# The Construction of Flux-Limiting Advection Algorithms through Program Generation

*Ilja Heitlager, Robert van Engelen & Lex Wolters*

High Performance Computing Division

Department of Computer Science, Leiden University

P.O. Box 9512, 2300 RA Leiden, The Netherlands

{ilja,robert,llexx}@cs.leidenuniv.nl

**Abstract**

The CTADEL code generator is a tool for the automatic generation of PDE based scientific models and generates efficient code for various computational platforms from an independent high level language description. It was applied for the generation of code to compute the dynamical tendencies within the HIRLAM weather forecast model. In this paper we will use CTADEL for the implementation of numerical advection schemes for the DELWAQ water quality model. This model is based on the finite volume discretization method. In this report we consider the generation of flux-limiter upwind schemes. These schemes show less numerical diffusion, but are still monotone, which are important properties for the modeling of advection within water quality models. We discuss the generation of these methods and the optimization of the resulting conditional loop nests within CTADEL.

## 1 Introduction

The need for portability and the ever increasing complexity of numerical methods impede the fast development of programs for scientific computing. The strong demand for efficiency causes the specification of the program to be contaminated with low-level implementation details about the target architecture. This makes the program less portable, maintainable and even intelligable. Numerical methods become more and more complex, thereby increasing the burden of the program developer. It makes the job of program writing errorprone and cumbersome.

This inhibiting effect was experienced with the HIRLAM[1] weather forecast system [3] and this experience has led to the development of the CTADEL code generator [21] for PDE-based models. It is a Prolog based computer aided program implementation tool, which generates multiple platform specific – with platform we denote the combination of architecture and compiler – implementations from a single abstract high level specification. The high level language simplifies the specification of the PDE-based models and since the problem is specified on a higher level, application information is available, allowing

[1] The HIRLAM system was developed by the HIRLAM-project group, a cooperative project of Denmark, Finland, Iceland, Ireland, The Netherlands, Norway, Spain and Sweden

more optimizations or code restructuring to be performed [19]. The code generator was successfully applied for the generation of efficient Fortran code to compute the explicit dynamical tendencies in the HIRLAM weather model [20].

After its successful employment for HIRLAM, we are currently investigating the use of CTADEL for the generation of code for the water quality model DELWAQ [2] from Delft Hydraulics. Water quality modeling is the description of the evolution of substances in water due to transport, such as forcings by the flow of water (advection), and processes, like chemical reactions. It is described by PDEs of the advection-diffusion-reaction type, but in the actual model however a distinction is made between the advection-diffusion part or transport kernel, and the reaction part or process kernel.

We begin our investigations with spatial discretization techniques for the advection equation. This is a linear hyperbolic partial differential equation. Modeling hyperbolic equations is not straightforward. First order accurate finite difference methods suffer from excessive numerical diffusion. Higher order discretizations can produce dispersive ripples, which is an undesired property since these ripples have no physical meaning. In many cases the quantities described by the advection equation are positive, for which the oscillations can produce negative values. This causes the model to blow up if other parts of the model cannot handle these negative values, as is the case for the reaction kernel for water-quality modeling [2, 22].

As we will discuss in Section 3 higher order oscillation free schemes can not be constructed with linear finite difference methods. Therefore a lot of research has been done in the field of the numerical solution of hyperbolic PDEs or more specifically of advection equations. This has resulted in a wide variety of more and more complex numerical methods which are hard to implement [22]. As we will show program generation is a useful tool for the fast and efficient implementation of these different discretization techniques.

In this report we focus on the class of discretization methods called flux-limiting TVD methods [10, 15]. This is a class of non-linear schemes based on the finite volume discretization approach. This class of methods was originally developed for the solution of momentum equations in compressible flows. These methods handle shocks very well, show little numerical diffusion, and produce no ripples. The treatment of shocks is of course of less concern in water quality modeling, but because of the latter two characteristics, these methods are now finding widespread use in the solution of advection equations in incompressible flows.

Examples of code generation for finite volume methods are Roache and Steinberg [17], which use the MACSYMA computer algebra system to derive the finite volume method from the continuous equations, and a joined project between Delft Hydraulics and Twente University [6]. In the latter project code is generated for the ISNaS flow solver based on the finite-volume method. The code generator is implemented in the REDUCE computer algebra system. Both articles do not treat flux limiting methods.

This paper is organized as follows. In the next section we shortly derive the advection equation. This is done as an introduction to the advection equation and the finite volume method. In Section 3 we discuss why linear methods are not sufficient to overcome the diffusion-oscillation controversy. After this section we derive the finite volume method and flux limiter schemes in Section 4. We elaborate the specification of flux limiter schemes for logically rectangular domains in Section 5. The schemes will result in perfectly nested conditional loops. We distinguish three canonical code forms to write these . The transformation of the CTADEL specification to these canonical forms is explained in

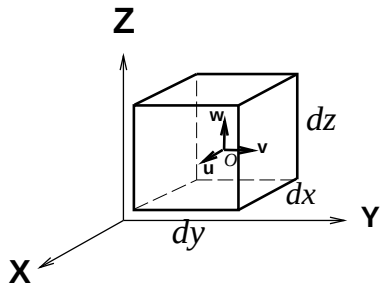


Figure 1: An infinitesimal cube in rectangular coordinates

Section 6. After this the performance of these forms for a third order limited scheme is shown in Section 7. Finally conclusions are drawn in Section 8.

# 2 The Advection Equation

As an introduction to the finite volume method, we start with the derivation of the advection equation from the general conservation principle.

Consider an infinitesimal cube in rectangular coordinates $[x, y, z]$ as is show in Figure 1. The cube has sides with length $dx$, $dy$ and $dz$. A quantity $q$ is given as a cell-centered average in point $O$ with velocity $\vec{V} = [u, v, w]^T$. This quantity could for example be temperature, density or concentration of a species. The general conservation law for a quantity $q$ in integral form is given as

$$\frac{\partial}{\partial t}\int_\Omega q\, d\Omega + \int_{\partial\Omega} q\vec{V}\cdot\vec{n}ds = 0\ , \tag{1}$$

with $\Omega$ the volume of the cube, $d\Omega$ an element of the volume, $\partial\Omega$ the surface of the cube, $\vec{n}$ the outer unit normal and $ds$ an element of the area. It states that the sum of the rate of change inside the volume $\Omega$ and what flows over the boundary $\partial\Omega$ must be zero. For the infinitesimal cube in Cartesian coordinates the integral over volume $\Omega$ can be written as

$$\frac{\partial}{\partial t}\int_\Omega q\, d\Omega \Leftrightarrow \frac{\partial q}{\partial t}\, dx\, dy\, dz\ . \tag{2}$$

The cube has six cell faces and the flux $F = q\vec{V}\cdot\vec{n}\, ds$ at these cell faces are found by Taylor expansion about point $O$, the center of the cube. Neglecting higher order terms, the right face flux at $\left(x + \frac{1}{2}dx\right)$ is given as

$$\begin{aligned} F_{right} &= \left[q + \left(\frac{\partial q}{\partial x}\right)\frac{dx}{2}\right]\left[u + \left(\frac{\partial u}{\partial x}\right)\frac{dx}{2}\right] dy\, dz \\ &= q\, u\, dy\, dz + \tfrac{1}{2}\left[u\left(\frac{\partial q}{\partial x}\right) + q\left(\frac{\partial u}{\partial x}\right)\right] dx\, dy\, dz \\ &= q\, u\, dy\, dz + \tfrac{1}{2}\frac{\partial q\, u}{\partial x}\, dx\, dy\, dz\ . \end{aligned} \tag{3}$$

The integral over the surface of the cube therefore simplifies to a sum of the six face fluxes

of the cube

$$\begin{aligned} \int_{\partial\Omega} q\vec{V}\cdot d\vec{A} \quad \Leftrightarrow \quad & F_{right} - F_{left} + \\ & F_{top} - F_{bottom} + \\ & F_{front} - F_{back} \\ \Leftrightarrow \quad & \left[\frac{\partial q\,u}{\partial x} + \frac{\partial q\,v}{\partial y} + \frac{\partial q\,w}{\partial z}\right] dx\,dy\,dz \ . \end{aligned} \tag{4}$$

The substitution of Eq. (2) and Eq. (4) into Eq. (1) results in the conservation law for a quantity $q$ in differential form, which is a linear hyperbolic partial differential equation

$$\frac{\partial q}{\partial t} + \frac{\partial q\,u}{\partial x} + \frac{\partial q\,v}{\partial y} + \frac{\partial q\,w}{\partial z} = 0 \ . \tag{5}$$

This is the conservative form of the advection equation. In vector notation it is given as

$$\begin{aligned} \frac{\partial q}{\partial t} &= -\nabla\cdot q\vec{V} \\ &= -\vec{V}\nabla q - q\nabla\cdot\vec{V} \ . \end{aligned} \tag{6}$$

In the case of water modeling the flow is incompressible, which means that the density is independent of time and space and therefore the continuity equation is given as [4]

$$\nabla\cdot\vec{V} = 0 \ . \tag{7}$$

This allows Eq. (6) to be simplified as

$$\begin{aligned} \frac{\partial q}{\partial t} &= -\vec{V}\nabla q \\ &= -u\frac{\partial q}{\partial x} - v\frac{\partial q}{\partial y} - w\frac{\partial q}{\partial z} \ , \end{aligned} \tag{8}$$

which is called the divergent form for the advection equation. For more details on the derivation of the advection equation, the reader is referred to [4, 9, 11] .

# 3 Linear Discretization Schemes

For the numerical solution of the advection equation in practical scientific computing, a proper discretization method needs to be chosen. As stated in Section 1, many methods exist, each with their own numerical and computational characteristics. The basic discretization technique is the finite difference technique. For this method the continuous differential operators are directly, and since the advection equation is linear, the discrete scheme is also linear.

Consider the one dimensional advection equation

$$\frac{\partial q}{\partial t} = -u\frac{\partial q}{\partial x} \quad \text{for } x \in [x_l, x_r] \ . \tag{9}$$

From an algorithmic point of view the most simple finite difference method is the second-order central difference scheme (CDS)

$$\frac{\partial q_i}{\partial t} = -u\frac{q_{i+1} - q_{i-1}}{2\Delta x} \ , \tag{10}$$

where $q(x)$ is discretized on an equidistant grid with mesh size $\Delta x$, and $q_i$ is an abbreviation for $q(x_l + i\Delta x)$. We do not consider the discretization in time. But in fact, where applicable in this paper, explicit time integration is assumed.

Although the CDS is second order accurate, it produces dispersive ripples [22]. It was already stated in Section 1 that the generation of dispersive ripples can be very undesirable. In case of positive quantities, negative values could be generated. This can be solved pragmatically by adjusting the right hand side of Eq. (10) to zero if it is negative, but this approach will make the scheme non conservative: the adjustment to zero introduces additional artificial mass. But since this is a computational inexpensive method, it has its application in practice, e.g. the HIRLAM weather model [3].

A finite difference scheme, which is not hindered by the generation of ripples, is the first order upwind difference scheme (UDS)

$$\frac{\partial q_i}{\partial t} = -u \begin{cases} \dfrac{q_i - q_{i-1}}{\Delta x} & \text{if } u > 0 \\ \dfrac{q_{i+1} - q_i}{\Delta x} & \text{otherwise .} \end{cases} \tag{11}$$

The stencil is aligned with the direction of the flow, hence the name upwind scheme. Due to this alignment, a selector, a conditional checking the sign of $u$, is necessary to adjust the scheme to the actual direction of the flow. Unfortunately, since this method is only first order accurate, it suffers from excessive numerical diffusion [22].

In the search for higher order, but oscillation free schemes the notion of *monotone* methods is introduced. This property implies that once the initial data $q_i$ is positive, it remains positive. Unfortunately, the class of monotone methods is greatly restricted since it is at most first order accurate. Related to this class are the *monotonicity preserving* methods. It is a weaker constraint than the monotonicity constraint. The monotonicity preservation condition states that if the initial data $q_i$ is monotone, $q_i^0 \geq q_{i+1}^0$, the solution will stay monotone at all time levels, $q_i^n \geq q_{i+1}^n$ for all n, which means that no ripples will be introduced. So if the solution is monotone and positive, no undershoots – potentially negative values – will be generated.

However, it was found by Godunov [5] that *linear* monotonicity preserving methods are equal to monotone methods, therefore at most first order accurate. This implies that every linear discretization scheme will be at most first order accurate. As a result of the Godunov theorem we have to rely on non-linear methods or *high-resolution* methods in order to construct higher order monotonicity preserving methods. More details can be found in [10, 15, 22].

# 4 Non-linear Discretization Schemes

A wide variety of non-linear positive discretization schemes exist, but most of these numerical methods are rather ad-hoc. A class of methods which has a firmer theoretical basis is the class of flux-limiter methods. These methods belong to the class of Total Variation diminishing (TVD), a concept introduced by Harten [7]. It can be shown that TVD methods are monotonicity preserving, and therefore high-order positive schemes can be constructed. The class of flux-limiter methods was first studied by Sweby [18].

Flux-limiting methods are based on the integral Eq. (1), formulation of the advection equation, which results in the finite volume approach [10, 16]. The approach can be

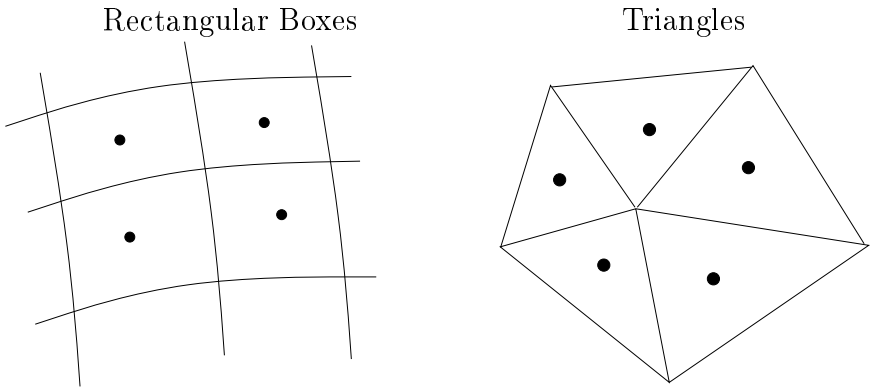


Figure 2: Finite Volume Discretizations

understood as a discretization of the domain into cells of some form and a description of the cell updates by computation of the fluxes over the cell. The derivation of the method is actually equal to the derivation of the differential form of the advection equation as discussed in Section 2, but in opposite direction. Starting from a continuous PDE a cell/flux description is found by the application of the Gauss divergence theorem on the integral of this continuous PDE over the volume of the cell. The properties of the scheme are altered by changing the interpolation function of the flux. To prevent the generation of ripples and therefore negative values, these fluxes are *limited*, hence the name flux-limiter methods.

## 4.1 Finite Volume Discretization

The first step in the finite volume discretization is the division of the domain into adjacent cells of some form like a rectangular box or a triangle in two dimensions, see Figure 2, or their three dimensional extensions, a cube and a tetrahedron. By application of the Gauss divergence theorem on the advection equation per cell, a description based on the fluxes over the cell faces is found. The Gauss divergence theorem for a vector field $\vec{F}$ over a closed volume $\Omega$ is given as

$$\int_\Omega \nabla \cdot \vec{F} d\Omega = \int_{\partial\Omega} \vec{F} \cdot \vec{n}\, ds\ . \tag{12}$$

The vector $\vec{n}$ is the unit outer normal of the surface. After discretization of the domain into cells and enumeration of the cells, we take the integral of the conservative advection equation Eq. (6) over the volume of the cell. By application of Eq. (12) we find

$$\begin{aligned} \frac{\partial}{\partial t}\int_\Omega q_i\, d\Omega &= -\int_\Omega \nabla \cdot q_i \vec{V}\, d\Omega \\ &= -\int_{\partial\Omega} q_i \vec{V} \cdot \vec{n}\, ds\ , \end{aligned} \tag{13}$$

where $q_i$ is the cell centered average of quantity $q$ at cell $i$. Actually we see that the conservation law Eq. (1) reappears.

Depending on the choice of the cell, the surface of the cell consists of a number of cell faces, for example, four cell faces in the case of rectangular boxes and three in the

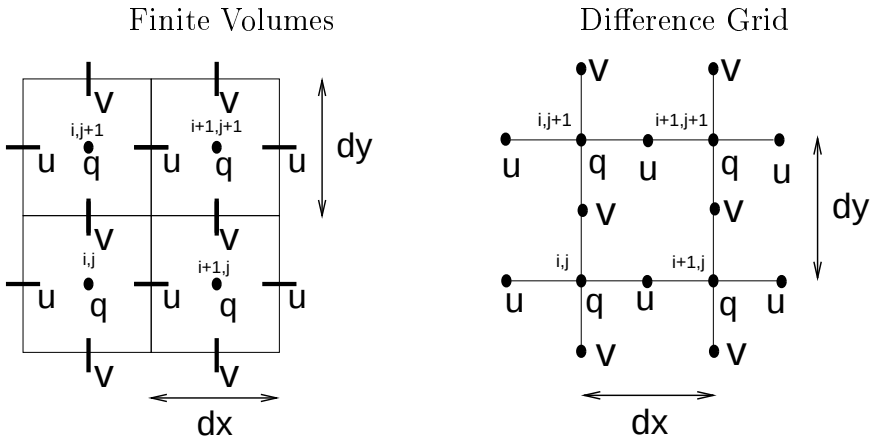


Figure 3: Regular Finite Volume Discretization

case of triangles, as is shown in Figure 2. The integral of the flux over the surface of a cell simplifies to a sum of fluxes over the cell faces, such that Eq. (13) reduces to

$$\begin{aligned} \frac{\partial q_i}{\partial t}\Omega_i &= \int_{\partial\Omega} F_i \cdot \vec{n}\, ds \\ &= \textstyle\sum_l F_{i,l} A_{i,l} \ , \end{aligned} \tag{14}$$

where $\Omega_i$ is the discrete volume of cell $i$, $l$ is the number of the face for cell $i$, $A_{i,l}$ is the corresponding surface area of the cell face, and $F_{i,l} = \vec{V}_{i,l} q_{i,l} \cdot \vec{n}$ is the flux at face $l$ for cell $i$. In the case of rectangular boxes $l$ ranges from 1 to 4 for each cell. The quantity $q$ however is not defined at the cell face, but in the center of the cell, and therefore some kind of interpolation is necessary.

## 4.2 Interpolation of the flux

To continue the discussion on flux interpolation we restrict ourself to the discretization using boxes. The quantity $q$ is given at the cell center of the box and the velocities are given normal to the cell face. In this case the grid reduces to a (logically) rectangular two dimensional grid. In this approach the finite volume grid coincides with the so-called Arakawa-C grid for finite differences [20], as shown in Figure 3. The quantities are given as cell centered averages at the whole points of the grid. The normal velocities at the cell faces are given at the half points of the grid. These kind of grids have the advantage that neighboring points are found by direct addressing, no addition adjacency information is therefore necessary.

After discretization with rectangular boxes of size $dx$, $dy$, $dz$ and application of the Gauss divergence theorem Eq. (12) on the integral of the advection equation Eq. (5) over the box we have

$$\begin{aligned} \tfrac{\partial q_{i,j}}{\partial t} dx\, dy\, dz = \ & \left(f(u,q)_{i+\frac{1}{2},j} - f(u,q)_{i-\frac{1}{2},j}\right) dy\, dz + \\ & \left(g(v,q)_{i,j+\frac{1}{2}} - g(v,q)_{i,j-\frac{1}{2}}\right) dx\, dz \ , \end{aligned} \tag{15}$$

where $f(u,q) = u\,q$ is the flux function in the $x$-direction and $g(v,q) = v\,q$ the flux function in the $y$-direction. With $f(q)_{i+\frac{1}{2},j}$ we denote that the arguments of the function

need to be evaluated at that particular point, in other words $f(u,q)_{i+\frac{1}{2},j} = u_{i+\frac{1}{2},j} q_{i+\frac{1}{2},j}$. As previously stated the quantity $q$ is only defined at the whole grid points, like $i+1, j$ and $i, j$. The value at $i+\frac{1}{2}, j$ is found by linear interpolation.

For the specification of linear interpolation schemes we make use of the Hildebrand difference operators [8] combined with the index free notation, which is a preliminary to the actual specification in CTADEL. In the index free notation, the half/whole grid point specification is a property of a specific field. For example the quantity $q$ is defined on whole grid points, therefore $q$ actually denotes $q_{i,j}$. The velocity field $u$ is staggered in the $x$-direction only, therefore $u$ actually denotes $u_{i+\frac{1}{2},j}$. This means that the flux function $f(u,q)$ in index free notation actually denotes $f(u_{i+\frac{1}{2},j}, q_{i,j}) = u_{i+\frac{1}{2},j}\, q_{i,j}$. The Hildebrand shift operator $E_x$ denotes a shift of a whole grid point in the $x$-direction. This means that $E_x\, q = q_{i+1,j}$ and $E_x\, u = u_{i+1\frac{1}{2},j}$. Larger shifts or negative shifts are given by a super index, a negative shift of two is denoted as: $E_x^{-2} q = q_{i-2,j}$. We introduce the linear interpolation function $F_x$, using the Hildebrand difference operators

$$F_x(f) = \frac{f + E_x(f)}{2} \;, \tag{16}$$

and analogously for the $y$-direction. Using the interpolation operator $F$ the flux functions at the cell faces can be interpolated as

$$\begin{array}{rclcrcl} f_{i+\frac{1}{2},j} & = & f(u, F_x(q)) & = & f(u, \frac{q+E_x(q)}{2}) & = & u_{i+\frac{1}{2},j} \frac{q_{i,j}+q_{i+1,j}}{2} \\ g_{i,j+\frac{1}{2}} & = & g(v, F_y(q)) & = & g(v, \frac{q+E_y(q)}{2}) & = & v_{i,j+\frac{1}{2}} \frac{q_{i,j}+q_{i,j+1}}{2} \;. \end{array} \tag{17}$$

The interpolation of $f_{i-\frac{1}{2},j}$ is similar to $f_{i+\frac{1}{2},j}$, but one grid point shifted.

The interpolation form for which the interpolation function is applied to the quantity $q$ only, it is called *state*-interpolation. The interpolation function $F_x$ could also be applied to the complete flux function, which gives

$$f_{i+\frac{1}{2},j} = F_x(f(u,q)) = \frac{f(u,q) + E_x(f(u,q))}{2} = \frac{u_{i+\frac{1}{2},j}\, q_{i,j} + u_{i+1\frac{1}{2},j}\, q_{i+1,j}}{2} \tag{18}$$

This form of interpolation is called *flux*-interpolation. Both forms of interpolation have a potentially different numerical behavior. For more details the reader is referred to [10,13].

The order of the finite volume method is altered by choice of the interpolation function. Linear interpolation results in a second order central scheme [10]. A first order, oscillation free, approximation is found by a one sided upwind scheme (given in index free notation)

$$F_x(f(u,q)) = \begin{cases} f(u,q) & \text{if } u > 0 \\ E_x(f(u,q)) & \text{otherwise} \;. \end{cases} \tag{19}$$

By using a larger stencil higher order interpolation functions are constructed. A wide variety of higher order interpolation schemes can be constructed using the $\kappa$-interpolation function introduced by van Leer [14]

$$F_x(f) = f + \frac{1+\kappa}{4}\left(E_x(f) - f\right) + \frac{1-\kappa}{4}\left(f - E_x^{-1}(f)\right) \tag{20}$$

with $-1 \le \kappa \le 1$. For the values $\kappa = 1, -1$ and $\frac{1}{3}$ the scheme is equal to respectively a second-order central, a second-order upwind, and a third order upwind biased scheme. Note that in the case of upwind schemes, the $\kappa$-interpolation function Eq. (20) should be extended with an upwind selector, like the first order upwind scheme Eq. (19).

## 4.3 Flux-Limiting methods

The construction of linear advection schemes ends up in a paradox: we either have a monotone, but first order and diffusive method or we have on non-monotone, but higher order and less-diffusive method. We would like to have a scheme which has the best of both worlds: a scheme which is of higher order and therefore less diffusive, but which still is monotone. As stated in Section 3 this can not be achieved with linear methods, because of the Godunov theorem, but instead we have to rely on non-linear methods.

The solution is to use a combination of the two schemes: use a first order monotone method and switch over to a higher order method where it is save to do so, such that no wiggles are introduced. Or seen from an opposite view, use a higher-order method and add some dissipation, which is to switch over to a first order method, to damp the ripples when strong gradients exist. The selection is done on the actual solution, which makes the method non-linear. This is the way flux-limiting methods work. Based on the smoothness of the solution a choice is made between a higher-order interpolation function ($F_x^H$) and a first order interpolation function ($F_x^L$). If the higher-order method is seen as a first-order method plus a correction, the limited flux interpolation function can be written as [15]

$$F_x = F_x^L + \Psi\left(F_x^H - F_x^L\right) \ , \tag{21}$$

where $\Psi$ is the switching function or limiter. When $\Psi$ is equal to zero we have a first order method and when it is equal to one we have a higher order method. Any value between zero and one results in a blend between the two methods.

The initial definition Eq. (21) of a flux limiter scheme allows a lot of freedom for the specification of a scheme, but following Zijlema and Wesseling [23] the class of one-dimensional flux-limiting methods can been reduced to a canonical form, where the interpolation function is given as a limited upwinding method and different schemes are found by choosing a specific limiter function. The interpolation function is given as

$$F_x(f) = \begin{cases} f_i + \frac{1}{2}\Psi(r_+)\left(f_i - f_{i-1}\right) & \text{if } u_{i+\frac{1}{2}} > 0 \\ f_{i+1} + \frac{1}{2}\Psi(r_-)\left(f_{i+2} - f_{i+1}\right) & \text{otherwise} \ , \end{cases} \tag{22}$$

or given in index-free notation with Hildebrand operators

$$F_x(f) = \begin{cases} f + \frac{1}{2}\Psi(r_x^+)\Delta f & \text{if } u > 0 \\ E_x f + \frac{1}{2}\Psi(r_x^-)\Delta E_x f & \text{otherwise} \ , \end{cases} \tag{23}$$

where the operator $\Delta$ denotes the forward difference $\Delta f_i = f_{i+1} - f_i$. The functions $r^+$ and $r^-$ are called smoothness monitor and give an indication of the smoothness of the solution and therefore an indication of the application anti-diffusive term. They are defined as

$$\begin{aligned} r_x^+(q) &= \frac{q_{i+1} - q_i + \epsilon}{q_i - q_{i-1} + \epsilon} = \frac{\Delta q + \epsilon}{\nabla q + \epsilon} \\ r_x^-(q) &= \frac{q_{i+1} - q_i + \epsilon}{q_{i+2} - q_{i+1} + \epsilon} = \frac{\Delta q + \epsilon}{\Delta E_x q + \epsilon} \ , \end{aligned} \tag{24}$$

where $\nabla$ denotes the backward difference $\nabla f_i = f_i - f_{i-1}$ and $\epsilon$, a small enough number (e.g. $\epsilon = 10^{-7}$), to protect the smoothness monitors against division by zero, which is the case for uniform solutions.

Monotonicity preserving schemes are found by constructing real limiters following the monotonicity domain of Sweby [18], as is shown in Figure 4. In order to be monotonicity

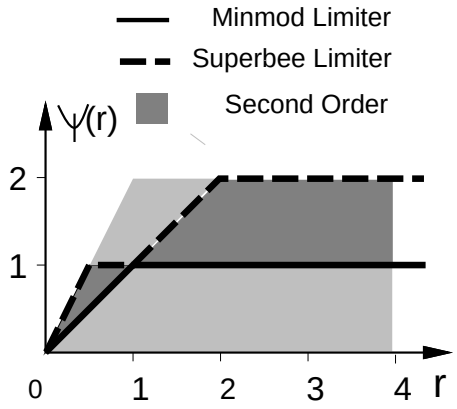


Figure 4: The Sweby Monotonicity Domain

preserving, the value of $\Psi$ should be inside the grey area for every value of $r$. The dark grey area denotes the second order monotonicity preserving area. Based on the monotonicity domain of Sweby, a wide variety of limiters exist. For example the lower bound of second order monotone schemes is outlined by the *Minmod* limiter

$$\Psi(r) = max(0, min(r, 1)) \; . \tag{25}$$

A limiter which follows the upper bound of the Sweby domain is the *Superbee* limiter

$$\Psi(r) = max(0, min(2r, 1), min(r, 2)) \; . \tag{26}$$

All limiters constructed following the monotonicity domain of Sweby fall in the domain set out by these two limiters [18]. There is no such a thing as the ultimate limiter, but the choice of a limiter depends on the modeled problem.

Within the framework of canonical form Eq. (22), a linear $\Psi$ operator can be constructed, which resembles the linear non-monotone $\kappa$ scheme Eq. (20). This operator is given as

$$\Psi(r)_\kappa = \frac{1}{2}(1+\kappa)\, r + \frac{1}{2}(1-\kappa) \; , \tag{27}$$

with $-1 \leq \kappa \leq 1$. A limited version of the $\kappa$ scheme is found by limiting Eq. (27) to the Sweby domain, Figure 4, resulting in the following limiter

$$\Psi_\kappa(r) = max[0, min(2, \frac{1}{2}(1+\kappa)r + \frac{1}{2}(1-\kappa), 2r)] \; , \tag{28}$$

with $-1 \leq \kappa \leq 1$. The limited form of the $\kappa = \frac{1}{3}$ third order upwind biased scheme is found for $\kappa = \frac{1}{3}$, see [13].

# 5 Abstract Specification in CTADEL

The specification of finite volume schemes consist of three parts: the definition of the domain by cells, the definition of the flux interpolation functions and the cell update function. As an example of how finite volume schemes are specified in CTADEL we consider

| Name | Symbol | Notation | Meaning |
|---|---|---|---|
| Shift | $E_x$ | `fd_shift(F,X)` | $E_x f_i = f_{i+1}$ |
| | $E_x^N$ | `fd_shift(F,X,N)` | $E_x^N f_i = f_{i+N}$ |
| Forward Difference | $\Delta_x$ | `fd_forward(F,X)` | $\Delta_x f_i = f_{i+1} - f_i$ |
| Backward Difference | $\nabla_x$ | `fd_backward(F,X)` | $\nabla_x f_i = f_i - f_{i-1}$ |
| Central Difference | $\delta_x$ | `fd_central(F,X)` | $\delta_x f_i = f_{i-1} - f_{i+1}$ |

Table 1: The Hildebrand difference operators

the two-dimensional advection equation. We use rectangular boxes as cell form, such that the finite volume grid fits in the logically rectangular grid with discrete size $(i, j) \in [1..\ell] \times [1..m]$, as was discussed in Section 4.2.

The first part of the CTADEL specification is given by the declaration of the concentration field `q` and the normal velocity fields `u` and `v` at the cell faces

```
q    :: real(0 .. infinity) ~ "kg/m^3"   field (x(grid), y(grid)) on i=1..l by j=1..m.
u    :: real ~ "m/s"   field (x(half), y(grid)) on i=0..l by j=1..m.
v    :: real ~ "m/s"   field (x(grid), y(half)) on i=1..l by j=0..m.
```

The fields are all of type real. Range information and units are declared if appropriate. The concentration is defined as an positive real and since it is a cell centered value as is shown in Figure 3, it is defined at the whole grid points in each direction, denoted by `x(grid), y(grid)` grid staggering type. The `u` and the `v` fields are given at the cell faces in the $x$-direction and $y$-direction respectively. After the specification of the grid staggering, the size of the fields are given. Note that the domains for the `u` and `v` are one point larger in their particular direction. After definition of the staggering information, the size of the domain is set.

The second part of the specification consists of the definition of the cell face interpolation functions. We use a field for the cell faces in each direction. The declaration of the cell faces field is similar to the velocity fields

```
f   :: real ~ "kg/s"   field (x(half), y(grid)) on i=0..l by j=1..m.
g   :: real ~ "kg/s"   field (x(grid), y(half)) on i=1..l by j=0..m.
```

We now have to define the interpolation functions itself. Expressions in CTADEL are given by implicit indexing with the use of the Hildebrand difference operators as was shown in Section 4.2. The operators, their definition and their CTADEL name are shown in Table 1.

With the use of these operators the difference expressions are specified which define the cell face interpolation functions. As an example we consider the first order upwind scheme, Eq. (19), with zero inflow boundaries. The cell face in the $x$ and $y$-direction are defined as is shown in Figure 5.

Three operators are introduced for the specification of conditional expressions. A conditional expression is denoted by the binary infix operator `if`. The first argument is the expression and is only assigned if the second argument the condition evaluates to true. Conditional expressions with two branches are constructed by using the binary case operator `\\`. Multiple branches are denoted by appending extra operators. The postfix `otherwise` operator denotes the last branch in a case construct and is added for readability. Nested conditionals are constructed by using parenthesis to denote the associativity. By using the operators, we are able to define the uds scheme from Figure 5 in CTADEL as

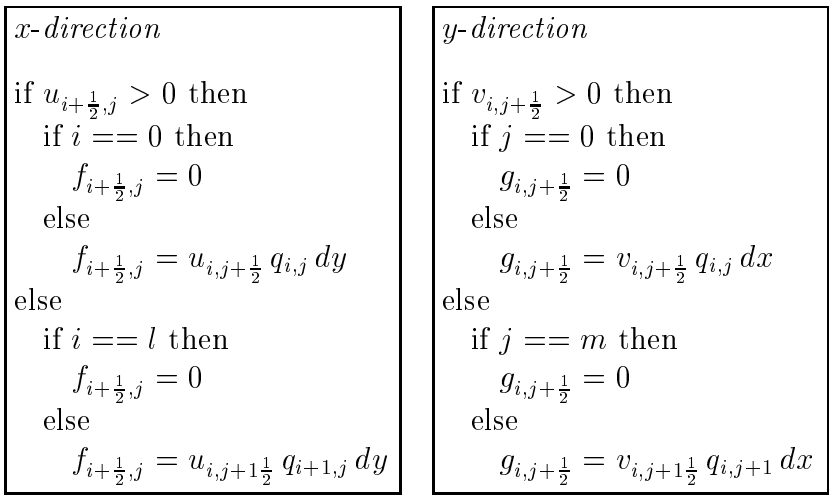


Figure 5: Flux definition for UDS Scheme

```
f = (0                   if i==0 \\
     u*q*dy              otherwise) if u > 0 \\
    (0                   if i==l \\
     u*fd_shift(q,x)*dy  otherwise).
g = (0                   if j==0 \\
     v*q*dx              otherwise) if v > 0 \\
    (0                   if j==m \\
     v*fd_shift(q,y)*dx  otherwise).
```

The third part is the definition of the update function for the cell (its time derivative) Eq. (14). The cell update is the sum of all cell faces.

$$\frac{\partial q_{i,j}}{\partial t} = -\frac{f_{i,j} - f_{i-1,j} + g_{i,j} - gi, j - 1}{dy\, dx}\ . \tag{29}$$

The update for the two-dimensional problem is given as

```
dqdt :: real(0 .. inf) ~ "kg/m^3/s" field (x(grid), y(grid)) on i=1..l by j=1..m.
dqdt = -(fd_backward(f,x) + fd_backward(g,y))/(dy*dx).
```

## 5.1 Exploiting Structural Symmetry using Operators

There is a structural equivalence between the definition of the interpolation function for the two cell faces. From a local point of view, that is seen from the cell face itself, the schemes are exactly the same. Due to the logically rectangular grid, this results globally in an interchange of indices. An advantage over regular programming languages is that this structural equivalence can be exploited by using operators. This allows the first order upwind scheme to be written as

```
uds(Q,U,X) := (0             if pnt(X) == lb(pnt(X)) \\
               U             otherwise) if U>0  \\
              (0             if pnt(X) == ub(pnt(X)) \\
               fd_shift(U,X) otherwise)
f = u*uds(q,u,x)*dy.
g = v*uds(q,v,y)*dx.
```

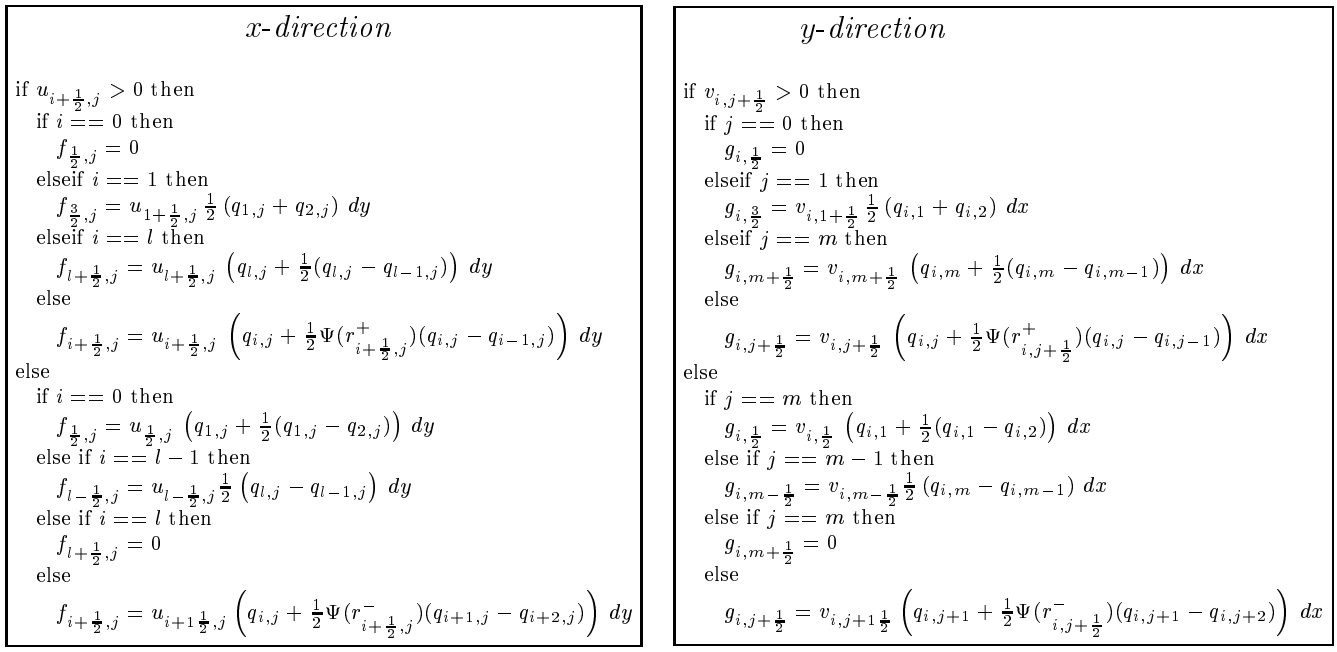


Figure 6: Flux definition for the limited $\kappa = \frac{1}{3}$ scheme

The capital identifiers `Q`, `U` and `X` denote formal parameters, to be replaced by actual parameters when the operator is actually referenced. The operator `pnt` returns the discrete index of a continuous coordinate. The `lb` and `ub` operators return the lower and upper bound of the defined field in the specified direction, so `lb(i)` returns `0` for `f` and `1` for `g` and `ub(i)` returns `l` for `f` and `m` for `g`.

The use of symbolic operators also simplifies the specification of *state* versus *flux* interpolation functions as was discussed in Section 4.2. In the examples above a state interpolation is given. A flux interpolation is simply defined by putting the multiplication with the velocity field inside the operator

```
f = uds(u*q,u,x)*dy.   % Flux interpolation
g = uds(v*q,v,y)*dx.   % Flux interpolation
```

## 5.2 Specification of Flux Limiter Schemes

Within the current framework the specification of flux limiter schemes follows naturally. The interpolation function becomes more complicated, since it is extended to higher order schemes.

The higher order schemes have larger stencils and order reduction is necessary to reduce the stencil near the boundary. As an example we consider the specification of the limited $\kappa = \frac{1}{3}$ scheme given in Section 4.3. The scheme consists of an upwind and a downwind branch. Each of the branches contains the scheme and three exceptions near boundaries to reduce the stencil, shown in Figure 6 [13]. We first declare the two slope monitor functions $r^+$ and $r^-$ as defined in Eq. (24)

```
rp(Q,X) := (fd_forward(Q,X)+eps)/(fd_backward(Q,X)-eps).
rm(Q,X) := (fd_forward(Q,X)+eps)/(fd_forward(fd_shift(Q,X),X)-eps).
```

At the boundaries the central scheme Eq. (16) is also used. This scheme is as declared as

```
% Van Leer Kappa Scheme
kappa(R, K)              := 1/2*(1+K)*R + 1/2*(1-K) if -1 <= K and K <= 1 \\
                            undefined               otherwise.

% Differential Limiters
vanleer(R)               := (R+abs(R))/(1+R).
albada(R)                := (R^2+R)/(1+R^2).
ospre(R)                 := 3/2*(R^2+R)/(1+R+R^2).

% Non Differential Limiters
sweby(R, P)              := max(0,max(min(P*R,1),min(R,P))).
chakravarthyosher(R, P)  := max(0,min(R,P)).
mc(R)                    := max(0,min((1+R)/2,min(2,2*R))).
plk(R, K, M)             := max(0,min(M,min(kappa(R,K),2*R))).
splk(R, K, M)            := max(0,min(min(M,kappa(R,K),2*R),min(1/2*(1-K)*R+1/2*(1+K),2*R))).

rk(R, K)                 := (R+abs(R))*(-R^2+(3*K)*R-K)/(1+R)^2  if R <=1 and -1 <= K and K < 0 \\
                            (2+K)*R-K/(1+R)                      if R > 1 and -1 <= K and K < 0 \\
                            (R+abs(R))*((1+K)*R+1-K)/(1+R)^2     if 0 <= K and K <= 1 \\
                            undefined                            otherwise.

% Derived Limiters
superbee(R)              := sweby(R,2).
minmod(R)                := chakravarthyosher(R,1).              % Sweby(R,1).
davis(R)                 := plk(R, -1, 1).
koren(R)                 := plk(R, 1/3, 2).
smart(R)                 := plk(R, 1/2, 4).
umist(R)                 := splk(R, 1/2, 2).
muscl(R)                 := splk(R, 0, 2).
```

Figure 7: CTADEL specification of Limiters

```
cds(Q,X) := (Q + fd_shift(Q,X))/2.
```

The limited $\kappa$ scheme with order reduction is defined by the following operator, which simply replaces the `uds` operator in the UDS example of Section 5

```
kscheme(Q,U,X)  :=
  (0                                                          if pnt(X)==lb(pnt(X))   \\
   cds(Q,X)                                                   if pnt(X)==lb(pnt(X))+1 \\
   Q+1/2*fd_forward(Q,X)                                      if pnt(X)==ub(pnt(X))   \\
   Q+1/2*koren(rp(Q,X))*fd_forward(Q,X)                       otherwise
  ) if U > 0 \\
  (fd_shift(Q,X) - 1/2*fd_forward(fd_shift(Q,X),X))           if pnt(X)==lb(pnt(X))   \\
   cds(Q,X)                                                   if pnt(X)==ub(pnt(X))-1 \\
   0                                                          if pnt(X)==ub(pnt(X))   \\
   fd_shift(Q,X) - 1/2*koren(rm(Q,X))*fd_forward(fd_shift(Q,X),X) otherwise
  ).
```

The operator `koren` is the $\kappa = \frac{1}{3}$ limiter $\Psi$ and is taken from a CTADEL library, given in Figure 7. The limiters are defined as symbolic operators. The library is constructed by recursive application of more generic functions. For example the Piecewise Linear Kappa limiter (`plk`) is based on the linear $\kappa$ scheme (`kappa`). The derived $\kappa = \frac{1}{3}$ limiter is an instance of this `plk` limiter.

# 6 Code Generation

The definition of the flux interpolation function in all directions consist of a conditional expression. The conditionals either denote scheme exceptions resulting from order reduction

**A.** *Fully Conditional Code*

```
DOALL j = 1, m

  DOALL i = 0, l

    IF (0.LT.u(i,j)) THEN
      IF (0.EQ.i) THEN
        f(i,j) = 0E0
      ELSE
        f(i,j) = q(i,j)*u(i,j)*dy
      ENDIF
    ELSE
      IF (i.EQ.l) THEN
        f(i,j) = 0E0
      ELSE
        f(i,j) = q(i+1,j)*u(i,j)*dy
      ENDIF
    ENDIF

  ENDDOALL

ENDDOALL
```

**B.** *Loop Partitioned Code*

```
DOALL j = 1, m
  DOPAR
    DOALL i = 1, l-1
      IF (0.LT.u(i,j)) THEN
        f(i,j) = q(i,j)*u(i,j)*dy
      ELSE
        f(i,j) = q(i+1,j)*u(i,j)*dy
      ENDIF
    ENDDO

    IF (0.LT.u(0,j)) THEN
      f(0,j) = 0E0
    ELSE
      f(0,j) = q(1,j)*u(0,j)*dy
    ENDIF

    IF (u(l,j).LE.0) THEN
      f(l,j) = 0E0
    ELSE
      f(l,j) = q(l,j)*u(l,j)*dy
    ENDIF
  ENDDOPAR
ENDDOALL
```

**C.** *Fully Distributed Code*

```
DOPAR
 DOALL j = 1, m
  IF (0.LT.u(0,j)) THEN
    f(0,j) = 0E0
  ENDIF
 ENDDOALL
 DOALL j = 1, m
  IF (u(l,j).LE.0) THEN
    f(l,j) = 0E0
  ENDIF
 ENDDOALL
 DOALL j = 1, m
  DOALL i = 0, l-1
   IF (u(i,j).LE.0) THEN
     f(i,j) = q(i+1,j)*u(i,j)*dy
   ENDIF
  ENDDOALL
 ENDDOALL
 DOALL j = 1, m
  DOALL i = 1, l
   IF (0.LT.u(i,j)) THEN
     f(i,j) = q(i,j)*u(i,j)*dy
   ENDIF
  ENDDOALL
 ENDDOALL
ENDDOPAR
```

Figure 8: Canonical Code Forms

or denote upwind-downwind branches. Fields are implemented using multi-dimensional arrays and are computed by loop constructs. The naive implementation of the abstract CTADEL specification, as discussed in Section 5, in Fortran will result in perfectly nested conditional loops. The conditionals for the order reduction near the boundaries are simple checks on the loop variables, which are determinable at run-time. This optimization of the code is not straightforward and we will therefore discuss this transformation in this section.

We consider again the first order UDS scheme with zero inflow boundaries for a two dimensional rectangular domain as defined in Section 5. It serves as an example of the complexity of the interpolation specification. For higher order schemes such as the limited $\kappa = \frac{1}{3}$ scheme from Section 5.2 the complexity increases as higher order schemes have more exceptions on the loop indices and also the expressions are more complex. In Figure 8 three different codes for the computation of the UDS scheme cell face interpolation in the $x$-direction are given. The loop for the computation of the cell face in the $y$-direction is similar but with shifts applied to the opposite index.

The first code **A** is the direct implementation of the original abstract CTADEL specification in Section 5. This code is easy readable, but there is overhead due to the conditionals on the loop variables. The second code **B** is semantically equivalent to **A**. The conditionals on the loop variables are optimized as a result of loop partitioning. This code has the least serial overhead and will perform best on serial architectures. The third code **C** is the fully distributed form and is semantically equivalent to codes **A** and **B**, but with the loops distributed and again the index variable conditionals optimized. This code will perform best on vector machines, since these statements are translated to well vectorizable guarded assignments. We call codes **A** till **C** canonical forms for the computation of the cell faces for logically rectangular domains.

These three code forms are semantically equivalent, but will perform differently on the various architectures. Code restructuring will therefore be necessary to assure the performance of the generated code. Code **A** however does not vectorize very well. Conditional loop nests are vectorized using mask arrays (i.e. logical arrays) following the *if-conversion* technique [24]. However no distinction is made between conditionals for loop indices and other kinds of conditionals leading to the introduction of unnecessary logical arrays. The difference between **B** and **C** is that code **B** has `IF-THEN-ELSE` statements and code `C` has `IF-THEN` statements. If the latter is more efficient on vector machines, the compiler should be able to transform code **B** into **C**, but this will result in six loops (three times `IF-THEN-ELSE` instead of the four loops of **C** and therefore will always perform slightly worse.

The translation from code **A** to either code **B** or **C** is done by loop partitioning [12]. However, the conditional on the loop index is inside the upwind conditional. It is not permitted to just simply interchange the conditionals. This would mean a violation of semantics: The body of the conditional – in this case another conditional – is evaluated before the condition is evaluated to true. But in the case of the upwind-downwind selection this will not cause any harm and therefore it is clear that in cases like this it *is* allowed to inter change the conditionals. We found that restructuring compilers do not perform this loop partitioning, since they do not make this distinction between conditionals being interchangeable or not, and we have therefore added the generation of the various code forms to the target specific code transformations of CTADEL.

## 6.1 Code Restructuring in CTADEL

Within CTADEL system, code fragments are treated as algebraic expressions [21]. At the intermediate level, all statements are seen as operators. This is similar to the way conditionals are treated at the input level, discussed in Section 5. Code restructuring (either semantics preserving or not) is performed by the General Purpose Algebraic Simplifier (GPAS). Transformations are constructed by definition of sets of simple rewrite rules. Rules from these rule sets are applied by GPAS on an algebraic expression until no rule can be applied anymore, thereby transforming the expression from one canonical form to another canonical form. A similar restructuring approach has been adopted by Boyle [1]. Rule sets can either be combined in serial or in parallel. In serial composition, the rule sets are applied one after the other. In parallel composition rule sets are intermixedly applied.

Unlike optimizing compilers, the transformations in CTADEL are not applied in a black-box fashion. Because CTADEL is a open and fully interactive environment, the user has full freedom to apply arbitrary rule sets to the desired expressions. The user decides which code form **A**, **B** or **C** will be produced. This decision is translated to a sequence of rule sets. GPAS will then transform the initial code form to the desired code form. The transformations described here are only applied to the cell face interpolation functions.

## 6.2 The Transformation Steps

We will now discuss the transformation process step by step. We recall the template for the computation of the flux interpolation function of the UDS scheme after its initial specification, Figure 9. It is given in a pseudo Fortran. The `DOALL` statement is introduced

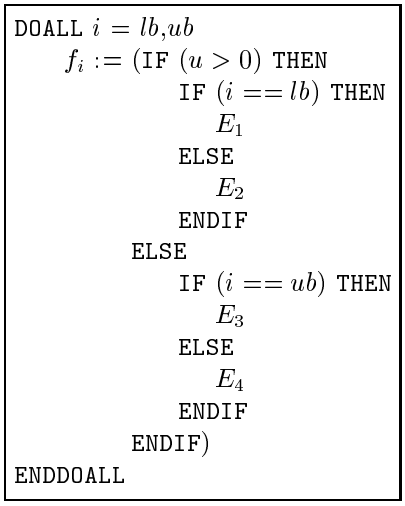


Figure 9: Template Upwind Scheme

due to the semantics of the declarative assignment from the initial specification. The if-statements are actually conditional expressions and there are no data dependencies ($f$ does not occur in the expressions $E_k$ , $k \in [1, 4]$).

In order to retrieve code form **A**, it is sufficient to propagate the assignment inside the conditional statements, such that the code is in imperative form, instead of functional form. To retrieve code forms **B** or **C** four rule sets and an intermediate code form is introduced where the if-nest is decoupled to a set of guarded assignments. The first rule set transforms the initial specification to the decoupled intermediate form. After decoupling the conditionals on the loop variables flattened by the second rule set. Flattening allows the conditions to be interchanged such that the conditionals on the index variables can be optimized. We have now retrieved code form **C**. Loopfusion, performed by the fourth rule set will transform code form **C** to code form **B**. We will now discuss the four rule sets.

### 6.2.1 IF-decoupling

The first decoupling stage of the transformation process consists of three distinct steps: expansion of conditionals, assignment distribution and doall distribution.

In the first transformation step the conditionals are fully expanded by propagation of the negated conditions to the other branches. This is allowed since there are no dependencies between the various branches of the if-nest. Therefore the expanded if-nest can be executed in parallel, resulting in a fully decoupled, commutable set of guarded assignments. The transformation is given as

$$
\begin{array}{l}
f_i := \texttt{IF } (c_1) \texttt{ THEN} \\
\qquad E_1 \\
\quad \texttt{ELSE} \\
\qquad E_2 \\
\quad \texttt{ENDIF)}
\end{array}
\quad \Rightarrow \quad
\begin{array}{l}
f_i := \texttt{DOPAR} \\
\qquad \texttt{IF } (c_1)\ E_1 \\
\qquad \texttt{IF } (\neg c_1)\ E_2 \\
\quad \texttt{ENDDOPAR}
\end{array}
$$

where the left part of rule denotes the pattern to be matched and the right part denotes the pattern to be substituted. If-statements are expanded from the inner loop to

the outer loop, such that nested if's are decoupled as well. After expansion of the if-nest the assignment is distributed over the decoupled if-nest, such that the imperative form is retrieved, by the following rule

```
f_i :=DOPAR                          DOPAR
        IF (c_1) E_1                     IF (c_1) f_i := E_1
        IF (c_2) E_2        ⇒            IF (c_2) f_i := E_2
    ENDDOPAR                         ENDDOPAR
```

After decoupling of the if-statements, the loop is fully distributed by propagation of the doall loop over the statements.

```
                                 DOPAR
DOALL i = lb,ub                      DOALL i = lb,ub
    DOPAR                                S_1
        S_1                          ENDDOALL
        S_2          ⇒               DOALL i = lb,ub
    ENDDOPAR                             S_2
ENDDOALL                             ENDDOALL
                                 ENDDOPAR
```

After expansion, distribution and transformation of the logical expressions to DNF form, the initial template Figure 9 is transformed to the intermediate canonical form, shown in following code fragment

```
DOPAR
    DOALL i = lb,ub
        IF (u > 0) IF (i == lb) f := E_1
    ENDDOALL
    DOALL i = lb,ub
        IF (u > 0) IF (i <> lb) f := E_2
    ENDDOALL
    DOALL i = lb,ub
        IF (0 ≥ u) IF (i == ub) f := E_3
    ENDDOALL
    DOALL i = lb,ub
        IF (0 ≥ u) IF (i <> ub) f := E_4
    ENDDOALL
ENDDOALL
```

In this canonical form, the if-nest is fully decoupled by expansion. We now have a set of parallel executable (doubly) guarded assignments.

### 6.2.2 IF-conversion

The simplification of the conditionals on the loop variables is hindered by the fact that they are "inner" conditionals, that is guarded by the upwind-downwind conditionals. Straightforward simplification would be a violation of semantic constraints, but as already stated this is not always so. It is possible to exchange the conditionals in some cases. In CTADEL, the absence of side effects allows interchanging of conditionals that are comprised of so-called *total* expressions. The concept of total expressions originates from calculus, where a total function denotes a function which is defined on the total domain. We define total expressions inductively as follows

**Definition** A function $f : D \to R$ with domain $D$ and range $R$, where $D$ and $R$ are either the complete boolean, integer, real or complex domain, is called *total* if for all $x \in D$ we have that $f(x) \in R$. An algebraic expression $e$ is called a *total expression* if

- $e$ consists of a constant or a variable;
- $e$ is a function application $e = f(e_1, \ldots, e_k)$ such that function $f$ of arity $k > 0$ is a total function and $e_1, \ldots, e_k$ are total expressions.

Examples of total functions are `sin`, `abs` and multiplication, but also boolean operators like `or` and the greater-than operator ($>$). Non-total functions are division and square root. In our algebraic environment we consider array references as total functions as well. Domain inference on the index variables is performed to the define the actual dimensions of the array such that totality of array references is guaranteed. Totality is tested similar to its definition. First the atomic terms are tested for totality (constants and variables are assumed so) and then the complete expression is checked by testing all its function compositions.

After testing the "inner" conditional expressions for totality, the nested-if is flattened by combining conditions to a single conjunction. The conjunctive operator is considered a commutative operator and therefore allows the interchange of the conditionals. This transformation is defined as

$$\texttt{IF } (c_1) \texttt{ IF } (c_2)\ f := E_1 \quad \underset{\Rightarrow}{\textbf{if } c_2 \textbf{ is total}} \quad \texttt{IF } (c_1 \wedge c_2)\ f := E_1$$

Now that the if's are flattened, it is possible to interchange the conditions, such that the tests on loop variables can be optimized.

## 6.3 Index Conditional Optimization

The specification of cell face interpolation functions contain tests on loop variables, where the tests are either equalities or inequalities to lower and upperbounds of the loops. Two rules are applied to optimize these tests. The first rule is the equality test

$$\begin{array}{l}\texttt{DOALL } i = l,u \\ \quad \texttt{IF } (i == c \wedge C_r)\ S \\ \texttt{ENDDOALL}\end{array} \quad \Rightarrow \quad \texttt{IF } (c \geq l \wedge c \leq u) \texttt{ IF } ([C_r]_{i=c})\ [S]_{i=c}$$

The operator $[C]_{i=c}$ denotes that a certain function $C$ is evaluated with index $i = c$. In order to assure that the transformation is semantically equivalent, a conditional is introduced to test if the iteration is within the iteration space. This guard can be removed at generation time by range inference, which may find that the condition is true. The range information is extra application information, which is otherwise not available in regular programming languages.

Another simplification is removal of the inequality to lower or upper bounds, which can be considered as a kind of loop peeling. The simplification of the inequality concerning the lower bound is given as

```
DOALL i = lb,ub                          DOALL i = lb + 1,ub
    IF (i <> c) S      if c == lb            S
ENDDOALL                  ⇒              ENDDOALL
```

and analogous for the upper bound.

After these transformation the fully distributed code **C** of the initial specification code is found. From this code, code **B** is found by loopfusion and (re-)combining if-statements.

### 6.3.1 Loop Fusion

Loop fusion is the just the inverse transformation of loop distribution transformation. For the kind of schemes considered here however the fusion is hindered by unequal loop bounds. The fusion is continued after the equalization of all loops to the same inner domain. This is performed by a loop peeling transformation, in which first the lower bounds are equalized and than the upper bounds. Loop interchange for the combination of the various loops is allowed, since the loops are composed by the parallel statement composition operator.

The equalization of the lower bounds is defined by the following rule

```
DOPAR                                    DOPAR
    DOALL i = l1,u1                          DOALL i = l1,l2 − 1
        S1                                       S1
    ENDDOALL                                 ENDDOALL
    DOALL i = l2,u2          if l1 < l2      DOALL i = l2,u1
        S2                       ⇒               S1
    ENDDOALL                                 ENDDOALL
ENDDOPAR                                     DOALL i = l2,u2
                                                 S2
                                             ENDDOALL
                                         ENDDOPAR
```

After the equalization of the lower bounds, the equalization of the upper bounds is performed, which is given as

```
DOPAR                                    DOPAR
    DOALL i = l,u1                           DOALL i = l,u2
        S1                                       S1
    ENDDOALL                                 ENDDOALL
    DOALL i = l,u2           if u1 > u2      DOALL i = u2 + 1,u1
        S2                       ⇒               S1
    ENDDOALL                                 ENDDOALL
ENDDOPAR                                     DOALL i = l,u2
                                                 S2
                                             ENDDOALL
                                         ENDDOPAR
```

After equalization more loopfusion can be performed. The fused statements contain only conditionals for upwind-downwind selection. After fusion the code is further optimized by combining upwind-downwind pairs.

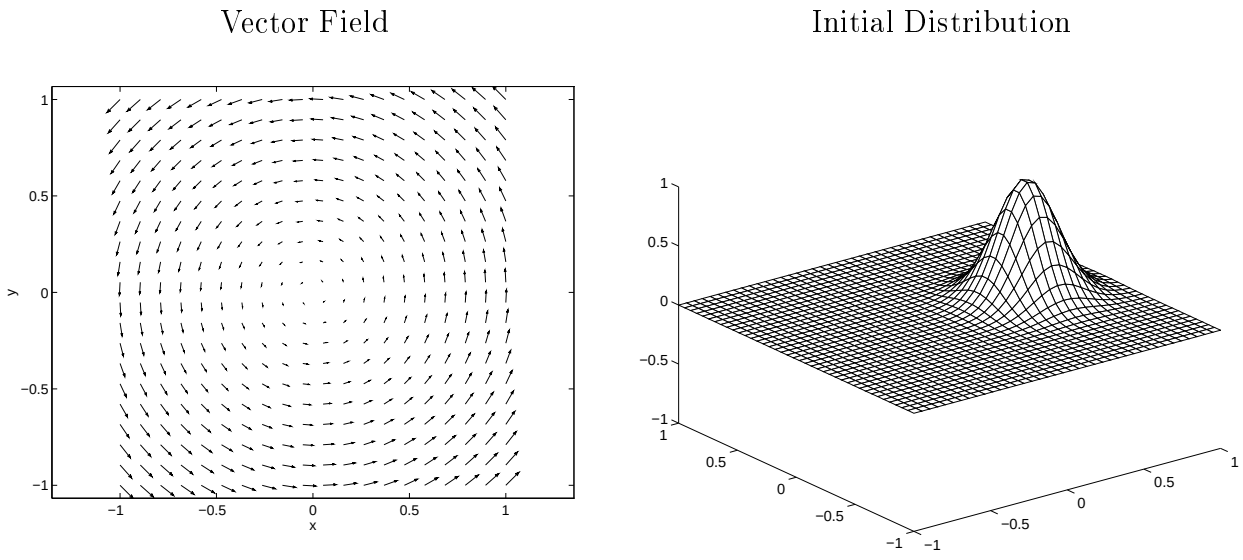


Figure 10: The Molenkamp-Crowley Test

```
IF (c1) THEN
    S1
ENDIF
IF (c2) THEN
    S2
ENDIF
```

**if** $c_1 == \neg c_2$ $\Rightarrow$

```
IF (c1) THEN
    S1
ELSE
    S2
ENDIF
```

We use here another criterium for the combination of if-statements than used in [19]. This is a more strict condition. The check is performed by structurally comparing the conditions in DNF form, like in [19]. Because of this potential further combination of upwind-downwind pairs, loop normalization instead of peeling to equalize the loops is not performed. It would simply ruin the possibilities for combining conditionals.

# 7 Performance

In the previous section we have discussed the generation of the canonical code forms for the computation of the cell face interpolation functions. We will now discuss the performance of these canonical code forms for serial and vector architectures. As a test case we take the two dimensional Molenkamp-Crowley test [22]. This is a well accepted test for the numerical qualities of advection schemes. In this test the propagation of an initial distribution in a rotating flow within a rectangular domain is measured, see Figure 10. Since the exact solution of the advection equation after one rotation is equal to the initial distribution, the test gives a good indication on the numerical qualities of the applied scheme. It fits our purposes as well, because it allows us to measure the computational performance of the different canonical code forms. Due to the rotating flow a fifty percent hit rate for the upwind/downwind conditionals is achieved.

We consider the limited $\kappa = \frac{1}{3}$ scheme, completed with order reduction near the boundaries as was discussed in Section 5.2. The scheme is tested on a domain of 20 by 20 grid points to a domain of 240 by 240 grid points with an increase of 20 by 20 grid points.

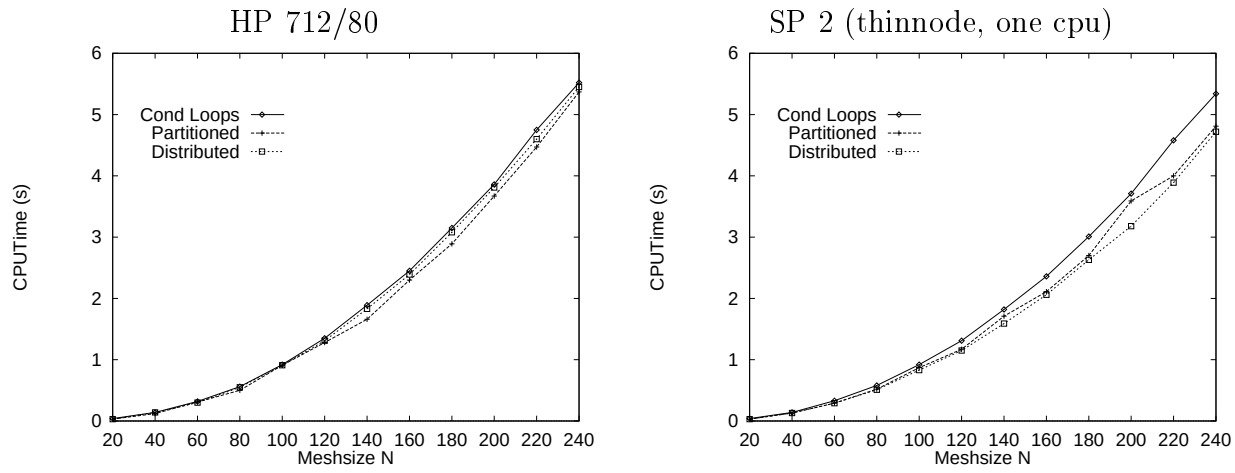


Figure 11: Timings for the serial architectures

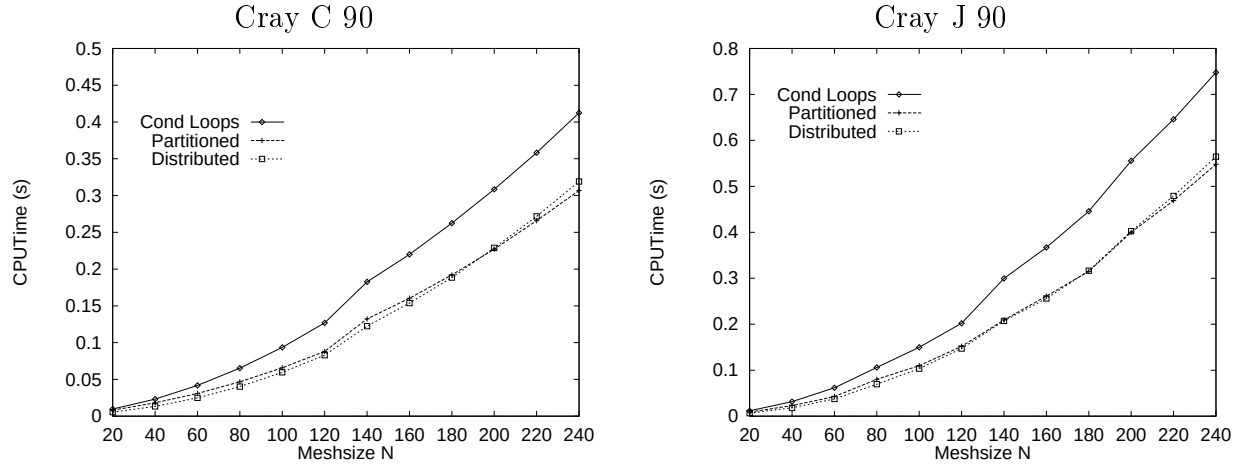


Figure 12: Timings for the vector architectures

The tested platforms are two serial architectures: a HP 712/80 Workstation (`fort77 +O2`) and a single-cpu thinnode SP2 (`xlf -O3`) and two vector architectures: a Cray C 90 (`cft -Oscalar3 -Ovector3`) and a Cray J 90 (`cft -Oscalar3 -Ovector3`), both on one CPU. For the test the first ten time steps of a Four Stage Runge Kutta timestepping scheme are timed, so the interpolation function and the update are computed 40 times.

In Figure 11 and Figure 12 timings are shown for serial and vector architectures respectively. On the HP there is hardly any difference between the three codes. On the SP 2 there is also not much difference. Surprisingly the distributed code performs slightly better than the partitioned code. The loop overhead is larger than the overhead of the conditionals. For serial architectures the performance is bound by the costs for memory references. Which is the reason why only a moderate improvement is achieved. On vector architectures the distributed code performs best. It was expected that codes **B** and **C** would perform better than code **A**, since the latter is vectorized using the if-conversion technique [24]. This will introduce new logical vector arrays and operations, which was actually confirmed by the `hpm` hardware performance monitor for the Cray machines. The

compiler is presumable able to transform code **B** and **C** to an almost equal form, which could explain why there is hardly any difference between those two code forms.

# 8 Summary and Further Research

In this paper we have discussed the relation between the finite difference and finite volume discretization methods. Finite difference discretizations are easily automated by using operator overloading techniques. Instead, the finite volumes derivation relies on integral transformations. We focused on the specification of such methods within CTADEL. Some new transformations where introduced to allow efficient encoding of the conditions in order to achieve optimal code. We showed that code generations allow *very fast* change of schemes. By code generation, knowledge on efficient implementation is hidden from the user. From this better readable and understandable specifications are possible.

Currently we are incorporating the generation activities within the DELWAQ model itself. Our goal is to develop new schemes based on multi dimensional flux limiter methods and implicit time integration for this model. The ability of automatic code generation for implicit time integration methods is very interesting, since a minor change in the numerical specification (time level $n$ to $n+1$) involves heavy programming to implement this change of specification. The generation of multi-dimensional schemes requires extra study. Such schemes are not as well documented as one-dimensional schemes and the implementation of such schemes is less straightforward.

# Acknowledgements

The authors like to thank Paul ten Brummelhuis and Mart Borsboom of Delft Hydraulics for providing the subject and for the many discussions. The CTADEL project was initiated in close cooperation with Gerard Cats of KNMI, the Royal Dutch Meteorological Office, who is grately thanked for careful reading and pointing out the relevance of the subject for weather modeling. Barry Koren from the Mathematical Institute in Amsterdam (CWI) is also thanked for providing background knowledge on advection modeling.

# References

[1] James M. Boyle, Terence J. Harmer, and Victor L. Winter. The TAMPR program transformation system: Design and applications. In E. Arge, A.M. Bruaset, and H.P. Langtangen, editors, *Modern Software Tools for Scientific Computing*. Brikhauser, 1997.

[2] Delft Hydralics, P.O.Box 177, 2600 MH Delft, The Netherlands. *Delwaq Users Manual, Version 4.0*, 1994.

[3] P. Källén (Ed). *Hirlam Documentation Manual*. SMHI, S-60176 Norrköping, Sweden, 1996.

[4] Robert W. Fox and Alan T. McDonald. *Introduction to Fluid Dynamics*. John Wiley and Sons, New York, third edition, 1995.

[5] S.K. Godunov. Finite difference method for numerical computation of discontinuous solutions of the equations of fluid dynamics. *Matematicheskii Sbornik*, 47:271–306, 1959. (Cornell Aeronautical Lab. Translation from Russian).

[6] V.V. Goldman, J.A. Van Hulzen, A.E. Mynett, A.S. Posthuma, and H.J Van Zuylen. The application of computer algebra for the discretization and coding of the navier-stokes equations. In A.M. Cohen, L. van Gastel, and S.M. Verduyn Lunel, editors, *Computer Algebra in Industry 2*, pages 131–149. John Wiley and Sons, Ltd, 1995.

[7] A. Harten. High resolution schemes for hyperbolic conservation laws. *J. Comput. Phys.*, 49:357–393, 1983.

[8] Francis B. Hildebrand. *Finite-Difference Equations and Simulations.* Prentice-Hall, Inc., Englewood Cliff, New Jersey, 1968.

[9] Charles Hirsch. *Numerical Computation of Internal and External Flows : Fundamentals of Numerical Discretization*, volume 1. John Wiley and Sons, Chichester, England, 1988.

[10] Charles Hirsch. *Numerical Computation of Internal and External Flows : Computational Methods for Inviscid and Viscous Flows*, volume 2. John Wiley and Sons, Chchester, England, 1990.

[11] Frank P. Incropera and David P. DeWitt. *Fundamentals of Heat and Mass Transfer.* John Wiley & Sons, New York, fourth edition, 1996.

[12] Peter M. W. Knijnenburg and Aart J. C. Bik. On reducing overhead in loops. In *Compilers for Parallel Computers*, pages 299 – 311, 1996.

[13] Barry Koren. A robust upwind discretization method for advection, diffusion and source terms. In Cornelis B. Vreugdenhil and Barry Koren, editors, *Numerical Methods for Advection-Diffusion Problems*, chapter 5, pages 117–138. Vieweg, Braunschweig, Wiesbaden, 1993.

[14] Bram Van Leer. Upwind-differencing methods for aerodynamic problems governed by the euler equations. In B.E. Engquist, S. Osher, and R.C.J. Sommerville, editors, *Large-Scale Computations in Fluid Mechanics*, volume 22, pages 327–336. American Mathematical Society, 1985.

[15] Randall J. Leveque. *Numerical Methods for Conservation Laws.* Lecture Notes in Mathematics. Birkäuser, Basel, ETH, Zürich, 1992.

[16] R. Peyret and T.D. Taylor. *Computational Methods for Fluid Flow.* Springer Series in Computational Physics. Springer-Verlag, New York Heidelberg Berlin, 1983.

[17] Stanly Steinberg and Patrick J. Roache. Using MACSYMA to write finite-volume based PDE solvers. *American Society of Mechanical Engineers, Pressure Vessels and Piping Division (Publication) PVP*, 205:81–96, 1990.

[18] P. K. Sweby. High-resolution schemes using flux limiters for hyperbolic conservation laws. *SIAM J. Numer. Anal.*, 21:995–1011, 1984.

[19] Robert van Engelen, Ilja Heitlager, Lex Wolters, and Gerard Cats. Incorporating application dependent information in an automatic code generating environment. In *Proceedings of the 1997 International Conference on Supercomputing*, pages 180–187, New York, July 1997. ACM.

[20] Robert van Engelen, Lex Wolters, and Gerard Cats. CTADEL: A generator of multi-platform high performance codes for PDE-based scientific applications. In *Proceedings of the 1996 International Conference on Supercomputing*, pages 86–93, New York, May 1996. ACM.

[21] Robert van Engelen, Lex Wolters, and Gerard Cats. Tomorrow's weather forecast: Automatic code generation for atmospheric modeling. *IEEE Computational Science and Engineering*, 4(3):22–31, July/September 1997.

[22] Cornelis B. Vreugdenhil and Barry Koren, editors. *Numerical Methods for Advection-Diffusion Problems*, volume 45 of *Notes on numerical fluid mechanics*. Vieweg, Braunschweig, Wiesbaden, 1993.

[23] M. Zijlema and P. Wesseling. Higher order flux-limiting methods for steady state multi-dimensional convection-dominated flow. Technical Report 95-131, Delft University of Technology, Department of Technical Mathematics and Computer Science, 1995.

[24] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. Frontier Series. ACM Press, 1991.