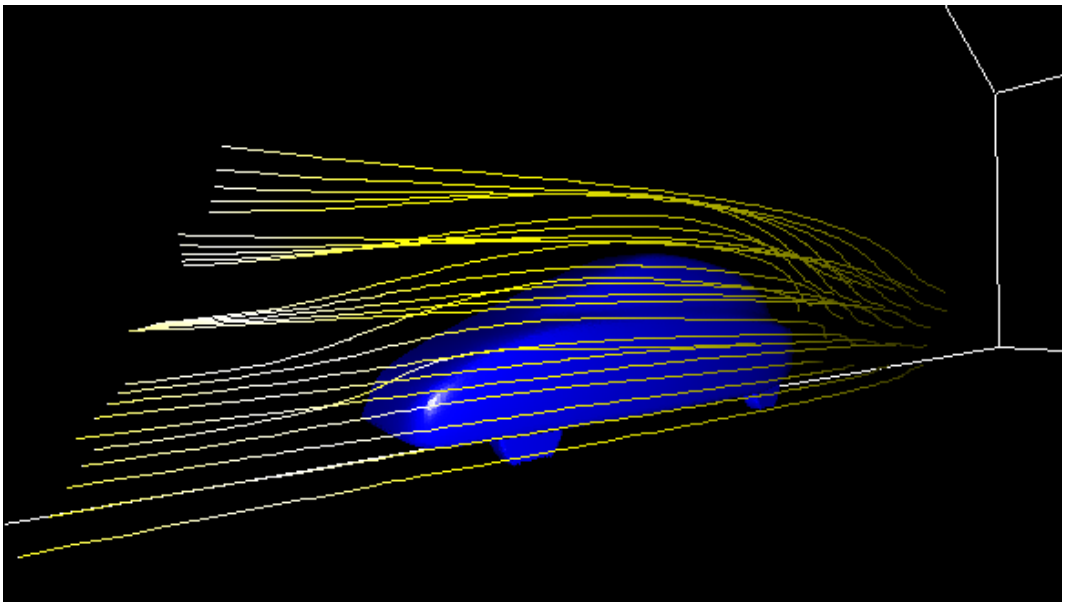
FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

Methods for fast Particle Tracing in unstructured Meshes

Volker Daum



Diplomarbeit

Methods for fast Particle Tracing in unstructured Meshes

Volker Daum

Diplomarbeit

Aufgabensteller:	Prof. Dr. U. Rüde und Prof. Dr. G. Greiner
Betreuer:	Prof. Dr. U. Rüde und Prof. Dr. G. Greiner
Bearbeitungszeitraum:	15.1.2004 – 15.07.2004

Erklärung:

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 9. September 2004 .....

Abstract

Numerical simulations of flow fields, such as air current simulations of a car model in a wind tunnel or fluid simulations produce vector fields as output, often on unstructured grids. One of the standard methods to visualize such vector data is particle tracing and there exist many more methods that employ particle tracing as a first step in processing the data.

Particle tracing on unstructured grids is faced with the problem of locating the cell the particle is in, after it has been advanced during a step. Usually this is done by performing a local cell search, for which standard and also more specialized methods [1] are available. Since such a local cell search can be quite computationally costly, Reck and Greiner [9] presented a locally exact method that eliminates the cell search by stepping from one face of a cell to the next, which turns the cell search into a simple lookup of a neighboring cell. This also solves the often quite difficult problem of choosing a step size for the method. The disadvantage of this method is, that the approach to calculate the particle path analytically in each cell is limited to tetrahedral meshes and linear basis functions. It also needs a lot of precomputations and the memory to store their results to perform well.

The algorithms presented in this work pick up the basic idea of only crossing one cell at a time, but calculate the particle path numerically with specially adapted Runge-Kutta methods. The main advantage of these algorithms is their greater flexibility, as they can be applied to arbitrary cell types and interpolation methods.

Zusammenfassung

Die numerische Simulation von Strömungsfeldern, wie zum Beispiel die Simulation der Umströmung eines Automodells in einem Windtunnel oder auch die Simulation von Flüssigkeiten, erzeugen Vektordaten. Oftmals werden diese auf unstrukturierten Gittern berechnet und repräsentiert. Für die Visualisierung solcher Daten ist Particle Tracing eine der Standardmethoden und viele andere Verfahren verwenden Particle Tracing als einen ersten Schritt in der Verarbeitung der Daten.

Auf unstrukturierten Gittern stellt sich bei Particle Tracing jedoch das Problem, nach jedem Schritt ermitteln zu müssen, in welcher Gitterzelle sich das Partikel nun befindet. Normalerweise erfolgt dies durch eine lokale Zellsuche, für die sowohl Standardmethoden als auch spezialisiertere Ansätze [1] zur Verfügung stehen. Da aber eine lokale Zellsuche sehr Rechenintensiv sein kann haben Reck und Greiner in [9] ein lokal exaktes Verfahren vorgestellt, das sich in jedem Schritt von Zellgrenze zu Zellgrenze bewegt und dadurch eine Zellsuche unnötig macht. Bei einem solchen Verfahren entspricht das Auffinden der nächsten Zelle einem schlichten Ermitteln des Nachbarn der Zelle, die gerade bearbeitet wurde. Zusätzlich wird dadurch automatisch das Problem der Schrittweitensteuerung gelöst. Der Hauptnachteil dieses Algorithmus besteht darin, dass die analytische Berechnung der Partikelbahn in jeder Zelle nur bei Tetraedern mit linearer Interpolation durchführbar ist. Außerdem werden einige vorbereitende Berechnungen und dementsprechend Speicherplatz für deren Ergebnisse benötigt.

Die Algorithmen, die im Rahmen dieser Arbeit präsentiert werden, nehmen denselben Ansatz wieder auf, immer nur eine Zelle am Stück zu durchqueren, berechnen aber den Partikelpfad in jeder Zelle numerisch. Dazu werden speziell angepasste Runge-Kutta Verfahren verwendet. Der daraus resultierende Vorteil ist eine größere Flexibilität dieser Algorithmen, da sie mit beliebigen Zelltypen und Interpolationsmethoden arbeiten können.

Contents

Chapter 1

Introduction

Many important applications like simulations of fluid or air currents yield two or three dimensional vector fields. These are sometimes represented on regular, but often also on highly unstructured meshes. For visualizing such vector fields, a number of different approaches like glyph and texture based methods exist. One of the most widely used methods is particle tracing.

Particle tracing is suited to visualize global as well as local features of the data, and can in most cases be used interactively. Particle tracing is also the basis of other visualization techniques like Streak-lines, Streak-ribbons, Line Integral Convolution (LIC) [3] or other texture based approaches (see for example [6]).

The areas of application for particle tracing are very varied and range from the more obvious flow visualizations, like objects in a wind tunnel, to the visualization of field lines in potential fields (the necessary vector field is the gradient of the potential field) or the movement of particles in a gravity field. For some examples see also the images of figure 1.0.1.

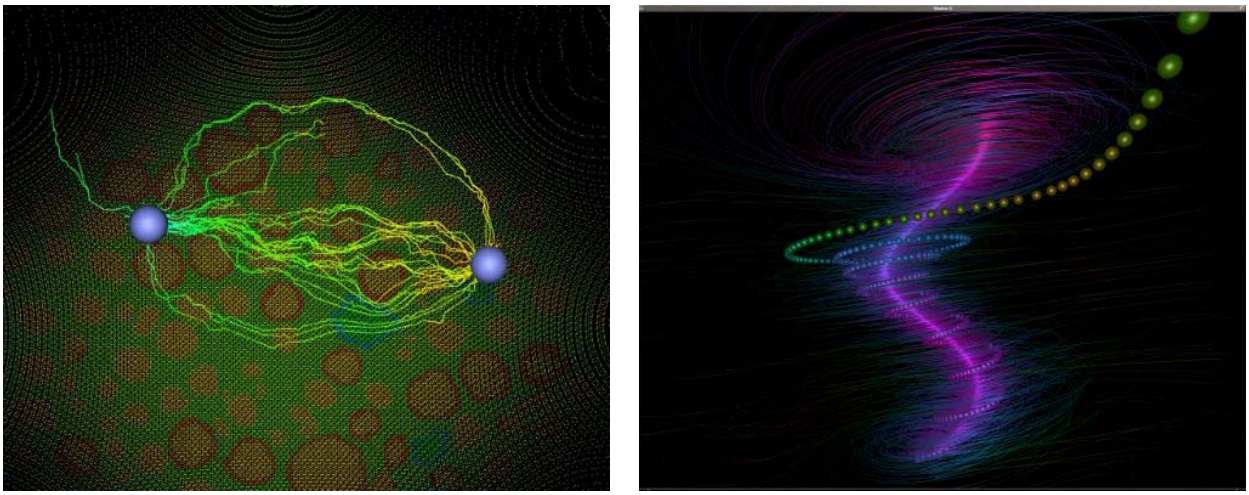
1.1 Particle Tracing

Particle tracing calculates the movement of a massless particle through the vector field. This is done by treating the data as a vectorial function $\boldsymbol{f}$ on the domain, and integrating over this data, which corresponds to solving an initial value problem for an ordinary differential equation, in time x and particle position $\boldsymbol{y}(x)$. The starting point of the particle provides the initial values x_0 and $\boldsymbol{y}(x_0) = \boldsymbol{y}_0$.

$$\begin{aligned} \boldsymbol{y}(x_0) &= \boldsymbol{y}_0 \\ \boldsymbol{y}'(x) &= \boldsymbol{f}(x, \boldsymbol{y}(x)) \end{aligned} \tag{1.1.1}$$

There are a wide variety of single and multi step methods available to solve this equation. The most popular family of methods used for this are the Runge-Kutta methods, which will be presented in chapter 2. Since they are one step methods they are easy to handle and compute, and it is possible to adapt them to the needs we have for the work in unstructured meshes.

Since the data is usually only represented at the vertices of the mesh, the vector data has to be interpolated to every position needed. For this work it will be assumed that the



(a) 2D cross section of a cat sciatic nerve under a bipolar electric stimulation. (C. Butson)

(b) Visualization of a simulated tornado dataset. (M. Ikits, J. D. Brederson)

Figure 1.0.1: **Examples of particle tracing applications:** Images courtesy of the Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, USA (`http://www.sci.utah.edu`)

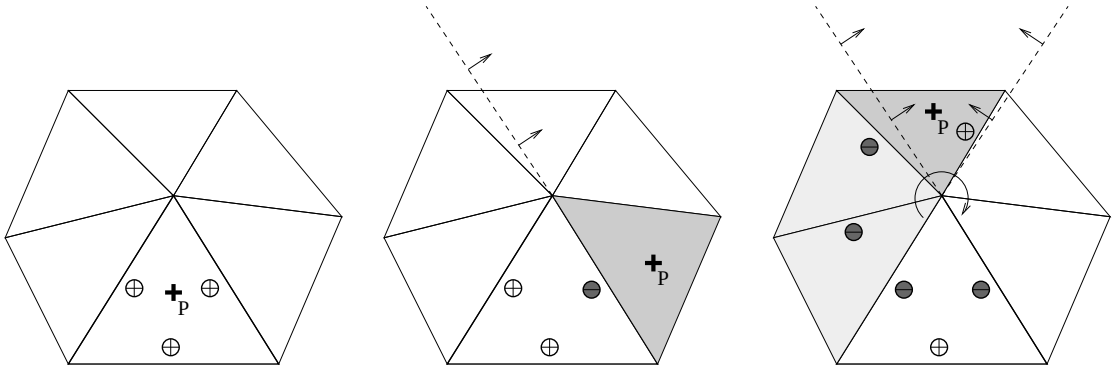
application provides this interpolation or has other means to provide vector data for every position in question.

For this interpolation the application will not only need the particle position, but also the mesh cell the particle is in. For structured meshes this is very easy to determine and can be done with a few simple calculations. In unstructured meshes this is a much harder problem. Usually some kind of local cell search is employed to find the right cell. Such a cell search can computationally be quite costly, therefore it is the main intent of this work to eliminate it.

1.2 Unstructured Meshes

Most data of interest is either the output of a simulation or the result of measurements like MRI (Magnetic Resonance Imaging). While measured data is usually represented on structured meshes, this is a different matter in simulations. For simulations it is often advantageous if the mesh can be adapted to the geometry that is involved in the simulation. For example unstructured meshes can only represent more or less rectangular domains, which means that a lot of computing power is wasted if a non-rectangular domain has to be embedded in a rectangular one. It is also very common that very differently scaled objects or events have to be simulated on the same mesh. Although it is certainly possible to do mesh refinement on structured meshes as well, unstructured meshes, are much better able to adapt to the needs at hand.

There is a wide range of cell types used in unstructured meshes. The most common unstructured meshes are tetrahedral and curvilinear (a deformed rectangular mesh). But since structured meshes have the advantage of being computationally more efficient, it is also often the case that structured and unstructured meshes are combined so that the advantages of



(a) Point P is inside

(b) Point P is outside for one face

(c) Point P is outside for two faces

Figure 1.3.1: **Local cell search by barycentric coordinates:** Barycentric coordinates are calculated for the lower cell. The space described by the dashed lines and the arrows has to be searched for the position P. The search is recursively continued in the dark shaded cells. In case c) the fan has to be checked (light shaded cells) until the right cell to continue (dark shaded cell) is found.

both can be used. This leads to highly unregular cells at the interfaces between the structured and unstructured domains inside the mesh. For example pyramidal cells are often used in this case.

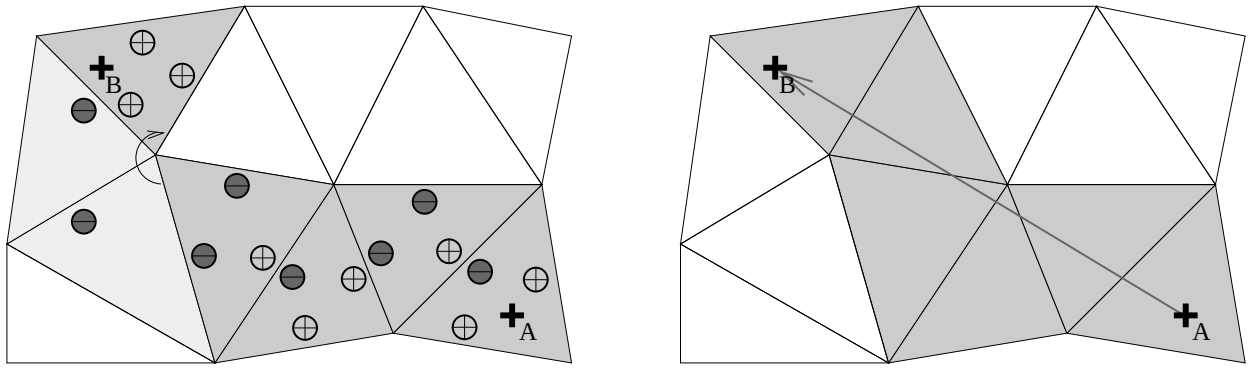
1.3 Previous work

Standard methods for particle tracing in unstructured meshes use Runge-Kutta methods for the integration, and have to perform local cell searches to find the mesh cell for the current particle position. One possibility for doing such a local cell search in tetrahedral meshes, utilizes barycentric coordinates.

This method starts by calculating the barycentric coordinates of the particle position, for the current cell. Each coordinate is associated with one of the faces. If the coordinate is greater than zero the particle is on the "right" side of the face, if it is smaller than zero it is on the "wrong" side, and therefore definitely not inside the current cell.

In 2D this results in three cases. In the case that the particle is in the cell for which the coordinates have been calculated, all of the coordinates will be greater than zero (see figure 1.3.1(a)). If one of the coordinates is smaller than zero, then the particle has to be somewhere in the half space on the opposite side of the negative coordinates' face. The next cell to examine is the cell which shares this face with the current cell (see figure 1.3.1(b)). If instead there are two negative coordinates, then the particle is in the intersection of the halfspaces defined by the two faces with the negative coordinates. To find the first cell that lies in this intersection of halfspaces, the triangle fan around the vertex that connects the edges with the negative coordinates has to be examined (see figure 1.3.1(c)). To this end all the edges separating the cells around the fan are checked for in/out conditions, until the first edge is found for which the particle is "inside". The cell described by this edge and the previous edge, has to be in the same halfspace that the particle is in.

In 3D we get four cases instead, that are treated in much the same way. The negative



(a) barycentric coordinates

(b) ray casting

Figure 1.3.2: **Local cell search:** The images show a local cell search beginning in point A. The shaded cells are the ones examined by the algorithms.

coordinates together with the faces they are associated with, describe an intersection of half-spaces, that has to be searched in order to find the right cell.

Similar algorithms are possible for non-tetrahedral cells as well. The barycentric coordinates in the algorithm are only used as a means to determine on which side of a face the particle is. Naturally this can be done with normal plane equations for an arbitrary number of faces as well. How the different in and out results for arbitrary elements should be handled is quite nontrivial. A heuristic solution is probably simpler and faster than an overkill of special cases, as it is basically only important to always step in the right direction (e.g. into the right half space). One approach would be to always consider the neighbor across the face, for which the distance to the particle position is the greatest. It is certainly also possible to come up with special treatments like the fan processing above for certain configurations, but these will always be limited to certain cell types.

Another method that could be used is to cast a ray from the old to the new particle position, and just step through the cells along this ray, until the right one is found (see figure 1.3.2(b)).

There also already exist some more specialized methods for unstructured meshes. For curvilinear meshes Buning [1] proposed a method named "stencil walk", which uses a transformation from the deformed mesh in the physical space to a non-deformed, rectangular mesh in the so called computational space. The center of the current cell in computational space $\boldsymbol{c}$ is used as a starting point for the algorithm. The distance vector in physical space between the starting point $\boldsymbol{c}$ transformed to physical space, and the particle position (in physical space), is transformed to computational space. This transformed distance vector is then used as an indicator to which of the neighboring cells (9 cells in 3D) the algorithm should step next. The position $\boldsymbol{c}$ in computational space is also moved according to the transformed distance vector and used as starting point for the next iteration of the algorithm (see also [7]).

The particle tracing could also be done completely in an undeformed mesh in computational space, but this would involve a transformation of the data as well. Besides being quite computationally costly, this transformation can also introduce further inaccuracies. Both approaches using a computational space have the main disadvantage that they are limited to curvilinear meshes.

For particle tracing in tetrahedral meshes Reck and Greiner [9] presented a method that determines the analytically correct particle path in each cell and intersects it with the faces of the cell, to determine the point through which the particle leaves the cell. The advantage of this approach aside from its high accuracy is that it completely eliminates the cell search. Since the face through which the particle leaves the cell is known, it is only necessary to determine the neighbor of the current cell to continue the trace and thus step from cell to cell. The downside of this method is that the approach of analytically computing the particle path is limited to tetrahedrons and linear interpolation functions and that is has to perform a lot of precomputations.

The methods presented in this work (chapter 3) use the same idea of stepping from cell face to cell face, but compute the particle path numerically. Therefore they are able to deal with an arbitrary, convex cell type and arbitrary interpolation methods. Since they are based on standard Runge-Kutta methods, these will be introduced in the next chapter.

Chapter 2

Runge-Kutta methods

2.1 Notation

Before we start looking at the details of Runge-Kutta methods it is helpful to introduce and explain some notational conventions. All vector values, like $\boldsymbol{f}$, will be denoted in bold font. Vector products are specially marked by a dot, as in $\boldsymbol{n} \cdot \boldsymbol{p}$. All other products are normal scalar, or vector-matrix products depending on the values involved. In general all equations and calculations in this work hold, no matter whether we consider the vector values to be 2D or 3D (or even 1D).

Since Runge-Kutta methods are iterative, they calculate the values of $\boldsymbol{y}$ at discrete samples of x. We therefore define the stepsize h_i and the sample points x_i as

$$x_{i+1} = x_i + h_i$$

where x_0 is taken from the problem definition (1.1.1). Furthermore $\boldsymbol{y}_i$ denotes the approximated value of $\boldsymbol{y}$ at position x_i.

$$\boldsymbol{y}_i \approx \boldsymbol{y}(x_i)$$

For some of the more complicated derivations, we will also introduce the following notational simplifications:

$$\begin{aligned} \frac{d}{dx}\boldsymbol{f}(x, \boldsymbol{y}) &= \boldsymbol{f}_x(x, \boldsymbol{y}) \\ \frac{d}{dy}\boldsymbol{f}(x, \boldsymbol{y}) &= \boldsymbol{f}_{\boldsymbol{y}}(x, \boldsymbol{y}) \end{aligned}$$

Further in the i-th step we will write

$$\begin{aligned} \boldsymbol{y}(x_i) &= \boldsymbol{y} \\ \boldsymbol{f}(x_i, \boldsymbol{y}(x_i)) &= \boldsymbol{f} \\ \boldsymbol{f}_x(x_i, \boldsymbol{y}(x_i)) &= \boldsymbol{f}_x \\ \boldsymbol{f}_{\boldsymbol{y}}(x_i, \boldsymbol{y}(x_i)) &= \boldsymbol{f}_{\boldsymbol{y}} \end{aligned}$$

For all derivations the application of a previous equation, will be denoted by the equation number above the equal sign. Similarly a "!" will denote that a Taylor expansion (see [12]) has been done.

2.2 Origins

The simplest method for solving equation (1.1.1) is to approximate $\boldsymbol{y}(x)$ locally as a linear function, that is characterized by its tangent. This is equivalent to ignoring all terms of $\mathcal{O}(h^2)$ or above from the Taylor series of $\boldsymbol{y}(x+h)$.

$$\begin{aligned}\boldsymbol{y}(x+h) &\stackrel{!}{=} \boldsymbol{y}(x) + h\boldsymbol{y}'(x) + \mathcal{O}(h^2) \\ &= \boldsymbol{y}(x) + h\boldsymbol{f}(x, \boldsymbol{y}(x)) + \mathcal{O}(h^2)\end{aligned}$$

By doing this approximation stepwise, we get the following iterative scheme that is known as Euler's method.

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h_i \boldsymbol{f}(x_i, \boldsymbol{y}_i) \tag{2.2.1}$$

For examining the accuracy of this method it is necessary to define what kind of errors we want to look at. We will write a general, explicit one step method as

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h_i \boldsymbol{\Phi}(\boldsymbol{f}, x_i, \boldsymbol{y}_i) \tag{2.2.2}$$

For the Euler method $\boldsymbol{\Phi}(\boldsymbol{f}, x_i, \boldsymbol{y}_i) = \boldsymbol{f}(x_i, \boldsymbol{y}_i)$. The local error d_i is defined as the error introduced in the i-th step of the method (see figure 2.2.1(a)). It is therefore assumed that the starting point for this step is correct, this means that $\boldsymbol{y}_i = \boldsymbol{y}(x_i)$.

$$\begin{aligned}d_{i+1} &= \boldsymbol{y}(x_{i+1}) - \boldsymbol{y}(x_i) - h_i \boldsymbol{\Phi}(\boldsymbol{f}, x_i, \boldsymbol{y}(x_i)) \\ &= \boldsymbol{y}(x_i + h_i) - \boldsymbol{y}(x_i) - h_i \boldsymbol{\Phi}(\boldsymbol{f}, x_i, \boldsymbol{y}(x_i))\end{aligned} \tag{2.2.3}$$

In contrast to the local error d_i, the global error g_i is defined as the accumulated error over the course of the iteration (see figure 2.2.1(b)).

$$\begin{aligned}g_{i+1} &= \boldsymbol{y}(x_{i+1}) - \boldsymbol{y}_{i+1} \\ &= \boldsymbol{y}(x_i + h_i) - \boldsymbol{y}_i - h_i \boldsymbol{\Phi}(\boldsymbol{f}, x_i, \boldsymbol{y}_i))\end{aligned} \tag{2.2.4}$$

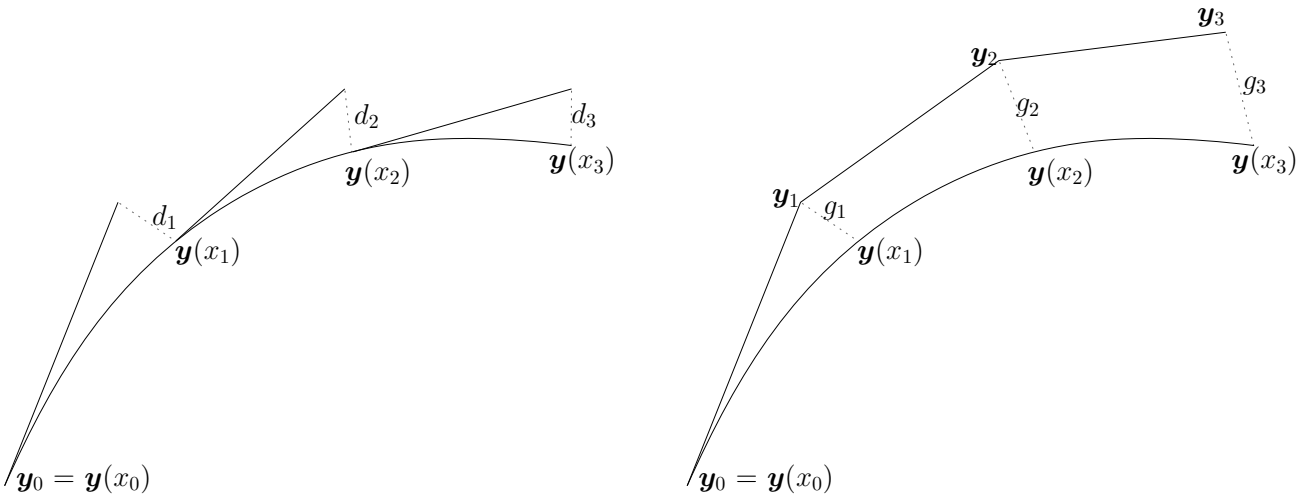
Hence g_i contains not only the sum of the local errors d_i, from all steps, but also any further error introduced by doing a step with the error from the previous steps already present in the starting values.

We will call a method for which $g_i = \mathcal{O}(h^p)$ an order p method. If there is no constant step size h, we will set $h = \max_i(h_i)$. It is possible to prove that for explicit, one step methods, if $d_i = \mathcal{O}(h^{p+1})$ then $g_i = \mathcal{O}(h^p)$ (see [5], [10], [11], and [4]). Therefore the Euler method which has a local error of $\mathcal{O}(h^2)$, is an order $p = 1$ method. Since this is a rather low accuracy, Runge extended a numerical integration scheme, the midpoint rule, which is known to be of order $p = 2$, to this kind of problem [5]. The midpoint rule is a scheme for problems of the following kind.

$$\begin{aligned}\boldsymbol{y}(x_0) &= \boldsymbol{y}_0 \\ \boldsymbol{y}'(x) &= \boldsymbol{f}(x)\end{aligned}$$

Which is iteratively solved as

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h\boldsymbol{f}(x_i + \frac{h}{2})$$



(a) Local error d_i along a curve.

(b) Global error g_i along a curve.

Figure 2.2.1: **Comparison of global and local error**

Applied to our problem (1.1.1) this looks like

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h\boldsymbol{f}(x_i + \frac{h}{2}, \boldsymbol{y}(x_i + \frac{h}{2}))$$

But when this method is applied to this kind of problem we need an intermediate value $\boldsymbol{y}(x_i + \frac{h}{2})$. To compute this value it is possible to apply a step of Euler's method. The lower accuracy of this Euler step is compensated by the fact that in all further uses of $\boldsymbol{y}(x_i + \frac{h}{2})$ in the method, it is multiplied by h. The resulting method is also know as the modified Euler's method.

These kinds of algorithms devised by Runge and Heun were then further generalized by Kutta, and are now known as Runge-Kutta methods. In the next chapter we will examine how Runge-Kutta methods can in general be described and derived.

2.3 Derivation

Runge-Kutta methods all have the same structure, of iteratively sampling the function $\boldsymbol{y}' = \boldsymbol{f}$ several times. In each of these sampling stages, the previous samples are used to compute the location where $\boldsymbol{f}$ will be evaluated next. Therefore each Runge-Kutta method can be characterized by the number of its stages, as an n-stage method. In general a n-stage Runge-Kutta method looks like this:

$$\begin{aligned} \boldsymbol{k}_1 &= \boldsymbol{f}(x_i, \boldsymbol{y}_i) \\ \boldsymbol{k}_j &= \boldsymbol{f}(x_i + q_j h_i, \boldsymbol{y}_i + \sum_{l=1}^{j-1} b_{jl}\boldsymbol{k}_l) \\ \boldsymbol{y}_{i+1} &= \boldsymbol{y}_i + h_i \sum_{j=1}^{n} a_j \boldsymbol{k}_j \end{aligned} \tag{2.3.1}$$

The parameters $a_1 \dots a_n$, $q_2 \dots q_n$ and $b_{21} \dots b_{n,n-1}$ have to be chosen so that the local error d_i in each step is minimized. To determine this we have to calculate the Taylor expansion of both, the exact solution $\boldsymbol{y}(x_{i+1}) = \boldsymbol{y}(x_i + h_i)$ and its approximation $\boldsymbol{y}_{i+1}$.

As an example this calculation will be done for a two stage method. For this case (2.3.1) simplifies to

$$\begin{aligned} \boldsymbol{k}_1 &= \boldsymbol{f}(x_i, \boldsymbol{y}_i) \\ \boldsymbol{k}_2 &= \boldsymbol{f}(x_i + q_2 h_i, \boldsymbol{y}_i + h_i b_{21} \boldsymbol{k}_1) \\ \boldsymbol{y}_{i+1} &= \boldsymbol{y}_i + h_i(a_1 \boldsymbol{k}_1 + a_2 \boldsymbol{k}_2) \end{aligned} \tag{2.3.2}$$

The local error of this method according to equation (2.2.3) is

$$\begin{aligned} d_i &= \boldsymbol{y}(x_{i+1}) - \boldsymbol{y}_{i+1} \\ &= \boldsymbol{y}(x_i + h_i) - (\boldsymbol{y}(x_i) + h_i(a_1 \boldsymbol{k}_1 + a_2 \boldsymbol{k}_2)) \end{aligned} \tag{2.3.3}$$

For the Taylor expansion series a derivative of $\boldsymbol{f}$ will be needed

$$\boldsymbol{f}'(x_i, \boldsymbol{y}(x_i)) = \boldsymbol{f}_x + \boldsymbol{f}_{\boldsymbol{y}} \boldsymbol{f} \tag{2.3.4}$$

Where $\boldsymbol{f}_{\boldsymbol{y}}$ is a quadratic matrix, since we derive a vectorial function $\boldsymbol{f}$, by a vectorial parameter $\boldsymbol{y}$. Now the expansion of $\boldsymbol{y}(x_{i+1})$ can be calculated as

$$\begin{aligned} \boldsymbol{y}(x_{i+1}) &= \boldsymbol{y}(x_i + h_i) \\ &\overset{!}{=} \boldsymbol{y}(x_i) + h_i \boldsymbol{y}(x_i)' + \frac{h_i^2}{2} \boldsymbol{y}(x_i)'' + \mathcal{O}(h_i^3) \\ &= \boldsymbol{y}(x_i) + h_i \boldsymbol{f} + \frac{h_i^2}{2} \boldsymbol{f}' + \mathcal{O}(h_i^3) \\ &\overset{(2.3.4)}{=} \boldsymbol{y}(x_i) + h_i \boldsymbol{f} + \frac{h_i^2}{2} (\boldsymbol{f}_x + \boldsymbol{f}_{\boldsymbol{y}} \boldsymbol{f}) + \mathcal{O}(h_i^3) \end{aligned} \tag{2.3.5}$$

Similarly an expansion of $\boldsymbol{y}_{i+1}$ (with a correct starting value of $\boldsymbol{y}(x_i) = \boldsymbol{y}_i$) can be done

$$\begin{aligned} \boldsymbol{y}_{i+1} &= \boldsymbol{y}(x_i) + h_i(a_1 \boldsymbol{k}_1 + a_2 \boldsymbol{k}_2) \\ &= \boldsymbol{y}(x_i) + h_i(a_1 \boldsymbol{f} + a_2 \boldsymbol{f}(x_i + q_2 h_i, \boldsymbol{y}(x_i) + b_{21} h_i \boldsymbol{f})) \\ &\overset{!}{=} \boldsymbol{y}(x_i) + h_i(a_1 \boldsymbol{f} + a_2(\boldsymbol{f} + h_i(q_2 \boldsymbol{f}_x + b_{21} \boldsymbol{f}_{\boldsymbol{y}} \boldsymbol{f}))) + \mathcal{O}(h_i^3) \\ &= \boldsymbol{y}(x_i) + h_i(a_1 + a_2) \boldsymbol{f} + h_i^2 a_2 (q_2 \boldsymbol{f}_x + b_{21} \boldsymbol{f}_{\boldsymbol{y}} \boldsymbol{f}) + \mathcal{O}(h_i^3) \end{aligned} \tag{2.3.6}$$

It is now possible to represent the error d_i as a polynome in h_i, with coefficients $\boldsymbol{c}_0 \dots \boldsymbol{c}_2$, which are obtained by subtracting (2.3.6) from (2.3.5).

$$\begin{aligned} d_i &= \boldsymbol{c}_0 + \boldsymbol{c}_1 h_i + \boldsymbol{c}_2 h_i^2 + \mathcal{O}(h_i^3) \\ \boldsymbol{c}_0 &= \boldsymbol{y}_i - \boldsymbol{y}_i \\ &= 0 \\ \boldsymbol{c}_1 &= \boldsymbol{f} - (a_1 + a_2) \boldsymbol{f} \\ &= (1 - a_1 - a_2) \boldsymbol{f} \\ \boldsymbol{c}_2 &= \frac{1}{2} (\boldsymbol{f}_x + \boldsymbol{f}_{\boldsymbol{y}} \boldsymbol{f}) - a_2 (q_2 \boldsymbol{f}_x + b_{21} \boldsymbol{f}_{\boldsymbol{y}} \boldsymbol{f}) \\ &= (\frac{1}{2} - a_2 q_2) \boldsymbol{f}_x + (\frac{1}{2} - a_2 b_{21}) \boldsymbol{f}_{\boldsymbol{y}} \boldsymbol{f} \end{aligned}$$

From this follows that for all coefficients $\boldsymbol{c}_0 \dots \boldsymbol{c}_2$ to be zero it is necessary to set

$$a_1 + a_2 = 1 \qquad a_2 q_2 = \tfrac{1}{2} \qquad a_2 b_{21} = \tfrac{1}{2} \tag{2.3.7}$$

If the Taylor expansion would have been done to a higher degree, it would have shown that it is not possible to eliminate the remaining error of $\mathcal{O}(h_i^3)$.

The degrees of freedom in setting the parameters can be reduced by requiring that each sample stage is at least an order 1 approximation to the solution. This means that

$$\sum_{l=1}^{l<i-1} b_{il} = q_i \qquad \forall i = 2, \dots, n \tag{2.3.8}$$

This requirement is often useful, but not necessary (see for example [8]). With it the equations (2.3.7) simplify to

$$a_1 + a_2 = 1 \qquad a_2 q_2 = \tfrac{1}{2} \qquad q_2 = b_{21} \tag{2.3.9}$$

By solving (2.3.9) for q_2, a one dimensional manifold in q_2 is obtained.

$$a_2 = \tfrac{1}{2q_2} \qquad a_1 = 1 - a_2 = 1 - \tfrac{1}{2q_2} \qquad q_2 = b_{21} \tag{2.3.10}$$

Choosing $q_2 = 1$ will yield Heun's method,

$$q_2 = b_{21} = 1 \qquad \Rightarrow \qquad a_1 = 1/2 \qquad a_2 = 1/2 \tag{2.3.11}$$

For $q_2 = 1/2$ it will become the modified Euler's method, that was already mentioned in section 2.2.

$$q_2 = b_{21} = 1/2 \qquad \Rightarrow \qquad a_1 = 0 \qquad a_2 = 1 \tag{2.3.12}$$

Those same derivations can be done for any Runge-Kutta method, but they grow very fast in complexity when the number of stages is increased. To formalize and simplify these calculations Butcher [2] developed the so called Butcher-Trees. Since this work is not concerned with methods above an order of $p = 4$, we will not discuss these in more detail. Another thing that is noteworthy, is that up to $n = 4$, the order of the Runge-Kutta method is equal to the number of its stages, hence $n = p$, while for higher orders more stages are necessary.

Chapter 3

Fast Particle Tracing in Unstructured Meshes

The problem with particle tracing on unstructured meshes is to find the cell the particle is in after each step, and in the case of Runge-Kutta methods, also the cells of the intermediate sample positions. The standard approach to solve this problem is to do a local cell search (see section 1.3). This cell search could be eliminated if the tracing method would only step from one cell to the next. If the mesh is stored such that each cell knows its neighbors it is only necessary to determine through which face the particle will leave it. This way the method can step from face to face and to determine the next cell to cross equals a simple lookup.

The resulting step sizes will sometimes be overly small, if the particle passes very close by a vertex, but on average step size and cell size will roughly correspond. As a side effect this automatically yields a certain adaptivity, which is quite useful. In large cells, the interpolated function is not very accurate and therefore a large step size is not harmful. Analoguously for small cells the accuracy of the interpolated data will be higher and so a small step size will help to conserve it.

The simplest method that could traverse the mesh like this, cell by cell would be an Euler method that adapts its step size such that it exactly hits the face through which it will leave the cell. Therefore if the face $\mathcal{F}$ through which the cell is left, is described by the plane $P = \{\boldsymbol{p} \,|\, \boldsymbol{n} \cdot \boldsymbol{p} = d\}$, we set

$$h_i = \frac{d - \boldsymbol{n} \cdot \boldsymbol{y}_i}{\boldsymbol{n} \cdot \boldsymbol{f}(x_i, \boldsymbol{y}_i)}$$

As was shown in section 2.3, Runge-Kutta methods operate by implicitly fitting an approximation of the Taylor polynomial to the underlying data. Hence we have to solve not a linear function, but instead a polynomial, to obtain the correct h for higher order methods. This gets increasingly difficult when the order of the method, and with it the order of the polynomial is increased. We therefore also explored a method different from just solving the intersection of those polynomials with the faces of the cell.

The two methods we present here are of order 2 and 3. It is usually not necessary to use an integration method that has a higher accuracy than the underlying data. As discussed in section 1.2, most of the data that will be visualized this way comes from finite element methods. The accuracy of these methods is largely dependent on the basis functions that are used in each cell. As many of these applications use only linear or quadratic basis function, our methods should be sufficient for them.

3.1 Order 2 Method (MARK2)

The first method presented here will be of order 2. It will be called MARK2, for "Mesh Adaptive Runge-Kutta of order 2".

For deriving an algorithm for the needs outlined above, it is helpful to simplify the requirements. Instead of considering a complete cell we will start with a single plane. Since a cell is essentially composed of intersecting planes it is not hard to generalize the method afterwards.

Let the plane P be defined as

$$\boldsymbol{n} \cdot \boldsymbol{p} = d \qquad \forall \boldsymbol{p} \in P$$

With $\boldsymbol{k}_1 = \boldsymbol{f}(x_i, \boldsymbol{y}_i)$ defined normally just as in any other Runge-Kutta method. The position where the second sample of $\boldsymbol{f}$ is taken depends on h_i. As h_i has to be chosen later in the last step of the method, it cannot be set now. Instead $\tilde{q}_2$ has to be defined as

$$\tilde{q}_2 := q_2 h_i = \frac{d - \boldsymbol{n} \cdot \boldsymbol{y}_i}{\boldsymbol{n} \cdot \boldsymbol{k}_1} \tag{3.1.1}$$

Then $\boldsymbol{y}_i + \tilde{q}_2 \boldsymbol{k}_1$ is $\in P$. This step is the same that an adaptive Euler method would take. Now $\boldsymbol{k}_2$ can be sampled.

$$\boldsymbol{k}_2 = \boldsymbol{f}(x_i + \tilde{q}_2, \boldsymbol{y}_i + \tilde{q}_2 \boldsymbol{k}_1) \tag{3.1.2}$$

In the equations (2.3.9), that define how the parameters for an order 2 method have to be set, this $\tilde{q}_2$ can be substituted.

$$a_2 q_2 = \frac{1}{2} \quad \Rightarrow \quad a_2 = \frac{1}{q_2} = \frac{h_i}{\tilde{q}_2} \tag{3.1.3}$$

$$a_1 + a_2 = 1 \quad \Rightarrow \quad a_1 = 1 - a_2 \tag{3.1.4}$$

These equations are then inserted into the final step of the algorithm

$$\begin{aligned} \boldsymbol{y}_{i+1} &= \boldsymbol{y}_i + h_i(a_1 \boldsymbol{k}_1 + a_2 \boldsymbol{k}_2) \\ &\overset{(3.1.4)}{=} \boldsymbol{y}_i + h_i(\boldsymbol{k}_1 + a_2(\boldsymbol{k}_2 - \boldsymbol{k}_1)) \\ &\overset{(3.1.3)}{=} \boldsymbol{y}_i + h_i \boldsymbol{k}_1 + h_i^2 \frac{\boldsymbol{k}_2 - \boldsymbol{k}_1}{\tilde{q}_2} \end{aligned} \tag{3.1.5}$$

Intersecting this quadratic step with the plane P, yields a scalar, quadratic equation, which can be solved for h_i.

$$\boldsymbol{n} \cdot \boldsymbol{y}_{i+1} = d \qquad \Rightarrow \qquad \boldsymbol{n} \cdot \boldsymbol{y}_i - d + h_i \boldsymbol{n} \cdot \boldsymbol{k}_1 + h_i^2 \frac{\boldsymbol{n} \cdot (\boldsymbol{k}_2 - \boldsymbol{k}_1)}{\tilde{q}_2} = 0 \tag{3.1.6}$$

With the knowledge of h_i, it is then simple to use equation (3.1.5) to calculate $\boldsymbol{y}_{i+1}$.

The step from the simplified algorithm for a single plane, to an algorithm that considers the whole cell is done by repeating the intersection steps for each face of the cell, until the correct intersection is found. In the case of the first Euler step this is pretty straightforward. For the quadratic step it is a bit more complicated, since the quadratic curve can cross the cell boundaries up to four times. The correct intersection is the one with the smallest positive h.

In practice we will not have to solve the quadratic equation for each face. In most cases where the derivatives of $\boldsymbol{f}$ are not too big (eg. $\boldsymbol{f}$ is smooth in this cell), the correct intersection will be the one with the face, that also the Euler step intersected. It is not necessary to search the other faces in this case. This shortcut works, due to the properties of a quadratic curve. A quadratic curve has, by definition a constant second derivative. This makes it impossible for such a curve to loop back onto itself or change its curvature behavior in any way, that a higher order polynomial could (see figure 3.1.1(f)). Additionally the Euler step is tangential to the quadratic curve. Therefore if the quadratic curve intersects the same face as the Euler step (with a positive h), then this face will contain the correct intersection. In practice this saves a lot of work as this case should be quite common as long as the flow is smooth. Figure 3.1.1 depicts some of the cases that can occur in a step of this method.

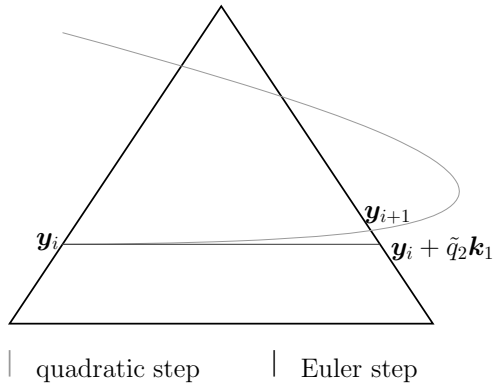
Put together all this yields the pseudo code presented as algorithm MARK2 at the end of this section. In practice the code is a bit more complicated, because often numerical inaccuracies can cause problems. These problems are treated in more detail in section 4.1.

The algorithm does not need any more function evaluations than a normal order 2 method. The only added computational effort are the calculations of the intersections, which should be compensated by the elimination of the cell search. Another advantage is that all evaluations of $\boldsymbol{f}$ are done on the cell boundaries. In practice this means that it is sufficient to implement only a 2D interpolation on the faces, which helps to further improve the speed of this method.

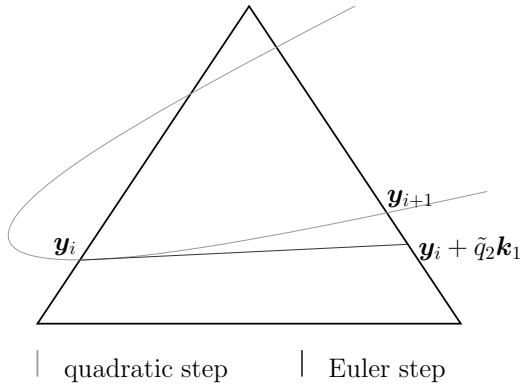
The algorithm is also very robust. There are no special cases which need handling. On the other hand it is a bit peculiar that q_2 has to be chosen differently in each step. Compared to standard Runge-Kutta methods this equals using a different method in each step. In itself this is not a problem, but the method could get unstable for some choices of q_2. In practice q_2 will be in the range of $0.5 \dots 2$, which should not cause any problems. In cells where q_2 gets considerably bigger or smaller than that, the data is so discontinuous that an instability in the method will not matter much, since the result will not be very accurate anyway.

It should be noted, that this method could have been derived differently as well. We could have employed a standard Runge-Kutta scheme, combined with a quadratic interpolation scheme to produce dense output for the particle path. Since this dense output curve is a polynomial, and would only have an error of $\mathcal{O}(h^3)$ it is identical to the according Taylor expansion of $\boldsymbol{y}$ (up to the quadratic term). Therefore intersecting this quadratic interpolation curve with the cell, is identical to what MARK2 does. It is just a different formulation of the same algorithm.

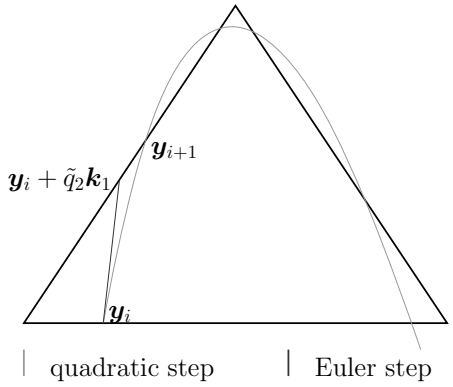
In figure 3.1.2 a step by step a trace of the method can be seen. The flow in the data is a circulation around the center of the domain, the same data that is used for the convergence analysis in section 5.1. It is nice to observe that almost always the linear, and the quadratic step intersect the same face, even though the mesh is quite coarse.



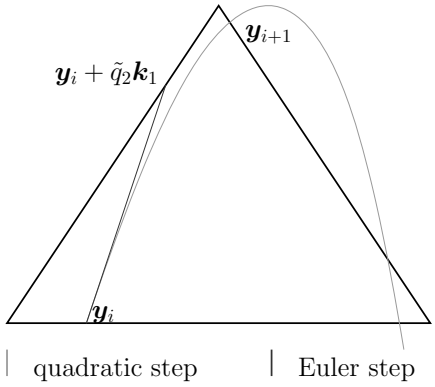
(a) Intersection with same face as the Euler step. The correct intersection is determined by the smallest h. Only one side has to be searched.



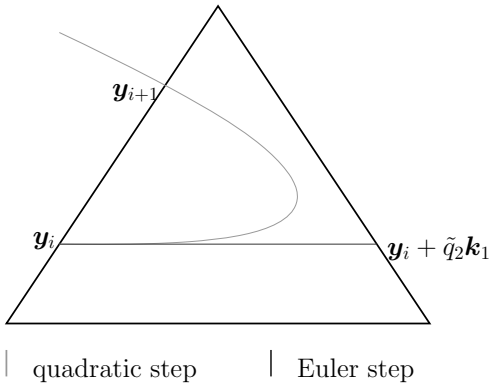
(b) Intersection with same face as the Euler step. The correct intersection is the one with $h > 0$. Only one face has to be searched.



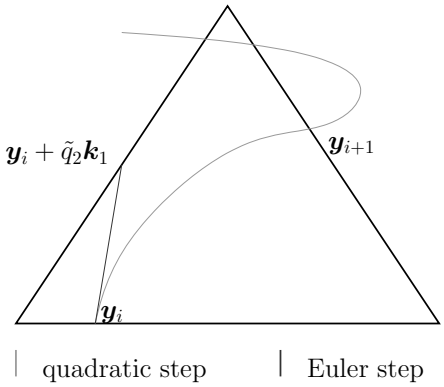
(c) Intersection with the same face as the Euler step. The correct intersection is determined by the smallest positive h. Only one side has to be searched.



(d) Intersection with another side than the Euler step. All sides have to be searched for the smallest $h > 0$.



(e) Intersection with another side than the Euler step. Here the quadratic curve even bends back to the side from which it originated.



(f) This picture illustrates what *cannot* happen in a quadratic step. A quadratic curve *cannot* change its curvature behavior and *cannot* bend back onto itself. The curve shown here is cubic.

Figure 3.1.1: **Examples of intersections for the quadratic step of MARK2**

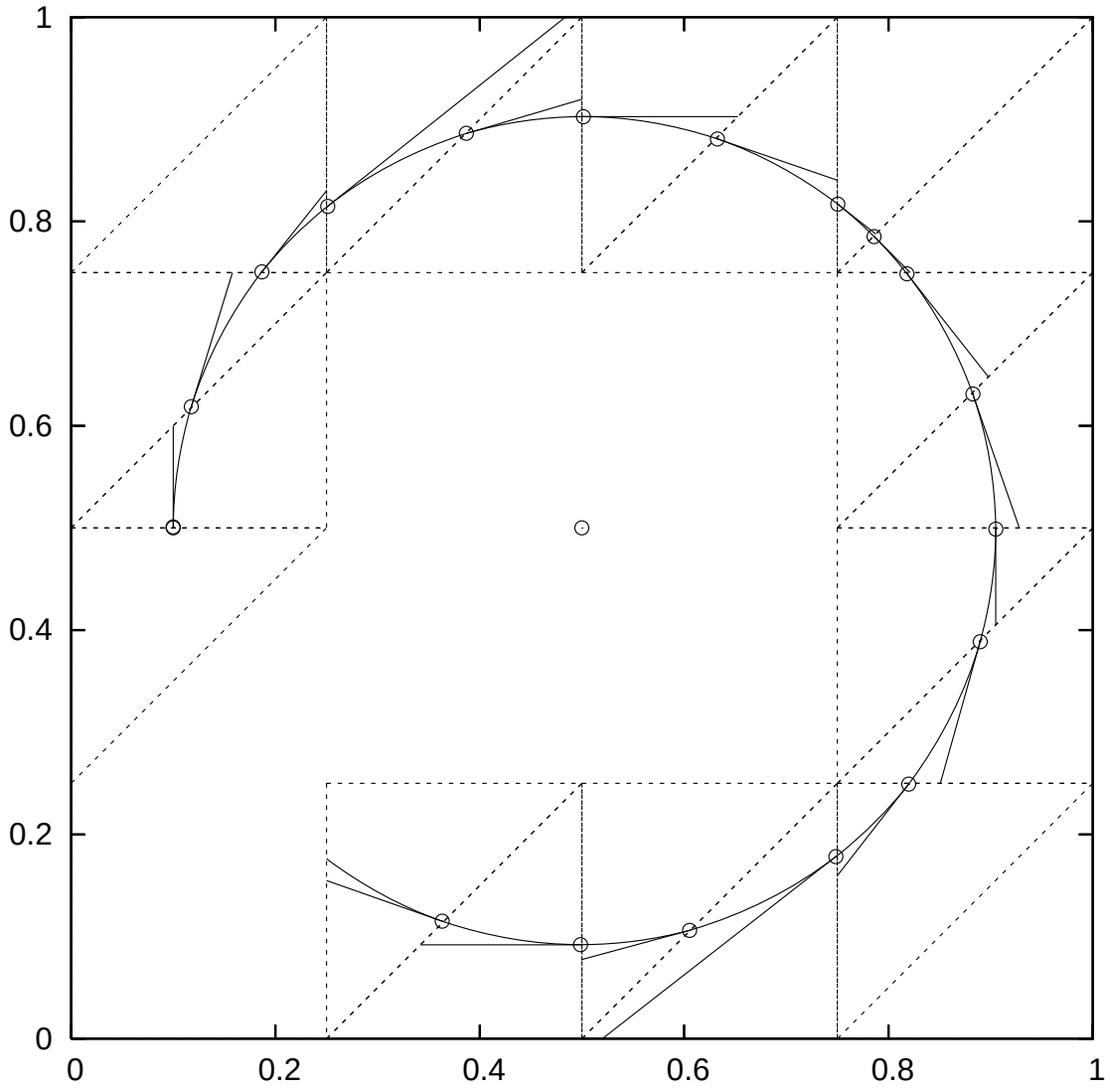


Figure 3.1.2: **MARK2** 2 **trace:** The flow in the data of this plot is circling around the center of the domain. The actual intersections with the cell boundaries that the method determines, are marked by circles. The plot shows for each iteration, the linear Euler step, and the second quadratic step.

Begin Algorithm: MARK2

Require: starting point $(x_0, \boldsymbol{y}_0)$, starting element $\mathcal{E}$
Require: interpolating function $\boldsymbol{f}$
Require: end time t_{max}

$\mathcal{F}_{old} = \text{NULL}$
$i = -1$

while $t_{max} > x_{i+1}$ **do**
  $i = i + 1$
  $\boldsymbol{k}_1 = \boldsymbol{f}(\mathcal{E}, x_i, \boldsymbol{y}_i)$

  // the Euler step
  $\mathcal{F}_E = \text{NULL}$
  for all faces $\mathcal{F}$ of $\mathcal{E}$ **do**
    $(\boldsymbol{n}, d) = \text{getPlane}(\mathcal{F})$
    if $\boldsymbol{n} \cdot \boldsymbol{k}_1 == 0$ **then** // $\boldsymbol{n}$ and $\boldsymbol{k}_1$ are parallel
      continue// try next face
    end if
    $\tilde{q}_2 = (d - \boldsymbol{n} \cdot \boldsymbol{y}_i)/(\boldsymbol{n} \cdot \boldsymbol{k}_1)$
    if $\tilde{q}_2 \leq 0$ **then** // negative distance
      continue// try next face
    end if
    $\tilde{\boldsymbol{y}} = \boldsymbol{y}_i + \tilde{q}_2 \boldsymbol{k}_1$
    if $\tilde{\boldsymbol{y}} \in \mathcal{F}$ **then**
      $\mathcal{F}_E = \mathcal{F}$
      break// found the correct face; exit loop
    end if
  end for

  if $\mathcal{F}_E == \text{NULL}$ **then** // failed to find an intersection with $h > 0$
    // the curve did not leave the previous cell; return to it
    $x_{i+1} = x_i$
    $\boldsymbol{y}_{i+1} = \boldsymbol{y}_i$
    $\mathcal{E} = \text{getNeighbor}(\mathcal{E}, \mathcal{F}_{old})$
    continue// next step
  end if

  $\tilde{x} = x_i + \tilde{q}_2$
  $\boldsymbol{k}_2 = \boldsymbol{f}(\mathcal{E}, \tilde{x}, \tilde{\boldsymbol{y}})$

  // intersect the quadratic
  // try the face determined by the Euler step first
  $(\boldsymbol{n}, d) = \text{getPlane}(\mathcal{F}_E)$

  // calculate the coefficients for the quadratic equation

42: $c_0 = \boldsymbol{n} \cdot \boldsymbol{y}_i - d$
43: $c_1 = \boldsymbol{n} \cdot \boldsymbol{k}_1$
44: $c_2 = (\boldsymbol{n} \cdot (\boldsymbol{k}_2 - \boldsymbol{k}_1))/(2\tilde{q}_2)$
45:
46: // solve the quadratic equation
47: **if** $c_2 == 0$ **then** // linear equation
48: **if** $c_1 == 0$ **then** // parallel
49: $h = -1$ // invalidate h
50: **else**
51: // same intersection as the Euler step; finished
52: $h = \tilde{q}_2$
53: $\boldsymbol{y}_{i+1} = \tilde{\boldsymbol{y}}$
54: $\mathcal{F}_{old} = \mathcal{F}_E$
55: $\mathcal{E} = \text{getNeighbor}(\mathcal{E}, \mathcal{F}_E)$
56: **continue**// next step
57: **end if**
58: **else**
59: $D = c_1^2 - 4c_0c_2$
60: **if** $D < 0$ **then** // no intersection
61: $h = -1$ // invalidate h
62: **else**
63: $h_0 = (-c_0 - \sqrt{D})/c_2$
64: $h_1 = (-c_0 + \sqrt{D})/c_2$
65: **if** $h_0 > 0$ **then**
66: $h = h_0$
67: **else**
68: $h = h_1$
69: **end if**
70: **end if**
71: **end if**
72:
73: **if** $h > 0$ **then** // success
74: $\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h(\boldsymbol{k}_1 + h(\boldsymbol{k}_2 - \boldsymbol{k}_1)/(2\tilde{q}_2))$
75: **if** $\tilde{\boldsymbol{y}} \in \mathcal{F}_{old}$ **then**
76: $x_{i+1} = x_i + h$
77: $\mathcal{F}_{old} = \mathcal{F}_E$
78: $\mathcal{E} = \text{getNeighbor}(\mathcal{E}, \mathcal{F}_E)$
79: **continue**
80: **end if**
81: **end if**
82:
83: // intersect with the remaining faces, and find intersection with minimal h
84: $h_{min} = \infty$
85: **for all** faces $\mathcal{F}$ in $\mathcal{E}$ **do**
86: **if** $\mathcal{F} == \mathcal{F}_E$ **then** // already tried that face
87: **continue**// try next face

88: **end if**
89:
90: $(\boldsymbol{n}, d) = \text{getPlane}(\mathcal{F})$
91:
92: // calculate the coefficients for the quadratic equation
93: $c_0 = \boldsymbol{n} \cdot \boldsymbol{y}_i - d$
94: $c_1 = \boldsymbol{n} \cdot \boldsymbol{k}_1$
95: $c_2 = (\boldsymbol{n} \cdot (\boldsymbol{k}_2 - \boldsymbol{k}_1))/(2\tilde{q}_2)$
96:
97: // solve the quadratic equation
98: **if** $c_2 == 0$ **then** // linear equation
99: **if** $c_1 == 0$ **then** // parallel
100: **continue**// try next face
101: **end if**
102: $h = -c_0/c_1$
103: **else**
104: $D = c_1^2 - 4c_0c_2$
105: **if** $D < 0$ **then** // no intersection
106: **continue**// try next face
107: **end if**
108: $h_0 = (-c_0 - \sqrt{D})/c_2$
109: $h_1 = (-c_0 + \sqrt{D})/c_2$
110: **if** $h_0 > 0$ **then**
111: $h = h_0$
112: **else**
113: $h = h_1$
114: **end if**
115: **end if**
116:
117: **if** $h > 0$ and $h < h_{min}$ **then** // success
118: $\tilde{\boldsymbol{y}} = \boldsymbol{y}_i + h(\boldsymbol{k}_1 + h(\boldsymbol{k}_2 - \boldsymbol{k}_1)/(2\tilde{q}_2))$
119: **if** $\tilde{\boldsymbol{y}} \in \mathcal{F}$ **then**
120: // found new minimal intersection; record results
121: $h_{min} = h$
122: $\boldsymbol{y}_{i+1} = \tilde{\boldsymbol{y}}$
123: $\mathcal{F}_{old} = \mathcal{F}$
124: **end if**
125: **end if**
126: **end for**
127:
128: $x_{i+1} = x_i + h_{min}$
129: $\mathcal{E} = \text{getNeighbor}(\mathcal{E}, \mathcal{F}_{old})$
130: **end while**

End Algorithm: MARK2

3.2 Order 3 Method (MARK3)

3.2.1 Outline

When the approach we took for the MARK2 method is extended to an order 3 method, several problems turn up. First we obviously have to solve a cubic equation in this case. What makes matters worse is that the computation of the coefficients for this cubic equation also starts to become quite complicated. It turns out that in order to determine the q_i for the method, it is even necessary to go up to a 4-stage method (for details see chapter 3.3).

All of this makes the complexity of such a method grow very quickly if the order is increased. Therefore we tried a different approach for an order 3 method.

For the development of the new algorithm, which will be called MARK3 (for "Mesh Adaptive Runge-Kutta of order 3"), it is again helpful to consider only a single plane P at first.

$$\boldsymbol{n} \cdot \boldsymbol{p} = d \qquad \forall \boldsymbol{p} \in P$$

Our starting point will be a standard Runge-Kutta order 3 method. Just as this was done in chapter 2.3, it is possible to derive conditions for the parameter set of such a method.

$$\begin{array}{rclrclrcl} a_1 + a_2 + a_3 & = & 1 & a_2 q_2 + a_3 q_3 & = & \frac{1}{2} & a_2 q_2^2 + a_3 q_3^2 & = & \frac{1}{3} \\ a_3 b_{32} q_2 & = & \frac{1}{6} & b_{21} & = & q_2 & b_{31} + b_{32} & = & q_3 \end{array} \tag{3.2.1}$$

Let $q_2 = 1$ and h_i such, that the first step, corresponds to the one the Euler method would take.

$$\begin{array}{rcl} h_i & = & \dfrac{d - \boldsymbol{n} \cdot \boldsymbol{y}_i}{\boldsymbol{n} \cdot \boldsymbol{k}_1} \\ \bar{\boldsymbol{y}}_1 & = & \boldsymbol{y}_i + h_i \boldsymbol{k}_1 \quad \text{with} \quad \bar{\boldsymbol{y}}_1 \in P \end{array} \tag{3.2.2}$$

From this follows that $\bar{\boldsymbol{y}}_1$ is an approximation for $\boldsymbol{y}(x + h_i)$ with a local error of $\mathcal{O}(h_i^2)$. Since the value that the order 3 method produces is also an approximation for $\boldsymbol{y}(x + h_i)$, with a lower error of $\mathcal{O}(h_i^4)$, they must be very close. As $\bar{\boldsymbol{y}}_1$ lies on the plane P, the value from the order 3 method cannot be too far from it as well. The idea is now that this small distance can be closed by an Euler "correction step", without introducing any error of lower order than that of the Runge-Kutta method. These considerations lead to the following algorithm.

Begin Algorithm: MARK3 basic

---

x_i = given
$\boldsymbol{y}_i$ = given

The standard step
$\boldsymbol{k}_1 = \boldsymbol{f}(x_0, \boldsymbol{y}_0)$
$h_i = (d - \boldsymbol{n} \cdot \boldsymbol{y}_0)/(\boldsymbol{n}\boldsymbol{k}_1)$
$\bar{x}_1 = x_i + h_i$
$\bar{\boldsymbol{y}}_1 = \boldsymbol{y}_i + h_i \boldsymbol{k}_1$

$\boldsymbol{k}_2 = \boldsymbol{f}(\bar{x}_1, \bar{\boldsymbol{y}}_1)$

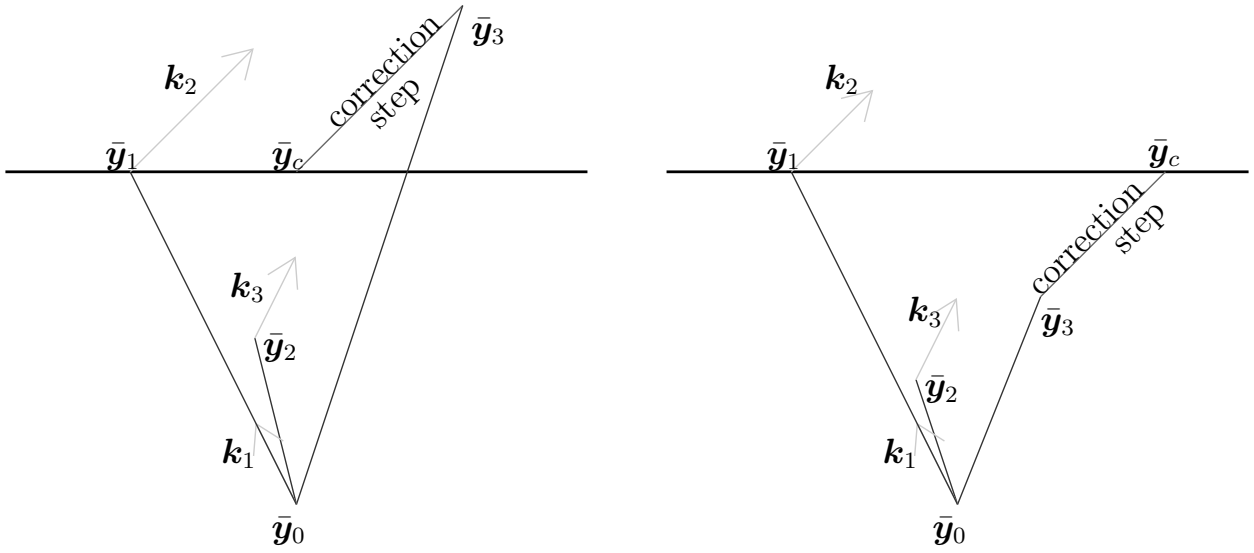


Figure 3.2.1: **Correction step:** The correction step along $\boldsymbol{k}_2$ from $\bar{\boldsymbol{y}}_3$ to $\bar{\boldsymbol{y}}_c$, closes the distance between the calculated point and the face of the cell. It can be either a forward step (right picture) or a backward step (left picture), depending on $\boldsymbol{k}_2$ and ϵ.

$\bar{x}_2 = x_i + q_3 h_i$
$\bar{\boldsymbol{y}}_2 = \boldsymbol{y}_i + h_i(b_{31}\boldsymbol{k}_1 + b_{32}\boldsymbol{k}_2)$

$\boldsymbol{k}_3 = \boldsymbol{f}(\bar{x}_2, \bar{\boldsymbol{y}}_2)$
$\bar{x}_3 = x_i + h_i$
$\bar{\boldsymbol{y}}_3 = \boldsymbol{y}_i + h_i(a_1\boldsymbol{k}_1 + a_2\boldsymbol{k}_2 + a_3\boldsymbol{k}_3)$

The correction step
$\epsilon = (d - \boldsymbol{n}\cdot\bar{\boldsymbol{y}}_3)/(\boldsymbol{n}\boldsymbol{k}_2)$
$\bar{x}_c = \bar{x}_3 + \epsilon$
$\bar{\boldsymbol{y}}_c = \bar{\boldsymbol{y}}_3 + \epsilon\boldsymbol{k}_2$

$x_{i+1} = \bar{x}_c$
$\boldsymbol{y}_{i+1} = \bar{\boldsymbol{y}}_c$

End Algorithm: MARK3 basic

Additionally with the choice of $q_2 = 1$, the parameter restrictions of (3.2.1) can be simplified to a one dimensional manifold in q_3.

$$\begin{array}{lllllllll} q_2 & = & b_{21} = 1 & a_3 & = & \frac{1}{6q_3(1-q_3)} & a_2 & = & \frac{2-3q_3}{6(1-q_3)} \\ a_1 & = & \frac{q_3(4-3q_3)-1}{6q_3(1-q_3)} & b_{32} & = & q_3(1-q_3) & b_{31} & = & q_3^2 \end{array} \tag{3.2.3}$$

Now that this algorithm has been presented and motivated, it remains to be proven.

3.2.2 Proof

To prove that MARK3 is an order 3 method, as claimed we have to show that its local error d_i is $\mathcal{O}(h_i^4)$. To do this the first thing that needs to be shown, is that the length ϵ of the correction step is indeed small. The basis for this is that $\bar{\boldsymbol{y}}_1$ (the result of the Euler step) and $\bar{\boldsymbol{y}}_3$ (the result of the standard order 3 method) are very close, as mentioned above.

$$\begin{aligned} \bar{\boldsymbol{y}}_1 &= \boldsymbol{y}(x+h_i) + \mathcal{O}(h_i^2) && (3.2.4)\\ \bar{\boldsymbol{y}}_3 &= \boldsymbol{y}(x+h_i) + \mathcal{O}(h_i^4) && (3.2.5)\\ &\Downarrow \\ \bar{\boldsymbol{y}}_1 - \bar{\boldsymbol{y}}_3 &= \mathcal{O}(h_i^2) && (3.2.6) \end{aligned}$$

We will now use this to establish that

$$\begin{aligned} \epsilon &= \frac{d - \boldsymbol{n}\cdot\bar{\boldsymbol{y}}_3}{\boldsymbol{n}\cdot\boldsymbol{k}_2} \\ &\overset{(3.2.6)}{=} \frac{d - \boldsymbol{n}\cdot(\bar{\boldsymbol{y}}_1 + \mathcal{O}(h_i^2))}{\boldsymbol{n}\cdot\boldsymbol{k}_2} \\ &= \frac{d - \boldsymbol{n}\cdot\bar{\boldsymbol{y}}_1 + \mathcal{O}(h_i^2)}{\boldsymbol{n}\cdot\boldsymbol{k}_2} \\ &\overset{(3.2.2)}{=} \frac{\mathcal{O}(h_i^2)}{\boldsymbol{n}\cdot\boldsymbol{k}_2} && (3.2.7) \end{aligned}$$

Under the assumption that $\boldsymbol{n}\cdot\boldsymbol{k}_2 = \mathcal{O}(1)$, we get

$$\epsilon = \mathcal{O}(h_i^2) \qquad (3.2.8)$$

The correction step for which ϵ is used, is an Euler step, from $\boldsymbol{y}(x_i+h_i)$ to $\boldsymbol{y}(x_i+h_i+\epsilon)$. Therefore the following holds:

$$\begin{aligned} \boldsymbol{y}_c &= \boldsymbol{y}(x_i+h_i) + \epsilon\boldsymbol{y}'(x_i+h_i) \\ &= \boldsymbol{y}(x_i+h_i+\epsilon) + \mathcal{O}(\epsilon^2) && (3.2.9) \end{aligned}$$

As a side effect the approximation $\boldsymbol{y}_c$ that will result for $\boldsymbol{y}(x_i+h_i+\epsilon)$ will be $\in P$. For doing the correction step correctly $\boldsymbol{y}' = \boldsymbol{f}$ would have to be sampled at x_i+h_i, but it turns out that $\boldsymbol{k}_2$ can be substituted for this sample without introducing too much additional error.

$$\begin{aligned} \boldsymbol{y}'(x_i+h_i) - \boldsymbol{k}_2 &= \boldsymbol{f}(x_i+h_i, \boldsymbol{y}(x_i+h_i)) - \boldsymbol{f}(x_i+h_i, \boldsymbol{y}(x_i) + h_i\boldsymbol{k}_1) \\ &\overset{!}{=} \boldsymbol{f}(x_i+h_i, \boldsymbol{y}(x_i) + h_i\boldsymbol{f} + \mathcal{O}(h_i^2)) - \boldsymbol{f}(x_i+h_i, \boldsymbol{y}(x_i) + h_i\boldsymbol{f}) \\ &\overset{!}{=} \boldsymbol{f}(x_i+h_i, \boldsymbol{y}(x_i) + h_i\boldsymbol{f}) + \mathcal{O}(h_i^2) - \boldsymbol{f}(x_i+h_i, \boldsymbol{y}(x_i) + h_i\boldsymbol{f}) \\ &= \mathcal{O}(h_i^2) && (3.2.10) \end{aligned}$$

The final step, is to put all of the above together to show that $\bar{\boldsymbol{y}}_c$, which is $\in P$ is indeed close enough to $\boldsymbol{y}$ for a local error of $\mathcal{O}(h_i^4)$.

$$\begin{aligned} \bar{\boldsymbol{y}}_c &= \bar{\boldsymbol{y}}_3 + \epsilon\boldsymbol{k}_2 \\ &\overset{(3.2.5)}{=} \boldsymbol{y}(x_i+h_i) + \epsilon\boldsymbol{k}_2 + \mathcal{O}(h_i^4) \end{aligned}$$

$$
\begin{aligned}
&\overset{(3.2.10)}{=} && \boldsymbol{y}(x_i + h_i) + \epsilon(\boldsymbol{y}'(x_i + h_i) + \mathcal{O}(h_i^2)) + \mathcal{O}(h_i^4) \\
&\overset{(3.2.8)}{=} && \boldsymbol{y}(x_i + h_i) + \epsilon\boldsymbol{y}'(x_i + h_i) + \mathcal{O}(h_i^4) \\
&\overset{(3.2.9)}{=} && \boldsymbol{y}(x_i + h_i + \epsilon) + \mathcal{O}(\epsilon^2) + \mathcal{O}(h_i^4) \\
&\overset{(3.2.8)}{=} && \boldsymbol{y}(x_i + h_i + \epsilon) + \mathcal{O}(h_i^4) \qquad (3.2.11)
\end{aligned}
$$

To sum it up, $\bar{\boldsymbol{y}}_c \in P$ is a sufficiently good approximation of $\boldsymbol{y}(x + h_i + \epsilon)$ for an order 3 method, as long as $\boldsymbol{n} \cdot \boldsymbol{k}_2 = \mathcal{O}(1)$. Therefore MARK3 with $x_{i+1} = x_c$ and $\boldsymbol{y}_{i+1} = \boldsymbol{y}_c$ is an order 3 method, as claimed.

What remains to do, to obtain the complete algorithm, is to make the step from intersecting a plane to intersecting the cell, and to examine what happens if $\boldsymbol{n} \cdot \boldsymbol{k}_2$ is not of $\mathcal{O}(1)$.

3.2.3 The Algorithm

To go from the simplified algorithm for intersecting a single plane, to intersecting the cell is more complicated with this algorithm, than with the previous one.

The first problem that needs to be discussed, is where $\boldsymbol{k}_3$ will be sampled. The location according to the algorithm is $\boldsymbol{y}_i + h_i(b_{31}\boldsymbol{k}_1 + b_{32}\boldsymbol{k}_2)$, but the cell containing this point is unknown. If $b_{31} + b_{32} = q_3$ is sufficiently small, and $\boldsymbol{k}_2$ is not too different from $\boldsymbol{k}_1$, this location will still be in the same cell, or at least be so close, that any interpolated value at this position with the wrong cell information will still be accurate enough.

The choice of $q_3 = 1/2$, was used for all the tests in chapter 5. It is a reasonably small choice for the argument outlined above, and it yields a standard Runge-Kutta scheme, that is known to be stable and robust, for the calculation up to $\bar{\boldsymbol{y}}_3$. With this choice the other parameters get set according to the equations (3.2.3).

$$
a_1 = 1/6 \qquad a_2 = 1/6 \qquad a_3 = 4/6 \qquad q_2 = b_{21} = 1 \qquad q_3 = 1/2 \qquad b_{31} = b_{32} = 1/4 \quad (3.2.12)
$$

Another problem that the algorithm has to face, is what happens if $\boldsymbol{n} \cdot \boldsymbol{k}_2 = \mathcal{O}(h_i)$. In this case, the accuracy of the correction step breaks down, which means that overall accuracy is lost. The first step in treating this case is to detect it properly. The simple approach of checking whether $\boldsymbol{n} \cdot \boldsymbol{k}_2 < C$, for some constant C, is not in general the right way to do it. Intuitively it makes no sense that $|\boldsymbol{k}_2|$ has an influence on whether a fallback is necessary. Therefore the condition $(\boldsymbol{n} \cdot \boldsymbol{k}_2)/|\boldsymbol{k}_2| < C$ (where $\boldsymbol{n}$ is assumed to be already normalized), seems to be reasonable. That this intuition is indeed correct in most cases, can be seen if we examine ϵ in more detail than we did in equation (3.2.7). For this more detailed look at ϵ it is also necessary to reexamine the distance between $\bar{\boldsymbol{y}}_1$ and $\bar{\boldsymbol{y}}_3$ in more detail.

$$
\begin{aligned}
\bar{\boldsymbol{y}}_1 - \boldsymbol{y}(x_i + h_i) &\overset{!}{=} \boldsymbol{y}(x_i) + h_i\boldsymbol{f} - \boldsymbol{y}(x_i) - h_i\boldsymbol{f} - \frac{h_i^2}{2}\boldsymbol{y}''(x_i) + \mathcal{O}(h_i^3) \\
&= -\frac{h_i^2}{2}\boldsymbol{f}' + \mathcal{O}(h_i^3) \\
&= -\frac{h_i^2}{2}(\boldsymbol{f}_x + \boldsymbol{f}_{\boldsymbol{y}}\boldsymbol{f}) + \mathcal{O}(h_i^3) \qquad (3.2.13) \\
\bar{\boldsymbol{y}}_3 &= \boldsymbol{y}(x_i + h_i) + \mathcal{O}(h_i^4) \\
&\overset{(3.2.13)}{=} \bar{\boldsymbol{y}}_1 + \frac{h_i^2}{2}(\boldsymbol{f}_x + \boldsymbol{f}_{\boldsymbol{y}}\boldsymbol{f}) + \mathcal{O}(h_i^3) \qquad (3.2.14)
\end{aligned}
$$

Equation (3.2.14) can now be used to show, that

$$\begin{aligned}
\epsilon &= \frac{d - \boldsymbol{n}\cdot\bar{\boldsymbol{y}}_3}{\boldsymbol{n}\cdot\boldsymbol{k}_2} \\
&\overset{(3.2.14)}{=} \frac{d - \boldsymbol{n}\cdot(\bar{\boldsymbol{y}}_1 + (\boldsymbol{f}_x + \boldsymbol{f}_{\boldsymbol{y}}\boldsymbol{f}) + \mathcal{O}(h_i^3))}{\boldsymbol{n}\cdot\boldsymbol{f}(x_i + h_i, \boldsymbol{y}(x_i) + h_i\boldsymbol{f})} \\
&\overset{!}{=} \frac{d - \boldsymbol{n}\cdot\bar{\boldsymbol{y}}_1 + \frac{h_i^2}{2}\boldsymbol{n}\cdot(\boldsymbol{f}_x + \boldsymbol{f}_{\boldsymbol{y}}\boldsymbol{f}) + \mathcal{O}(h_i^2)}{\boldsymbol{n}\cdot(\boldsymbol{f} + h_i(\boldsymbol{f}_x + \boldsymbol{f}_{\boldsymbol{y}}\boldsymbol{f}))} \\
&= \frac{h_i^2\boldsymbol{n}\cdot(\boldsymbol{f}_x + \boldsymbol{f}_y\boldsymbol{f}) + \mathcal{O}(h_i^3)}{\boldsymbol{n}\cdot(\boldsymbol{f} + h_i(\boldsymbol{f}_x + \boldsymbol{f}_{\boldsymbol{y}}\boldsymbol{f}))}
\end{aligned} \tag{3.2.15}$$

Provided that $\boldsymbol{f}$ is not dependent on x, eg. $\boldsymbol{f}$ is not time dependent, then $\boldsymbol{f}_x = 0$ and $|\boldsymbol{f}|$ can be cancelled out of the equation. Hence for not time dependent data, it is reasonable to check $(\boldsymbol{n}\cdot\boldsymbol{k}_2)/|\boldsymbol{k}_2|$, since $|\boldsymbol{k}_2| = |\boldsymbol{f}| + \mathcal{O}(h_i)$. If $\boldsymbol{f}$ is time dependent, then we cannot cancel $|\boldsymbol{f}|$ from equation (3.2.15), and therefore have to take the absolute size of $\boldsymbol{k}_2$ into account. In this case the simpler condition $\boldsymbol{n}\cdot\boldsymbol{k}_2$ is the right one to choose. The flip side of this choice is, that it can lead to more false positives. In practice both of these conditions become true when the particle path is almost parallel to the face it crosses, since $(\boldsymbol{n}\cdot\boldsymbol{k}_2)/|\boldsymbol{k}_2| = \cos\alpha$ where α is the angle enclosed by $\boldsymbol{n}$ and $\boldsymbol{k}_2$.

A fallback for this situation is easily found. If the particle position cannot be adjusted to lie on the cell boundary, it is left untouched and a a local cell search is performed instead. Experiments showed that in cases where the fallback is needed, the step size is often almost twice as long as in non-fallback cases. Therefore it makes sense to half the step size in the case of a fallback. This can be done by considering $q_2 = 2$, which allows a reuse of the already determined value for $\boldsymbol{k}_2$. One fallback step, especially if it was computed with the halved step size, will often not be sufficient to leave the current cell. As it is very likely that the following step will intersect the same face in about the same angle as the previous one, thus triggering another fallback, the treatment of a fallback has to be extended. Whenever a fallback is triggered the algorithm will revert to a standard Runge-Kutta method until the current cell is left. The whole fallback treatment outlined is not very costly. The first step, costs almost nothing in addition to a normal step, because $\boldsymbol{k}_2$ can be reused and most often $\boldsymbol{k}_3$ will not have been calculated yet, when the algorithm detects the need for the fallback. Practice shows that almost never more than three steps will be needed to leave the current cell. Additionally all the cell searches will terminate after at most one step, as the step size has already been set to half the size of the current cell. Altogether the fallback corresponds to a standard Runge-Kutta method with cell searching, that had its step size adapted to the current element size.

Another problem case occurs when the correction step intersects the plane that was found with the first Euler step, outside the actual face. Under certain circumstances it is possible to just continue the correction step until it hits a neighboring face. To see why this works we have to look at the distance δ along $\boldsymbol{k}_2$, between $\boldsymbol{y}_c$ and the intersection with the neighboring face, which we will call $\boldsymbol{y}_{cc}$. The situation will then look like in figure 3.2.2. We can use (3.2.6) to show that

$$\begin{aligned}
\bar{\boldsymbol{y}}_c - \bar{\boldsymbol{y}}_1 &= \bar{\boldsymbol{y}}_3 + \epsilon\boldsymbol{k}_2 - \bar{\boldsymbol{y}}_1 \\
&\overset{(3.2.6)}{=} \boldsymbol{y}(x + h_i) + \mathcal{O}(h_i^4) + \epsilon\boldsymbol{k}_2 - \boldsymbol{y}(x + h_i) - \mathcal{O}(h_i^2) \\
&= \mathcal{O}(h_i^2)
\end{aligned}$$

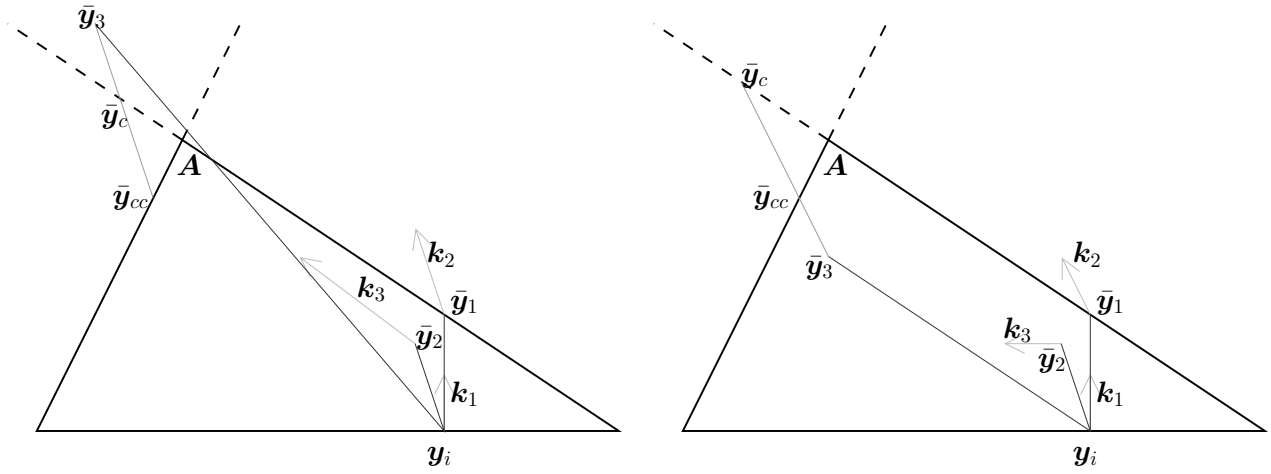


Figure 3.2.2: **Correction error:** In these images the correction step intersects outside the face, that the first Euler step intersected.

Since the corner (in 2D) or edge (in 3D) $\boldsymbol{A}$ of the polygon lies in between $\bar{\boldsymbol{y}}_c$ and $\bar{\boldsymbol{y}}_1$ we also have

$$\boldsymbol{A} - \bar{\boldsymbol{y}}_c = \mathcal{O}(h_i^2) \tag{3.2.16}$$

If the neighboring $\mathcal{F}_2$ is described by $P_2 = \{\boldsymbol{p} \,|\, \boldsymbol{n}_2 \cdot \boldsymbol{p} = d_2\}$, we can use this information to compute

$$\begin{aligned}
\delta &= \frac{d_2 - \boldsymbol{n}_2 \cdot \bar{\boldsymbol{y}}_c}{\boldsymbol{n}_2 \cdot \boldsymbol{k}_2} \\
&= \frac{d_2 - \boldsymbol{n}_2 \cdot \bar{\boldsymbol{y}}_c}{\boldsymbol{n}_2 \cdot \boldsymbol{k}_2} \\
&\overset{(3.2.16)}{=} \frac{d_2 - \boldsymbol{n}_2 \cdot (\boldsymbol{A} + \mathcal{O}(h_i^2))}{\boldsymbol{n}_2 \cdot \boldsymbol{k}_2} \\
&= \frac{d_2 - \boldsymbol{n}_2 \cdot \boldsymbol{A} + \mathcal{O}(h_i^2)}{\boldsymbol{n}_2 \cdot \boldsymbol{k}_2} \\
&= \frac{\mathcal{O}(h_i^2)}{\boldsymbol{n}_2 \cdot \boldsymbol{k}_2}
\end{aligned} \tag{3.2.17}$$

Again we have to assume that $\boldsymbol{n}_2 \cdot \boldsymbol{k}_2 = \mathcal{O}(1)$ to get $\delta = \mathcal{O}(h_i^2)$. The distance for the Euler correction step is then $\epsilon + \delta = \mathcal{O}(h_i^2)$, which is good enough for the proof in chapter 3.2.2. If instead $\boldsymbol{n}_2 \cdot \boldsymbol{k}_2 = \mathcal{O}(h_i)$, we again have to rely on the fallback. Since finding the neighbors and calculating the additional intersections means quite some work and because this case should be relatively rare as well, we could also use the fallback right away. Hence searching the neighbors for an intersection makes most sense for cells with a high number of faces, since then the normals of the neighboring faces will be similar to the normal of the original face and the risk of $\boldsymbol{n}_2 \cdot \boldsymbol{k}_2$ being small, which would trigger the fallback, is lower.

Put together we get the pseudocode shown in the algorithm MARK3 at the end of this section. In figure 3.2.3 this method (with $q_3 = 1/2$) solves the same problem, of a whirl around the center of the domain that was already shown in chapter 3.1. It can be observed, that for this data, $\bar{\boldsymbol{y}}_2$ is never outside of the current cell, in all non-fallback steps, which is a good indication that our choice of q_3 should be fine in all cases where the data is sufficiently smooth.

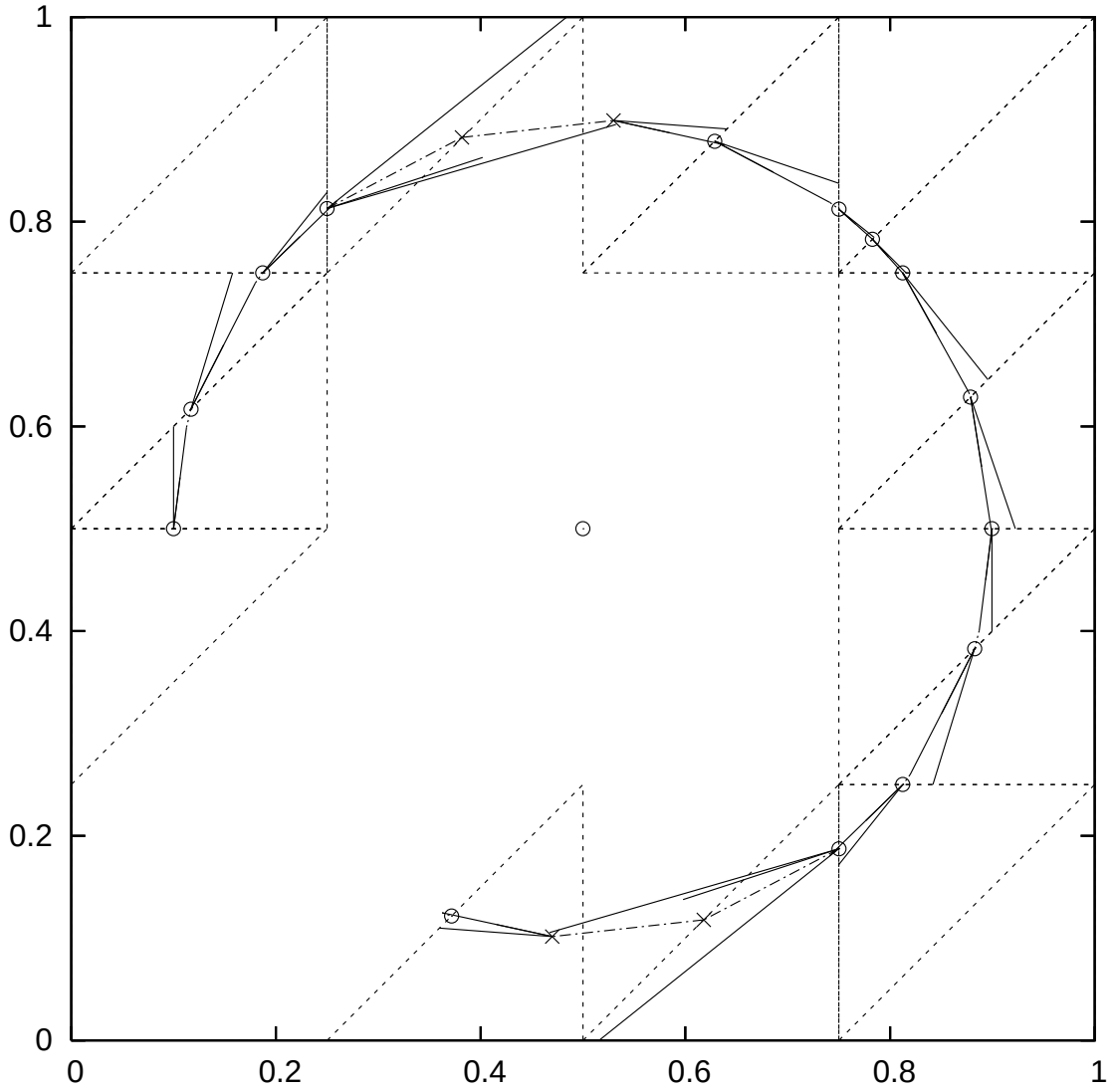


Figure 3.2.3: **MARK3 trace:** The flow in the data of this plot is circling around the center of the domain. The actual intersections with the cell boundaries that the method determines, are marked by circles. The illustration shows for each iteration the three steps ($\bar{\boldsymbol{y}}_1$, $\bar{\boldsymbol{y}}_2$ and $\bar{\boldsymbol{y}}_3$) calculated by the standard Runge-Kutta 3 method (although the step for $\bar{\boldsymbol{y}}_2$ is barely visible), and the correction step to $\bar{\boldsymbol{y}}_c$. Notice also that $\bar{\boldsymbol{y}}_2$ is never sampled outside the current cell, for a normal step. The fallback steps are plotted with dashed lines.

Begin Algorithm: MARK3

Require: starting point $(x_0, \boldsymbol{y}_0)$, starting cell $\mathcal{E}$
Require: interpolating function $\boldsymbol{f}$
Require: end time t_{max}
Require: constant C for fallback condition

$\mathcal{F}_{old} = \text{NULL}$
$i = -1$

while $t_{max} > x_{i+1}$ **do**
  $i = i + 1$
  $\boldsymbol{k}_1 = \boldsymbol{f}(\mathcal{E}, x_i, \boldsymbol{y}_i)$

  // the Euler step
  $\mathcal{F}_E = \text{NULL}$
  for all faces $\mathcal{F}$ of $\mathcal{E}$ **do**
    $(\boldsymbol{n}, d) = \text{getPlane}(\mathcal{F})$
    if $\boldsymbol{n} \cdot \boldsymbol{k}_1 == 0$ **then** // $\boldsymbol{n}$ and $\boldsymbol{k}_1$ are parallel
      continue// try next face
    end if
    $h = (d - \boldsymbol{n} \cdot \boldsymbol{y}_i)/(\boldsymbol{n} \cdot \boldsymbol{k}_1)$
    if $h \leq 0$ **then**
      continue// try next face
    end if
    $\bar{\boldsymbol{y}}_1 = \boldsymbol{y}_i + h\boldsymbol{k}_1$
    if $\bar{\boldsymbol{y}}_1 \in \mathcal{F}$ **then**
      $\mathcal{F}_E = \mathcal{F}$
      break// found correct intersection; break loop
    end if
  end for

  if $\mathcal{F}_E == \text{NULL}$ **then** // failed to find an intersection with $h > 0$
    // the curve did not leave the previous cell; return to it
    $x_{i+1} = x_i$
    $\boldsymbol{y}_{i+1} = \boldsymbol{y}_i$
    $\mathcal{E} = \text{getNeighbor}(\mathcal{E}, \mathcal{F}_{old})$
    continue// next step
  end if

  $\bar{x}_1 = x_i + h$
  $\boldsymbol{k}_2 = \boldsymbol{f}(\mathcal{E}, \bar{x}_1, \bar{\boldsymbol{y}}_1)$

  $\bar{x}_2 = x_i + h/2$
  $\bar{\boldsymbol{y}}_2 = \boldsymbol{y}_i + (h/4)(\boldsymbol{k}_1 + \boldsymbol{k}_2)$
  $\boldsymbol{k}_2 = \boldsymbol{f}(\mathcal{E}, \bar{x}_2, \bar{\boldsymbol{y}}_2)$

41: $\bar{x}_3 = x_i + h/6$
42: $\bar{\boldsymbol{y}}_3 = \boldsymbol{y}_i + (h/6)(\boldsymbol{k}_1 + \boldsymbol{k}_2 + 4\boldsymbol{k}_3)$
43:
44: **if** $(\boldsymbol{n} \cdot \boldsymbol{k}_2)/|\boldsymbol{k}_2| < C$ **then** // test for fallback
45: // perform fallback (see below)
46: $(\mathcal{E}, x_{i+1}, \boldsymbol{y}_{i+1}) = \text{fallback}(\bar{x}_3, \bar{\boldsymbol{y}}_3, \mathcal{E}, \boldsymbol{k}_1, \boldsymbol{k}_2, \boldsymbol{f})$
47: $\mathcal{F}_{old} = \text{NULL}$ // invalidate last face, since $\boldsymbol{y}_{i+1}$ is not on a face
48: **continue**// next step
49: **end if**
50:
51: // perform the correction step
52: $(\boldsymbol{n}, d) = \text{getPlane}(\mathcal{F}_E)$
53: $\epsilon = (d - \boldsymbol{n} \cdot \bar{\boldsymbol{y}}_3)/(\boldsymbol{n} \cdot \boldsymbol{k}_2)$
54: $\boldsymbol{y}_{i+1} = \bar{\boldsymbol{y}}_3 + \epsilon \boldsymbol{k}_2$
55: **if** $\boldsymbol{y}_{i+1} \in \mathcal{F}_E$ **then** // correction step was successful
56: $x_{i+1} = \bar{x}_3 + \epsilon$
57: $\mathcal{F}_{old} = \mathcal{F}_E$
58: **else** // correction step was not successful
59: // search the neighboring sides, for minimal intersection with the correction step
60: $\epsilon_{min} = \infty$
61: **for all** faces $\mathcal{F}$ in $\mathcal{E}$ neighboring $\mathcal{F}_E$ **do**
62: $(\boldsymbol{n}, d) = \text{getPlane}(\mathcal{F})$
63: **if** $\boldsymbol{n} \cdot \boldsymbol{k}_2 == 0$ **then** // correction step is parallel to the face
64: **continue**// try next face
65: **end if**
66: $\epsilon = (d - \boldsymbol{n} \cdot \bar{\boldsymbol{y}}_3)/(\boldsymbol{n} \cdot \boldsymbol{k}_2)$
67: $\bar{\boldsymbol{y}}_c = \bar{\boldsymbol{y}}_3 + \epsilon \boldsymbol{k}_2$
68: **if** $|\epsilon| < |\epsilon_{min}|$ and $\bar{\boldsymbol{y}}_c \in \mathcal{F}$ **then**
69: $\epsilon_{min} = \epsilon$
70: $\boldsymbol{n}_{min} = \boldsymbol{n}$
71: $\boldsymbol{y}_{i+1} = \bar{\boldsymbol{y}}_c$
72: $\mathcal{F}_{old} = \mathcal{F}$
73: **end if**
74: **end for**
75:
76: // check if a correction step to a neighbor was successful
77: **if** $\mathcal{F}_{old} == \text{NULL}$ or $(\boldsymbol{n}_{min} \cdot \boldsymbol{k}_2)/|\boldsymbol{k}_2| < C$ **then**
78: // perform fallback (see below)
79: $(\mathcal{E}, x_{i+1}, \boldsymbol{y}_{i+1}) = \text{fallback}(\bar{x}_3, \bar{\boldsymbol{y}}_3, \mathcal{E}, \boldsymbol{k}_1, \boldsymbol{k}_2, \boldsymbol{f})$
80: $\mathcal{F}_{old} = \text{NULL}$
81: **continue**// next step
82: **end if**
83:
84: $x_{i+1} = \bar{x}_3 + \epsilon_{min}$
85: **end if**
86:

87: // correction step was successful; determine next cell
88: $\mathcal{E} = \text{getNeighbor}(\mathcal{E}, \mathcal{F}_{old})$
89: **end while**

End Algorithm: MARK3

Begin Algorithm: Fallback for MARK3

1: **function** $(\mathcal{E}_{new},\, x_{new},\, \boldsymbol{y}_{new}) = \text{fallback}(x_i, y_i, h, \mathcal{E}, \boldsymbol{k}_1, \boldsymbol{k_2}, \boldsymbol{f})$
2: // first step in fallback with previously computed $\boldsymbol{k}_1$ and $\boldsymbol{k_2}$
3: // consider $q_2 = 2$
4: $h = h/2$
5: // choosing $q_3 = 1/2$, should make a cell search unnecessary
6: $\bar{x}_2 = x_i + h/2$
7: $\bar{\boldsymbol{y}}_2 = \boldsymbol{y}_i + h(\boldsymbol{k}_1 + 13\boldsymbol{k}_2)/14$
8: $\bar{x}_3 = x_i + h$
9: $\bar{\boldsymbol{y}}_3 = \boldsymbol{y}_i + h(\boldsymbol{k}_1 - 2\boldsymbol{k}_2 + 7\boldsymbol{k}_3)/6$
10: $\mathcal{E}_{new} = \text{localCellSearch}(\mathcal{E}, \bar{\boldsymbol{y}}_3)$
11:
12: **while** $\mathcal{E}_{new} == \mathcal{E}$ **do** // continue with normal Runge-Kutta 3 until cell is left
13: $\boldsymbol{k}_1 = \boldsymbol{f}(\mathcal{E}, \bar{x}_3, \bar{\boldsymbol{y}}_3)$
14: $\bar{x}_1 = \bar{x}_3 + h$
15: $\bar{\boldsymbol{y}}_1 = \bar{\boldsymbol{y}}_3 + h\boldsymbol{k}_1$
16:
17: $\mathcal{E}_{new} = \text{localCellSearch}(\mathcal{E}, \bar{\boldsymbol{y}}_1)$
18: $\boldsymbol{k}_2 = \boldsymbol{f}(\mathcal{E}_{new}, \bar{x}_1, \bar{\boldsymbol{y}}_1)$
19: $\bar{x}_2 = \bar{x}_3 + h/2$
20: $\bar{\boldsymbol{y}}_2 = \bar{\boldsymbol{y}}_3 + h(\boldsymbol{k}_1 + \boldsymbol{k}_2)/4$
21:
22: $\mathcal{E}_{new} = \text{localCellSearch}(\mathcal{E}, \bar{\boldsymbol{y}}_2)$
23: $\boldsymbol{k}_3 = \boldsymbol{f}(\mathcal{E}_{new}, \bar{x}_3, \bar{\boldsymbol{y}}_3)$
24: $\bar{x}_3 = \bar{x}_3 + h$
25: $\bar{\boldsymbol{y}}_3 = \bar{\boldsymbol{y}}_3 + h(\boldsymbol{k}_1 + \boldsymbol{k}_2 + 4\boldsymbol{k}_3)/6$
26:
27: $\mathcal{E}_{new} = \text{localCellSearch}(\mathcal{E}, \bar{\boldsymbol{y}}_3)$
28: **end while**
29: **return** $(\mathcal{E}_{new},\, \bar{x}_3,\, \bar{\boldsymbol{y}}_3)$
30: **end function**

End Algorithm: Fallback for MARK3

3.3 Suggestions for future work

In this chapter we will examine some ideas and further extensions of the methods presented, that were developed while working on this topic.

It was already mentioned in section 3.2.1, that we tried to extend the MARK2 method to order 3. Although we decided to drop this approach it will be presented here for completeness. Again it is not possible to fix q_2 and q_3 for this method in advance. Instead values for $\tilde{q}_2 := q_2 h$ and $\tilde{q}_3 := q_3 h$ have to be determined. Sadly it is not possible to determine a direct relationship between $\tilde{q}_2$ and $\tilde{q}_3$ without involving h (which is at this time still undefined). This problem can be solved by going up from a 3 to a 4-stage order 3 method. This introduces enough additional degrees of freedom that it is possible to set $\tilde{q}_2$, $\tilde{q}_3$ and $\tilde{q}_4$ independently of each other. Just as this was done in section 2.3 it is possible to derive a set of conditions for the remaining parameters, which guarantee a local error of $\mathcal{O}(h^4)$.

$$\begin{aligned} a_1 + a_2 + a_3 + a_3 + a_4 &= 1 \\ a_2 q_2 + a_3 q_3 + a_4 q_4 &= \frac{1}{2} \\ a_2 q_2^2 + a_3 q_3^2 + a_4 q_4^2 &= \frac{1}{3} \\ a_3 b_{32} q_2 + a_4 (b_{42} q_2 + b_{43} q_3) &= \frac{1}{6} \end{aligned} \tag{3.3.1}$$

The parameters b_{ij} can be set depending on q_i.

$$\begin{array}{lllllll} b_{31} = q_3 d_{31} & & b_{32} = q_3 d_{32} & & & & \\ b_{41} = q_4 d_{41} & & b_{42} = q_4 d_{42} & & b_{43} = q_4 d_{43} & & \\ & & \text{with} & & & & \\ d_{31} + d_{32} = 1 & & & & d_{41} + d_{42} + d_{43} = 1 & & \end{array} \tag{3.3.2}$$

where d_{ij} can be freely chosen. Using all of the above $a_1 \dots a_4$ can be expressed in terms of $q_2 \dots q_4$.

$$\begin{aligned} a_1 &= 1 - a_2 - a_3 - a_4 \\ a_2 &= \frac{q_4 - q_3 + (d_{42} q_2 + d_{43} q_3)(3q_3 - 2) - d_{32} q_2 (3q_4 - 2)}{6 q_2 F} \\ a_3 &= \frac{q_2 - q_4 - (d_{42} q_2 + d_{43} q_3)(3q_2 - 2)}{6 q_3 F} \\ a_4 &= \frac{q_3 - q_2 + d_{32} q_2 (3q_2 - 2)}{6 q_4 F} \\ & \text{with} \\ F &= d_{32} q_2 (q_2 - q_4) - (d_{42} q_2 + d_{43} q_3)(q_2 - q_3) \end{aligned} \tag{3.3.3}$$

These equations are then substituted into the final step of the Runge-Kutta method. We also have to substitute

$$\begin{array}{lll} q_2 = \tilde{q}_2 / h_i & q_3 = \tilde{q}_3 / h_i & q_4 = \tilde{q}_4 / h_i \\ F = \tilde{F} / h_i^2 = (d_{32} \tilde{q}_2 (\tilde{q}_2 - \tilde{q}_4) - (d_{42} \tilde{q}_2 + d_{43} \tilde{q}_3)(\tilde{q}_2 - \tilde{q}_3)) / h_i^2 & & \end{array}$$

All this put together then yields the following, quite complex equation for the final step in the algorithm.

$$\begin{aligned}
\boldsymbol{y}_{i+1} &= \boldsymbol{y}_i + h_i(a_1\boldsymbol{k}_1 + a_2\boldsymbol{k}_2 + a_3\boldsymbol{k}_3 + a_4\boldsymbol{k}_4) \\
&= \boldsymbol{y}_i + h_i(\boldsymbol{k}_1 + a_2(\boldsymbol{k}_2 - \boldsymbol{k}_1) + a_3(\boldsymbol{k}_3 - \boldsymbol{k}_1) + a_4(\boldsymbol{k}_4 - \boldsymbol{k}_1)) \\
&= \boldsymbol{c}_0 + \boldsymbol{c}_1 h_i + \boldsymbol{c}_2 h_i^2 + \boldsymbol{c}_3 h_i^3
\end{aligned}$$

with

$$\begin{aligned}
\boldsymbol{c}_0 &= \boldsymbol{y}_i \\
\boldsymbol{c}_1 &= \boldsymbol{k}_1 \\
\boldsymbol{c}_2 &= \frac{\boldsymbol{k}_2 - \boldsymbol{k}_1}{2\tilde{q}_2\tilde{F}}(\tilde{q}_3(d_{42}\tilde{q}_2 + d_{43}\tilde{q}_3) - d_{32}\tilde{q}_2\tilde{q}_4) - \frac{\boldsymbol{k}_3 - \boldsymbol{k}_1}{2\tilde{q}_3\tilde{F}}\tilde{q}_2(d_{42}\tilde{q}_2 + d_{43}\tilde{q}_3) + \frac{\boldsymbol{k}_4 - \boldsymbol{k}_1}{2\tilde{q}_4\tilde{F}}d_{32}\tilde{q}_2^2 \\
\boldsymbol{c}_3 &= \frac{\boldsymbol{k}_2 - \boldsymbol{k}_1}{6\tilde{q}_2\tilde{F}}(\tilde{q}_4 - \tilde{q}_3 + 2d_{32}\tilde{q}_2 - 2(d_{42}\tilde{q}_2 + d_{43}\tilde{q}_3)) \\
&\quad + \frac{\boldsymbol{k}_3 - \boldsymbol{k}_1}{6\tilde{q}_3\tilde{F}}(\tilde{q}_2 - \tilde{q}_4 + 2(d_{42}\tilde{q}_2 + d_{43}\tilde{q}_3)) \\
&\quad + \frac{\boldsymbol{k}_4 - \boldsymbol{k}_1}{6\tilde{q}_4\tilde{F}}(\tilde{q}_3 - \tilde{q}_2 - 2d_{32}\tilde{q}_2)
\end{aligned} \tag{3.3.4}$$

Intersecting this step with the plane then yields a scalar, cubic equation that can be solved for h_i.

$$\begin{aligned}
\boldsymbol{n} \cdot \boldsymbol{y}_{i+1} &= d \\
\boldsymbol{n} \cdot \boldsymbol{c}_0 + \boldsymbol{n} \cdot \boldsymbol{c}_1 h_i + \boldsymbol{n} \cdot \boldsymbol{c}_2 h_i^2 + \boldsymbol{n} \cdot \boldsymbol{c}_3 h_i^3 &= d
\end{aligned} \tag{3.3.5}$$

To solve this cubic equation a fixed number (two should be sufficient) of Newton iterations can be applied. Using Newton should be accurate enough and definitely faster then employing Cardano's formula.

It is obvious that the calculation of the coefficients c_2 and c_3 is quite complicated and therefore time consuming. The MARK3 method does not need anything like this. Also $\tilde{q}_3$ and $\tilde{q}_4$ have to be set so, that k_3 and k_4 are not sampled outside of the current cell. This is the same problem that turned up in the MARK3 method, but here it already affects two sample points, which further compounds the problem. It is also not possible to employ any shortcuts when intersecting the cell faces with the cubic curve, as this was possible for the quadratic curve of MARK2 (see section 3.1. The only advantage that this method would have is a higher robustness, as there are no special cases that have to be treated. All in all MARK3 clearly seems to be the better method.

The opposite approach of constructing an order 2 method similar to MARK3 fairs not much better. A method like this would retain all its problems, like the special cases that have to be treated and its complexity would be comparable to that of MARK2. Therefore this approach is clearly inferior to MARK2 at order 2.

Finally we will take a look at the possibilities of extending MARK3 to even higher orders. To retain a local error of a higher order than $\mathcal{O}(h^4)$ it is necessary to improve the accuracy of the correction step. This can be achieved by employing a higher order method like MARK2, which is especially well suited here, because it only samples $\boldsymbol{f}$ on the cell boundaries. Since it is not clear whether the starting point for the correction step is still in the current cell, this assures that $\boldsymbol{f}$ does not get sampled in the wrong cell. Furthermore it is not possible to

choose $q_2 = 1$ in a 4-stage, order 4 method. Therefore we have to add an additional stage, if we want to preserve the first Euler step, that yields an estimate for h. Other choices of q_2 would make it impossible to use $\boldsymbol{k}_2$ for the correction step, which means we would have to add another sample stage anyway.

It would also be imaginable to replace the first Euler step in MARK3, by MARK2. The point found on a face of the cell by MARK2 would then be closer to the final approximation of $\boldsymbol{y}$, and therefore ϵ would be smaller. This should make it possible to again employ a simple Euler step for the correction step. For methods of even higher order MARK2 could be employed for both, the first step onto a face of the cell and the correction step.

In general we can note that for higher order methods, the complexity grows very quickly. It gets increasingly complicated to intersect the higher order polynomials, that these methods implicitly calculate, with the cell boundaries. If very high accuracy is required, it is therefore probably advisable not to increase the order of the methods indefinitely, but instead to use multiple steps to cross one cell. Something like MARK2 could be used to detect if the cell is left with the current step, and to shorten this step so that it ends on the cell boundary.

Chapter 4

Implementation

To experiment with the methods presented in this work, two programs have been developed: A 2D program in Matlab/Octave which is mainly intended to illustrate how the algorithms work, and to do some convergence analysis (see section 5.1) and a 3D C++ implementation that is geared towards speed and practical usability (see section 5.2). Additionally to presenting these programs in more detail, we will also discuss some troubles and pitfalls one can run into, when trying to implement MARK2 and MARK3.

4.1 Implementation pitfalls

The real world limitations of computers often make the implementation of algorithms, more complicated than they appear on paper. This is also true for MARK2 and MARK3.

One of the problems that turns up in the implementation of both algorithms, is that most often the point determined in one step is not exactly on the cell boundary, due to numerical inaccuracies. Hence this point, which is the starting point for the next step, can be slightly inside or outside the current cell. This can lead to an intersection of the Euler step with the face that the algorithm started from. To compensate this, both algorithms take an additional input argument that specifies the side on which the starting point is supposed to be, so that it can be ignored for the intersection with the Euler step.

Similar numerical inaccuracies can also cause problems in the algorithms for checking whether an intersection is inside a face, or in the raycasting version of the local cell search.

Another not quite obvious special case is, if the curve only touches a face in a step, and not actually leaves the cell. In this case the Euler step, of the next iteration, will only find intersections with a negative h in the current cell. Hence the algorithm has to return to the previous cell to continue. This case is already being treated in the pseudocodes for MARK2 and MARK3 shown in sections 3.1 and 3.2 respectively.

4.2 2D Matlab/Octave Implementation

The 2D implementation was created to do fast prototyping and testing of the new algorithms. For this purpose Matlab/Octave seemed to be an appropriate tool, because it is simple and allows fast development. It makes it also easy to create detailed step by step visualizations (as seen in the figures 3.1.2 and 3.2.3), which helps to identify problem cases and pitfalls in

the algorithms, such as those detailed in the previous section 4.1. The octave implementation is also useful for doing convergence analysis (see section 5.1).

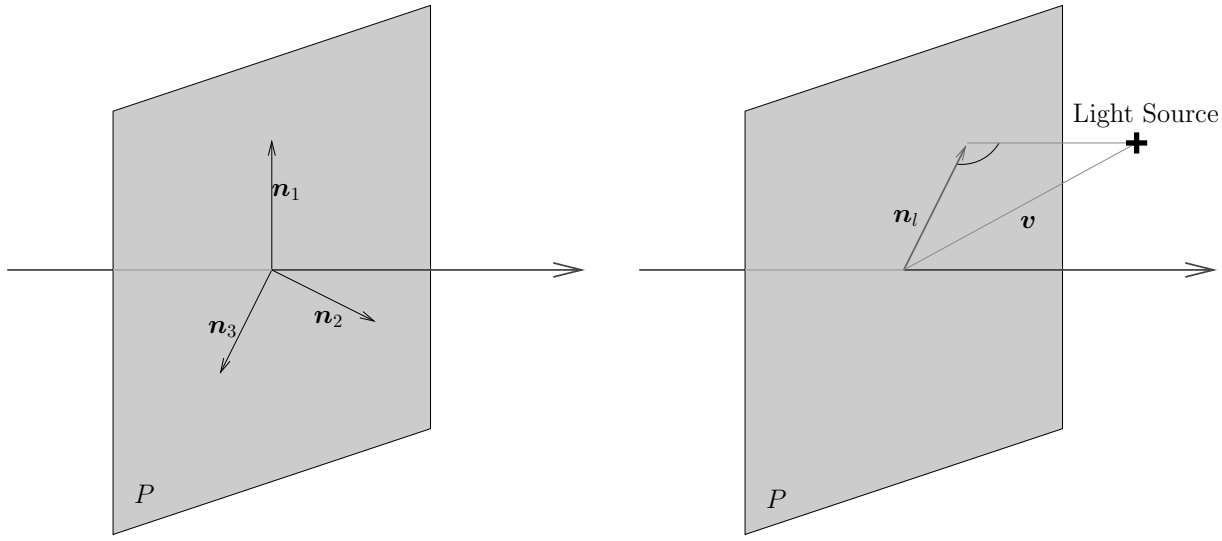
4.3 3D C++ Implementation

The C++ implementation, in contrast to the Matlab/Octave program, is intended to be fast and useable in real applications. To this end an external grid library for unstructured grids (UGLI see Appendix A) was used. The use of an external library makes it easier for an application to interface with the particle tracing methods, and it guarantees that the algorithms are clearly separate from the representation of the mesh. This makes it easier to port them to any other mesh representation that an application might use.

Furthermore the classes and interfaces that encapsulate the algorithms are designed in a way that makes it easy to plug them into an application using UGLI without too much work. All algorithms presented in this work, have been unified under one abstract interface class, which makes it possible to seamlessly switch between them. A similar interface class has been created for the interpolation function. The existence of this class allows an application to switch the interpolation function at runtime if this is needed.

The visualization for this implementation was done in plain OpenGL, using GLUT as a windowing toolkit. Since proper illumination of the particle paths is an important visual cue, to help interpret the results of a trace, it was necessary to implement some sort of shading for the generated traces. This line shading is more complicated than surface shading, as the normal of a line is not uniquely defined. Instead the tangent of the particle path only defines a two dimensional normal space, in which a normal can be picked (see figure 4.3.1(a)). By convention the normal that is used for the calculation is the one, which minimizes the angle between the light vector and the normal (see figure 4.3.1(b)). By applying this convention it is even possible to eliminate the normal from the lighting equation as shown in [13], where the light vector and the tangent vector are used to index into an OpenGL texture map, to determine the lighting. Since the speed of the visualization was not that important for this work, our program just calculates the normal vector on the fly for each frame.

Additionally to the particle traces on the mesh, the program can also read in and display surface meshes in the object file format (OFF, `http://www.geomview.org/docs/html/geomview_41.html`), and extract and display the surface of the volume.



(a) The normal plane P for a line.

(b) The line normal $\boldsymbol{n}_l$ chosen for a specific light source.

Figure 4.3.1: **Normal plane:** All choices of a vector on the plane P, like $\boldsymbol{n}_1 \dots \boldsymbol{n}_3$ are normal to the line (see a). Figure b illustrates how a normal $\boldsymbol{n}_l$ for the line is chosen for a certain light source, depending on the light vector $\boldsymbol{v}$. The line normal $\boldsymbol{n}_l$ is the normalized vector between the intersection of the line with its normal plane P, and the projection of the light source on the plane P.

Chapter 5

Results

After the theory has been explained it is time to look at some results. We will examine performance and accuracy of MARK2 and MARK3, and compare them to standard Runge-Kutta methods. For MARK3 in all of these experiments the necessity of a fallback is detected by checking $(\boldsymbol{n} \cdot \boldsymbol{k}_2)/|\boldsymbol{k}_2| < 0.2$. The according fallback treatment is to half the step size and perform a regular Runge-Kutta order 3 method until the cell is left, as outlined in section 3.2.3.

This more practical chapter will be started by a convergence analysis, to verify our theoretical results, followed by benchmarks of the C++ implementation.

5.1 Convergence Analysis

To examine the global convergence one would normally progressively reduce the step size and examine the resulting error. Since both MARK2 and MARK3 choose their step size automatically depending on the mesh, we cannot reduce the step size directly, but instead have to refine the mesh. This will result in an increasingly small step size, although the reduction will not be completely steady. Since this mesh refinement is quite nontrivial to implement in the 3D C++ program, the convergence analysis was done with the 2D Matlab/Octave implementation.

The data used for calculating the trajectories is a circular flow, with its center $\boldsymbol{c}$ at $(0.5, 0.5)$ and a velocity, equal to the distance from the center. With a starting point of $(0.1, 0.5)$, and therefore a starting distance $d = 0.4$, an analytic solution can be calculated as

$$\begin{aligned} f(t) &= d \begin{pmatrix} -\cos(t) \\ \sin(t) \end{pmatrix} + \boldsymbol{c} \\ &= 0.4 \begin{pmatrix} -\cos(t) \\ \sin(t) \end{pmatrix} + \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} \end{aligned}$$

The mesh used is regular and covers the unit square. The coarsest mesh starts by dividing the domain into 4 intervals along the x and y-axis respectively, which corresponds to $4^2 \cdot 2 = 32$ triangles of a size of $1/32$. For the tests particle traces were calculated up to time $t = 20$, which corresponds roughly to 3.2 circles around the center.

The tables 5.1.1 and 5.1.2 show the resulting global errors, average step sizes, number of steps and overall convergence, for MARK2 and MARK3 respectively. The column convergence shows the factor by which the error is decreased when the number of intervals is doubled. It has to be mentioned that the average step size recorded does not exactly match 20/(intervals),

Intervals	global error	# of steps	average step size	convergence
4	$1.45441E^{-01}$	74	$2.71405E^{-1}$	-
8	$6.03030E^{-02}$	149	$1.34392E^{-1}$	2.41183
16	$1.91895E^{-02}$	287	$6.96978E^{-2}$	3.14249
32	$4.24938E^{-03}$	552	$3.62476E^{-2}$	4.51584
64	$1.06893E^{-03}$	1106	$1.80980E^{-2}$	3.97533
128	$2.43181E^{-04}$	2213	$9.04286E^{-3}$	4.39563
256	$6.44109E^{-05}$	4403	$4.54351E^{-3}$	3.77546

Table 5.1.1: **MARK2 error**

Intervals	global error	# of steps	# of fallbacks	# of steps in fallback	average stepsize	convergence
4	$6.25432E^{-03}$	74	7	9	$2.8340E^{-1}$	-
8	$1.03650E^{-02}$	147	0	0	$1.3793E^{-1}$	0.60340
16	$1.76292E^{-03}$	298	14	36	$6.7751E^{-2}$	5.87943
32	$8.16933E^{-05}$	552	14	22	$3.6398E^{-2}$	21.57983
64	$3.53247E^{-05}$	1113	14	35	$1.8019E^{-2}$	2.31263
128	$4.03001E^{-06}$	2292	52	132	$8.7391E^{-3}$	8.76542
256	$6.32162E^{-07}$	4502	92	229	$4.4456E^{-3}$	6.37496

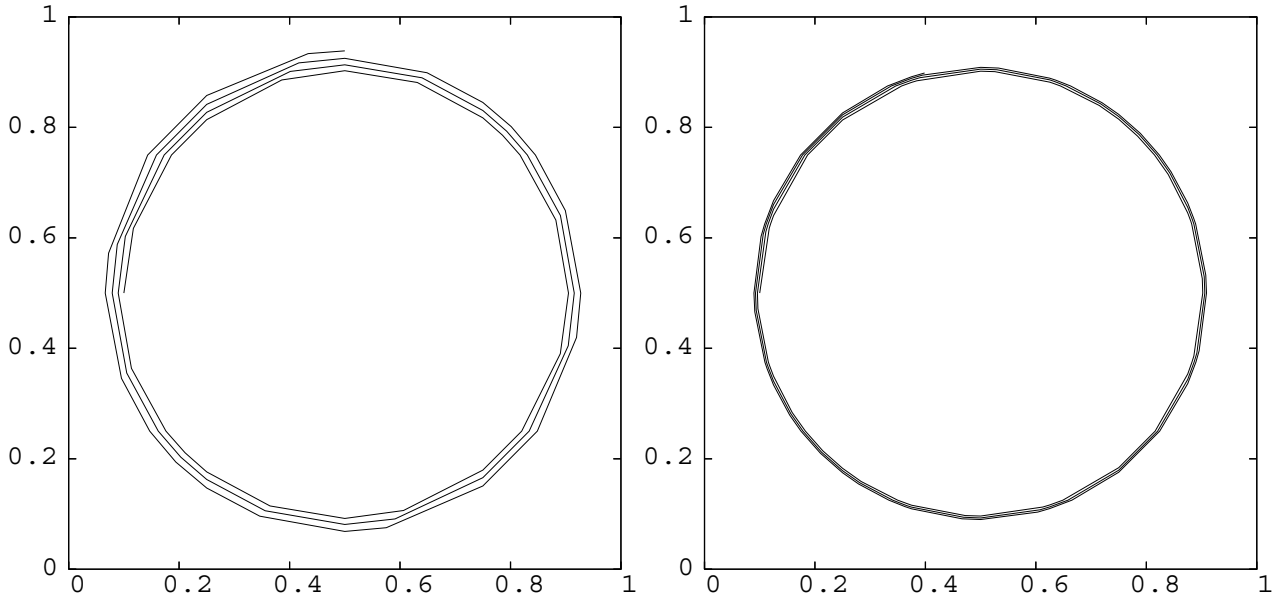
Table 5.1.2: **MARK3 error**

because both methods were implemented to stop iterating after a full step, which means they only stop after they hit a cell face. Therefore they always advance in time a little bit more than required. Additionally steps with stepsize zero (see section 4.1) were not counted for this calculation. For MARK3 the table also displays the number of fallbacks that were necessary.
In the figures 5.1.1 and 5.1.2 some of the results from the experiments are plotted. It is easy to see that already for the relatively coarse meshes the results "look" very good, which is a good sign, since these are visualization techniques after all. The evolution of the error during the trace has been plotted in figures 5.1.3 and 5.1.4. These plots are helpful in identifying any problematic or abnormal behavior of the algorithms, like the "negative" convergence of MARK3 in the step from 4 to 8 intervals (see below).

All in all it can be said that MARK2 performs very consistent and the convergence rate is roughly equal to the expected $\mathcal{O}(h^2)$, eg. when the number of intervals is doubled, the step size gets cut in half, and therefore the error is reduced by a factor of 4. The periodic behavior shown in the error plots is due to the data being traced. The accuracy of the method gets worse when the step size is big and probably also when q_2 becomes slightly degenerate. Since both of these parameters are dictated by the mesh and how it is traversed, all of these conditions repeat in each cycle.

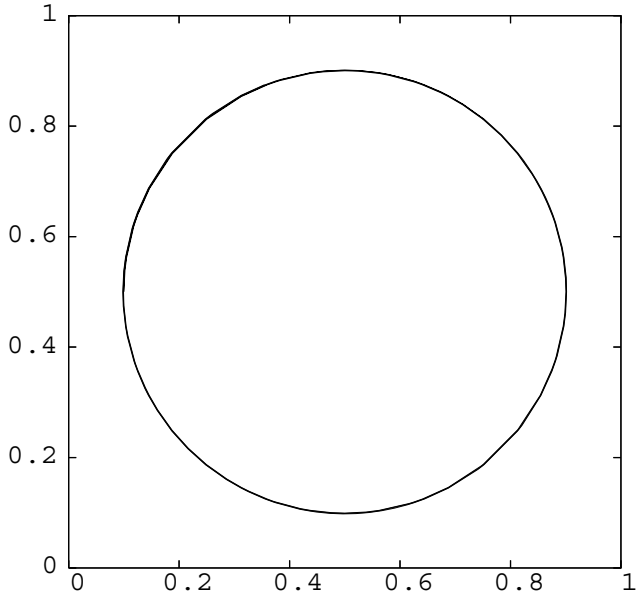
For MARK3 the results are a bit less satisfying. First it has to be mentioned the results for MARK3 vary for different choices of the constant C which governs the decision of whether a fallback is necessary (for these experiments the choice was $C = 0.2$). The number of fallbacks actually taken is a little bit erratic as can be seen in table 5.1.2 and depends not only on the total number of steps taken, but also on which path the particle actually takes through the mesh.

The most interesting numbers to look at for MARK3 are the plots of the error in fig-

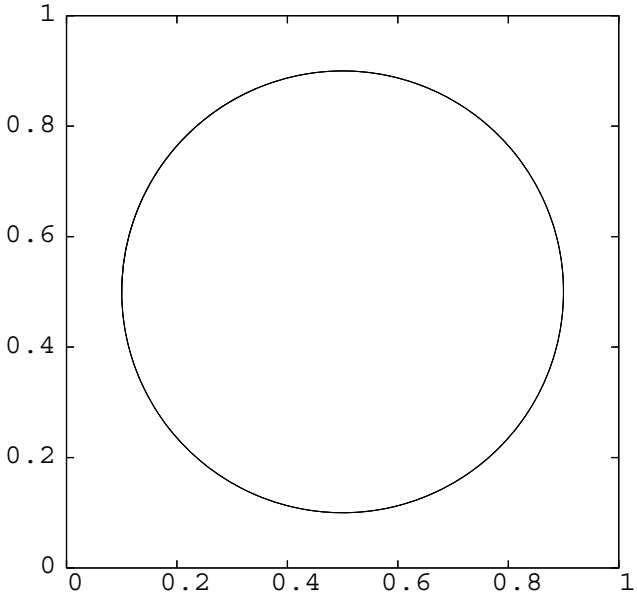


(a) 4 Intervals

(b) 8 Intervals

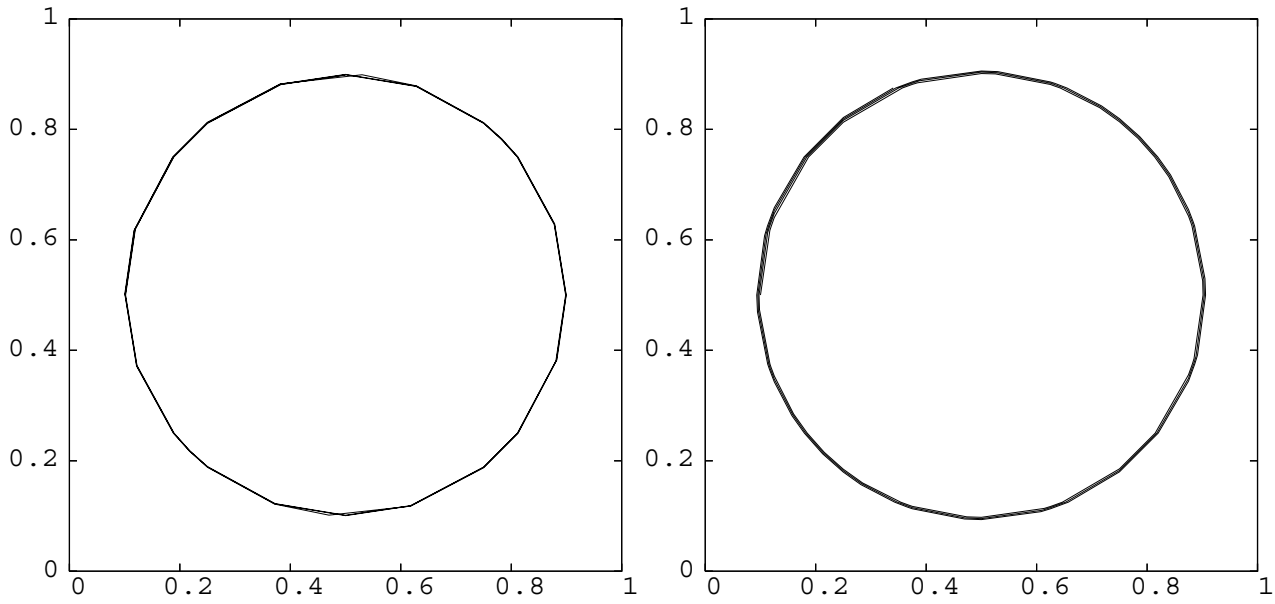


(c) 16 Intervals



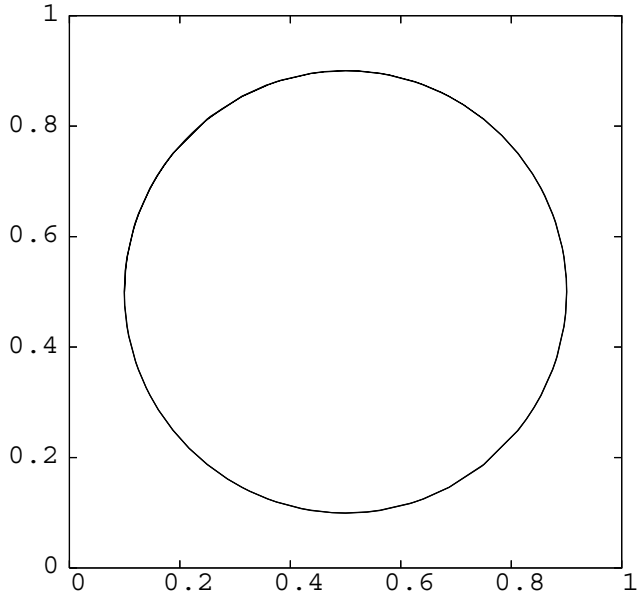
(d) 32 Intervals

Figure 5.1.1: **MARK2 trajectories**

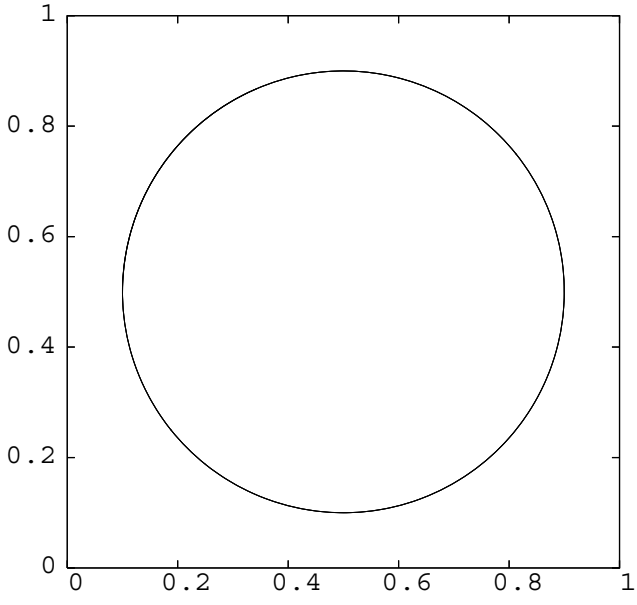


(a) 4 Intervals

(b) 8 Intervals

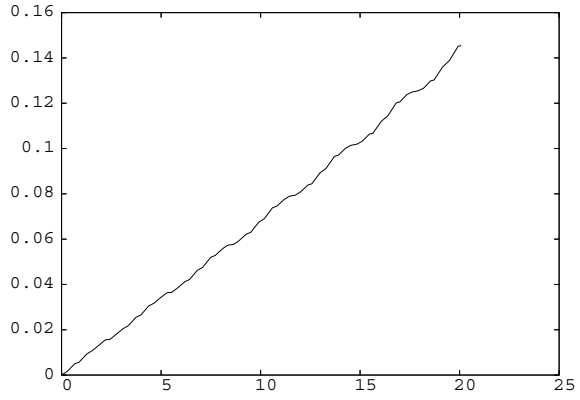


(c) 16 Intervals

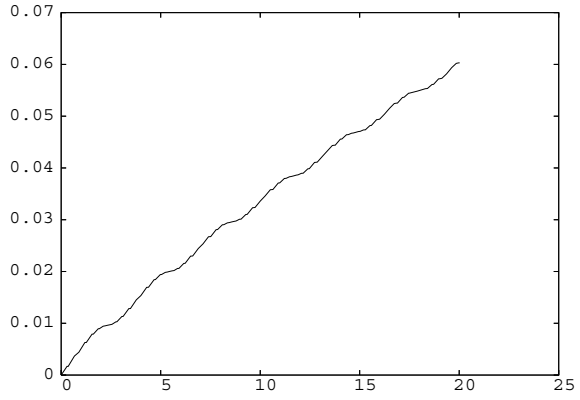


(d) 32 Intervals

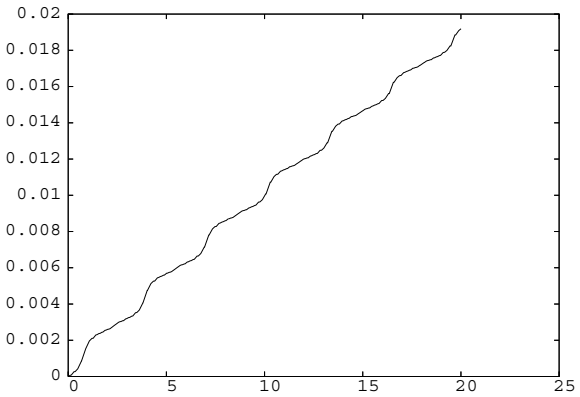
Figure 5.1.2: **MARK3 trajectories:**



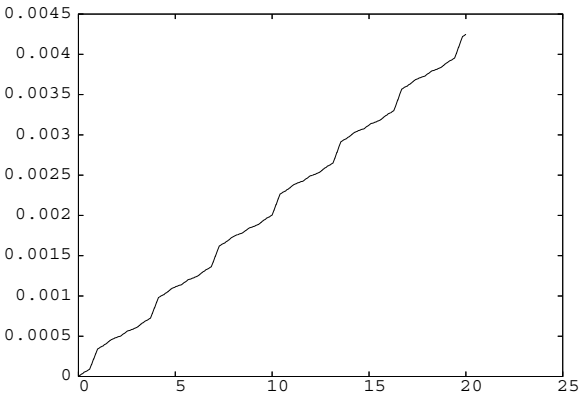
(a) 4 Intervals



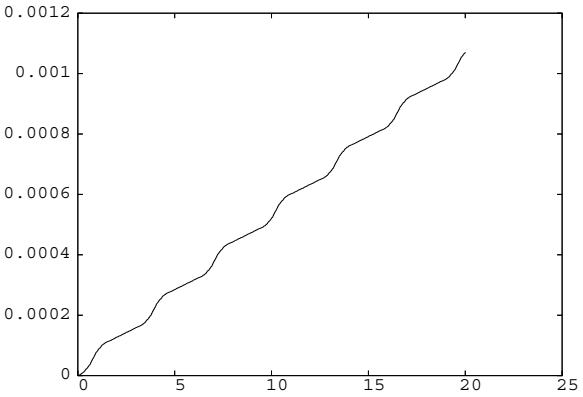
(b) 8 Intervals



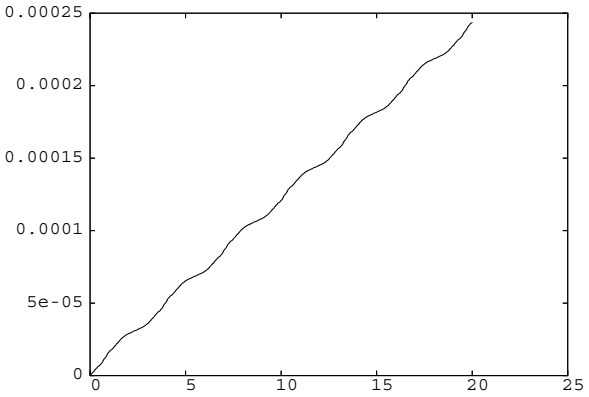
(c) 16 Intervals



(d) 32 Intervals

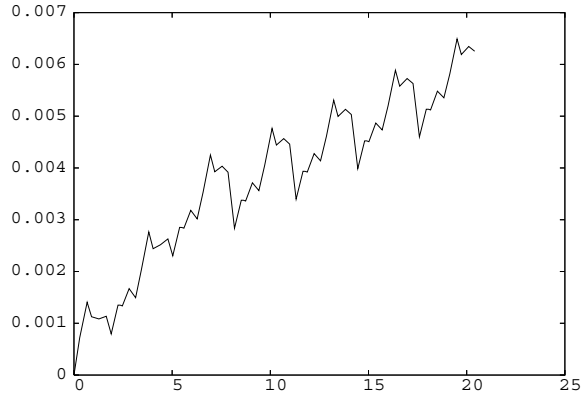


(e) 64 Intervals

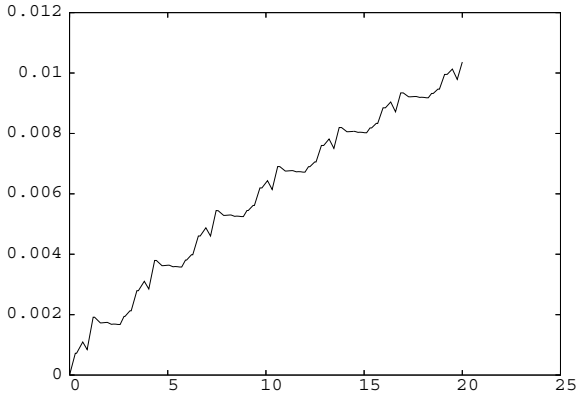


(f) 128 Intervals

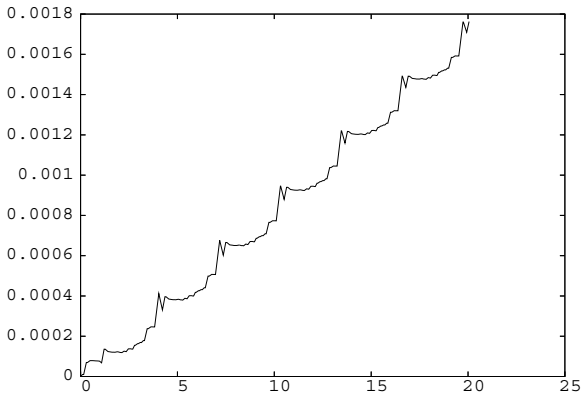
Figure 5.1.3: **MARK2 error plots**



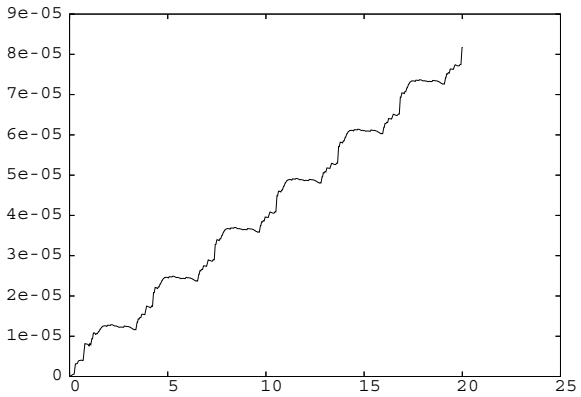
(a) 4 Intervals



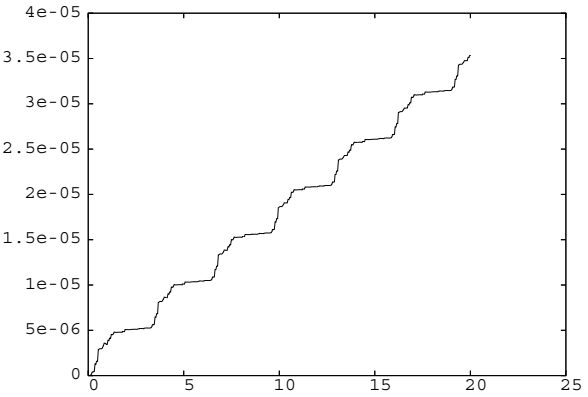
(b) 8 Intervals



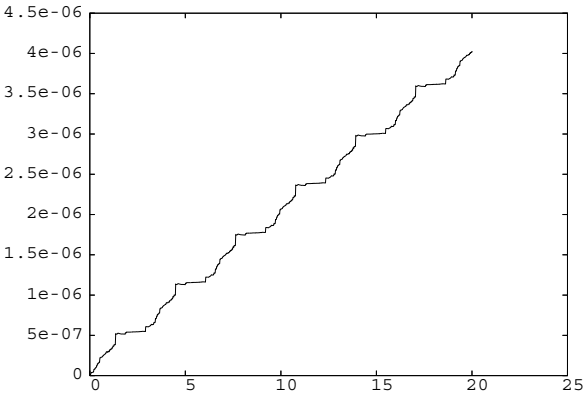
(c) 16 Intervals



(d) 32 Intervals



(e) 64 Intervals



(f) 128 Intervals

Figure 5.1.4: **MARK3 error plots**

Mesh	# of vertices	# of cells
Small synthetic dataset	4735	20020
Large synthetic dataset	18783	82656
Small sphere	1550	8193
Car dataset	89881	457874
Shuttle dataset	190584	1058775

Table 5.2.1: **Mesh sizes**

ure 5.1.4. In some of these plots it is clearly visible that the error takes leaps at certain points. These points coincide with the points where the angle α between the normal $\boldsymbol{n}$ and $\boldsymbol{k}_2$ is bad, but not bad enough to trigger the fallback. The problem is not quite as bad as it might look though. The error that the correction step makes, grows non-linearly with α getting smaller and it seems to have a higher constant factor than the error of the underlying Runge-Kutta method. Therefore as long as the error of the Euler step is dominating, a plot of the error will always be scaled so, that a similar stepping behavior is visible. As seen in the error plots with 4 and 8 intervals, there seem to be two sources of error with opposite "directions", vying for dominance. The sources are most likely the error of the normal Runge-Kutta method and that of the correction step. This behavior is probably also the explanation of the "negative" convergence of MARK3 when the number of intervals is increased from 4 to 8 (see table 5.1.2). The performance for 8 intervals is probably not that bad, which can be seen when comparing the absolute error with that of MARK2. Instead the performance for 4 intervals is so disproportionately good. A guess would be that for 4 intervals the errors for the correction step and the normal Runge-Kutta method, cancel each other out to a certain degree, which is also visible in the plots of the error. We can therefore constitute that although the convergence of MARK3 is not very steady it is superior to MARK2 and the absolute error made by the method backs this up.

5.2 Benchmarking

To get some measurements of the actual speed of the algorithms, we benchmarked the C++ code with several different datasets. The five datasets used consist of a small and a large mesh with synthetic data of a circular flow, a dataset of a simulated current perturbed by a small sphere, and two more realistic datasets of the airflow around a car and a shuttle. The small sphere, car and shuttle dataset were also used in [9]. All datasets are described in more detail below, in the sections 5.2.1 through 5.2.1. All of these meshes are tetrahedral and their respective sizes can be seen in table 5.2.1. The machine used to run the benchmarks is an Intel Pentium 4 with 2.4 GHz and 4 GB of main memory.

For all datasets a total of 25 trace lines are generated, so that the measured data like runtimes and stepsizes, gets somewhat averaged. Therefore all the data presented in the tables 5.2.2, is either averaged (stepsize and maximum stepsize) or summed (number of steps and runtime) over all 25 traces. The measured time consists only of the time taken for the actual generation of the traces, without the global cell search necessary to determine the starting points or anything else, like rendering time.

Since for some of the datasets the traces are rather short, the whole set of 25 traces had to be recomputed several times in order to produce meaningful runtimes. This accumulated

runtime is then averaged by the number of times the traces are being computed, to generate the average runtime for the computation of only one set of traces. This is the time denoted in table 5.2.2.

The images in figure 5.2.1 through 5.2.4 illustrate what exactly is being traced on the different datasets. In all of these images MARK2 was used for the creation of the particle traces.

For all local cell searches necessary, in the standard methods as well as in MARK3, raycasting was used (see section 1.3). The Runge-Kutta method of order 3 used in the benchmarks is the same that MARK3 is based on. Its parameters are therefore:

$$q_2 = 1 \qquad q_3 = 1/2 \qquad b_{31} = 1/4 \qquad b_{32} = 1/4$$
$$a_1 = 1/6 \qquad a_2 = 1/6 \qquad a_3 = 4/6$$

Heun's and the modified Euler's method are presented in section 2.3.

All non-adaptive methods are supplied with a stepsize h roughly equal to the average stepsize used by the adaptive methods, to make the result comparable. Nevertheless this practice can be questionable, as will be discussed in section 5.2.2.

5.2.1 Datasets

Small synthetic dataset

The synthetic data that was generated for this benchmark is a perfect whirl around the center of the domain.(see figure 5.2.1(a)). While this data is not very interesting, it allows the traces to be run indefinitely long without the particles leaving the domain. This is useful to generate longer runtimes in the benchmarks, which allow a more detailed comparison. The disadvantage of this dataset is that the cell sizes over the domain are very regular, which denies the adaptive methods any advantages they might gain through their stepsize control.

Large synthetic dataset

The data generated for this dataset is similar to that of the small synthetic dataset (see figure 5.2.1(b)).The only difference to the small synthetic dataset is that the resolution and therefore the number of vertices and cells is much higher.

Small sphere dataset

The small sphere dataset, consists of two spheres. The outer sphere is the boundary of the domain, the inner one is an obstacle to the flow simulated in the domain. The obstacle produces a disturbance in the simulated flow. The seedpoints for the traces were set so, that they go right through these disturbances (see figure 5.2.2). The central traces do not manage to go around the obstacle and therefore end on it. Since this happened independently of the particle tracing method used, it should not influence the results by much.

Car dataset

The car dataset depicts the airflow around a car (see figure 5.2.3). The surface model shown in the images is generated from the surface of the volume mesh. The mesh is very fine close to the surface of the car and gets coarser when moving away from it. This explains the large

discrepancy of the average and maximum stepsize recorded for the adaptive methods, for this dataset.

Shuttle dataset

Similarly to the car dataset, the shuttle dataset shows an airflow around an obstacle, in this case a space shuttle (see figure 5.2.4). The surface shown in the images is again extracted from the volume mesh. Just as this was the case for the car dataset, the mesh has a higher resolution close to the shuttle, which results in a high variance in the stepsizes taken by the adaptive methods.

5.2.2 Discussion

At first glance the benchmark results for MARK2 do not look too good. Several benchmarks show a worse runtime for MARK2 than for the standard methods. There are several reasons for these results.

First it is arguable whether setting the stepsize of the standard methods to the average stepsize of the adaptive methods is fair. Depending on the dataset and mesh used the number of steps actually taken varies quite a bit. If the number of steps is taken into account MARK2 seems to be about as fast as Heun's method. The reason for this is that for the average stepsize determined by the adaptive methods, the cell search for the standard methods usually will never have to search more than two cells. This also explains why Heun's method is faster than the modified Euler's method. Heun operates with a $q_2 = 1/2$ instead of the modified Euler's $q_2 = 1$. From this follows that the first point Heun's method has to search for is much closer to the starting point and therefore much more likely to be in the same cell. This in turn makes the cell search for this step cheaper on average.

Hence we can conclude that for the first cell search Heun's method often only needs to search the current cell. For the second step it will usually not have to search more then the current and a neighboring cell. MARK2 on the other hand has to intersect a linear and a quadratic step with the cell boundaries. While the intersection for the linear step can be determined with about the same effort that one step of cell searching would need, the computation of the intersection of the quadratic step should be more expansive. MARK2 thus spends about as much time intersecting the cell boundaries as Heun's method spends cell searching.

What remains as the main advantage of MARK2 is its ability to automatically adapt its stepsize to the current cell size. For a standard method, the average step size determined by the adaptive methods is more or less optimal in terms of accuracy versus runtime. If the stepsize is chosen larger then the standard methods spend much more time cell searching per step. Hence even though the number of steps goes down, the runtime does not decrease by the same factor. In some of the benchmarks it was even possible to observe that the runtime first went slightly up, when the stepsize was increased. For a smaller stepsize than the one used in these benchmarks, accuracy will only increase significantly if the accuracy of the data provided by $\boldsymbol{f}$ is higher than the accuracy of the particle tracing method. If this is not the case than the additional steps employed to cross a cell are wasted. Furthermore in a real application this optimal stepsize is not known, but has to be supplied by the user. MARK2 does not need this user input. In the case of a trace that passes through very non-uniformly sized cells, it is even impossible to supply just one such step size, and a standard method

will either waste a lot of effort by tracing the large cells with too many steps, or it will lose accuracy by traversing too many small cells in one step.

All these advantages certainly also apply to MARK3. But MARK3 also performs much better than the standard Runge-Kutta order 3 method. In fact it performs only slightly slower than MARK2 and the standard order 2 methods. This is not very surprising as MARK3 only has to calculate the intersections of two linear steps with the cell boundaries, in comparison to the intersections of a linear and a quadratic step that MARK2 has to calculate. The downside of MARK3 is that its convergence is rather erratic. The traces for the small synthetic dataset even showed a total visible error larger than that of the order 2 methods. This suggests that MARK3 is a very fast method, but the behavior of its error in problem cases has to be further researched, to make it more reliable.

Small synthetic dataset

Method	# steps	average step size	maximum step size	time in seconds
MARK2	53996	$1.855527E^{-02}$	$8.471258E^{-02}$	0.34
MARK3	56236	$1.785381E^{-02}$	$9.049525E^{-02}$	0.38
Heun	55575	$1.800000E^{-02}$	$1.800000E^{-02}$	0.34
Modified Euler	55575	$1.800000E^{-02}$	$1.800000E^{-02}$	0.36
Runge-Kutta 3	55575	$1.800000E^{-02}$	$1.800000E^{-02}$	0.61

Large synthetic dataset

Method	# steps	average step size	maximum step size	time in seconds
MARK2	92087	$1.086763E^{-02}$	$5.565922E^{-02}$	0.59
MARK3	96644	$1.036814E^{-02}$	$5.978267E^{-02}$	0.68
Heun	100000	$1.000000E^{-02}$	$1.000000E^{-02}$	0.61
Modified Euler	100000	$1.000000E^{-02}$	$1.000000E^{-02}$	0.64
Runge-Kutta 3	100000	$1.000000E^{-02}$	$1.000000E^{-02}$	1.06

Small sphere dataset

Method	# steps	average step size	maximum step size	time in seconds
MARK2	1419	$2.679955E^{-01}$	$1.731418E^{+00}$	0.0098
MARK3	1503	$2.534700E^{-01}$	$1.731639E^{+00}$	0.0110
Heun	1420	$2.600000E^{-01}$	$2.600000E^{-01}$	0.0095
Modified Euler	1420	$2.600000E^{-01}$	$2.600000E^{-01}$	0.0101
Runge-Kutta 3	1420	$2.600000E^{-01}$	$2.600000E^{-01}$	0.0160

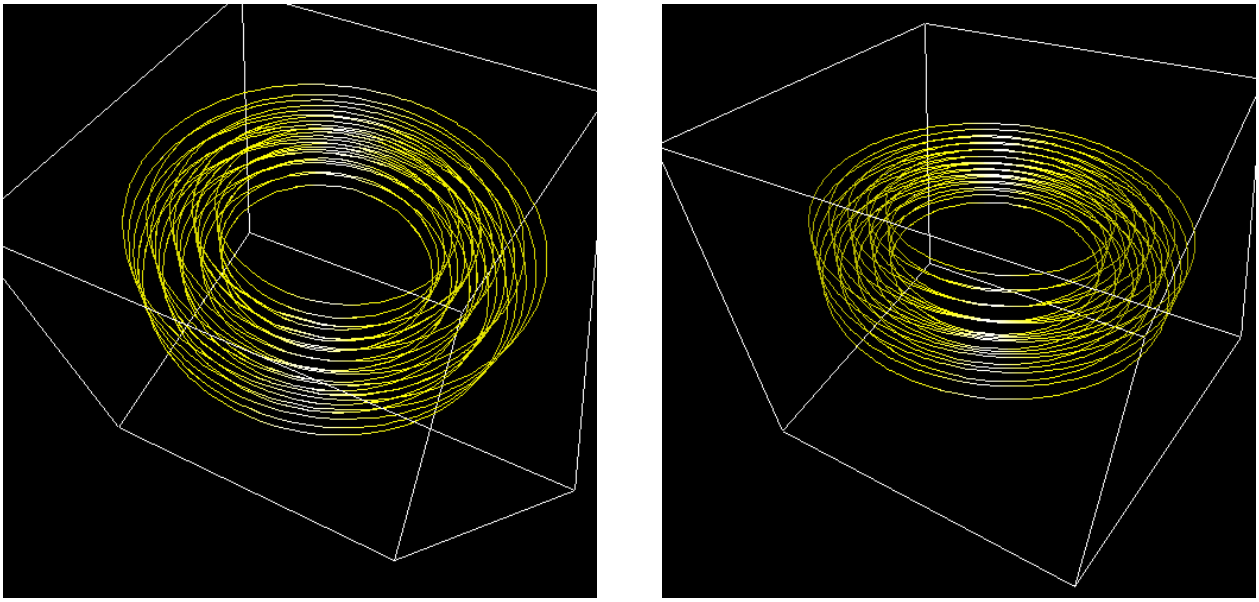
Car dataset

Method	# steps	average step size	maximum step size	time in seconds
MARK2	3724	$1.212791E^{+01}$	$1.893675E^{+02}$	0.027
MARK3	3887	$1.158405E^{+01}$	$1.892938E^{+02}$	0.030
Heun	3350	$1.200000E^{+01}$	$1.200000E^{+01}$	0.026
Modified Euler	3350	$1.200000E^{+01}$	$1.200000E^{+01}$	0.027
Runge-Kutta 3	3350	$1.200000E^{+01}$	$1.200000E^{+01}$	0.043

Shuttle dataset

Method	# steps	average step size	maximum step size	time in seconds
MARK2	5427	$8.861345E^{+00}$	$8.743934E^{+01}$	0.040
MARK3	5702	$8.418706E^{+00}$	$8.735314E^{+01}$	0.045
Heun	4550	$8.800000E^{+00}$	$8.800000E^{+00}$	0.035
Modified Euler	4550	$8.800000E^{+00}$	$8.800000E^{+00}$	0.038
Runge-Kutta 3	4550	$8.800000E^{+00}$	$8.800000E^{+00}$	0.058

Table 5.2.2: **Benchmark results**



(a) Small synthetic dataset (b) Large synthetic dataset

Figure 5.2.1: **Images of the synthetic datasets**

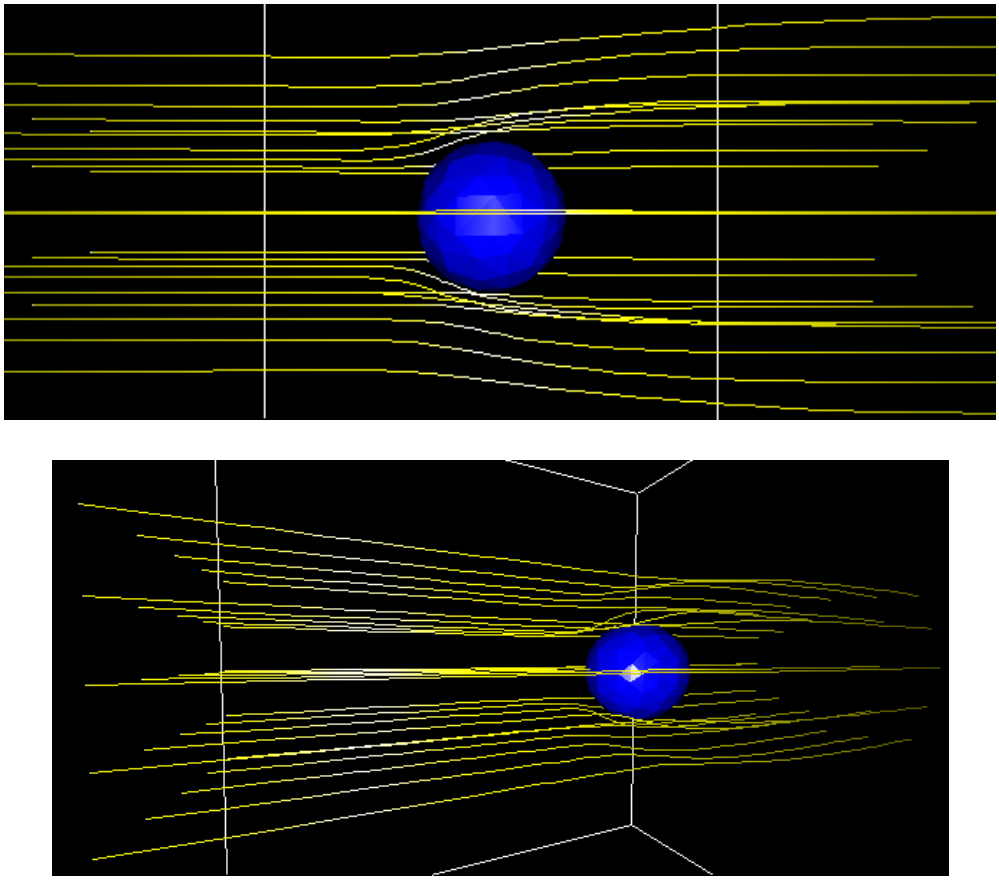


Figure 5.2.2: **Images of the small sphere dataset**

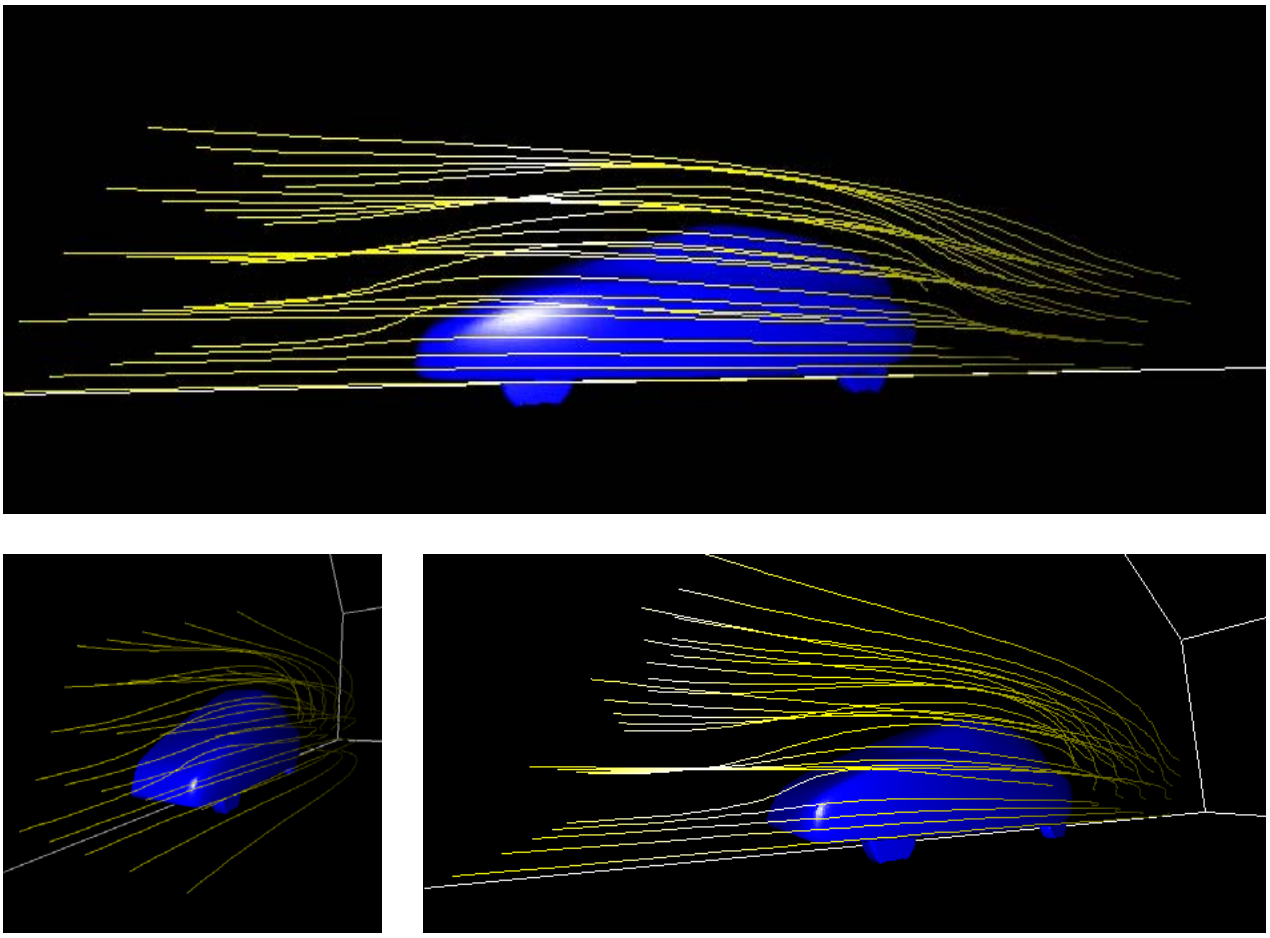


Figure 5.2.3: **Images of the car dataset**

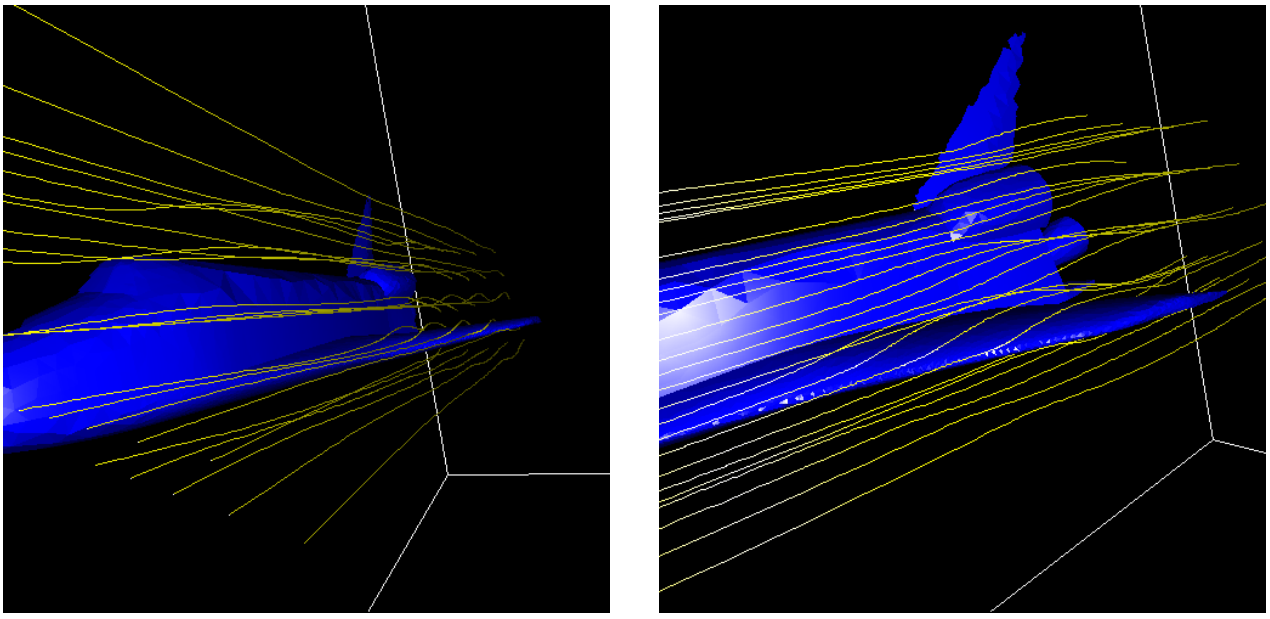


Figure 5.2.4: **Images of the shuttle dataset**

Chapter 6

Conclusions and Outlook

Two new methods for particle tracing in unstructured meshes have been presented. Their ability to adapt their step sizes to the current cell size, eliminates the need for the user to explicitly specify a step size and makes the local cell search after each step unnecessary. While MARK3 shows a very promising performance, it has to be researched further to make it more stable. MARK2 on the other hand performs not much faster than the standard methods, but in turn is very stable and has all the advantages of a method with automatic step size control (see section 5.2.2).

The major shortcoming of both methods is their limited accuracy. For a lot of applications like the visualization of the output of finite element methods with linear or quadratic basis functions they should still be sufficient. Nevertheless it would be rewarding to try to increase the order, and therefore the accuracy of these methods. Some ideas how this could be done were already discussed in section 3.3. The other approach to increase accuracy, to decrease the step size and therefore to take several steps per element is also far less trivial than it would appear, because one would have to check each step whether the particle left the current cell. Furthermore something like this can only be done if the step size that would cross the cell in one step is already known. Therefore it might also be interesting to try variations of MARK2 or MARK3 for an approach like this.

As a last point, it remains to be investigated what influence these methods would have when used together with one of the visualization techniques that build on top of particle tracing.

Appendix A

UGLI - Unstructured Grid Library

UGLI is a library to manage unstructured meshes. It has been written by Ben Bergen (`http://www10.informatik.uni-erlangen.de/~ben`) in the course of his work on a doctorate degree. Although it was only used in serial for this work, UGLI has the capability to be used in a parallel environment. The main advantage of UGLI is that it is simple. A lot of other mesh libraries provide almost an overkill of features, which often makes them hard to use. Another advantage, especially for this work is that UGLI explicitly saves all the connectivity present in a mesh. This means that every element in the mesh saves references to its vertices and faces, every face references to its vertices and the elements it is part of and every vertex references to all the faces and elements it is part of. It is therefore easy and fast to determine the neighbor of an element across a certain face, which is an important feature for MARK2 and MARK3. The disadvantage of explicitly storing that much connectivity is the memory consumption.

Additional data can be associated with each entity in the mesh by its unique ID. It is also possible to store two three dimensional vectors and two scalars with each vertex, which was sufficient for the storage of the flow fields needed for this work.

UGLI can obtained from `http://www10.informatik.uni-erlangen.de/~ben/UGLi/UGLi/doc/html/index.html` .

Acknowledgements

I would like to thank Prof. Dr. Ulrich Rüde, Prof. Dr. G. Greiner, Ben Bergen and Frank Reck for their help and support. Thanks also to Alexander Hausner and Nils Thürey for proofreading and corrections.

List of Figures

List of Tables

Bibliography

[1] P. Buning. Numerical algorithms in cfd post-processing, computer graphics and flow visualization in computational fluid dynamics. *von Karman Institute for Fluid Dynamics Lecture Series*, 1989.

[2] J.C. Butcher. Coefficients for the study of runge-kutta integration processes. *Journal of the Australian Mathematical Society*, 3:185–201, 1963.

[3] B. Cabral and L. Leedom. Imaging vector fields using line integral convolution. In J.T. Kajiya, editor, *Computer Graphics Proceedings*, volume 27, pages 263–270, Los Angeles, California, August 1993. ACM SIGGRAPH, Addison-Wesly Publishing Company, Inc.

[4] H. Grabmüller. Numerik 1. Lecture Script, 2001. `http://www.am.uni-erlangen.de/am1/members/grabmue/lect.notes.html`.

[5] E. Hairer, S.P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations*, volume 1 Nonstiff Problems. Springer, second revised edition, 2000.

[6] G. Li, U.D. Bordoloi, and H. Shen. Chameleon: An interactive texture-based rendering framework for visualizing three-dimensional vector fields. Oct 2003.

[7] G. M. Nielson, H. Hagen, and H. Müller. *Scientific Visualization*. IEEE Computer Society, Los Alamitos, California, 1997.

[8] J. Oliver. A curiosity of low-order explicit runge-kutta methods. *Mathematics of Computation*, 29(132):1032–1036, October 1975.

[9] F. Reck and G. Greiner. Fast and accurate integration of vector fields in unstructured grids. *it+ti, Informationstechnik und Technische Informatik*, 30(6):331–338, December 2002.

[10] H. R. Schwarz. *Numerische Mathematik*. Teubner, 3 edition, 1993.

[11] J. Stoer and R. Bulirsch. *Numerische Mathematik*, volume 2. Springer, 3 edition, 1990.

[12] W. Walter. *Analysis*, volume 1. Springer, 6 edition, 2001.

[13] M. Zöckler, D. Stalling, and H.C. Hege. Interactive visualization of 3d-vector fields using illuminated stream lines. In *Proceedings of the 7th conference on Visualization*, pages 107–113. IEEE Computer Society Press, 1996.