DIPLOMARBEIT

Implementation of Different Semi-Lagrangian Convection Schemes for the Simulation of Room Air Flow

Ausgeführt zum Zweck der Erlangung des akademischen Grades eines
Dipl.- Ing. (FH) für Computersimulation
am Fachhochschul-Diplomstudiengang Computersimulation St. Pölten

unter der Leitung von

Dipl.- Ing. Dr. Stefan Barp
und Dipl.- Ing. Dr. Christian Harlander

ausgeführt von

Oliver Dunkl
Matr.Nr. 0210095007

St. Pölten, September 2006

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Implementierung von verschiedenen Algorithmen für die online Strömungsberechnung. Online Strömungsberechnungen sind Berechnungen von Strömungen die während der Simulationsdurchführung eine Änderung der Umgebung zulassen. Zur Berechnung dieser Strömungen werden unterschiedliche Semi-Lagrange Verfahren in einem Computational Fluid Dynamic (CFD) Code verwendet. Unter Computational Fluid Dynamic versteht man Berechnungen von Strömungen mittels der Unterstützung von Computern.

Im ersten Teil der Arbeit werden verschiedene Interpolationsarten auf Genauigkeit und Rechenzeit verglichen. Besondere Sorgfalt wird hierbei auf die lineare Interpolation, die Polynominterpolation und die Spline Interpolation genommen, da diese Interpolationsarten am öftesten verwendet werden.

Der zweite Teil widmet sich der Implementierung dieser Interpolationsarten. Es wird der theoretische Hintergrund dieser Algorithmen gezeigt und wie sie in einer Programmiersprache implementiert werden können. Außerdem wird eine Möglichkeit beschrieben wie diese Algorithmen in einen bestehenden CFD Code implementiert werden können.

Im letzten Teil dieser Arbeit werden verschiedene ausgewählte Beispiele anhand eines eigens entwickelten Simulators untersucht, die zuvor analytisch berechnet wurden. Anhand dieses Simulators können die Ergebnisse der ausgewählten Interpolationsarten gut visualisiert und verglichen werden.

Abstract

This thesis investigates the implementation of different algorithms for the real time fluid flow. Real time fluid flow means that it is possible to change the environment of the simulation during the simulation. This fluid flow should be calculated with a Computational Fluid Dynamic (CFD) code by different semi-Lagrangian schemes. Computational Fluid Dynamic means the calculation of fluid flow with the support of computers.

In the first part of this work different interpolation schemes are determined for their accuracy and time of calculation. Special emphasis and care is taken to analyse the linear interpolation, the polynomial interpolation and the spline interpolation scheme because that are the most important interpolation schemes.

The second part of this work shows the implementation of the above mentioned interpolation schemes. The theoretical background of the interpolation schemes are shown as well as implementation in an arbitrary programming language. Additionally there is shown the possibility how to implement such an algorithm in an established CFD code.

In the last part of this work we discuss some selected examples, which are analytically calculated. These examples are performed with a specifically developed simulator. With that simulator the results of these selected interpolation schemes are compared and visualized.

Acknowledgement

First of all I would like to express my thanks to my primary person who looks after my thesis Dipl.-Ing. Dr. Stefan Barp, who provided me with a lot of feedback to my research occupation at the ETH Zurich, and with interesting discussions about that work. My deepest appreciations are due to AFC and the research group Air & Climate at the ETH in Zurich for their perfect working environment and their support for my thesis.

I would like to extend a special thank to my secondary person who looks after my thesis Dipl.-Ing. Dr. Christian Harlander who gaves me his stimulus in the subject matter of the simulation technique especially in the discretization methods and for the help with words and deeds.

I am very grateful to Saikat Roy for his corrections of my work. I owe gratitude to number of my colleagues at the research group Air & Climate at the ETH Zurich for the interesting discussions, their support and cooperation especially Daniel Rusch, Yves Brise, Basil Weber and Farid Dshabarow.

After that I would thank Dr. Alfred Moser and Daniel Rusch for their diversified ventures. They made a lot of fun and that was a good opportunity to get to know the other colleagues.

Finally I wish to thank my parents for their continuous support during all the years of my study.

Contents

Chapter 1

Introduction

In the world of fluid flows are important, e.g. rising smoke, clouds and mist in the flow of rivers and oceans. This fact is the reason why some physical scientists developed mathematical models for most fluid flows occurring in the nature. In the time of Euler, Navier and Stokes these developments have led to the so-called Navier-Stokes equations. In (1.1) the Navier-Stokes equation describes the incompressible fluid flow.

$$\frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla)\vec{u} = -\nabla p + \nu \nabla^2 \vec{u} + \rho \vec{f} \tag{1.1}$$

In (1.1) $\vec{u}$ is a vector representing the velocity of an infinitesimal element of mass at a point in 3D space, p is the scalar pressure at the same point, ρ is the mass density at the point and is assumed constant throughout the medium, ν is the viscosity of the medium, and $\vec{f}$ is a vector acceleration due to some constant external force on the infinitesimal element, usually taken to be gravity.

In general, these algorithms strive for accuracy, are quite complex and time consuming. Additionally, there are no analytical solutions available in many cases. Therefore, since the computers have revolutionized our life, a lot of efforts were made to find approaches to solve these equations numerically.

Since computers became faster and faster over the last few years, programmers and computer graphic designers took the challenge of developing numerical algorithms for real time fluid effects for the computer games industry. These computer games provide users a plausible virtual world, which includs fluid-like effects. For developing such computer games it is important both that the simulation looks convincing and is fast enough for the familiar home computers. For that reason the numerical algorithms for solving the Navier-Stokes equation were taken once again and were optimized for speed by Jos Stam [Sta03].

1.1 Introduction to real time fluid flow

Real time fluid flows are so called by the computer game industry because of calculating and presenting some fluid-like effects at the same time running a sequence in the game. The computer game industry applied some special numerical algorithm for that fluid-like effects in the game, mainly using the algorithms of Jos Stam [Sta03] which are based on the physical equations of fluid flow, namely the Navier-Stokes equations.

Fig. 1.1 shows the conventional process needed to solve such CFD problems. CFD is an abbreviation for Computational Fluid Dynamics. Many of the commercial and non-commercial CFD products take this course for solving CFD problems. These products are for example: *Fluent* [Inc06b], *CFX* [Inc06a] for the commercial products and *Gerris* [Pop06] and *OpenFOAM* [Ltd06] for OpenSource products. In contrast to the

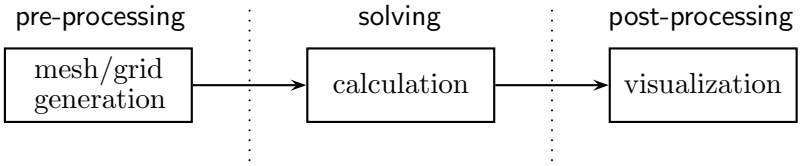


Fig. 1.1: Conventional process of a CFD calculation

conventional approach in Fig. 1.1, the basic approach of Real Time Fluid Flows is that the calculation and the visualization are combined in one simulation step which is shown in Fig. 1.2. The biggest advantage of the Real Time Fluid Flow is to visualize the results of a CFD simulation at the same time at which the calculation of that simulation is running. An advantage of them, is that the effective time of the simulation is less than the simulation time for the conventional approach. Of course, there isn't an advantage with no disadvantage. The consequence of decreasing the effective simulation time is that the accuracy suffers.

1.1.1 A fluid in a box

For the numerical solution of the Navier-Stokes equations we need a finite representation for our fluids. The useful approach is to subdivide the space as in Fig. 1.3 into a finite region of space with identical cells and sample the fluid at each cell's center [Sta03]. In this work we will only describe a fluid in two dimensions, but this approach is not restricted to two dimensions. To use this approach for three dimensions is straightforward.

Therefore the fluid is modeled on a square grid like the one shown in Fig. 1.3. There is an additional layer of grid around the fluid domain to simplify the treatment of the boundaries. The velocity and all scalar values are defined in the center of each cell. In

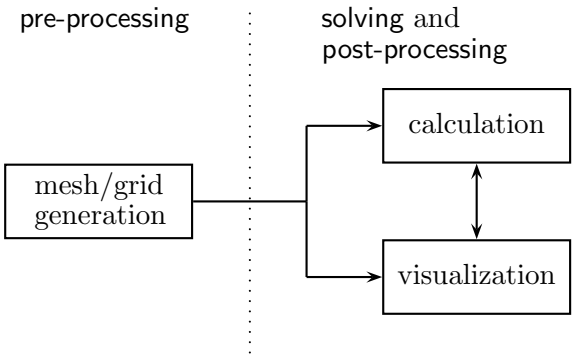


Fig. 1.2: Basic approach of Real Time Fluid Flows

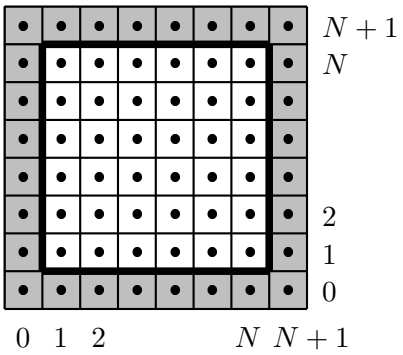


Fig. 1.3: Computational grids which are described in that thesis. The grid contains an extra layer of cells to account for the boundary conditions.

practice two arrays are allocated for the velocity with a size of `size=(N+2)*(N+2)`.

```
static float u[size], v[size], u_prev[size], v_prev[size];
```

Single dimensional arrays are preferred over double ones for efficiency purposes [Sta03]. These arrays will be referenced by the following macro.

```
#define IX(i,j) ((i)+(N+2)*(j))
```

The cell `(i,j)` of the horizontal component of the velocity is given by the entry `u[IX(i,j)]`. It is also assumed that the physical length of each side of the grid is 1 so that the grid spacing is given by `h=1/N` [Sta03].

1.1.2 Basic approach to calculate a scalar value

In this thesis the advective transport of a scalar field by the velocity field is investigated. The basic procedure to solve the advection step is shown in Fig. 1.4. In Fig. 1.4(a) the velocity field is shown. For a better oversight only one velocity vector is displayed in the grid.

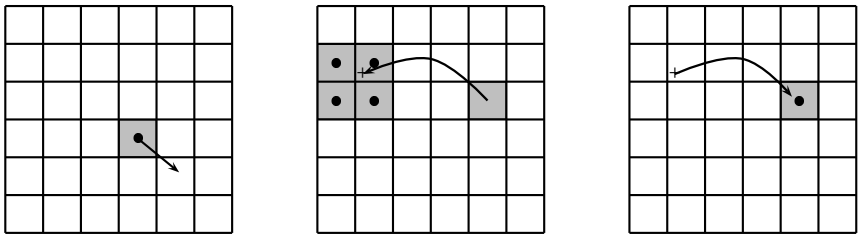


Fig. 1.4: Basic idea behind the advection step. The searched point (a) will be traced back in time through the velocity field, interpolate the point (b) and write it back to the searched point (c).

In this thesis the density is taken as a scalar value. The basic idea behind the advection step is instead of moving the cell centers forward in time through the velocity field 1.4(a), we look for the particles which end up exactly at the cell centers by tracing backwards in time from the cell centers, which is shown in Fig. 1.4(b) and Fig. 1.4(c). The magnitude of the scalar value, that these particles carry is simply obtained by interpolating the scalar value at its starting location from the closest neighbours.

1.2 Presentation of the problem

The approach of real time fluid flows is also interesting for CFD calculations and visualizations i.e. for room air flow simulations. If these algorithms of real time fluid flow are used in CFD calculations an important fact is the accuracy. The accuracy has to be as good as possible for solving such CFD problems.

In this case the accuracy of the real time fluid flows will be reduced to the time discretization. If explicit numerical procedures are used, only small time steps can be applied. This is because of the criterion of the stability of the convection term. This causes huge computing time for simulating long time periods. For this reason there are numerical procedures for the weather prediction which permit much higher time steps for the simulations. These procedures are called **semi-Lagrangian convection schemes**.

The following points are investigated in this thesis:

1. *Numerical accuracy of the schemes.* How accurate are the schemes compared to analytical results. First the accuracy of the interpolations are investigated with respect to the analytical results of a mathematical function. Second the semi-Lagrangian convection schemes are investigated with analytical results from other papers, and third the combination of the interpolation and the semi-Lagrangian schemes are investigated.

2. *Calculation time of the schemes.* An important factor of such schemes is the calculation time of the procedures. We investigate the calculation time with their floating point of operations (FLOPS) of each procedure.

3. *Finding the optimal procedure for the interpolation and the semi-Lagrangian schemes.* Establishing point 1 and 2, gives an optimal procedure for the schemes. It is possible that the scheme which is included in the solver is already the optimal scheme.

4. *Implementation of the schemes.* How easy is it to implement the algorithms in an existing real time CFD solver.

In this work the accuracy of the interpolation schemes is presented along with the implementation in a small real time fluid solver, and second the semi-Lagrangian convection scheme is presented and then the implementation of the schemes are described.

1.3 Motivation

Today there are lot of products which follow the conventional solution of CFD solvers, showed in Fig. 1.1. To get a broad overview of the results of some CFD problems it is nice to simulate and present the results quickly and also it will be nice to intervene to the simulation just by calculating.

After the development of the real time fluid flow for Games [Sta03] and the previous work of Jos Stam it is possible to implement such methods in a scientific CFD solver which calculates the problems and present the results at the same time. In addition, interaction with the real time CFD solver can also be permitted during the simulation, which means i.e. one can move some objects during the simulation run and see the result simultaneously.

In contrast to the computer games where the accuracy of the calculation is not so important but only the visualization where the results look like fluid-effects, for the CFD calculations it is important that the accuracy is as good as possible.

For this reason this thesis investigates special methods of the numerical algorithms for the weather prediction which are also important for the room air flow. The numerical weather prediction has been developed as a procedure which is stable and relatively accurate for long time steps. This is also important for simulating room air flow.

Chapter 2

Introduction to different interpolation schemes

Sometimes there is only a set of points $x_1, x_2, \ldots, x_N$ (i.e. points from a measuring experiment) given from a function $f(x)$ and we don't know the analytical expression for $f(x)$. To get points between these given points, you can fit a curve between these data points or find a function which is closely to these data points. This is called *curve fitting* and the interpolation is a special case of curve fitting, in which the function has to go exactly through the data points. There are a few of different interpolation schemes which are more or less accurate.

2.1 Presentation of the problem of the interpolation

The idea how to implement interpolation schemes in a Real Time CFD solver is that in every center of a grid cell a scalar value and the value of the velocity vector is stored. (2.1) and Fig. 2.1 shows that the point $P_{x,y}$ at which the scalar value will be interpolated has to be between its four neighboring grid points. However, it is also important to know the values between these fixed grid points ($P_{i,j}$).

$$\begin{aligned} P_{i,j} \leq x \leq P_{i+1,j} \quad & \text{and} \\ P_{i,j} \leq y \leq P_{i,j+1} \quad & \text{for} \;\; i,j = 1,..,N-1 \end{aligned} \tag{2.1}$$

It is important that we define in which arrangement that points appears. We assume that the points are arranged counterclockwise.

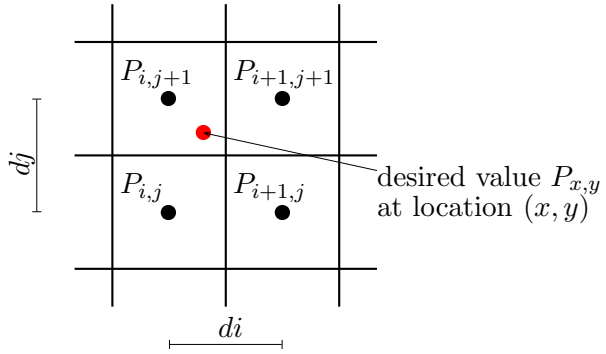


Fig. 2.1: Four neighboring points for a 2D interpolation

$$\begin{aligned} P_1 &= P_{i,j} \\ P_2 &= P_{i+1,j} \\ P_3 &= P_{i+1,j+1} \\ P_4 &= P_{i,j+1} \end{aligned} \tag{2.2}$$

There are some interpolation schemes which will be used to interpolate the searched values at point $P_{x,y}$. A few interpolation schemes will be described in the next sections.

2.2 Linear interpolation

The easiest way to interpolate some measured data is the piecewise linear interpolation. It is the easiest interpolation method because only two data points are required. The biggest drawback of this interpolation scheme is that you need a lot of data points if you want to get a very accurate interpolation there. The linear interpolation works by drawing a straight line between these two neighboring data points and returns an appropriate point along that line.

2.2.1 Fundamentals of one-dimensional linear interpolation

For the better understanding the linear interpolation scheme is described in one dimension. One dimensional linear interpolation means that there is one s-value for a given x-value which you can see in Fig. 2.2. The linear interpolation in 1D could be reduced

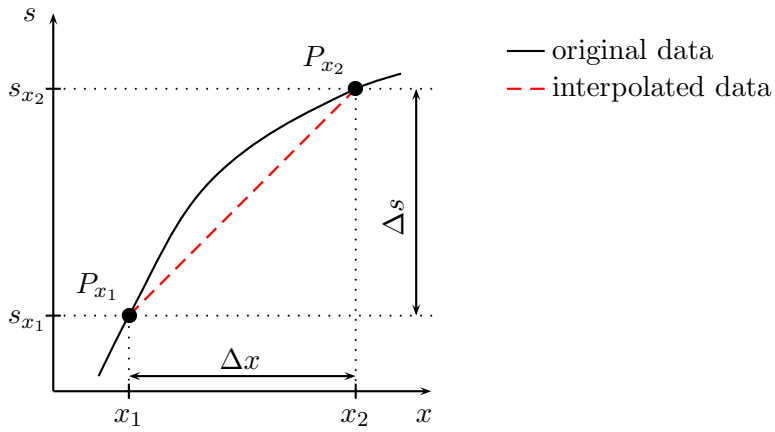


Fig. 2.2: Principle of 1D linear interpolation

to the calculation of the slope of a right-angled triangle. With that equations in (2.3) there could be solved the linear interpolation between the points P_{x_1,y_1} and P_{x_2,y_2}.

$$\begin{aligned} s &= k \cdot x + d \\ k &= \frac{\Delta s}{\Delta x} = \frac{s_{x_2} - s_{x_1}}{x_2 - x_1} \\ d &= s_{x_1} \end{aligned} \tag{2.3}$$

Linear interpolation is shown in (2.4). Each x-value gives an interpolated s-value back.

$$s = \frac{s_{x_2} - s_{x_1}}{x_2 - x_1} \cdot (x + x_1) + s_{x_1} \tag{2.4}$$

For the real time CFD solver which will be described in this thesis, the linear interpolation in one dimension is not so interesting as the interpolations in multiple dimensions. For that reason the next section will describe the linear interpolation in more than one dimension.

2.2.2 The linear interpolation in multiple dimensions

For a better understanding, here the linear interpolation in more than one dimension will only be described the interpolation in two dimensions, but the interpolation in more than two dimensions is straightforward.

One idea to interpolate data points in two dimensions is to perform a linear interpolation in two directions. First, the linear interpolation will be performed in one direction and then in the other direction. This procedure is also called *bilinear interpolation*. To

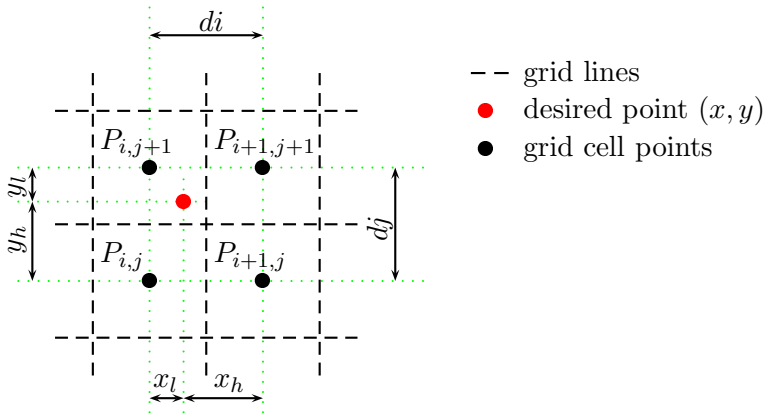


Fig. 2.3: Principle arrangement of the two dimensional linear interpolation

calculate the two dimensional linear interpolation there is only a weight needed. The distance of the given cell points is di in x-direction respectively dj in y-direction. It it necessary to break the distances di and dj down between the desired point and the grid cell points into a high value and a low value.

$$\begin{aligned} di &= x_h + x_l = P^x_{i+1,j} - P^x_{i,j} \\ dj &= y_h + y_l = P^y_{i,j+1} - P^y_{i,j} \end{aligned} \tag{2.5}$$

After the splitting di and dj into a high and a low value these values have to be weighted. In (2.6) there is the value for the x-direction and the value for the y-direction weighted.

$$\begin{aligned} t &\equiv \frac{x_l}{di} \\ u &\equiv \frac{y_l}{dj} \end{aligned} \tag{2.6}$$

The value t is the ratio factor of the x-direction and the value u is the ration factor of the y-direction, and they can be assumed values between 0 and 1. After defining the arrangement of that points, which is shown in (2.2) the desired point $P_{x,y}$ get its value with (2.7).

$$s_{x,y} = (1-t)(1-u) \cdot s_1 + t(1-u) \cdot s_2 + tu \cdot s_3 + (1-t)u \cdot s_4 \tag{2.7}$$

2.2.3 Implementation of the two-dimensional linear interpolation

After a look at the basic theoretically background of the linear interpolation in two dimensions, these discussed equations will be implemented into some algorithms. These algorithms will be written in C but I think it is not very difficult to implement these algorithms in another programming language. There is no special reason why the programming language C is used for that algorithms.

It will be assumed that we know the coordinates x and y of the desired point $P_{x,y}$. To get the value at this point a simple trace back will be performed in time through the velocity field from the cell centers [Sta03], which is shown in Fig. 1.4. First it has to be detected that the x and y values of the point $P_{x,y}$ are not outside of the working space of the Real Time CFD solver. That means these values are not allowed to be in the boundary space which is shown in Fig. 2.4. If the value of x or y is greater than $N+0.5$

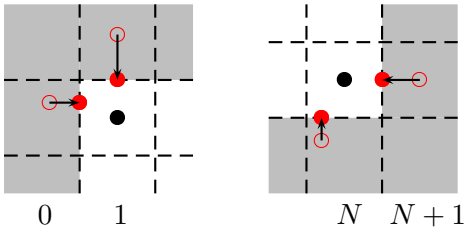


Fig. 2.4: Boundary check for the desired point $P_{x,y}$

or smaller than 0.5, the desired point will be moved to the border of the simulation space. After that check, if the values of x and y are outside of the simulation space, the nearest neighbours has to be found.

List. 2.1: Boundary check

```
if( x < 0.5 ) x=0.5;        if( y < 0.5 ) y=0.5;
if( x > N+0.5 ) x=N+0.5;    if( y > N+0.5 ) y=N+0.5;
```

The nearest neighbour in the x-direction will be found with List. 2.2. This type of getting the nearest neighbours works only if di and dj equals 1, that means that the space between two grid points has to be 1 in that case.

List. 2.2: Find nearest neighbours

```
/* if di and dj are 1 */
i0=(int)x; i1=i0+1;         j0=(int)y; j1=j0+1;
```

The next step is to ascertain the ratio of the x and the y value, which is described in (2.6). Fig. 2.3 shows the allocation of the ratio of the x and y values.

List. 2.3: Ratio of the 2D linear interpolation

```
t=x-i0; u=y-j0; /* if di and dj are 1 */
```

The last step of the two-dimensional linear interpolation is to calculate the scalar value on the point $P_{x,y}$ like in (2.7). We assume that the scalar value is `s`. The scalar is also an array like the velocity fields `u` and `v` of the same size `size=(N+2)*(N+2)` which is

shown in Fig. 1.3. `IX(i,j)` gives the index of the array back with the index i, j, see Section 1.1.1 on page 2, `s0` in List. 2.4 is the scalar field one time step earlier that `s`.

List. 2.4: Two-dimensional interpolation

```
s[IX(i,j)] = (1-t)*(1-u)*s0[IX(i0,j0)] +
             t*(1-u)*s0[IX(i1,j0)] +
             t*u*s0[IX(i1,j1)] +
             (1-t)*u*s0[IX(i0,j1)];
```

2.2.4 Example of the two-dimensional linear interpolation

Now for a better understanding a simple example of the bilinear interpolation is shown. In this case we take a 4 x 4 grid in two dimensions, that means we have 16 centered cell points which are shown with a red cross in Fig. 2.5, left. The scalar value on each point of the grid is $sin(x) \cdot sin(y)$. On the right hand side of Fig. 2.5 the analytical data for each point are shown. This interpolation takes the four neighbouring grid points which

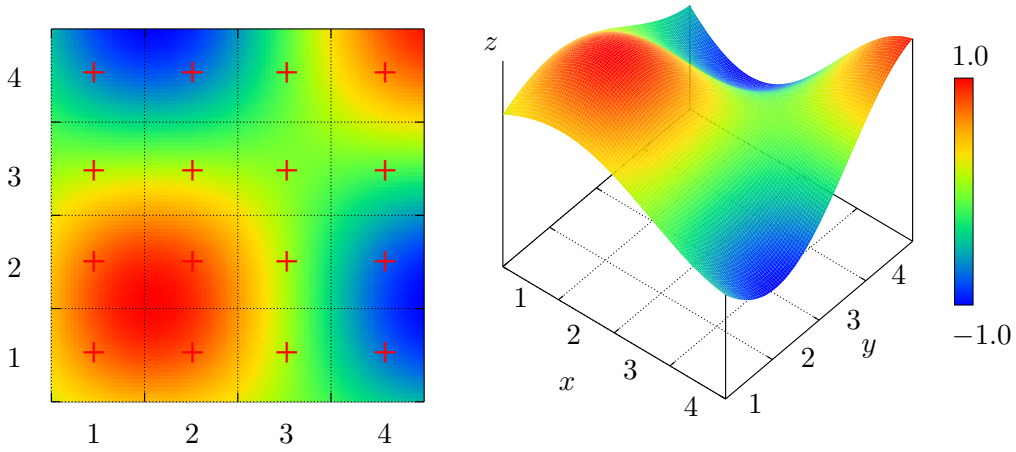


Fig. 2.5: Analytical data of $sin(x) \cdot sin(y)$ in the map view on the left side and in the surface view on the right side.

is described in the previous section and it shows a coarse approach to the analytical result in Fig. 2.5. To find out how precise the result of the linear interpolation is, it is necessary to calculate the error between the linear interpolation and the analytical results. To know if that interpolation is appropriate it is important to find out the failure of the bilinear interpolation.

2.2.5 Determination of the error of the linear interpolation

The error of the linear interpolation in two dimensions will be determined with (2.8). $E_{x,y}$ is the error value of the point $P_{x,y}$ and $S_{x,y}$ is the interpolated scalar value of the point $P_{x,y}$.

$$E_{x,y} = |O_{x,y} - S_{x,y}| \tag{2.8}$$

$O_{x,y}$ describes the analytical value of the point $P_{x,y}$, in that case, which is described in the previous example, $O_{x,y} = sin(x) \cdot sin(y)$.

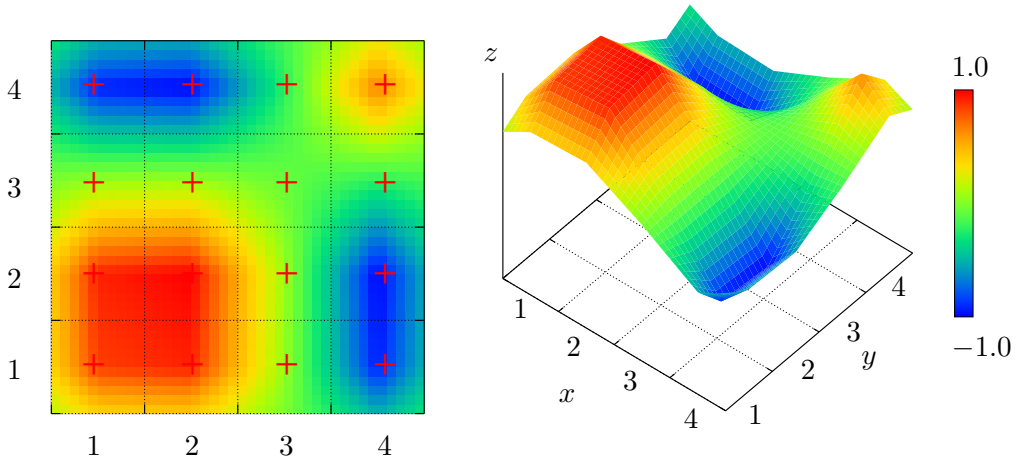


Fig. 2.6: Bilinear interpolation with a grid size of 4 x 4

List. 2.5: 2D error calculation

```
/* s_val is the interpolated value of the point x,y */
error = fabs((sin(x)*sin(y)) - s_val);
```

List. 2.5 shows how to implement (2.8) in the existing code. The value `s_val` describes the interpolated scalar value of the point $P_{x,y}$. Fig. 2.7 shows the allocation of the error over the two dimensional grid of the linear interpolation from Fig. 2.5. It shows an error value between 0 and 1. The red region on the left side of Fig. 2.7 means only that the value of the error is higher than 0.25. It shows that the error on the border of the bilinear interpolation is very high.

The function `fabs()` returns the absolute value of a floating point value. It is important to include `math.h` in the source code, if that isn't in the source code it will not be compiled without failures. To get an estimation of the overall error there will be

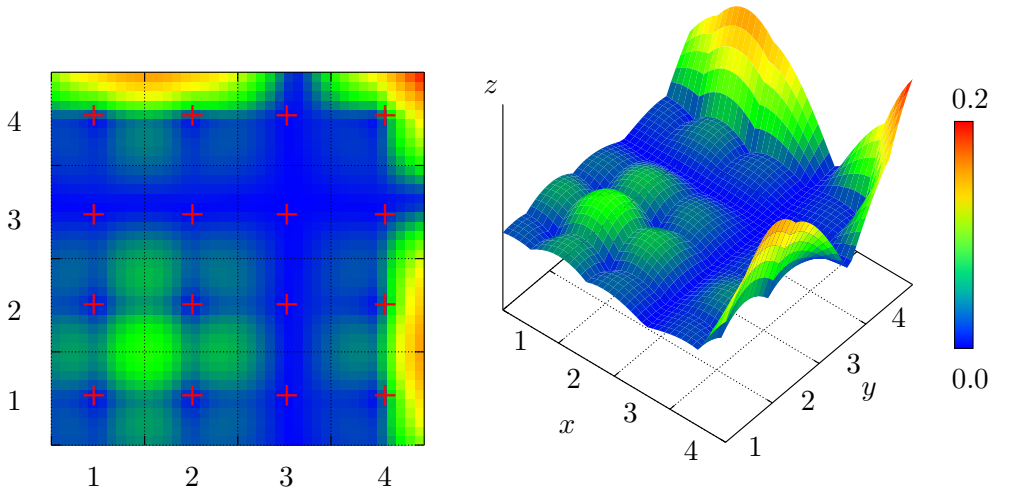


Fig. 2.7: Error of the bilinear interpolation with a grid size of 4 x 4.

defined how many points in the grid have an error over 0.2. Since we take the absolute value of the difference between interpolated scalar value and original analytical scalar value, the maximum error could only be between 0 and 1. List. 2.6 shows how to get the numbers of points with an error above 0.2.

List. 2.6: Error calculation

```
if( error > 0.2 ) nerror++;
```

In the case of the 4 x 4 grid the number of errors bigger than 0.2 are:

```
grid:40x40, cells:1600, nerror:78, percentage:4.875000
```

That means that between the fixed cell center points there are 10 points for the linear interpolation. So we have a grid with 40 x 40 linear interpolated scalar values and 1600 cells. `nerror` returns the number of points which are over the error limit of 0.2. The last value `percentage` returns the percentage of the number of errors above the error limit.

2.2.6 Increase accuracy with increasing the cell points

One method to increase the precision of the linear interpolation is to take more cell points, that means to increase the grid refinement. The scalar of the original data would be kept in the same range, only the number of cell points are more then in the previous example. Figure. 2.8 shows the same interpolation like in Fig. 2.7 but with more grid cells.

This result looks more than the analytical result in Fig. 2.5 than the result with the 4 x 4 grid in Fig. 2.6. Fig. 2.9 shows the error of the 8 x 8 grid. This figure shows that the 8 x 8 grid is more precise than the 4 x 4 grid which is shown in Fig. 2.7. To investigate the influence of the number of cells for the bilinear interpolation, calculations with different grid sizes are performed.

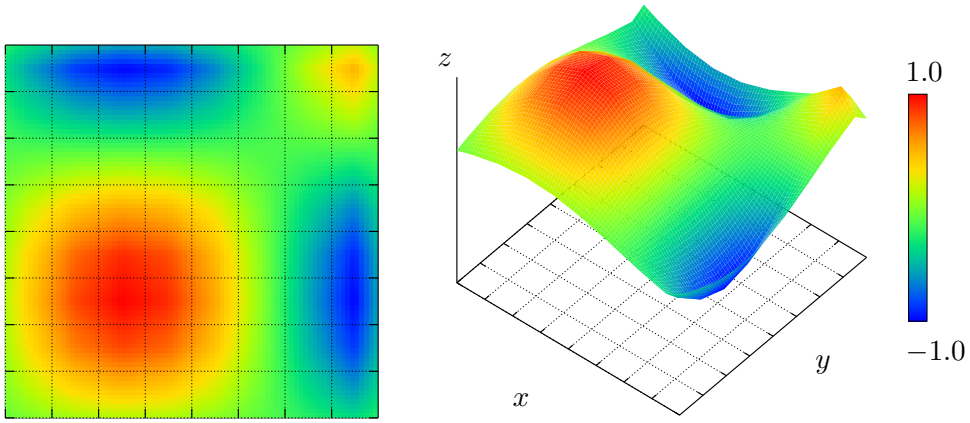


Fig. 2.8: Bilinear interpolation with a grid size of 8 x 8

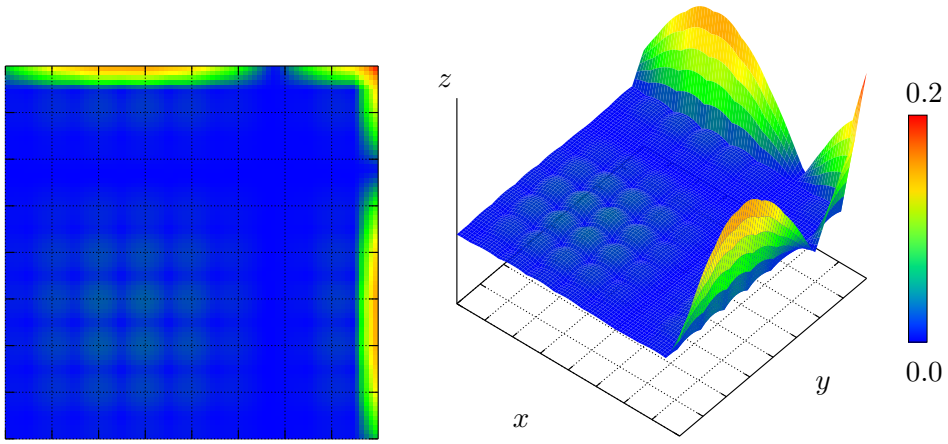


Fig. 2.9: Error of the bilinear interpolation with a grid size of 8 x 8

Tab. 2.1 shows the number of errors and the percentage of the error related to the number of cells. There the first two columns describe the size of the grid, e.g. a 4 x 4

grid size	cells	nerror	percentage
4 x 4	1600	78	4.875
8 x 8	6400	70	1.093
16 x 16	25281	41	0.162
32 x 32	101761	49	0.048
64 x 64	409600	188	0.046
128 x 128	1638400	328	0.020

Tab. 2.1: Calculation of the error of the different grid sizes

grid has 1600 cells. The third value shows the number of cells for which the error is larger than 0.2. The last column shows the percentage of the number of cells for which the error is larger than 0.2. Figure. 2.10 shows the percentage of the error over the

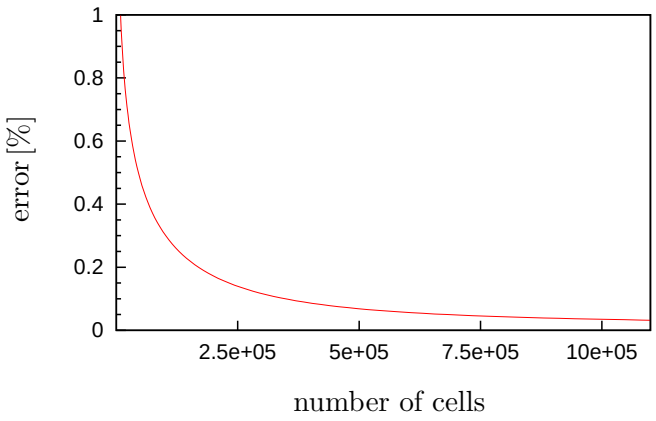


Fig. 2.10: Error of different grid sizes

number of grid cells. It shows that the error would be scatter if the number of grid cells rises. So the error of the interpolation is quite nice but also important is the time which will be spend for that interpolation.

2.2.7 Number of operations of the linear interpolation

Since the execution time of the program is very different on each computer, there the operations will be described. To specify how often a function in a program is called, a *profiler* is needed. In that case the *GNU profiler* is used. To evaluate that program with the profiler, a special compiling flag is needed. The GNU compiler takes the flag *-pg* to compile the program for profiling. After that the profile information could be written into a file with the following command

```
\$ gprof ./a.out ./gmon.out > prof_info
```

For the the linear interpolation in two dimensions the number of calls of the function which calculates (2.7) is specified. An output of the profiler is shown in List. 2.7. The function `interp` calculates (2.7) and is called 1681 times.

List. 2.7: Output of the profiler

```
 time   seconds   seconds   calls   Ts/call   Ts/call   name
  0.00      0.00      0.00    1681      0.00      0.00   interp
```

This means that the interpolation is called as well 1681 times. In Tab. 2.2 there are some columns from the typical profiler output shown. The first column shows the grid size, the second column shows the percentage of the total time which the program spent in the function `interp`. The third column shows the total number of seconds which the program spent in the function `interp`. The fourth column shows the number of calls of the function and the fifth column shows the time per call of the function.

Fig. 2.11 shows the graphical evaluation of the data in Tab. 2.2. Since only the total time is interesting the program can also be started with the UNIX® command `time`. The last column of Tab. 2.2 shows the total time calculated by the command `time`.

List. 2.8: Output of the time command

```
0.04s user 0.00s system 93% cpu 0.038 total
```

All of that time measures are performed on a **HP** compaq nx9005 notebook with a **AMD** Athlon™ XP-M 2400+ and 457 MB physical memory. Figure. 2.12 shows the

grid size	time [%]	time [s]	calls	time per call [ns]	total time [s]
4 x 4	0.00	0.00	1681	0.00	0.038
8 x 8	0.00	0.01	6561	0.00	0.125
16 x 16	33.35	0.06	25600	781.66	0.426
32 x 32	55.58	0.10	102400	977.08	1.244
64 x 64	51.57	0.34	410881	815.75	2.908
128 x 128	70.04	2.14	1640961	1300.00	9.381

Tab. 2.2: Calculation table of linear interpolation for `interp`

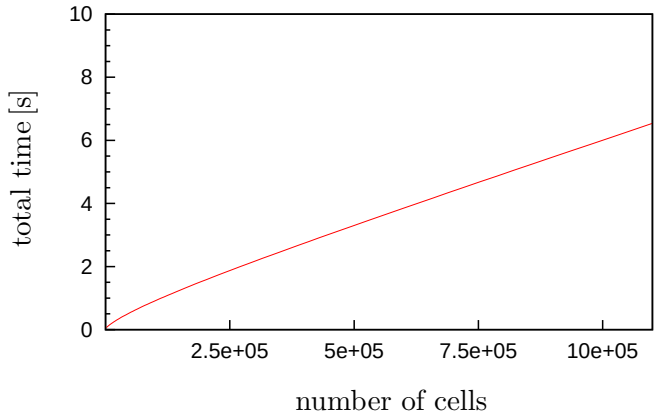


Fig. 2.11: Calculation time of different grid sizes with linear interpolation

comparison between the number of errors and the calculation time of the bilinear interpolation. It shows that the calculation time increases almost linear, and it will get higher and higher when the percentage of the error decreases.

The optimal range of the bilinear interpolation will be in the middle of both curves. In that area the calculation time will not be so big and the percentage of the error will be small enough for getting sufficient results.

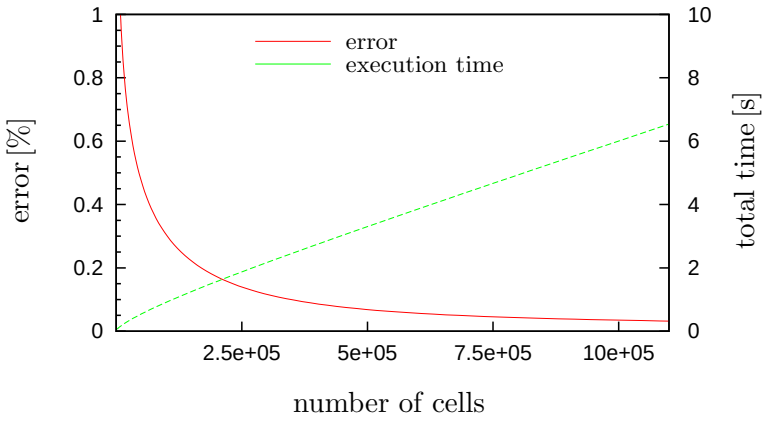


Fig. 2.12: Error and execution time of different grid sizes with the bilinear interpolation

2.3 Polynomial interpolation

The polynomial interpolation is the solution of the problem to find a polynomial from $n+1$ data points. There is an interpolation polynomial of degree n with $n+1$ data points. In the polynomial interpolation the neighbouring points are not connected with a straight line, like in the linear interpolation, but more points are pulled up for the interpolation.

There are a few basic approaches to solve this polynomial, (e.g *Newton* approach, *Lagrange* approach, ...). Many people denote the polynomial interpolation with the Lagrangian approach as Lagrangian interpolation and many papers are presently working with that approach [FSJ01]. The typical Lagrangian approach of the polynomial interpolation is shown in (2.9) and (2.10).

$$s_j(x) = s_j \prod_{k=0;k\neq j}^{n} \frac{x - x_k}{x_j - x_k} \tag{2.9}$$

$$s(x) = \sum_{j=0}^{n} s_j(x) \tag{2.10}$$

2.3.1 Fundamentals of the polynomial interpolation

The number of points (minus one) used in an interpolation scheme is called *order* of the interpolation. Increasing the order does not necessarily increase the accuracy, especially in the polynomial interpolation [PTVF88].

The big difference between the linear interpolation and the polynomial interpolation is that the polynomial interpolation takes more than two data points for the interpolation and puts a higher order polynomial into these data points. That means that the polynomial interpolation takes n data points for an interpolation of degree $n-1$. Figure 2.13 shows the principle representation of the polynomial interpolation.

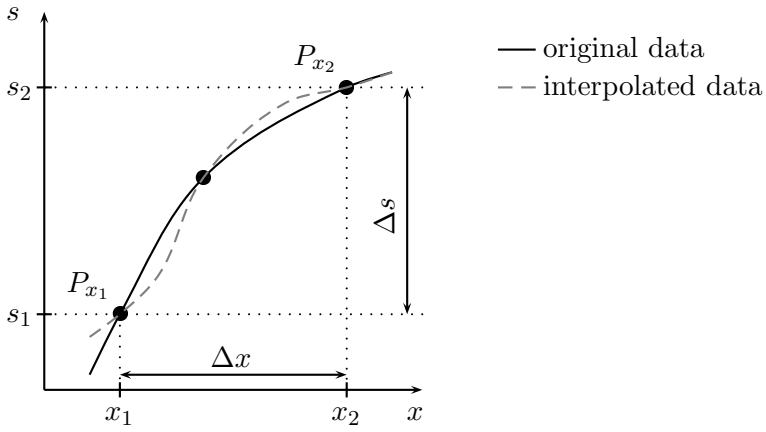


Fig. 2.13: Principle of 1D polynomial interpolation

2.3.2 Two-dimensional polynomial interpolation

The two-dimensional polynomial interpolation works as well as the two-dimensional linear interpolation. The principle to take the four neighbors for the interpolation described in Section 2.1 would also be applied for the polynomial interpolation. The basic idea of that two dimensional polynomial interpolation is to break up the problem in a succession of one dimensional interpolations.

Figure 2.14 shows the polynomial interpolation in two dimensions with four neighboring points. The comparison between that figure and Fig. 2.6 shows that their is no difference. This is because in that case the two dimensional interpolation with four neighboring points would break up into two polynomial interpolation in one dimension with two neighboring points. That means that their are rather two linear interpolations realized. On that reason there have to be implemented an other numerical

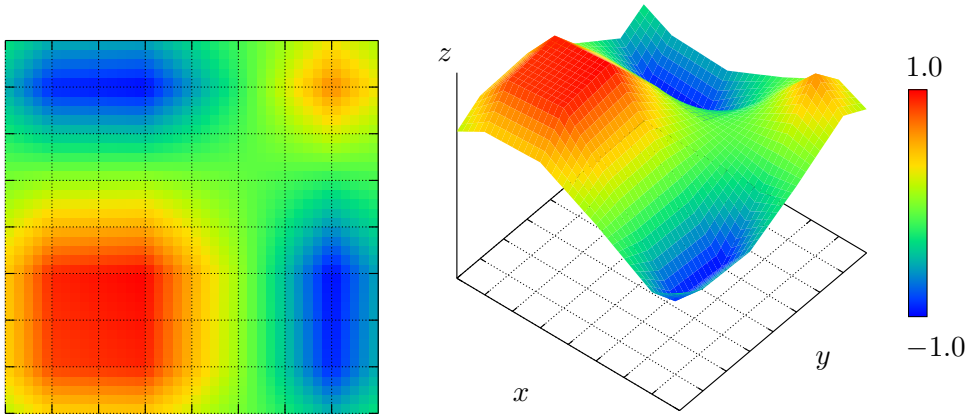


Fig. 2.14: Polynomial interpolation in two dimensions with a high order of accuracy with a grid refinement of 8 x 8

interpolation scheme or the neighboring points have to be incremented. So first an other interpolation scheme would be described, called *bicubic interpolation*. That interpolation scheme is a special numerical scheme for smoothness.

2.3.3 Implementation of the bicubic interpolation scheme

For the implementation of the bicubic interpolation scheme we took the algorithm from [PTVF88]. The bicubic interpolation scheme requires the user to specify at each grid point not just the function $y(x_1, x_2)$, but also the gradients $\partial y/\partial x_1 \equiv y_1, \partial y/\partial x_2 \equiv y_2$ and the cross derivative $\partial^2 y/\partial x_1 \partial x_2 \equiv y_{12}$. With the following properties a cubic interpolation function in the scaled coordinates t and u can be found:

- The values of the function and the specified derivatives are reproduced exactly on the grid points.
- The values of the function and the specified derivatives change continuously as the interpolating point cross from one grid square to another.

It is important to understand that nothing in the equations of bicubic interpolation requires that the derivatives are correct. The smoothness properties are tautologically forced, and have nothing to do with the accuracy of the interpolation. It is a separate problem to decide how to obtain the values that are specified. The better the values are, the more accurate the interpolation will be.

The best is to know the derivatives analytically, or to be able to compute them with numerical algorithms at the grid points. Next best is to determine them by numerical differencing. We decides to take *centered differencing* [FP02]. In (2.11) the function for the centered differencing method for calculating the first derivative is shown.

$$\left(\frac{\partial y}{\partial x}\right)_i \approx \frac{y_{i+1} - y_{i-1}}{x_{i+1} - x_{i-1}} \tag{2.11}$$

For the implementation we need a two-dimensional array and two one-dimensional arrays for the calculation of the centered differencing method, see Fig. 2.15. Listing 2.9 shows how to implement the gradients and the cross derivative.

The first line in Listing 2.9 calculates the gradient $\partial y/\partial x_1$, the second line calculates the gradient $\partial y/\partial x_2$ and the third and the fourth calculates the cross derivatives $\partial^2 y/\partial x_1 \partial x_2$. The two-dimensional array `y1a` stores the gradient $\partial y/\partial x_1$, the two-

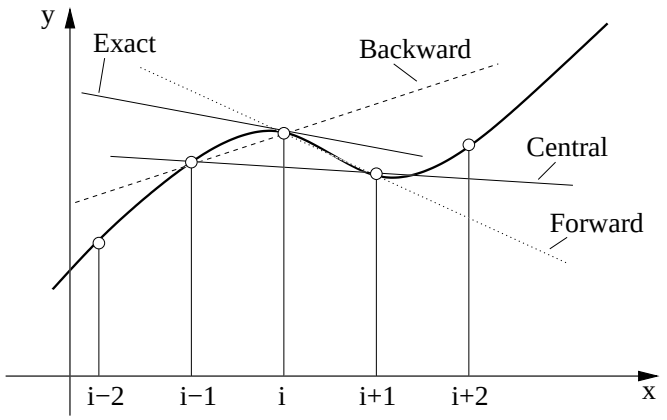


Fig. 2.15: Definition of the derivative and its approximations

dimensional-array `y2a` stores the gradient $\partial y/\partial x_2$ and the cross derivatives $\partial^2 y/\partial x_1 \partial x_2$ are stored in `y12a`. The values of the centered grid points are stored in the variable `ya` and their coordinates are stored in `x1a` and `x2a`.

List. 2.9: Centered differencing

```
y1a[j][k]=(ya[j+1][k]-ya[j-1][k])/(x1a[j+1]-x1a[j-1]);
y2a[j][k]=(ya[j][k+1]-ya[j][k-1])/(x2a[k+1]-x2a[k-1]);
y12a[j][k]=(ya[j+1][k+1]-ya[j+1][k-1]-ya[j-1][k+1]+
   ya[j-1][k-1])/((x1a[j+1]-x1a[j-1])*(x2a[k+1]-x2a[k-1]));
```

If we have the function y and the derivatives $y1, y2, y12$ we can do the bicubic interpolation within a grid square. There are two steps to calculate this interpolation.

- First obtain the sixteen quantities $c_{i,j}, i, j = 1, ..., 4$ using the routine `bcucof` of [PTVF88].
- The second step is to substitute the c's into any or all of the bicubic formulas for functions and derivatives shown in (2.13).

$$
\begin{aligned}
y(x_1, x_2) &= \sum_{i=1}^{4}\sum_{j=1}^{4} c_{ij} t^{i-1} u^{j-1} \qquad (2.12)\\
y1(x_1, x_2) &= \sum_{i=1}^{4}\sum_{j=1}^{4} (i-1) c_{ij} t^{i-2} u^{j-1} (dt/dx_1)\\
y2(x_1, x_2) &= \sum_{i=1}^{4}\sum_{j=1}^{4} (j-1) c_{ij} t^{i-1} u^{j-2} (du/dx_2)\\
y12(x_1, x_2) &= \sum_{i=1}^{4}\sum_{j=1}^{4} (i-1)(j-1) c_{ij} t^{i-2} u^{j-2} (dt/dx_1)(du/dx_2)
\end{aligned}
$$

Figure 2.16 shows a wrong result ot the bicubic interpolation of $sin(x) \cdot sin(y)$. For the calculation of this interpolation scheme we took the algorithms from [PTVF88].

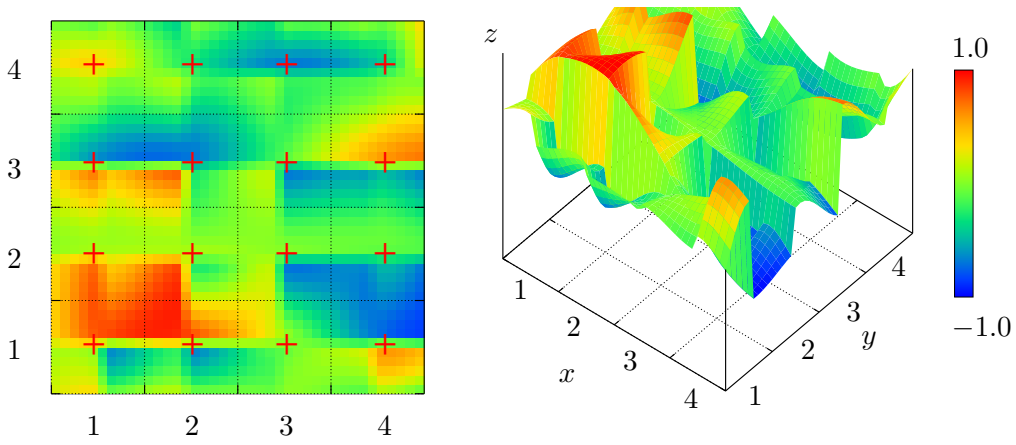


Fig. 2.16: Bicubic interpolation with a 4x4 grid of the function $z = sin(x) \cdot sin(y)$

2.3.4 Error calculation of the polynomial interpolation

To calculate the error of that interpolation scheme we need a working interpolation. Since we do not get a correct result, we can not determine the error of the bicubic interpolation scheme.

Also the calculation speed of that interpolation scheme is only interesting with a correct result. Since we do not get a correct result of the scheme we are not able to determine the calculation speed of that interpolation scheme.

2.3.5 Number of operations of the polynomial interpolation

The computing time is very important for calculating such interpolations. The computing time is directly proportional to the number of operations of the solving equation. In the equation of the Lagrangian interpolation, see (2.9) and (2.10), the operations at Tab. 2.3 with a degree N of the polynomial are required. There is a more detailed

addition	$\longrightarrow$	$(N+1)^2+N$
multiplication	$\longrightarrow$	$N(N+1)$
division	$\longrightarrow$	$N(N+1)$

Tab. 2.3: Number of operations of polynomial interpolations

description of these operations in [Sch94]. The most significant part of this operations is that one which increases with square. This results that only terms with N^2 are important for the number of operations. It's safe to say that the number of calculations could be ascertain with (2.13).

$$number\ of\ operations = 3N^2 + \mathcal{O}(N) \tag{2.13}$$

The important part of the operations is in $3N^2$ and the rest of the operations which are shown in Table 2.3 are pulled together in $\mathcal{O}(N)$.

2.3.6 Overshoots and clipping

If the measured data has a steep increase or a jump inside, and this area should be interpolated with a Lagrangian interpolating polynomial of degree N, there could be some overshoots in the interpolated data (Fig. 2.17.

One method to prevent such overshoots is to clip the data which are bigger or lower than a defined limit. This procedure is called *clipping* [Beh95].

$$u_1(x,y) = \begin{cases} u_{max} & \text{for} \quad u_{max} < u(x,y) \\ u_{min} & \text{for} \quad u_{min} > u(x,y) \\ u(x,y) & \text{for} \quad other \end{cases} \tag{2.14}$$

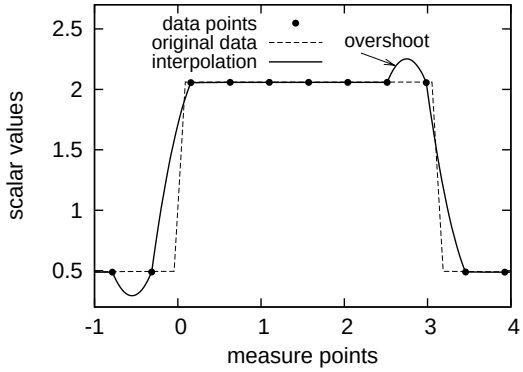


Fig. 2.17: Lagrangian interpolation with $N = 3$ data points and overshoots

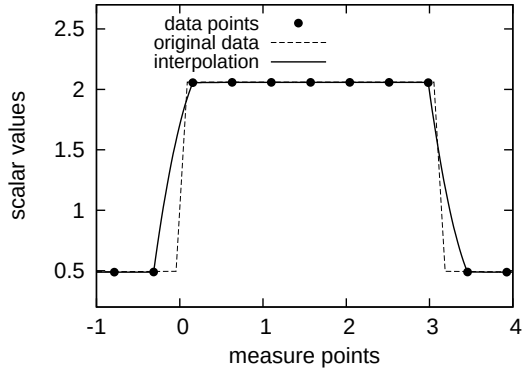


Fig. 2.18: Lagrangian interpolation with $N = 3$ data points and clipping

In Fig. 2.18 there are the same data points and interpolation data as plotted in Fig. 2.17, but the clipping method is applied on the interpolated data.

In List. 2.10 the measured data would be interpolated and inspected if there is a value which is bigger than `max_val` or lower than `min_val`. If the `val` is bigger than the `max_val`, the value `val` would become `max_val`, if the value `val` is lower than the `min_val` `val` would become `min_val`.

On the first line in List. 2.10 the value `val` would become the interpolated data which is calculated by the method `polint` and on the second line there is the clipping method which looks if the interpolated data are bigger or lower than the values `max_val` or `min_val` like in (2.14).

List. 2.10: Clipping procedure

```
polint(x,y,N,di,&val,&error);
val=(val<min_val) ? min_val:((val>max_val) ? max_val:val);
```

2.3.7 Quasi monotonic Lagrange interpolation

The idea of this interpolation [BS92] is that an interpolation with a high order is first calculated and then an interpolation with a lower order. After that an average of both interpolations is calculated with a weight factor. That means that the percentage of the interpolation with high order has to be as large as possible.

$$u_m = \alpha u_h + (1 - \alpha) u_l \tag{2.15}$$

$$0 \leq \alpha \leq 1 \tag{2.16}$$

The value u_h represents the interpolation with high order and u_l represents the interpolation with low order. In (2.16) the ratio factor is shown. The ratio factor will be calculated by (2.17) and (2.18). After that the ratio factor α is included in (2.15) to get u_m.

$$\begin{aligned} q_{max} &= u_{max} - u_l \\ q_{min} &= u_{min} - u_l \\ p &= u_h - u_l \end{aligned} \tag{2.17}$$

$$\alpha = \begin{cases} min(1, q_h/p) & \text{for} \quad p > 0 \\ min(1, q_l/p) & \text{for} \quad p < 0 \\ 0 & \text{for} \quad p = 0 \end{cases} \tag{2.18}$$

2.4 Bicubic Spline Interpolation

There is another common technique for obtaining smoothness for two or more dimensional interpolation. It is called the *bicubic spline interpolation*. The interpolation function is of the same form as (2.13), the values of the derivatives at the grid points are determined by one-dimensional splines. The benefit of the spline interpolation in contrast to the polynomial interpolation is that the spline interpolation do not evince oscillations for higher polynomial order.

2.4.1 Spline interpolation in one dimension

For a better understanding how to implement the bicubic interpolation the spline interpolation will be described in one dimension. Spline interpolations are used for smooth connections of different measured points, see Fig. 2.19. After the measured points are defined such splines are applied on the data points and the curve between that points will be drawn piecewise. For practical applications *cubic splines* are mostly used, that

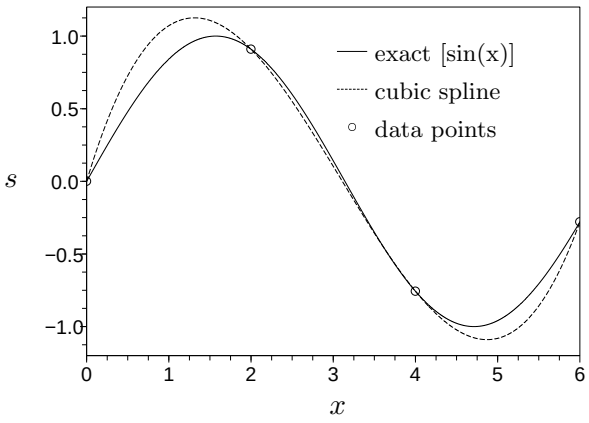


Fig. 2.19: Spline interpolation in one dimension

means that each polynomial between two data points is of degree 3. Cubic Splines are used because there is a good compromise between computational effort and accuracy [SC91] [BS92]. These pieces of curves are mathematically described with cubic polynomials,

$$s(x) = ax^3 + bx^2 + cx + d \tag{2.19}$$

where a, b, c and d are appropriate constants.

For getting a spline interpolation there are some rules necessary. Assumed that we have different data points in an interval $[A, B]$

$$A = x_0 < x_1 < \cdots < x_n = B \tag{2.20}$$

and their dedicated data points $y_0, y_1, \ldots, y_n$, the first rule is that each of the pieces of curves have the form like in (2.19). The second one is that every spline passes through each data point, shown in (2.21).

$$s_i(x_i) = y_i \quad \text{for} \quad i = 0, 1, ..., n \tag{2.21}$$

The next rule is that the spline forms a continuous function over $[A, B]$, like in (2.22).

$$s_i(x_{i+1}) = s_{i+1} \quad \text{for} \quad i = 0, 1, ..., n-1 \tag{2.22}$$

In addition it is necessary that the spline forms a smooth function over the interval $[A, B]$. To ensure that the spline forms a smooth function we need the first derivative.

$$s_i'(x_{i+1}) = s_{i+1}'(x_{i+1}) \quad \text{for} \quad i = 0, 1, ..., n-1 \tag{2.23}$$

The last point is that the connection points have to be the same crook, that means that there the crook has to be steady. To ensure that the crook is steady we need the second derivative of the splines.

$$s_i''(x_{i+1}) = s_{i+1}''(x_{i+1}) \quad \text{for} \quad i = 0, 1, ..., n-1 \tag{2.24}$$

With these equations the cubic spline interpolation is not yet ready. To finish the definition of the cubic spline interpolation we still need the boundary conditions. For the cubic spline interpolation there are two possibilities. The first possibility is that the second derivatives of the boundary data points are 0.

$$s_0''(x_0) = s_n''(x_n) = 0 \tag{2.25}$$

These cubic splines will also called *natural*. The second possibility is that the gradient of the boundary data points are pretended.

$$s_0'(x_0) = s_0; \;\; s_n'(x_n) = y_n \tag{2.26}$$

These boundary conditions have effects on the hole curve, but these effects will be decreased to the middle of the curve. The main benefit of the cubic spline interpolation is that it takes relative less computational power for a good result, so that in practical use cubic splines are often used.

The implementation of the one-dimensional cubic spline interpolation is described in [PTVF88]. There are only two functions which are necessary to calculate the cubic spline interpolation in one dimension.

List. 2.11: Cubic spline interpolation in one dimension

```
void spline(float x[], float y[], int n,
            float yp1, float ypn, float y2[]);
void splint(float xa[], float ya[], float y2a[],
            int n, float x, float *y);
```

It is important to understand that the function `spline` is called only once to process the entire tabulated function in arrays x_i and y_i. Once this has been done, values of the interpolated function for any value of x are obtained by calls to the separate routine `splint`.

2.4.2 Spline interpolation in multiple dimensions

The cubic spline interpolation in two or more dimensions is called *bicubic spline interpolation*. This interpolation scheme is a common technique for obtaining smoothness in two dimensions. The bicubic spline interpolation is a special case of the bicubic interpolation scheme, which is described in Section 2.3.2.

The difference to the bicubic interpolation scheme is that the values of the derivatives at the grid points are determined 'globally' by one-dimensional splines. To interpolate one functional value, one performs m one-dimensional splines across the rows of the table, followed by one additional one-dimensional spline down the newly created column.

List. 2.12: Bicubic spline interpolation

```
void splie2(float x1a[], float x2a[], float **ya,
            int m, int n, float **y2a);
void splin2(float x1a[], float x2a[], float **ya,
            float **y2a, int m, int n, float x1,
            float x2, float *y);
```

For quadratic regions m and n is equal, so we could simplify our variables.

$$N = m = n \tag{2.27}$$

Now we can say we have a $N \times N$ grid like at the linear interpolation schemes in Section 2.2. This two functions `splie2` and `splin2` are also described in [PTVF88].

2.4.3 Example of the bicubic spline interpolation

To compare the results from the bicubic spline interpolation with other interpolation schemes, we create an example with a 4×4 grid in two dimensions, that means we have 16 centered cell points. These centered cell points are marked as a red cross in Fig. 2.20 on the left picture. The function we analyse is again $sin(x) \cdot sin(y)$. On the right hand side you can see a much better approximation of the function $z = sin(x) \cdot sin(y)$ compared to the bilinear interpolation with a grid size of 4×4 which is described in Section 2.2.2 on page 8.

Fig. 2.21 shows the difference between the analytical result of the function $z = sin(x) \cdot sin(y)$ which is shown in Fig. 2.21(a) and the bicubic spline interpolation with different grid sizes which are shown in Fig. 2.21(b), (c) and (d). It shows that the accuracy increases if the grid size of the interpolation region increases.

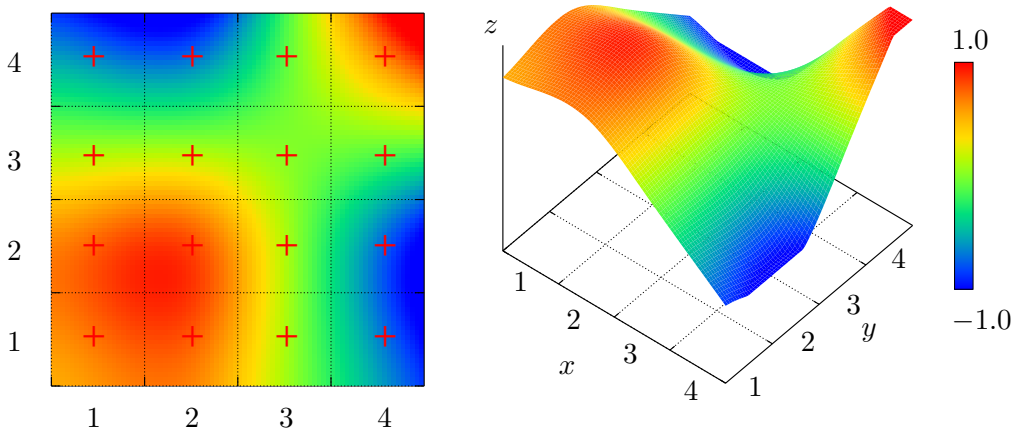
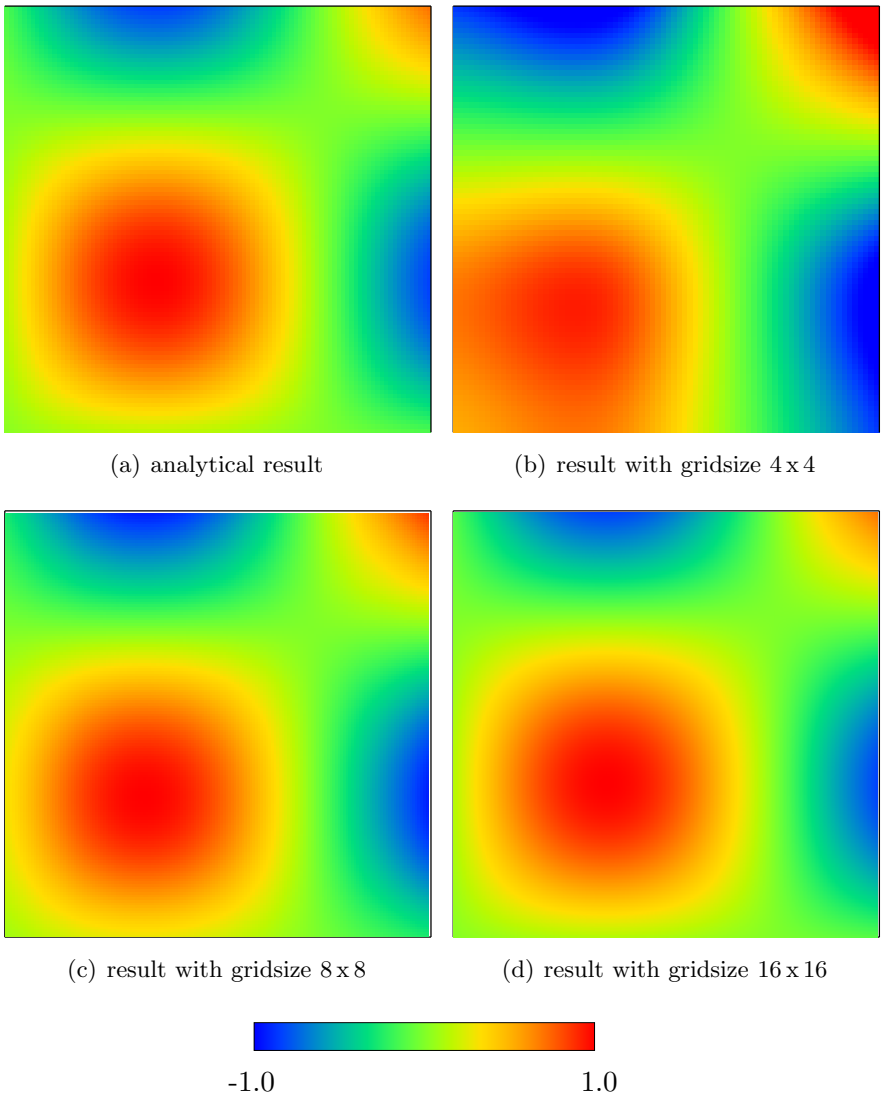


Fig. 2.20: Bicubic spline interpolation with a 4×4 grid of the function $z = sin(x) \cdot sin(y)$



(a) analytical result

(b) result with gridsize 4 x 4

(c) result with gridsize 8 x 8

(d) result with gridsize 16 x 16

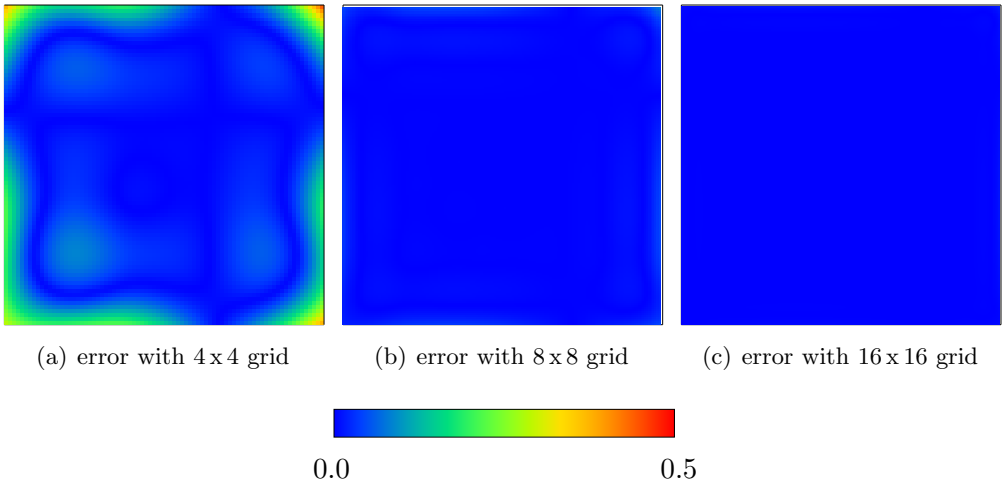
Fig. 2.21: Results of the bicubic spline interpolation with different grid sizes

To know how accurate this bicubic interpolation is, it is necessary to define the error of that interpolation scheme. This error will be described in the next section.

2.4.4 Determination of the error of the bicubic spline interpolation

The error of the bicubic interpolation scheme is described exactly as the error of the bilinear interpolation scheme in (2.8). The implementation is also exactly as the error implementation for the bilinear interpolation scheme in Section 2.2.5, page 10.

Figure 2.22 shows the error of bicubic spline interpolation with a 4 x 4 grid. It shows that the major error occurs on the boundary of the interpolation. To decrease that error we could increase the number of centered grid points with increasing the cells. Table 2.4 shows the number of errors which are bigger than 0.25 and the percentage of the error. It means we count all errors which are over 0.25 and divide them by the number of cells.



(a) error with 4 x 4 grid (b) error with 8 x 8 grid (c) error with 16 x 16 grid

Fig. 2.22: Error calculation of the bicubic spline interpolation

grid size	cells	nerror	percentage
4 x 4	1600	246	15.375
8 x 8	6400	1	0.015
16 x 16	25281	0	0.000
32 x 32	101761	0	0.000
64 x 64	409600	0	0.000
128 x 128	1638400	0	0.000

Tab. 2.4: Calculation of the error of the different grid sizes

2.4.5 Number of operations of the bicubic spline interpolation

That the execution time of that program is very different on each computer, there the operations will be described. How to create the Tab. 2.5 and read the columns, is described in Section 2.2.7, page 14. Tab. 2.5 is also produced by the *GNU profiler* which is also described in Section 2.2.7 in detail. Figure 2.23 shows the error of the bicubic

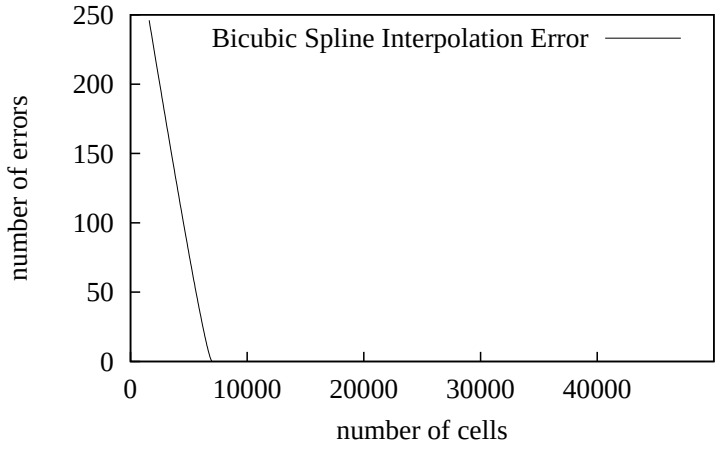


Fig. 2.23: Error of the bicubic spline interpolation

spline interpolation. Fig. 2.24 shows the calculation time of the function `splin2` which is essential for the bicubic spline interpolation. In contrast to that curve in Fig. 2.24 there is also the ideal speed curve shown. This ideal curve results from a function with no input and no output, see Lst. 2.13.

grid size	time [%]	time [s]	calls	time per call [μs]	total time [s]
4 x 4	0.00	0.01	1681	5.94	0.04
8 x 8	0.00	0.10	6561	15.24	0.18
16 x 16	3.03	0.33	25600	11.72	1.03
32 x 32	4.94	5.07	102400	50.80	6.88
64 x 64	6.39	29.11	410881	70.00	47.59
128 x 128	5.68	337.82	1640961	210.00	698.52

Tab. 2.5: Calculation time table of the bicubic spline interpolation

List. 2.13: Empty function

```
void empty_func() {}
```

Figure 2.24 shows that the bicubic spline interpolation takes a lot of time for the calculation, but Fig. 2.21 shows that there are not so many cells necessary for getting more accurate results. Figure 2.24 shows only the first 10 seconds because of the better view of this curves. More values are shown in Tab. 2.5.

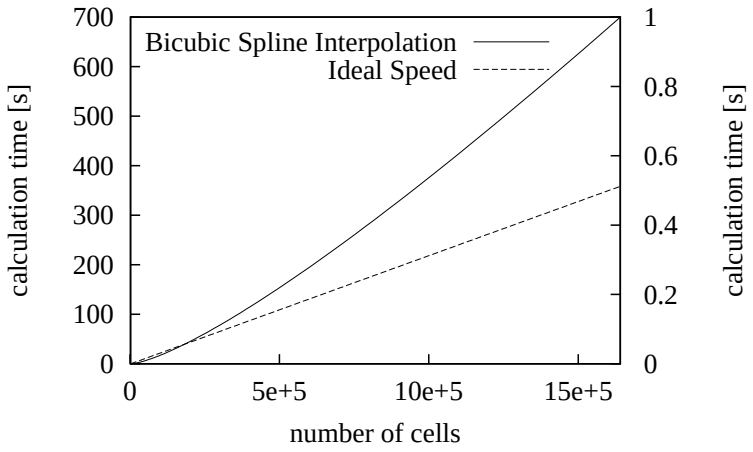


Fig. 2.24: Calculation time of the bicubic spline interpolation

Chapter 3

Implementation of different interpolation schemes

This chapter describes how to implement the specified interpolation schemes in Chapter 2 in an available CFD code [Sta03]. Jos Stam describes how to realize a fluid dynamics solver for games based on a complete C code. That solver is geared towards visual quality, and the emphasis of that solver is on stability and speed. This means that simulations with that solver can be advanced with arbitrary time steps.

In that CFD code, the amount of density of the particles is simply obtained by a linear interpolation, as described in Section 2.2, page 7. In the present chapter it will be described how to implement some interpolation schemes in the CFD code of Jos Stam.

3.1 Make the velocity field constant

To compare the results of the interpolation schemes with the analytical results, it is important to make the velocity field constant in one direction, either in x-direction or in y-direction. We choose the velocity field is constant in x-direction, this means only the vector u of the velocity field is working. For making the velocity field constant, only the function `advect` is necessary.

In the case the function `advect` is called with the parameters which belongs to the vector of the velocity in x-direction. Listing 3.1 shows the changed function `vel_step` after the simplification. In that function only some function calls are removed from the original CFD code.

List. 3.1: `vel_step` for constant velocity field

```
void vel_step(int N, float *u, float *v, float *u0,
              float *v0, float visc, float dt) {
   SWAP(u0,u); SWAP(v0,v);
   advect(N,1,u,u0,u0,v0,dt);
   advect(N,2,v,v0,u0,v0,dt,interp);
}
```

After that there is only the function call `advect` in the function `vel_step` working, the velocity field has only values in x-direction. Figure 3.1 shows the velocity vectors in x-direction at the center of each cell. For a better view there is only a 8×8 grid shown, it could be also bigger or smaller.

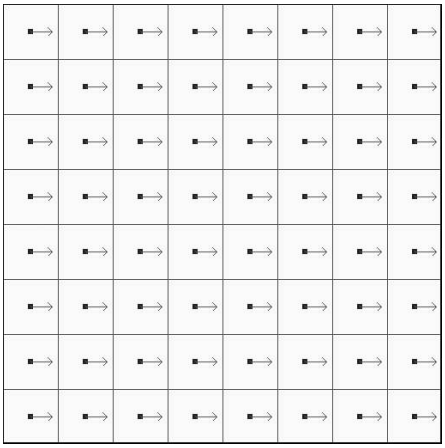


Fig. 3.1: Visualization of the constant velocity field $\vec{u}$

3.2 Bilinear interpolation scheme

This linear interpolation scheme is the easiest scheme and requires least effort to implement in the CFD code. The linear interpolation is already included by Jos Stam [Sta03]. The only function which we have to look at is the function `advect`. Section 2.2.3 describes these lines in Listing 3.2 more exactly.

List. 3.2: The function for the advection

```
void advect(int N, int b, float *d, float *d0, float *u,
            float *v, float dt) {
   int i, j, i0, j0, i1, j1;
   float x, y, s0, t0, s1, t1, dt0;

   dt0 = dt*N;
   for(i=1; i<=N; i++) {
     for(j=1; j<=N; j++) {
       x = i-dt0*i[IX(i,j)]; y = j-dt0*v[IX(i,j)];
       if(x<0.5) x=0.5; if(x>N+0.5) x=N+0.5;
       i0=(int)x; i1=i0+1;
       if(y<0.5) <=0.5; if(y>N+0.5) y=N+0.5;
       j0=(int)y; j1=j0+1;
       s1=x-i0; s0=1-s1; t1=y-j0; t0=1-t1;
       d[IX(i,j)]=s0*(t0*d0[IX(i0,j0)]+t1*d0[IX(i0,j1)]) +
                  s1*(t0*d0[IX(i1,j0)]+t1*d0[IX(i1,j1)]);
     }
   }
   set_bnd(N,b,d);
}
```

3.2.1 Bicubic interpolation scheme

We do not implement the bicubic interpolation scheme because the result of the algorithms in [PTVF88] seems to be wrong.

3.2.2 Bicubic spline interpolation scheme

To implement the bicubic spline interpolation algorithm from [PTVF88] it is necessary to create a function which switches the one-dimensional array to a two-dimensional array. This is necessary because Jos Stam works with one-dimensional arrays and the algorithms in [PTVF88] work with two-dimensional arrays.

List. 3.3: Switch array

```
void swap_array(int N, float *d0, float **s) {
   int i,j;

   for(i=1; i<=N; i++) {
      for(j=1; j<=N; j++) {
         s[i-1][j-1] = d0[IX(i,j)];
      }
   }
}
```

After small function mentioned as above, we have to rewrite the advection function for the bicubic interpolation scheme.

List. 3.4: Advection with bicubic splines

```
void advect(int N, int b, float *d, float *d0, float *u,
            float *v, float dt, int interp) {
   int i,j;
   float x,y,dt0,val,**ya,**y2a,*x1a,*x2a;

   x1a = malloc((N+2)*sizeof(float));
   x2a = malloc((N+2)*sizeof(float));

   ya = malloc((N+2)*sizeof(float));
   y2a = malloc((N+2)*sizeof(float));
   for(i=0; i<=N; i++) {
      ya[i] = malloc((N+2)*sizeof(float));
      y2a[i] = malloc((N+2)*sizeof(float));
   }

   for(i=0; i<=N; i++) {
      x1a[i] = i;
      x2a[i] = i;
   }

   swap_array(N,d0,ya);
   splie2(x1a,x2a,ya,N,N,y2a);
```

```
    dt0 = dt*N;
    FOR_EACH_CELL
       x=i-dt0*u[IX(i,j)]; y=j-dt0*v[IX(i,j)];
       if (x<0.5f) x=0.5f; if(x>N+0.5f) x=N+0.5f;
       if (y<0.5f) y=0.5f; if(y>N+0.5f) y=N+0.5f;

       splin2(x1a,x2a,ya,y2a,N,N,x,y,&val);
       d[IX(i,j)] = val;
    END_FOR

    free(x1a); free(x2a);
    for(i=0; i<=N; i++) {
       free(ya[i]); free(y2a[i]);
    }
    free(ya); free(y2a);
    set_bnd(N,b,d);
}
```

Chapter 4

Evaluation of different interpolation schemes

Since we do not get a working implementation for the polynomial bicubic interpolation we discuss only the bilinear interpolation scheme and the bicubic spline interpolation scheme. We discuss a simple example where the velocity field is constant and we look at the density. The second example is an analytical example from [FP02].

4.1 Simple wind tunnel

This example shows a simple wind tunnel like in Figure 4.1. On the left hand side of the wind tunnel there is an inlet and on the other side of the wind tunnel there is an outlet. The top and the bottom side of the wind tunnel defines a wall. To set the velocity field constant in the x-direction we have to change the code which is shown in Listing 4.1.

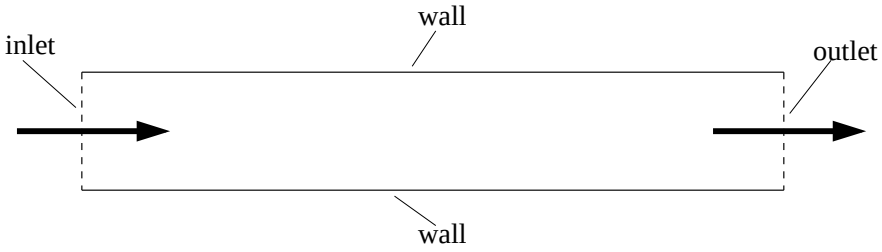


Fig. 4.1: Delineation of the simple wind tunnel

List. 4.1: Set the velocity in x-direction

```
static void set_velocity() {
   int i,j;
   for(i=0; i<=N; i++) {
      for(j=0; j<=N; j++) {
         u[IX(i,j)] = 0.01f;
         v[IX(i,j)] = 0.00f;
      }
   }
}
```

To compare these interpolation schemes we have to select some interesting variables, which are important for the simulation.

List. 4.2: Interesting variables

```
N=16; h=1.0f/N; dt=0.1f; diff=0.0f;
visc=0.0f; force=5.0f; source=100.0f;
```

N means the number of cells in one direction, in that case we have a grid with 16 cells in x-direction and 16 cells in y-direction. So that Jos Stam assumed in [Sta03] that the physical length of each side of the grid is one so that the grid spacing `h` is given by $1/N$. The fixed variable `dt` assumes the time spacing between two snapshots of the simulation. It can be set a diffusion value with the variable `diff`, that means if the `diff` variable is 0 there is no exchange with the neighboring grid cells. If the diffusion is bigger than 0 the density will spread across the grid cells. If the variable `visc` is 0 there is no viscosity in the grid cells. With the variables `force` and `source` you can adjust the value of the velocity and the value of the density.

Figure 4.2 shows the result with the bilinear interpolation scheme and Fig. 4.3 shows the result with the bicubic spline interpolation scheme. In this example no significant differences can be found between these interpolation schemes. It is also shown that there is no numerical diffusion at the bilinear and the bicubic spline interpolation.

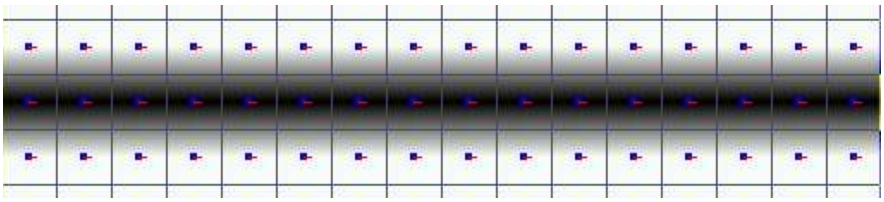


Fig. 4.2: Evaluation of the wind tunnel with the bilinear interpolation scheme

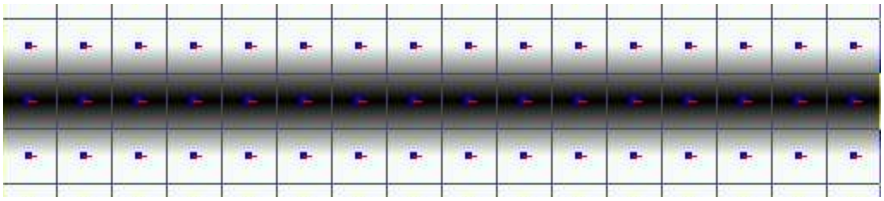


Fig. 4.3: Evaluation of the wind tunnel with the bicubic spline interpolation scheme

4.2 Convection of a step profile

Another popular test case is the convection of a step profile in a uniform flow oblique to grid lines. The mathematical background and the convergence is described in [FP02]. In Fig. 4.4 we show the profile for the case when the flow is at 45° to the grid $u = v$. In List. 4.3 we show the implementation of that flow. We have included this popular test case in our solver and looked at the results which are shown in Fig. 4.5.

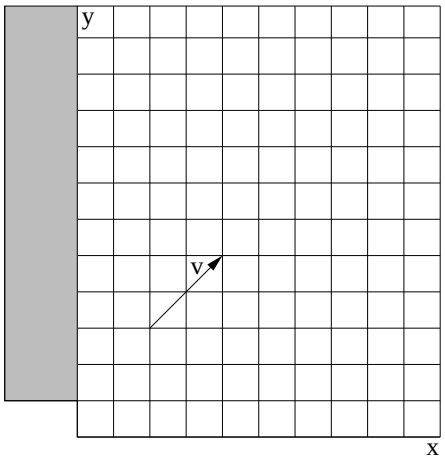


Fig. 4.4: Convection of a step profile in a uniform flow oblique to grid lines

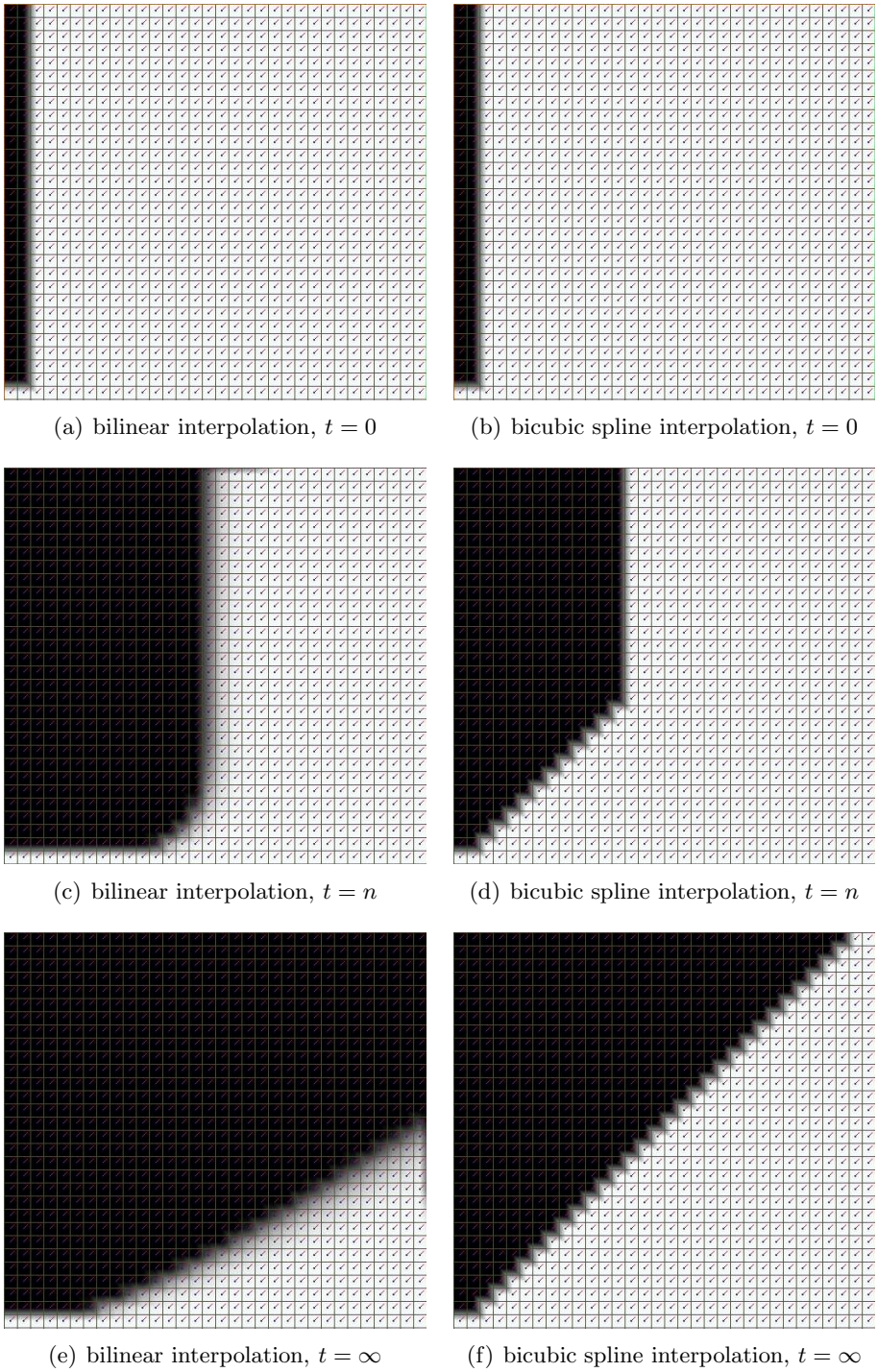
List. 4.3: Flow of 45°

```
static void set_veloctiy() {
   int i,j;
   for(i=0; i<=N; i++)
      for(j=0; j<=N; j++) {
         u[IX(i,j)] = 0.01f;
         v[IX(i,j)] = 0.01f;
      }
}
```

It is a popular test case in a uniform flow oblique to grid lines. It can be solved with the central difference scheme (CDS) or with the upwind difference scheme (UDS) which is described in [FP02]. The boundary conditions on the west and on the south side are values of ϕ like in [FP02] and the conditions at north and east are outflow boundaries. In Fig. 4.5 there is the profile of ϕ for the test case when the flow is 45° to the grid shown, obtained on a uniform 10 x 10 control volume (CV) grid.

Figure 4.5 shows also like the previous example that there is no numerical diffusion for the bilinear and the bicubic spline interpolation. In contrast to this two methods, which do not have any numerical diffusions, the methods which are shown in [FP02] have numerical diffusion. Figure 4.8 in [FP02] shows the two methods, UDS and CDS, which do have numerical diffusions.

We can say to the time calculation of these two interpolation schemes in practice that there is no essential difference between the bilinear interpolation, on the left side of Fig. 4.5 and the bicubic spline interpolation, on the right side of Fig. 4.5. That these two interpolation schemes are more or less equal for the time calculation it is only necessary to look for the accuracy of these interpolation schemes to select the better one for this application. That the bicubic spline interpolation do not have any numerical diffusions and the results are more accurate compared with the analytical results, the bicubic spline interpolation is more appropriate for that application.



(a) bilinear interpolation, $t = 0$

(b) bicubic spline interpolation, $t = 0$

(c) bilinear interpolation, $t = n$

(d) bicubic spline interpolation, $t = n$

(e) bilinear interpolation, $t = \infty$

(f) bicubic spline interpolation, $t = \infty$

Fig. 4.5: Result of the bilinear interpolation and the bicubic spline interpolation for different time steps

Chapter 5

Summary and outlook

In this thesis we have presented and implemented three different types of interpolations, the *bilinear interpolation*, the *bicubic interpolation* and the *bicubic spline interpolation*. The easiest and the quickest interpolation scheme for the implementation was the bilinear interpolation, because this scheme works with less mathematical effort. We can say that the bicubic spline interpolation is the best interpolation scheme for this application because the effort, the calculation time and the accuracy are in a good proportion. The bicubic interpolation does not work in that case because there is a possibility that the algorithm from [PTVF88] is wrong. My assumption is that the weight constants are selected wrong.

If we can make a statement which interpolation scheme is the best for this application, we probably have to check more than three interpolation schemes for their implementation, calculation time and accuracy. It will be interesting if the bicubic interpolation is working because there could also be compared that scheme with the other ones. After that it would also be interesting to implement other interpolation schemes because after there will be exposed which interpolation scheme is the optimal interpolation scheme for that applications.

It will also be interesting to split the interpolation schemes, e.g. at the boundary there could be applied the linear interpolation scheme and in the central of the simulation region there could be applied the spline interpolation for more accurate and faster results. On the other hand there could be implemented cubic spline interpolation schemes for high-order schemes and the linear interpolation for low-order schemes.

The next possible step will be to implement a backward integration in time which is called semi-Lagrangian integration for much faster calculation of the density and the velocity. This semi-Lagrangian scheme comes from the numerical weather review. They are working with big time steps even though they need exact results for their calculations.

It would also be interesting to make more practical examples with that interpolation schemes because in this examples in Section 4.1 and 4.2 there is no essential time difference between this two interpolation schemes, but it could be if there is a more complex example there. It is possible that there is more time difference if there are objects in the simulation region.

In practice such schemes are suggestive for the game industry, e.g. there could be calculated smoke during the game sequence. The smoke in the game could be calculated just in time if a figure of the game walks through the smoke. Other application in practice

could be a smoke simulation in buildings or a wind channel where objects could moved during the simulation and there is shown the result of the smoke or the wind instantly. An application with that schemes gives a good and fast overview how the smoke or the wind in a building could move, but for exact calculations there are conventional CFD applications which could calculate these problems more exact but they need more computer power.

List of Figures

List of Tables

Listings

Bibliography

[Beh95] J. Behrens. *Adaptive semi-Lagrange finite element method for the solution of the shallow water equation*. PhD thesis, December 1995.

[Beh96a] J. Behrens. An adaptive semi-Lagrangian advection scheme and its parallelization, 1996.

[Beh96b] J. Behrens. A parallel adaptive finite element semi-Lagrangian advection scheme for the shallow water equations, 1996.

[BS92] R. Bermejo and A. Staniforth. The conversion of semi-lagrangian advection schemes to quasi-monotonic schemes. *Monthly Weather Review*, 120(2622), January 1992.

[FP02] J. H. Ferziger and M. Perić. *Computational Methods for Fluid Dynamics*. Springer, third edition, 2002.

[FSJ01] R. Fedkiw, J. Stam, and H. W. Jensen. Visual simulation of smoke. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 15–22. ACM Press/ACM SIGGRAPH, 2001.

[Inc06a] ANSYS Inc. Computational fluid dynamics (CFD) software, September 2006. ANSYS Inc. Homepage: http://www.ansys.com.

[Inc06b] Fluent Inc. Computational fluid dynamics (CFD) software, September 2006. Fluent Inc. Homepage: http://www.fluent.com.

[Ltd06] Open CFD Ltd. Open Source CFD toolbox, September 2006. Open Field Operation and Manipulation (OpenFoam) Homepage: http://www.opencfd.co.uk.

[Pop06] Stephan Popinet. Open Source free software library for the solution of the partial differential equations, September 2006. Gerris Flow Solver Homepage: http://gfs.sf.net.

[PTVF88] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in* C. Cambridge University Press, 1988.

[SC91] A. Staniforth and J. Côtè. Semi-Lagrangian integration schemes for atmospheric models: A review. *Monthly Weather Review*, 119(2206), March 1991.

[Sch93] H. R. Schwarz. *Numerische Mathematik*, pages 94–149. B. G. Teubner Stuttgart, 1993.

[Sch94] B. Schmitt. *Numerik-Skripte 1994 Uni Marburg*. 1994.

[Sta03] J. Stam. Real-time fluid dynamics for games. Proceedings of the Game Developer Conference, 2003.