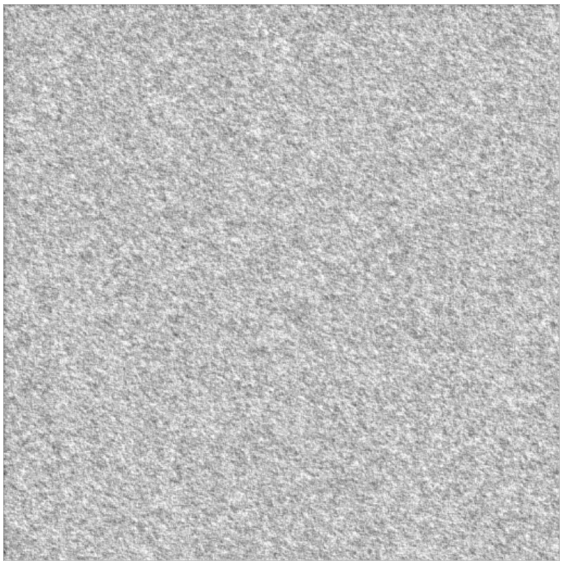


Figure B.7.: Texture created by autoregressive model.

to create (proto-)architectural structures. The intention was that the observer of such structures should be aware that there was some algorithmical processes used to generate the structure, but it should not be obvious what this process was alike. Also, since the structure is generated by an algorithm, parameters can be easily adjusted to target a specific situation. Examples of some structures created by the students during this course are shown in figures B.8, B.10 and B.9.

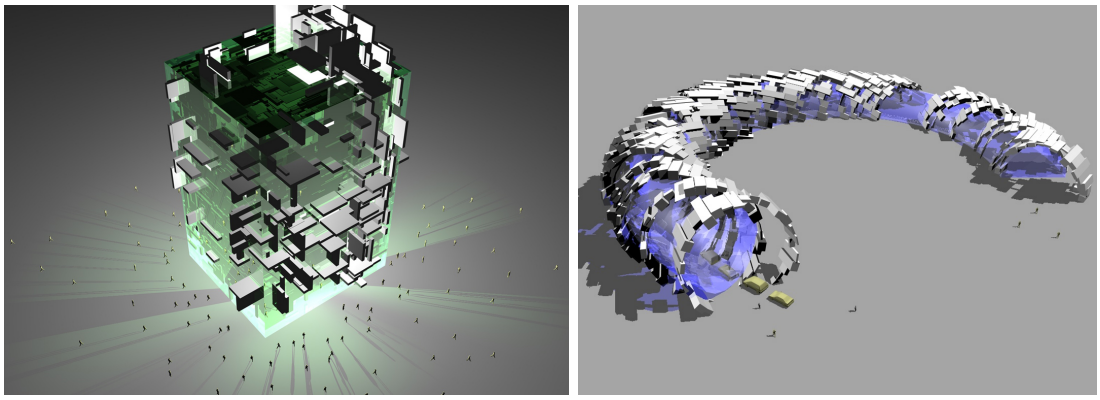


Figure B.8.: Proto architecture created with GroIMP (by Liang Liang).

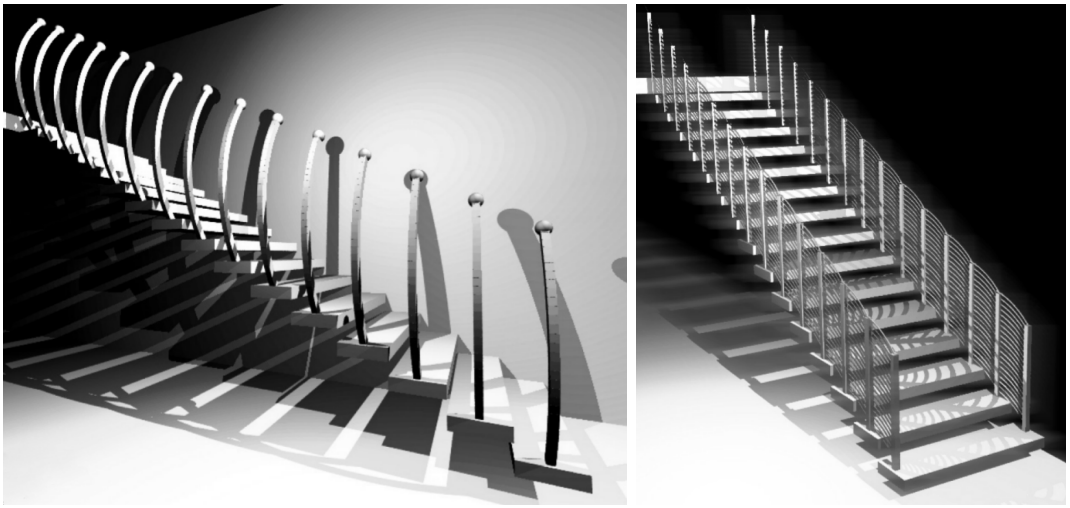


Figure B.9.: Stairs created with GroIMP (by Christopher Jarchow).

In order to make 3D structures modelled with GroIMP available to other software for post-processing, or to import complicated geometrical objects into GroIMP, import/export filters for the DXF file format and an import filter for the OBJ file format were added. To hold imported polygonal data, also a new geometric primitive type represented by the class `MeshNode` was introduced. A structure generated with GroIMP and post-processed after exporting it to DXF is shown in figure B.11.

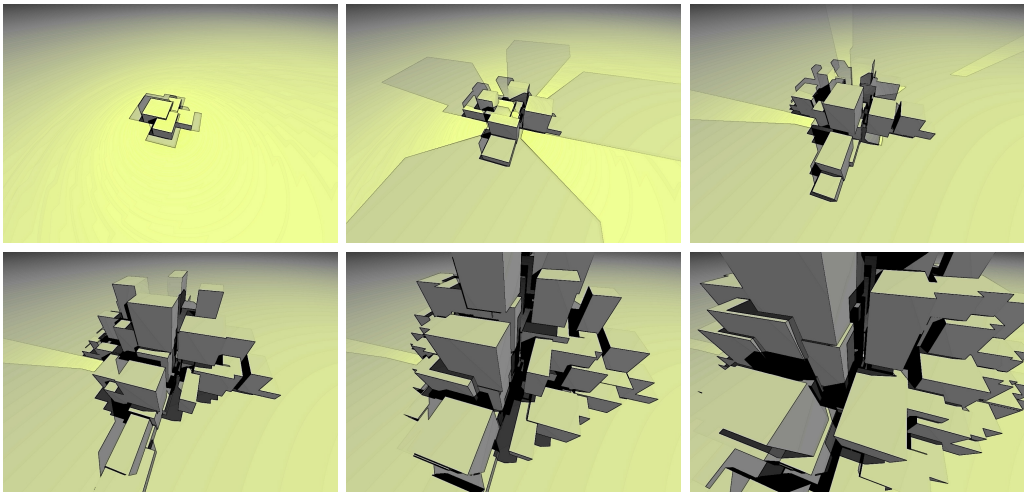


Figure B.10.: Proto architecture created with GroIMP (by Christopher Jarchow).

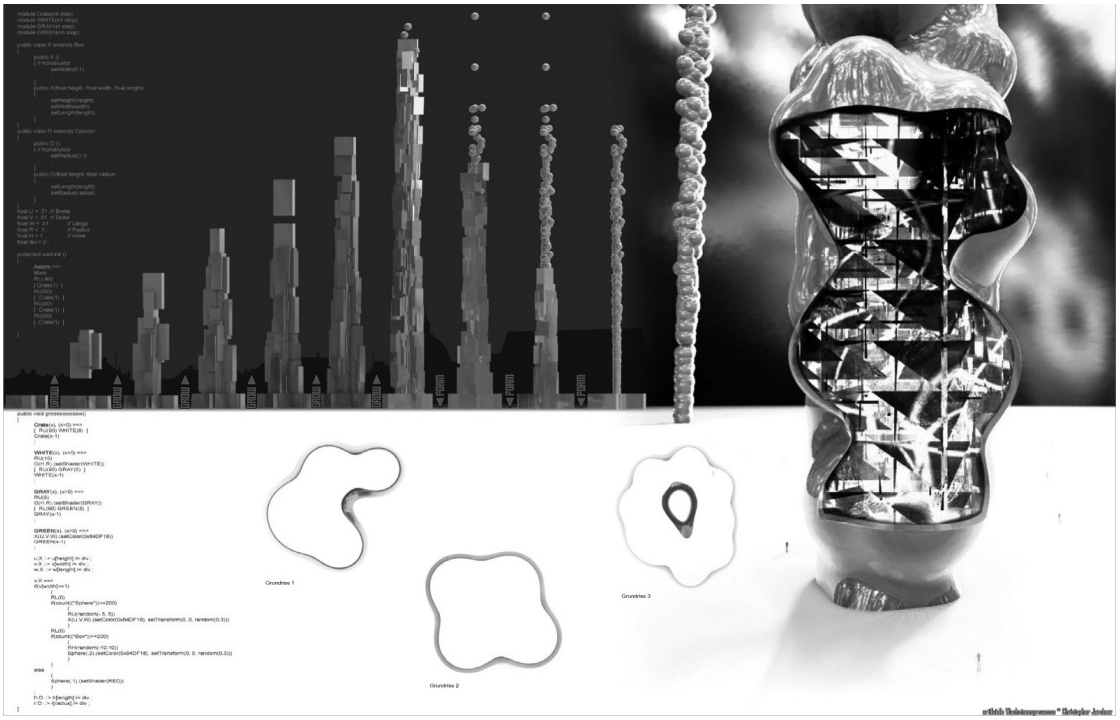


Figure B.11.: Skyscraper created with GroIMP (by Christopher Jarchow) [BK07].

Bibliography

[Abe99] R. Abegg. Untersuchungen über die chemischen Affinitäten – Abhandlungen aus den Jahren 1864, 1867, 1879 von C.M. Guldberg und P. Waage. In *Ostwald's Klassiker der exakten Wissenschaften*, volume 104. Verlag von Wilhelm Engelmann, Leipzig, 1899. 97

[Abr86] Henry I. Abrash. Studies concerning affinity. *Journal of Chemical Education*, 63(12):1044–1047, December 1986. 97

[Ad80] Harold Abelson and Andrea A. diSessa. *Turtle Geometry – The Computer as a Medium for Exploring Mathematics.* MIT Press, Cambridge, 1980. 53

[Ale10] Andrei Alexandrescu. *The D Programming Language.* Addison-Wesley, 2010. 77, 80

[Ama95] Herbert Amann. *Gewöhnliche Differentialgleichungen.* de Gruyter Berlin, 2nd edition, 1995. 5

[AU68] Alfred V. Aho and Jeffrey D. Ullman. The theory of languages. *Theory of Computing Systems*, 2(2):97–125, 1968. 48

[Bar75] V. K. Barwell. Special stability problems for functional differential equations. *BIT Numerical Mathematics*, 15(2):130–135, 1975. 14

[Bas83] Francis Bashforth. *An attempt to test the theories of capillary action by comparing the theoretical and measured forms of drops of fluid.* University Press, Cambridge, 1883. 32

[BBK65] R. E. Bellman, J. D. Buell, and R. E. Kalaba. Numerical integration of a differential-difference equation with a decreasing time-lag. *Communications of the ACM*, 8(4):227–228, 1965. 1, 39

[BD83] G. Bader and P. Deuflhard. A semi-implicit mid-point rule for stiff systems of ordinary differential equations. *Numerische Mathematik*, 41(3):373–398, 1983. 36

[BH72a] Rodger W. Baker and Gabor T. Herman. Simulation of organisms using a developmental model – part 1: Basic description. *International Journal of Bio-Medical Computing*, 3(3):201–215, July 1972. 56

[BH72b] Rodger W. Baker and Gabor T. Herman. Simulation of organisms using a developmental model – part 2: The heterocyst formation problem in blue-green algae. *International Journal of Bio-Medical Computing*, 3(4):251–267, October 1972. 56

[BH87] George D. Byrne and Alan C. Hindmarsh. Stiff ODE solvers: A review of current and coming attractions. *Journal of Computational Physics*, 70(1):1–62, 1987. 5

[Bic41] W. G. Bickley. Formulae for numerical differentiation. *The Mathematical Gazette*, 25(263):19–27, 1941. 43, 136

[Bic82] Theodore A. Bickart. P-stable and P[α,β]-stable integration/interpolation methods in the solution of retarded differential-difference equations. *BIT Numerical Mathematics*, 22(4):464–476, 1982. 14

[BK07] Günter Barczik and Winfried Kurth. From designing objects to designing processes: Algorithms as creativity enhancers. In *Predicting the Future. 25th eCAADe Conference Proceedings*, pages 887–894, 2007. 173

[BPW94] C.T.H. Baker, C.A.H. Paul, and D.R. Willé. Issues in the numerical solution of evolutionary delay differential equations. Numerical Analysis Report 248, University of Manchaster, 1994. 40

[BPW95] C.T.H. Baker, C.A.H. Paul, and D.R. Willé. Issues in the numerical solution of evolutionary delay differential equations. *Advances in Computational Mathematics*, 3:171–196, 1995. 40

[BS66] Roland Bulirsch and Josef Stoer. Numerical treatment of ordinary differential equations by extrapolation methods. *Numerische Mathematik*, 8(1):1–13, 1966. 36

[BSHK+08] Gerhard Buck-Sorlin, Reinhard Hemmerling, Ole Kniemeyer, Benno Burema, and Winfried Kurth. A rule-based model of barley morphogenesis, with special respect to shading and gibberellic acid signal transduction. *Annals of Botany*, 101(8):1109–1123, 2008. 138, 139

[BSKK05] Gerhard Buck-Sorlin, Ole Kniemeyer, and Winfried Kurth. Barley morphology, genetics and hormonal regulation of internode elongation modelled by a relational growth grammar. *New Phytologist*, 166(3):859–867, 2005. 138

[Bul64] Roland Bulirsch. Bemerkungen zur Romberg-Integration. *Numerische Mathematik*, 6:6–16, 1964. 35

[But63] J. C. Butcher. Coefficients for the study of Runge-Kutta integration processes. *J. Austral. Math. Soc.*, 3:185–201, 1963. 15

[But64a] J. C. Butcher. Implicit Runge-Kutta processes. *Mathematics of Computation*, 18(85):50–64, 1964. 15, 29, 30, 31

[But64b] J. C. Butcher. On Runge-Kutta processes of high order. *J. Austral. Math. Soc.*, 4:179–194, 1964. 14, 15, 19, 21

[But75] J. C. Butcher. A stability property of implicit Runge-Kutta methods. *BIT Numerical Mathematics*, 15(4):358–361, 1975. 14

[But76] J. C. Butcher. On the implementation of implicit Runge-Kutta methods. *BIT*, 16(3):237–240, 1976. 30

[But96] J. C. Butcher. A history of Runge-Kutta methods. *Applied Numerical Mathematics*, 20(3):247–260, 1996. 21

[Cas83a] J. R. Cash. Block Runge-Kutta methods for the numerical integration of initial value problems in ordinary differential equations Part I. The nonstiff case. *Mathematics of Computation*, 40(161):175–191, 1983. 26

[Cas83b] J. R. Cash. Block Runge-Kutta methods for the numerical integration of initial value problems in ordinary differential equations Part II. The stiff case. *Mathematics of Computation*, 40(161):193–206, 1983. 26

[Cas85] J. R. Cash. Block embedded explicit Runge-Kutta methods. *Computers & Mathematics with Applications*, 11(4):395–409, 1985. 25

[Cas03] J. R. Cash. Review paper: Efficient numerical methods for the solution of stiff initial-value problems and differential algebraic equations. *Proceedings of the Royal Society: Mathematical, Physical and Engineering Sciences*, 459(2032):797–815, 2003. 5, 13

[CFL28] R. Courant, K. Friedrichs, and H. Lewy. Über die partiellen Differenzengleichungen der mathematischen Physik. *Mathematische Annalen*, 100(1):32–74, 1928. 44

[CFL67] R. Courant, K. Friedrichs, and H. Lewy. On the partial difference equations of mathematical physics. *IBM Journal of Research and Development*, 11(2):215–234, 1967. 44

[CH52] C. F. Curtiss and J. O. Hirschfelder. Integration of stiff equations. *Proceedings of the National Academy of Sciences*, 38(3):235–243, March 1952. 34

[Chi71] F. H. Chipman. A-stable Runge-Kutta processes. *BIT Numerical Mathematics*, 11(4):384–388, 1971. 14

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956. 47

[Cho59] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2:137–167, 1959. 48

[CJC10] Vincent Le Chevalier, Marc Jaeger, and Paul-Henry Cournède. A framework for the simulation of coupled biophysical models, and its application to spatial processes interacting with resources. *Journal of computer science and technology*, 2010. (in press). 158

[CK90] J. R. Cash and Alan H. Karp. A variable order Runge-Kutta method for initial value problems with rapidly varying right-hand sides. *ACM Transactions on Mathematical Software*, 16(3):201–222, 1990. 26, 27

[CLM89] M. Calvo, F. Lisbona, and J. Montijano. On the stability of interpolatory variable-stepsize Adams methods in Nordsieck form. *SIAM Journal on Numerical Analysis*, 26(4):946–962, 1989. 33

[CN47] J. Crank and P. Nicolson. A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type. *Mathematical Proceedings of the Cambridge Philosophical Society*, 43(1):50–67, 1947. 135

[CN96] J. Crank and P. Nicolson. A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type. *Advances in Computational Mathematics*, 6(1):207–226, 1996. 135

[Col66] L. Collatz. *The Numerical Treatment of Differential Equations*. Springer, New York, 1966. 25, 32

[Cry73] Colin W. Cryer. A new class of highly-stable methods: A_0-stable methods. *BIT Numerical Mathematics*, 13(2):153–159, 1973. 14

[Cry74] Colin W. Cryer. Highly stable multistep methods for retarded differential equations. *SIAM Journal on Numerical Analysis*, 11(4):788–797, 1974. 14

[CSPH11] Mikolaj Cieslak, Alla N. Seleznyova, Przemyslaw Prusinkiewicz, and Jim Hanan. Towards aspect-oriented functional–structural plant modelling. *Annals of Botany*, 108(6):1025–1041, 2011. 1

[Dah56] Germund Dahlquist. Convergence and stability in the numerical integration of ordinary differential equations. *Mathematica Scandinavica*, 4:33–53, 1956. 32

[Dah63] Germund G. Dahlquist. A special stability problem for linear multistep methods. *BIT Numerical Mathematics*, 3(1):27–43, 1963. 12

[Dah76] Germund Dahlquist. Error analysis for a class of methods for stiff non-linear initial value problems. *Lecture Notes in Mathematics*, 1976. 14

[Dah78] Germund Dahlquist. G-stability is equivalent to A-stability. *BIT Numerical Mathematics*, 18(4):384–401, 1978. 14

[Deu83] P. Deuflhard. Order and stepsize control in extrapolation methods. *Numerische Mathematik*, 41(3):399–422, 1983. 35, 36

[Deu85] P. Deuflhard. Recent progress in extrapolation methods for ordinary differential equations. *SIAM Review*, 27(4):505–535, 1985. 5, 36

[DKH+07] Jan-Anton Dérer, Ole Kniemeyer, Reinhard Hemmerling, Gerhard Buck-Sorlin, and Winfried Kurth. Using the language XL for structural analysis. In Przemyslaw Prusinkiewicz, Jim Hanan, and Brendan Lane, editors, *5th International Workshop on Functional-Structural Plant Modelling*, pages P44, 1, Napier, New Zealand, nov 2007. 72

[DP78] J. R. Dormand and P. J. Prince. New Runge-Kutta algorithms for numerical simulation in dynamical astronomy. *Celestial Mechanics*, 18(3):223–232, 1978. 24

[DP80] J. R. Dormand and P. J. Prince. A family of embedded Runge-Kutta formulae. *Journal of Applied Mathematics*, 6(1):19–26, 1980. 25, 26

[Eul68] Leonard Euler. *Institutionum calculi integralis volumen primum in quo methodus integrandi a primis principiis usque ad integrationem aequationum differentialium primi gradus pertractatur*. Petropoli impensis academiae imperialis scientiarum, 1768. 15

[EVY+10] J. B. Evers, J. Vos, X. Yin, P. Romero, P. E. L. van der Putten, and P. C. Struik. Simulation of wheat growth and development based on organ-level photosynthesis and assimilate allocation. *Journal of Experimental Botany*, 61(8):2203–2216, 2010. 1

[Feh58] Erwin Fehlberg. Eine Methode zur Fehlerverkleinerung beim Runge-Kutta-Verfahren. *Zeitschrift für angewandte Mathematik und Mechanik*, 38(11/12):421–426, 1958. 19

[Feh68] Erwin Fehlberg. Classical fifth-, sixth-, seventh-, and eighth-order Runge-Kutta formulas with stepsize control. Technical Report 287, NASA, 1968. 21, 23

[Feh69a] Erwin Fehlberg. Klassische Runge-Kutta-Formeln fünfter und siebenter Ordnung mit Schrittweiten-Kontrolle. *Computing*, 4(2):93–106, 1969. 21, 22, 23

[Feh69b] Erwin Fehlberg. Low-order classical Runge-Kutta formulas with stepsize control and their application to some heat transfer problem. Technical Report 315, NASA, 1969. 23, 24, 25

[Feh70] Erwin Fehlberg. Klassische Runge-Kutta-Formeln vierter und niedrigerer Ordnung mit Schrittweiten-Kontrolle und ihre Anwendung auf Wärmeleitungsprobleme. *Computing*, 6:61–71, 1970. 23, 24

[Fer00] Pascal Ferraro. *Méthodes algorithmiques de comparaison d'arborescences.* PhD thesis, Institut National Polytechnique de Toulouse, Toulouse, France, November 2000. 74

[FFC82] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Commun. ACM*, 25(6):371–384, 1982. 170

[FH07] Roland W. Freund and Ronald H.W. Hoppe. *Stoer/Bulirsch: Numerische Mathematik 1.* Springer Berlin Heidelberg, 2007. 10

[Fic55] Adolf Fick. Ueber Diffusion. *Annalen der Physik*, 170:59–86, 1855. 103

[FK99] Thomas Früh and Winfried Kurth. The hydraulic system of trees: Theoretical framework and numerical simulation. *Journal of Theoretical Biology*, 201(4):251–270, December 1999. 104

[For88] Bengt Fornberg. Generation of finite difference formulas on arbitrarily spaced grids. *Mathematics of Computation*, 51(184):699–706, 1988. 43

[FP04] Pavol Federl and Przemyslaw Prusinkiewicz. Solving differential equations in developmental models of multicellular structures expressed using L-systems. In Marian Bubak, Geert van Albada, Peter Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2004*, volume 3037 of *Lecture Notes in Computer Science*, pages 65–72. Springer, Berlin / Heidelberg, 2004. 1, 133, 135

[Frü95] Thomas Früh. Entwicklung eines Simulationsmodells zur Untersuchung des Wasserflusses in der verzweigten Baumarchitektur. Berichte des Forschungszentrums Waldökosysteme A 131, Forschungszentrum Waldökosysteme, Göttingen, 1995. 104

[Gea67] C. W. Gear. The numerical integration of ordinary differential equations. *Mathematics of Computation*, 21(98):146–156, 1967. 33

[Gea71] C. William Gear. *Numerical Initial Value Problems in Ordinary Differential Equations.* Prentice-Hall, 1971. 34

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM}Language Specification – Second Edition.* Addision-Wesley, 2000. 62, 77, 85

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM}Language Specification – Third Edition.* Addision-Wesley, 2005. 1, 62, 82, 87, 88, 89

[Göd31] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik*, 38(1):173–198, 1931. 49

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991. 10

[Gra65] William B. Gragg. On extrapolation algorithms for ordinary initial value problems. *Journal of the Society for Industrial and Applied Mathematics: Series B, Numerical Analysis*, 2(3):384–403, 1965. 35, 36

[GS05] C. Godin and H. Sinoquet. Functional-structural plant modelling. *New Phytologist*, 166:705–708, 2005. 1

[GSDT85] Gopal K. Gupta, Ron Sacks-Davis, and Peter E. Tescher. A review of recent developments in solving ODEs. *ACM Comput. Surv.*, 17(1):5–47, March 1985. 5, 11, 30

[Han92] James Scott Hanan. *Parametric L-systems and Their Application To the Modelling and Visualization of Plants*. PhD thesis, University of Regina, 1992. 55, 56

[Har10] Konni Hartmann. Prozedurale Texturen in einem interaktiven Open-Source 3D-Modeller. Bachelor's thesis, Georg-August University of Göttingen, 2010. 168

[HBG+05] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodwardhor. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software*, 31(3):363–396, 2005. 117

[HEFS72] T. E. Hull, W. H. Enright, B. M. Fellen, and A. E. Sedgwick. Comparing numerical methods for ordinary differential equations. *SIAM Journal on Numerical Analysis*, 9(4):603–637, 1972. 5, 25

[Her69] G. T. Herman. Computing ability of a developmental model for filamentous organisms. *Journal of Theoretical Biology*, 25:421–435, 1969. 53

[Heu00] Karl Heun. Neue Methode zur approximativen Integration der Differentialgleichungen einer unabhängigen Veränderlichen. *Zeitschrift für Mathematik und Physik*, 45:23–38, 1900. 16

[HH55] P. C. Hammer and J. W. Hollingsworth. Trapezoidal methods of approximating solutions of differential equations. *Mathematical Tables and Other Aids to Computation*, 9(51):92–96, 1955. 31

[HKL+08] Reinhard Hemmerling, Ole Kniemeyer, Dirk Lanwert, Winfried Kurth, and Gerhard Buck-Sorlin. The rule-based language XL and the modelling environment GroIMP illustrated with simulated tree competition. *Functional Plant Biology*, 35(10):739–750, 2008. 72, 140

[Hor81] Mary Kathleen Horn. Scaled Runge-Kutta algorithms for handling dense output. Technical report, DFVLR, 1981. 27, 28

[Hor82] Mary Kathleen Horn. Scaled Runge-Kutta algorithms for treating the problem of dense output. Technical report, NASA Johnson Space Center, 1982. 27

[Hor83] Mary Kathleen Horn. Fourth- and fifth-order, scaled Runge-Kutta algorithms for treating dense output. *SIAM Journal on Numerical Analysis*, 20(3):558–568, 1983. 27, 28

[HOT78] François Hallé, Roelof A. A. Oldeman, and Philip B. Tomlinson. *Tropical Trees and Forests – An Architectural Analysis.* Springer Verlag, 1978. 72

[HSK10] Reinhard Hemmerling, Katarína Smoleňová, and Winfried Kurth. A programming language tailored to the specification and solution of differential equations describing processes on networks. In Adrian-Horia Dediu, Henning Fernau, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications*, volume 6031 of *Lecture Notes in Computer Science*, pages 297–308. Springer Berlin / Heidelberg, 2010. 104

[HT84] E. Hairer and H. Türke. The equivalence of B-stability and A-stability. *BIT Numerical Mathematics*, 24(4):520–528, 1984. 14

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979. 48

[HW02] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems.* Springer Berlin Heidelberg, 2nd edition, 2002. 5

[ISO11] ISO/IEC, Geneva, Switzerland. *Information technology – Programming languages – C++*, 3rd edition, 2011. 77

[Jen76] Richard D. Jenks. Problem #11: Generation of Runge-Kutta equations. *Newsletter, ACM SIGSAM Bulletin*, 10(1):6–7, Februrary 1976. 15

[Jür76] H. Jürgensen. Probabilistic L-systems. In A. Lindenmayer and G. Rozenberg, editors, *Automata, Languages, Development*, pages 211–225. North-Holland Publishing Company, 1976. 54

[Jus09] Ute Juschkat. Schrittweitensteuerung von Einschrittverfahren. Bachelor's thesis, Ruhr-Universität Bochum, 2009. 19

[KBSK03] Ole Kniemeyer, Gerhard Buck-Sorlin, and Winfried Kurth. Representation of genotype and phenotype in a coherent framework based on extended L-systems. In Wolfgang Banzhaf, Thomas Christaller, Peter Dittrich, Jan T. Kim, and Jens Ziegler, editors, *Advances in Artificial Life – Proceedings of the 7th European Conference on Artificial Life (ECAL)*, volume 2801 of *Lecture Notes in Artificial Intelligence*, pages 625–634. Springer Berlin / Heidelberg, 2003. 60

[KL87] Chris G. De Koster and Aristid Lindenmayer. Discrete and continuous models for heterocyst differentiation in growing filaments of blue-green bacteria. *Acta Biotheoretica*, 36(4):249–273, 1987. 56

[Kni04] Ole Kniemeyer. Rule-based modelling with the XL/GroIMP software. In Harald Schaub, Frank Detje, and Ulrike Brüggemann, editors, *The Logic of Artificial Life*, pages 56–65. AKA Akademische Verlagsgesellschaft Berlin, 2004. 60

[Kni08] Ole Kniemeyer. *Design and Implementation of a Graph Grammar Based Language for Functional-Structural Plant Modelling.* PhD thesis, Brandenburgische Technische Universität, Cottbus, 2008. 1, 60, 62, 66, 69, 86, 98, 114, 126, 128, 131, 158

[KP92] Peter E. Kloeden and Eckhard Platen. *Numerical Solution of Stochastic Differential Equations*, volume 23 of *Stochastic Modelling and Applied Probability.* Springer, 1st edition, 1992. 143

[KP03] Radoslaw Karwowski and Przemyslaw Prusinkiewicz. Design and implementation of the L+C modeling language. *Electronic Notes in Theoretical Computer Science*, 86(2):134–152, 2003. 1, 137

[Kua93] Yang Kuang. *Delay Differential Equations With Applications in Population Dynamics*, volume 191. Academic Press, 1993. 39

[Kup80] Ingbert Kupka. Van Wijngaarden grammars as a special information processing model. In P. Dembinski, editor, *Mathematical Foundations of Computer Science 1980*, volume 88 of *Lecture Notes in Computer Science*, pages 387–401. Springer Berlin / Heidelberg, 1980. 59

[Kur94] Winfried Kurth. Growth Grammar Interpreter GROGRA 2.4 – A software tool for the 3-dimensional interpretation of stochastic, sensitive growth grammars in the context of plant modelling. Berichte des Forschungszentrums Waldökosysteme B 38, Forschungszentrum Waldökosysteme, Göttingen, 1994. 59, 69

[Kur99] Winfried Kurth. *Die Simulation der Baumarchitektur mit Wachstumsgrammatiken.* Wissenschaftlicher Verlag Berlin, 1999. 59, 140

[Kur07] Winfried Kurth. Specification of morphological models with L-systems and Relational Growth Grammars. *Image - Journal of Interdisciplinary Image Science*, 5:50–79, February 2007. 61

[Kut01] Wilhelm Kutta. Beitrag zur näherungsweisen Integration totaler Differentialgleichungen. *Zeitschrift für Mathematik und Physik*, 46:435–453, 1901. 14, 17

[Lin68a] Aristid Lindenmayer. Mathematical models for cellular interaction in development – I. Filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280–299, March 1968. 51, 53

[Lin68b] Aristid Lindenmayer. Mathematical models for cellular interaction in development – II. Simple and branching filaments with two-sided inputs. *Journal of Theoretical Biology*, 18(3):300–315, March 1968. 51, 53, 54

[Lin71] Aristid Lindermayer. Developmental systems without cellular interactions, their languages and grammars. *Journal of Theoretical Biology*, 30(3):455–484, 1971. 51, 53

[Lin74] Aristid Lindenmayer. Adding continuous components to L-systems. In Grzegorz Rozenberg and Arto K. Salomaa, editors, *L Systems*, volume 15 of *LNCS*, pages 53–68. Springer Berlin/Heidelberg, 1974. 55, 56

[Lin75] Aristid Lindenmayer. Developmental algorithms for multicellular organisms: A survey of L-systems. *Journal of Theoretical Biology*, 54(1):3–22, 1975. 54

[LKBS10] Daniel Leitner, Sabine Klepsch, Gernot Bodner, and Andrea Schnepf. A dynamic root system growth model based on L-Systems. *Plant and Soil*, 332(1):177–192, 2010. 1

[LlCC09] Zhongping Li, Vincent le Chevalier, and Paul-Henry Cournède. Towards a continuous approach of functional-structural plant growth. In *Proceedings of the 2009 Plant Growth Modeling, Simulation, Visualization, and Applications*, PMA '09, pages 334–340, 2009. 1, 39

[LY99] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification.* Prentice Hall, 2nd edition, 1999. 62, 85

[Mer57] R. H. Merson. An operational method for the study of integration processes. In *Proceedings Symposium of Data Processing*, Salisbury, Australia, 1957. 21

[Mil26] W. E. Milne. Numerical integration of ordinary differential equations. *The American Mathematical Monthly*, 33(9):455–460, 1926. 32, 33

[Mil53] W. E. Milne. *Numerical Solution of Differential Equations.* John Wiley, New York, 1953. 25, 32

[MM13] L. Michaelis and M. L. Menten. Die Kinetik der Invertinwirkung. *Biochemische Zeitschrift*, 49:333–369, 1913. 102

[MN92] Makoto Murofushi and Hideko Nagasaka. On the internal stepsize of an extrapolation algorithm for IVP in ODE. *Numerical Algorithms*, 3(1):321–334, 1992. 35

[Moo86] Walter J. Moore. *Physikalische Chemie*. de Gruyter, Berlin, New York, 4th edition, 1986. 97, 101, 102

[Mou26] Forest Ray Moulton. *New methods in exterior ballistics*. The University of Chicago Press, September 1926. 33

[Nis80] Taishin Nishida. K0L-system simulating almost but not exactly the same development. In *Memoirs of the Faculty of Science*, volume 8 of *Series of Biology*, pages 97–122. Kyoto University, March 1980. 54

[NM10] Hiroya Nakao and Alexander S. Mikhailov. Turing patterns in network-organized activator-inhibitor systems. *Nature Physics*, 6(7):544–550, 2010. 146

[Nor62] Arnold Nordsieck. On numerical integration of ordinary differential equations. *Mathematics of Computation*, 16(77):22–49, 1962. 33

[NTT92] Hansrudi Noser, Daniel Thalmann, and Russel Turner. Animation based on the interaction of L-systems with vector force fields. In *Proceedings of the 10th International Conference of the Computer Graphics Society on Visual computing : integrating computer graphics with computer vision*, CG International '92, pages 747–761. Springer, 1992. 58

[Nys25] E. J. Nyström. Über die numerische Integration von Differentialgleichungen. *Acta Societatis Scientiarum Fennicæ*, 50(13), 1925. 19, 158

[O'R70] P. G. O'Regan. Step size adjustment at discontinuities for fourth order Runge-Kutta methods. *The Computer Journal*, 13(4):401–404, 1970. 27

[PCS+09] Przemyslaw Prusinkiewicz, Scott Crawford, Richard S. Smith, Karin Ljung, Tom Bennett, Veronica Ongaro, and Ottoline Leyser. Control of bud activation by an auxin transport switch. *Proceedings of the National Academy of Sciences*, 106(41):17431–17436, 2009. 137

[PdSPK01] Patricio Parada, Javier Ruiz del Solar, Wladimir Plagges, and Mario Kppen. Interactive texture synthesis. pages 434–439, 2001. 170

[PH90] P. Prusinkiewicz and J. Hanan. Visualization of botanical structures and processes using parametric L-systems. In Daniel Thalmann, editor, *Scientific Visualization and Graphics Simulation*, pages 183–201. J. Wiley & Sons, 1990. 55

[PH04] Matt Pharr and Greg Humphreys. *Physically Based Rendering*. Morgan Kaufmann Publishers Inc., 2004. 10

[PHM93] Przemylsaw Prusinkiewicz, Mark S. Hammel, and Eric Mjolsness. Animation of plant development. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 351–360. ACM, 1993. 1, 3, 58, 133, 135, 136

[PL90] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer, 1990. 1, 51, 52, 54, 56, 57, 72

[PR74] A. Prothero and A. Robinson. On the stability and accuracy of one-step methods for solving stiff systems of ordinary differential equations. *Mathematics of Computation*, 28(125):145–162, 1974. 14

[Pru86] Przemyslaw Prusinkiewicz. Graphical applications of L-systems. In *Proceedings on Graphics Interface '86/Vision Interface '86*, pages 247–253, 1986. 53, 54

[Pru87] Przemyslaw Prusinkiewicz. Applications of L-systems to computer imagery. In Hartmut Ehrig, Manfred Nagl, Grzegorz Rozenberg, and Azriel Rosenfeld, editors, *Graph-Grammars and Their Application to Computer Science*, volume 291 of *LNCS*, pages 534–548. Springer Berlin/Heidelberg, 1987. 54

[PST88] G. Papageorgiou, Th. Simos, and Ch. Tsitouras. Some new Runge-Kutta methods with interpolation properties and their application to the magnetic-binary problem. *Celestial Mechanics*, 44(1):167–177, 1988. 28

[PTVF07] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 3rd edition, 2007. 30

[QSS07] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Numerical Mathematics*. Springer Berlin Heidelberg, 2nd edition, 2007. 5, 6

[RD71] G. Rozenberg and P. G. Doucet. On 0L-languages. *Information and Control*, 19:302–318, 1971. 51, 53

[Rey87] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Comput. Graph.*, 21:25–34, 1987. 90

[Ric11] Lewis Fry Richardson. The approximate arithmetical solution by finite differences of physical problems involving differential equations, with an application to the stresses in a masonry dam. *Philosophical transactions of the Royal society of London*, 210:307–357, 1911. 34

[Ric27] Lewis Fry Richardson. The deferred approach to the limit. *Philosophical Transactions of the Royal Society of London*, 226:299–361, 1927. 34

[Ric70] J. P. Richter. *The notebooks of Leonardo da Vinci*. Dover Publications, 1970. 75

[Rog08] Stephan Rogge. Generierung von Baumdarstellungen in VRML für den Branitzer Park. Bachelor's thesis, BTU Cottbus, 2008. 72

[Rom55] W. Romberg. Vereinfachte numerische Quadratur. *Det Kongelige Norske Videnskabers Selskabs Forhandlinger*, 28:30–36, 1955. 35

[Ros67] J. Barkley Rosser. A Runge-Kutta for all seasons. *SIAM Review*, 9(3):417–452, 1967. 25

[Ros88] Javier F. Rosenblueth. System with time delay in the calculus of variations: The method of steps. *IMA Journal of Mathematical Control and Information*, 5(4):285–299, 1988. 40

[Roz73] Grzegorz Rozenberg. TOL systems and languages. *Information and Control*, 23(4):357–381, 1973. 55

[Run95] Carl Runge. Über die numerische Auflösung von Differentialgleichungen. *Mathematische Annalen*, 46:167–178, 1895. 14, 16

[Run05] Carl Runge. Ueber die numerische Auflösung totaler Differentialgleichungen. *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen*, 3:252–257, 1905. 143

[Sal73] Arto K. Salomaa. *Formal Languages*. Academic Press, 1973. 48

[Sal78] Arto K. Salomaa. *Formale Sprachen*. Springer, 1978. 51

[Sar65] D. Sarafyan. Multistep methods for the numerical solution of ordinary differential equations made self-starting. Technical Report 495, Mathematics Research Center, Madison, Wisconsin, 1965. 25, 32

[Sav98] Michael A. Savageau. Development of fractal kinetic theory for enzyme-catalysed reactions and implications for the design of biochemical pathways. *BioSystems*, 47:9–36, 1998. 102

[SB05] Josef Stoer and Roland Bulirsch. *Numerische Mathematik 2*. Springer Berlin Heidelberg, 5th edition, 2005. 11, 19, 20

[Sch91] William E. Schiesser. *The Numerical Method of Lines*. Academic Press, 1991. 1, 42, 44, 46

[SH10] Katarína Smoleňová and Reinhard Hemmerling. Growing virtual plants for virtual worlds. In *Proceedings of the 24th Spring Conference on Computer Graphics*, SCCG '08, pages 67–74. ACM, 2010. 69, 72

[Sha85] Lawrence F. Shampine. Interpolation for Runge-Kutta methods. *SIAM Journal on Numerical Analysis*, 22(5):1014–1027, 1985. 27, 28

[Sha86] Lawrence F. Shampine. Some practical Runge-Kutta formulas. *Mathematics of Computation*, 46(173):135–150, 1986. 26, 28

[She99] W. F. Sheppard. Central-difference formulæ. *Proceedings of the London Mathematical Society*, s1-31(1):449–488, 1899. 34

[SL69] Vimol Surapipith and Aristid Lindenmayer. Thioguanine-dependent light sensitivity of perithecial initiation in *Sordaria fimicola*. *Journal of General Microbiology*, 57(2):227–237, 1969. 55

[Smi84] Alvy Ray Smith. Plants, fractals, and formal lanugages. *Computer Graphics*, 18(3):1–10, 1984. 53

[Sti67] Claus Stimberg. Vereinfachte Herleitung von Runge-Kutta-Verfahren. *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik*, 47(6):413–414, 1967. 15

[Sto74] Josef Stoer. Extrapolation methods for the solution of initial value problems and their practical realization. In Dale Bettis, editor, *Proceedings of the Conference on the Numerical Solution of Ordinary Differential Equations*, volume 362 of *Lecture Notes in Mathematics*, pages 1–21. Springer Berlin / Heidelberg, 1974. 10.1007/BFb0066583. 36

[SWD76] L. F. Shampine, H. A. Watts, and S. M. Davenport. Solving nonstiff ordinary differential equations – the state of the art. *SIAM Review*, 18(3):376–411, 1976. 5

[SYHK64a] Kichiro Shinozaki, Kyoji Yoda, Kazuo Hozumi, and Tatuo Kira. A quantitative analysis of plant form – the pipe model theory — I. Basic analyses. *Japanese Journal of Ecology*, 14(3):97–105, June 1964. 72, 75

[SYHK64b] Kichiro Shinozaki, Kyoji Yoda, Kazuo Hozumi, and Tatuo Kira. A quantitative analysis of plant form – the pipe model theory — II. Further evidence of the theory and its application in forest ecology. *Japanese Journal of Ecology*, 14(4):133–139, August 1964. 72

[Tur37] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 42(2144):230–265, November 1937. 48

[Tur52] A. M. Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences*, 237(641):37–72, 1952. 103, 144

[vD71] Dirk van Dalen. A note on some systems of Lindenmayer. *Theory of Computing Systems*, 5(2):128–140, 1971. 51, 52, 53

[Vea97] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation.* PhD thesis, Standford University, December 1997. 143

[VEBS+10] J. Vos, J. B. Evers, G. H. Buck-Sorlin, B. Andrieu, M. Chelle, and P. H. B. de Visser. Functional-structural plant modelling: a new versatile tool in crop science. *Journal of Experimental Botany*, 61(8):2101–2115, 2010. 1

[Vel95] Todd Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. 93

[Vel98] Todd Veldhuizen. Expression templates. In Stanley B. Lippman, editor, *C++ Gems*, pages 475–488. Cambridge University Press, 1998. 93

[Vel00] Todd L. Veldhuizen. Just when you thought your little language was safe: "expression templates" in Java. Technical Report 539, Indiana University, Bloomington Indiana, USA, July 2000. 93

[Ver67] Loup Verlet. Computer "experiments" on classical fluids. I. Thermodynamical properties of lennard-jones molecules. *Physical Review*, 159(1):98–103, 1967. 37

[Ver68] Loup Verlet. Computer "experiments" on classical fluids. II. Equilibrium correlation functions. *Physical Review*, 165(1):201–214, 1968. 37

[Wal00] Wolfgang Walter. *Gewöhnliche Differentialgleichungen*. Springer Berlin Heidelberg, 7th edition, 2000. 5

[WG64] P. Waage and C.M. Guldberg. Studier over Affiniteten. In *Forhandlinger Videnskabs-Selskabet Christinia*, pages 35–45. 1864. 97

[Wid67] Olof B. Widlund. A note on unconditionally stable linear multistep methods. *BIT Numerical Mathematics*, 7(1):65–70, 1967. 14

[Yos93] Haruo Yoshida. Recent progress in the theory and application of symplectic integrators. *Celestial Mechanics and Dynamical Astronomy*, 56(1):37–43, 1993. 36

[You84] David A. Young. A local activator-inhibitor model of vertebrate skin patterns. *Mathematical Biosciences*, 72:51–58, 1984. 104