Numerical Methods for Engineers

Leif Rune Hellevik

Department of Structural Engineering, NTNU

Jan 26, 2018

Contents

Chapter 1

Introduction

1.1 Acknowledgements and dedications

This digital compendium is based on the compendium *Numeriske Beregninger* by my good friend and colleague Jan B. Aarseth who shared all of his work so generously and willingly. His compendium served as an excellent basis for further development and modernisation in terms of accessibility on the web, programming language, and naturally numerous theoretical extensions and removals. Primarily, the digital compendium is intended to be used in the course *TKT-4140 Numerical Methods* at NTNU.

The development of the technical solutions for this digital compendium results from collaborations with professor Hans Petter Langtangen, who has developed Doconce for flexible typesetting, and associate professor Hallvard Trætteberg at IDI, NTNU, who has developed the webpage-parser which identifies and downloads Python-code from webpages for integration in Eclipse IDEs. The the development of the digital compendium has been funded by the project IKTiSU. and NTNU Teaching Excellense 2015-2016.

Johan Kolstø Sønstabø was instrumental in the initial phase and the development of the first version of the Digital Compendium consisting of only Chapter 1 and for programming examples and development of exercises.

Fredrik Eikeland Fossan has contributed enormously by implementing python codes for most chapters. In particular, his contributions to the test procedures for MMS and to algorithms for testing of the order of accuracy of numerical schemes are highly appreciated.

Lucas Omar Müller taught the course TKT 4140 in the spring semester 2017. Apart from doing an excellent job in teaching by bringing in a wide range of new elements, he has also made numerous contributions in translating the compendium from Norwegian to English and by adding theoretical sections.

Marie Kjeldsen Bergvoll made important contributions during her summer-job with the translation of chapters 2-7 from LaTeX to Doconce. Tragically, Marie passed away due to a sudden illness while being in China working on her

master thesis in March 2017. Her choice of courses were motivated by interest, curiousity and a hunger for knowledge. The very same mentality brought her to countries like Germany and China, where English is not the official language. She was a energetic, independent, warm, and dedicated young student, with good analytical skills and a talent for mathematics.

Hans Petter Langtangen, passed away on October 10th in 2016. Hans Petter was an extraordinarily productive author, lecturer, supervisor and researcher. His books on software and methods for solving differential equations are widely used and have influenced these fields considerably. Ever since I meet Hans Petter for the first time, when he acted as my opponent at my PhD-defense in 1999 and later on as a collaborator at CBC, I have learned to know him as an enthusiastic and playful scholar with a great passion for science. He was truly a unique character in our field. Who else would write his own flexible markup language producing output to a range of web-formats and PDF, before setting off to produce piles of textbooks? But maybe even more important was his inspirational, energetic and natural ways of interaction with friends, colleagues, and collaborators in a great variety of disciplines. I miss him dearly both as a friend and collaborator.

During the work with this digital compendium, we have suffered the loss of *Hans Petter* and *Marie*, two fantastic but quite different persons. Hans Petter at the full bloom of his career, whereas Marie was a talented, young master student, just about to start her career.

To honor the memories of both *Hans Petter* and *Marie*, I dedicate this digital compendium to them.

Leif Rune Hellevik

Trondheim, October 19, 2017.

1.2 Check Python and LiClipse plugin

This section has the objective to verify that you have Python and the necessary Python Packages installed, and also that the Eclipse/LiClipse IDE plugin is

working. Download and run the code **systemCheck.py** in your Eclipse/LiClipse IDE. For an illustration on how to download the Eclipse/LiClipse plugin and how to use it see this video LiClipsePlugin.mp4 LiClipsePlugin.ogg

```
# src-ch0/systemCheck.py

def systemCheck():
    '''
    Check for necessary moduls needed in the course tkt4140:
        matplotlib
        numpy
        scipy
        sympy

    '''
    installed = " ... installed!"
    print ""
    print "Check for necessary modules needed for the course tkt4140"
    print ""

    print "check for matplotlib ",
    try:
        import matplotlib.pyplot
        print installed
    except:
        print " IMPORT ERROR; no version of matplotlib.pyplot found"


    print "check for numpy      ",
    try:
        import numpy
        print installed

    except:
        print " IMPORT ERROR; no version of numpy found"

    print "check for scipy      ",
    try:
        import scipy
        print installed
    except:
        print " IMPORT ERROR; no version of scipy found"


    print "check for sympy      ",
    try:
        import sympy
        print installed
    except:
        print " IMPORT ERROR; no version of sympy found"


if __name__ == '__main__':
    systemCheck()
```

1.3 Scientific computing with Python

In this course we will use the programming language **Python** to solve numerical problems. Students not familiar with Python are strongly recommended to work through the example Intro to scientific computing with Python before proceeding. If you are familiar with **Matlab** the transfer to Python should not be a problem.

Chapter 2

Initial value problems for Ordinary Differential Equations

2.1 Introduction

With an initial value problem for an ordinary differential equation (ODE) we mean a problem where all boundary conditions are given for one and the same value of the independent variable. For a first order ODE we get e.g.

$$y'(x) = f(x, y) \tag{2.1}$$
$$y(x_0) = a \tag{2.2}$$

while for a second order ODE we get

$$y''(x) = f(x, y, y') \tag{2.3}$$
$$y(x_0) = a,\ y'(x_0) = b \tag{2.4}$$

A first order ODE, as shown in Equation (2.2), will always be an *initial value problem*. For Equation (2.4), on the other hand, we can for instance specify the boundary conditions as follows,

$$y(x_0) = a,\ y(x_1) = b$$

With these boundary conditions Equation (2.4) presents a *boundary value problem*. In many applications boundary value problems are more common than initial value problems. But the solution technique for initial value problems may often be applied to solve boundary value problems.

Both from an analytical and numerical viewpoint initial value problems are easier to solve than boundary value problems, and methods for solution of initial value problems are more developed than for boundary value problems.

2.1.1 Example: A mathematical pendulum

Consider the problem of the mathematical pendulum (see Figure 2.1). By using Newton's second law in the θ-direction one can show that the following equation must be satisfied:

$$\frac{\partial^2\theta}{\partial\tau^2} + \frac{g}{l}\sin(\theta) = 0 \tag{2.5}$$

$$\theta(0) = \theta_0,\ \frac{d\theta}{d\tau}(0) = 0 \tag{2.6}$$

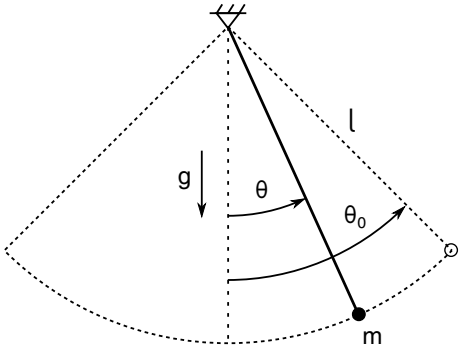

Figure 2.1: An illustration of the mathematical pendulum.

To present the governing equation (2.6) on a more convenient form, we introduce a dimensionless time t given by $t = \sqrt{\frac{g}{l}} \cdot \tau$ such that (2.5) and (2.6) may be written as:

$$\ddot{\theta}(t) + \sin(\theta(t)) = 0 \tag{2.7}$$

$$\theta(0) = \theta_0,\ \dot{\theta}(0) = 0 \tag{2.8}$$

The dot denotes derivation with respect to the dimensionless time t. For small displacements we can set $\sin(\theta) \approx \theta$, such that (2.7) and (2.8) becomes

$$\ddot{\theta}(t) + \theta(t) = 0 \tag{2.9}$$

$$\theta(0) = \theta_0,\ \dot{\theta}(0) = 0 \tag{2.10}$$

The difference between (2.7) and (2.9) is that the latter is linear, while the first is non-linear. The analytical solution of Equations (2.7) and (2.8) is given in Appendix G.2. in the Numeriske Beregninger.

2.1.2 n-th order linear ordinary differential equations

An n-th order linear ODE may be written on the generic form:

$$a_n(x)y^{(n)}(x) + a_{n-1}(x)y^{(n-1)}(x) + \cdots + a_1(x)y'(x) + a_0(x)y(x) = b(x) \quad (2.11)$$

where $y^{(k)}, k = 0, 1, \ldots n$ is referring to the k'th derivative and $y^{(0)}(x) = y(x)$.

If one or more of the coefficients a_k also are functions of at least one $y^{(k)},\ k = 0, 1, \ldots n$, the ODE is non-linear. From (2.11) it follows that (2.7) is non-linear and (2.9) is linear.

Analytical solutions of non-linear ODEs are rare, and except from some special types, there are no general ways of finding such solutions. Therefore non-linear equations must usually be solved numerically. In many cases this is also the case for linear equations. For instance it doesn't exist a method to solve the general second order linear ODE given by

$$a_2(x) \cdot y''(x) + a_1(x) \cdot y'(x) + a_0(x) \cdot y(x) = b(x)$$

From a numerical point of view the main difference between linear and non-linear equations is the multitude of solutions that may arise when solving non-linear equations. In a linear ODE it will be evident from the equation if there are special critical points where the solution changes character, while this is often not the case for non-linear equations.

For instance the equation $y'(x) = y^2(x),\ y(0) = 1$ has the solution $y(x) = \frac{1}{1-x}$ such that $y(x) \to \infty$ for $x \to 1$, which isn't evident from the equation itself.

2.2 Existence and uniqueness of solutions for initial value problems

If we are to solve an initial value problem of the type in Equation (2.2), we must first be sure that it has a solution and that the solution is unique. Such conditions are guaranteed by the following criteria:

The criteria for existence and uniqueness

Consider the domain $\mathcal{D} = \{(y, x) : |y - a| \leq c, x_0 \leq x \leq x_1\}$, with c an arbitrary constant. If $f(y, x)$ is Lipschitz continuous in y and continuous in x over $\mathcal{D}$, then there is a unique solution to the initial value problem (2.2)-(2.2) at least up to time $X^* = min(x_1, x_0 + c/S)$, where

$$S = \max_{(y,x)\in\mathcal{D}} |f(y, x)|.$$

We say that $f(y, x)$ is Lipschitz continuous in y over some domain $\mathcal{D}$ if there exists some constant $L \geq 0$ so that

$$|f(y, x) - f(y^*, x)| \leq L|y - y^*|$$

for all (y, x) and (y^*, x) in $\mathcal{D}$. Note that this requirement is more restrictive than plain continuity, which only requires that $|f(y, x) - f(y^*, x)| \to 0$ as $y \to y^*$. In fact, Lipschitz continuity requires that $|f(y, t) - f(y^*, t)| \to \mathcal{O}(|y - y^*|)$ as $y \to y^*$. To give a definition for L, consider that $f(y, x)$ is differentiable with respect to y in $\mathcal{D}$ and that its derivative $\frac{\partial f(y,x)}{\partial y}$ is bounded, then we can take

$$L = \max_{(y,x)\in\mathcal{D}} |\frac{\partial f(y, x)}{\partial y}|, \tag{2.12}$$

since

$$f(y, x) = f(y^*, x) + \frac{\partial f(v, x)}{\partial y}(y - y^*)$$

for some value v between y and y^*.

For (2.4), or higher order ODEs in general, one has to consider that such equations can be written as systems of first order ODEs. Similar criteria apply in that case to define the existence and uniqueness of solutions. However, special attention must be paid to the definition of Lipschitz continuity, in particular to the definition of appropriate bounds in terms of a given norm. This aspect will not be covered in this notes.

Fulfillment of the criteria for existence and uniqueness

Consider the initial value problem

$$y'(x) = (y(x))^2, \quad y(0) = a > 0.$$

It is clear that $f(y) = y^2$ is independent of x and Lipschitz continuous in y over any finite interval $|y - a| \leq c$ with $L = 2(a + c)$ and $S = (a + c)^2$. In this case, we can say that there is unique solution at least up to time $c/(a + c)^2$ and then take the value of c that maximizes that time, i.e. $c = a$, yielding a time of $1/(4a)$.

In this simple problem we can verify our considerations by looking at the exact solution

$$y(x) = \frac{1}{a^{-1} - x}.$$

It can be easily seen that $y(x) \to \infty$ as $x \to 1/a$, so that there is no solution beyond time $1/a$.

Violation of the criteria for existence and uniqueness

Consider the initial value problem

$$y'(x) = \sqrt{y(x)}, \quad y(0) = 0.$$

In this case we have that $f(y) = \sqrt{y}$ is not Lipschitz continuous near $y = 0$ since $f'(y) = 1/(2\sqrt{y}) \to \infty$ as $y \to 0$. Therefore, we can not find a constant L so that the bound in (2.12) holds for y and y^* near 0. This consideration implies that this initial value problem does not have an unique solution. In fact, it has two solutions:

$$y(x) = 0$$

and

$$y(x) = \frac{1}{4}x^2.$$

2.3 Taylor's method

Taylor's formula for series expansion of a function $f(x)$ around x_0 is given by

$$f(x) = f(x_0)+(x-x_0)\cdot f'(x_0)+\frac{(x-x_0)^2}{2}f''(x_0)+\cdots+\frac{(x-x_0)^n}{n!}f^{(n)}(x_0)+\text{higher order terms} \tag{2.13}$$

Let's use this formula to find the first terms in the series expansion for $\theta(t)$ around $t = 0$ from the differential equation given in (2.9):

$$\ddot{\theta}(t) + \theta(t) = 0 \tag{2.14}$$

$$\theta(0) = \theta_0,\ \dot{\theta}(0) = 0 \tag{2.15}$$

First we observe that the solution to the ODE in (2.15) may be expressed as an Taylor expansion around the initial point:

$$\theta(t) \approx \theta(0) + t\cdot\dot{\theta}(0) + \frac{t^2}{2}\ddot{\theta}(0) + \frac{t^3}{6}\dddot{\theta}(0) + \frac{t^4}{24}\theta^{(4)}(0) \tag{2.16}$$

By use of the initial conditions of the ODE in (2.15) $\theta(0) = \theta_0,\ \dot{\theta}(0) = 0$ we get

$$\theta(t) \approx \theta_0 + \frac{t^2}{2}\ddot{\theta} + \frac{t^3}{6}\dddot{\theta}(0) + \frac{t^4}{24}\theta^{(4)}(0) \tag{2.17}$$

From the ODE in (2.15) we obtain expressions for the differentials at the initial point:

$$\ddot{\theta}(t) = -\theta(t) \rightarrow \ddot{\theta}(0) = -\theta(0) = -\theta_0 \tag{2.18}$$

Expressions for higher order differentials evaluated at the initial point may be obtained by further differentiation of the ODE (2.15)

$$\dddot{\theta}(t) = -\dot{\theta}(t) \rightarrow \ddot{\theta}(0) = -\theta(0) = -\theta_0 \tag{2.19}$$

and

$$\theta^{(4)}(t) = -\ddot{\theta}(t) \rightarrow \theta^{(4)}(0) = -\ddot{\theta}(0) = \theta_0$$

Substitution of these differentials into (2.17) yields

$$\theta(t) \approx \theta_0\left(1 - \frac{t^2}{2} + \frac{t^4}{24}\right) = \theta_0\left(1 - \frac{t^2}{2!} + \frac{t^4}{4!}\right) \tag{2.20}$$

If we include n terms, we get

$$\theta(t) \approx \theta_0\cdot\left(1 - \frac{t^2}{2!} + \frac{t^4}{4!} - \frac{t^6}{6!} + \cdots + (-1)^n\frac{t^{2n}}{(2n)!}\right)$$

If we let $n \to \infty$ we see that the parentheses give the series for $\cos(t)$. In this case we have found the exact solution $\theta(t) = \theta_0 \cos(t)$ of the differential equation. Since this equation is linear we manage in this case to find a connection between the coefficients such that we recognize the series expansion of $\cos(t)$.

Taylor's Method for solution of initial value ODE..

1. Use the ODE (e.g. (2.2) or (2.4)) to evaluate the differentials at the initial value.
2. Obtain an approximate solution of the ODE by substitution of the differentials in the Taylor expansion (2.13).
3. Differentiate the ODE as many times needed to obtain the wanted accuracy.

2.3.1 Example: Taylor's method for the non-linear mathematical pendulum

Let's try the same procedure on the non-linear version (2.7)

$$\ddot{\theta}(t) + \sin(\theta(t)) = 0$$
$$\theta(0) = \theta_0,\ \dot{\theta}(0) = 0$$

We start in the same manner: $\theta(t) \approx \theta(0) + \frac{t^2}{2}\ddot{\theta}(0) + \frac{t^3}{6}\dddot{\theta}(0) + \frac{t^4}{24}\theta^{(4)}(0)$. From the differential equation we have $\ddot{\theta} = -\sin(\theta) \to \ddot{\theta}(0) = -\sin(\theta_0)$, which by consecutive differentiation gives

$$\dddot{\theta} = -\cos(\theta) \cdot \dot{\theta} \to \dddot{\theta}(0) = 0$$
$$\theta^{(4)} = \sin(\theta) \cdot \dot{\theta}^2 - \cos(\theta) \cdot \ddot{\theta} \to \theta^{(4)}(0) = -\ddot{\theta}(0)\cos(\theta(0)) = \sin(\theta_0)\cos(\theta_0)$$

Inserted above: $\theta(t) \approx \theta_0 - \frac{t^2}{2}\sin(\theta_0) + \frac{t^4}{24}\sin(\theta_0)\cos(\theta_0)$.

We may include more terms, but this complicates the differentiation and it is hard to find any connection between the coefficients. When we have found an approximation for $\theta(t)$ we can get an approximation for $\dot{\theta}(t)$ by differentiation: $\dot{\theta}(t) \approx -t\sin(\theta_0) + \frac{t^3}{8}\sin(\theta_0)\cos(\theta_0)$.

Series expansions are often useful around the starting point when we solve initial value problems. The technique may also be used on non-linear equations.

Symbolic mathematical programs like **Maple** and **Mathematica** do this easily.

2.3.2 Example: Newton's first differential equation

We will end with one of the earliest known differential equations, which Newton solved with series expansion in 1671.

$$y'(x) = 1 - 3x + y + x^2 + xy,\ y(0) = 0$$

Series expansion around $x = 0$ gives

$$y(x) \approx x \cdot y'(0) + \frac{x^2}{2}y''(0) + \frac{x^3}{6}y'''(0) + \frac{x^4}{24}y^{(4)}(0)$$

From the differential equation we get $y'(0) = 1$. By consecutive differentiation we get

$$\begin{array}{rclcrcl} y''(x) & = & -3 + y' + 2x + xy' + y & \rightarrow & y''(0) & = & -2 \\ y'''(x) & = & y'' + 2 + xy'' + 2y' & \rightarrow & y'''(0) & = & 2 \\ y^{(4)}(x) & = & y''' + xy''' + 3y'' & \rightarrow & y^{(4)}(0) & = & -4 \end{array}$$

Inserting above gives $y(x) \approx x - x^2 + \frac{x^3}{3} - \frac{x^4}{6}$.

Newton gave the following solution: $y(x) \approx x - x^2 + \frac{x^3}{3} - \frac{x^4}{6} + \frac{x^5}{30} - \frac{x^6}{45}$.

Now you can check if Newton calculated correctly. Today it is possible to give the solution on closed form with known functions as follows,

$$\begin{aligned} y(x) =& 3\sqrt{2\pi e} \cdot \exp\left[x\left(1 + \frac{x}{2}\right)\right] \cdot \left[\operatorname{erf}\left(\frac{\sqrt{2}}{2}(1+x)\right) - \operatorname{erf}\left(\frac{\sqrt{2}}{2}\right)\right] \\ &+ 4 \cdot \left[1 - \exp[x\left(1 + \frac{x}{2}\right)\right] - x \end{aligned}$$

Note the combination $\sqrt{2\pi e}$. See Hairer et al. [6] section 1.2 for more details on classical differential equations.

2.4 Reduction of Higher order Equations

When we are solving initial value problems, we usually need to write these as sets of first order equations, because most of the program packages require this.

Example 1:

$$y''(x) + y(x) = 0,\ y(0) = a_0,\ y'(0) = b_0$$

We may for instance write this equation in a system as follows,

$$\begin{aligned} y'(x) =& g(x) \\ g'(x) =& -y(x) \\ y(0) =& a_0,\ g(0) = b_0 \end{aligned}$$

Example 2:
Another of a third order ODE is:

$$\begin{aligned} &y'''(x) + 2y''(x) - (y'(x))^2 + 2y(x) = x^2 \\ &y(0) = a_0,\ y'(0) = b_0,\ y''(0) = c_0 \end{aligned} \tag{2.21}$$

We set $y'(x) = g(x)$ and $y''(x) = g'(x) = f(x)$, and the system may be written as

$$\begin{aligned} y'(x) =& g(x) \\ g'(x) =& f(x) \\ f'(x) =& -2f(x) + (g(x))^2 - 2y(x) + x^2 \end{aligned}$$

with initial values $y(0) = a_0,\ g(0) = b_0,\ f(0) = c_0$.

This is fair enough for hand calculations, but when we use program packages a more systematic procedure is needed. Let's use the equation above as an example.

We start by renaming y to y_0. We then get the following procedure:

$$\begin{aligned} y' &= y_0' = y_1 \\ y'' &= y_0'' = y_1' = y_2 \end{aligned}$$

Finally, the third order ODE in (2.21) may be represented as a system of first order ODEs:

$$\begin{aligned} y_0'(x) =& y_1(x) \\ y_1'(x) =& y_2(x) \\ y_2'(x) =& -2y_2(x) + (y_1(x))^2 - 2y_0(x) + x^2 \end{aligned}$$

with initial conditions $y_0(0) = a_0,\ y_1(0) = b_0,\ y_2(0) = c_0$.

General procedure to reduce a higher order ODE to a system of first order ODEs.

The general procedure to reduce a higher order ODE to a system of first order ODEs becomes the following:

Given the equation

$$\begin{aligned} y^{(m)} &= f(x, y, y', y'', \ldots, y^{(m-1)}) \\ y(x_0) &= a_0, y'(x_0) = a_1, \ldots, y^{(m-1)}(x_0) = a_{m-1} \end{aligned} \tag{2.22}$$

where

$$y^{(m)} \equiv \frac{d^m y}{dx^m}$$

with $y = y_0$, we get the following system of ODEs:

$$\begin{aligned} y_0' &= y_1 \\ y_1' &= y_2 \\ &\cdot \\ &\cdot \\ y_{m-2}' &= y_{m-1} \\ y_{m-1}' &= f(x, y_0, y_1, y_2, \ldots, y_{m-1}) \end{aligned} \tag{2.23}$$

with the following boundary conditions:

$$y_0(x_0) = a_0, y_1(x_0) = a_1, \ldots, y_{m-1}(x_0) = a_{m-1}$$

2.4.1 Example: Reduction of higher order systems

Write the following ODE as a system of first order ODEs:

$$\begin{aligned} y''' - y'y'' - (y')^2 + 2y &= x^3 \\ y(0) = a,\ y'(0) = b,\ y''(0) &= c \end{aligned}$$

First we write $y''' = y'y'' + (y')^2 - 2y + x^3$.
By use of (2.23) we get

$$\begin{aligned} y_0' &= y_1 \\ y_1' &= y_2 \\ y_2' &= y_1 y_2 + (y_1)^2 - 2y_0 + x^3 \\ y_0(0) = a,\ y_1(0) &= b,\ y_2 = c \end{aligned}$$

2.5 Differences

We will study some simple methods to solve initial value problems. Later we shall see that these methods also may be used to solve boundary value problems for ODEs.

For this purpose we need to introduce a suitable notation to enable us to formulate the methods in a convenient manner. In Figure 2.2) an arbitrary function y is illustrated as a continuous function of x, ie $y = y(x)$. Later y will be used to represent the solution of an ODE, which only may be represented and obtained for discrete values of x. We represent these discrete, equidistant values of x by:

$$x_j = x_0 + jh$$

where $h = \Delta x$ is assumed constant unless otherwise stated and j is an integer counter referring to the discrete values x_j for which the corresponding discrete value

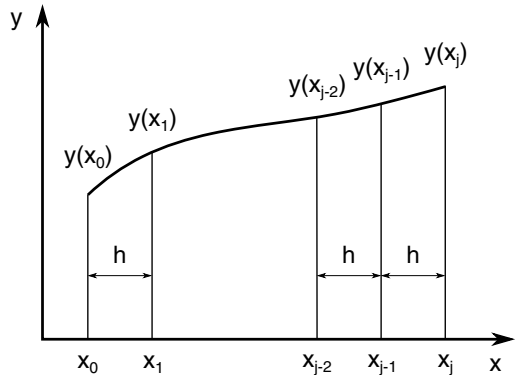

Figure 2.2: A generic function $y(x)$ sampled at equidistant values of x. .

$$y_j = y(x_j)$$

may be found (see Figure 2.3).

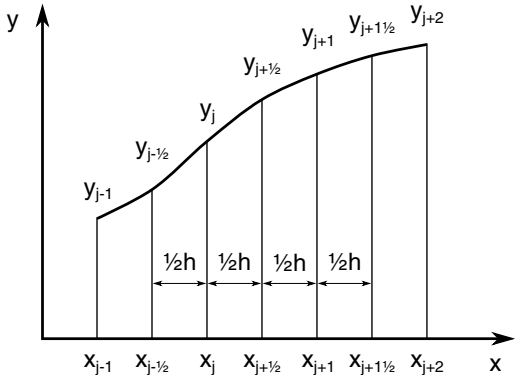

Figure 2.3: Illustration of how to obtain difference equations.

Having introduced this notation we may the develop useful expressions and notations for forward differences, backward differences, and central differences, which will be used frequently later:

Forward differences:

$$\Delta y_j = y_{j+1} - y_j$$

Backward differences:

$$\nabla y_j = y_j - y_{j-1} \tag{2.24}$$

Central differences:

$$\delta y_{j+\frac{1}{2}} = y_{j+1} - y_j$$

The linear difference operators Δ, ∇ and δ are useful when we are deriving more complicated expressions. An example of usage is as follows,

$$\delta^2 y_j = \delta(\delta y_j) = \delta(y_{1+\frac{1}{2}} - y_{1-\frac{1}{2}}) = y_{j+1} - y_j - (y_j - y_{j-1}) = y_{j+1} - 2y_j + y_{j-1}$$

However, for clarity we will mainly write out the formulas entirely rather than using operators.

We shall find difference formulas and need again:

Taylor's theorem:

$$\begin{aligned} y(x) =& y(x_0) + y'(x_0)\cdot(x-x_0) + \frac{1}{2}y''(x_0)\cdot(x-x_0)^2 + \\ & \cdots + \frac{1}{n!}y^{(n)}(x_0)\cdot(x-x_0)^n + R_n \end{aligned} \tag{2.25}$$

The remainder R_n is given by

$$R_n = \frac{1}{(n+1)!}y^{(n+1)}(\xi)\cdot(x-x_0)^{n+1} \tag{2.26}$$

where $\xi \in (x_0, x)$

Now, Taylor's theorem (2.25) may be used to approximate the value of $y(x_{j+1})$, i.e. the forward value of $y(x_j)$, by assuming that $y(x_j)$ and it's derivatives are known:

$$\begin{aligned} y(x_{j+1}) \equiv& y(x_j + h) = y(x_j) + hy'(x_j) + \frac{h^2}{2}y''(x_j) + \\ & \cdots + \frac{h^n y^{(n)}(x_j)}{n!} + R_n \end{aligned} \tag{2.27}$$

where the remainder $R_n = O(h^{n+1}),\ h \to 0$.

From (2.27) we also get

$$y(x_{j-1}) \equiv y(x_j - h) = y(x_j) - hy'(x_j) + \frac{h^2}{2}y''(x_j) + \cdots + \frac{h^k(-1)^k y^{(k)}(x_j)}{k!} + \ldots \tag{2.28}$$

In the following we will assume that h is positive. By solving (2.27) with respect to y' we may obtain a discrete forward difference approximation of y' at x_j:

$$y'(x_j) = \frac{y(x_{j+1}) - y(x_j)}{h} + O(h) \tag{2.29}$$

By solving (2.28) with respect to y' we obtain a discrete approximation at x_i:

$$y'(x_j) = \frac{y(x_j) - y(x_{j-1})}{h} + O(h) \tag{2.30}$$

By adding (2.28) and (2.27) together we get an approximation of the second derivative at the location x_j:

$$y''(x_j) = \frac{y(x_{j+1}) - 2y(x_j) + y(x_{j-1})}{h^2} + O(h^2) \tag{2.31}$$

By subtraction of (2.28) from (2.27) we get a backward difference approximation of the first order derivative at the location x_j:

$$y'(x_j) = \frac{y(x_{j+1}) - y(x_{j-1})}{2h} + O(h^2) \tag{2.32}$$

Notation:
We let $y(x_j)$ always denote the function $y(x)$ with $x = x_j$. We use y_j both for the numerical and analytical value, the intended meaning is hopfullye clear from the context.

Equations (2.29), (2.30), (2.31) and (2.32) then may be used to deduce the following difference expressions:

$$y'_j = \frac{y_{j+1} - y_j}{h} \ ; \text{ truncation error } O(h) \tag{2.33}$$

$$y'_j = \frac{y_j - y_{j-1}}{h} \ ; \text{ truncation error } O(h) \tag{2.34}$$

$$y''_j = \frac{y_{j+1} - 2y_j + y_{j-1}}{h^2} \ ; \text{ truncation error } O(h^2) \tag{2.35}$$

$$y'_j = \frac{y_{j+1} - y_{j-1}}{2h} \ ; \text{ truncation error } O(h^2) \tag{2.36}$$

In summary, equation (2.33) is a forward difference, (2.34) is a backward difference while (2.35) and (2.36) are central differences.

The expressions in (2.33), (2.34), (2.35) and (2.36) may also conveniently be established from Figure 2.4.

Whereas, (2.33) follows directly from the definition of the derivative, whereas the second order derivative (2.35) may be obtained as a derivative of the derivative by:

$$y''_j(x_j) = \left(\frac{y_{j+1} - y_j}{h} - \frac{y_j - y_{j-1}}{h} \right) \cdot \frac{1}{h} = \frac{y_{j+1} - 2y_j + y_{j-1}}{h^2}$$

and an improved expression for the derivative (2.36) may be obtained by averaging the forward and the backward derivatives:

$$y'_j = \left(\frac{y_{j+1} - y_j}{h} + \frac{y_j - y_{j-1}}{h} \right) \cdot \frac{1}{2} = \frac{y_{j+1} - y_{j-1}}{2h}$$

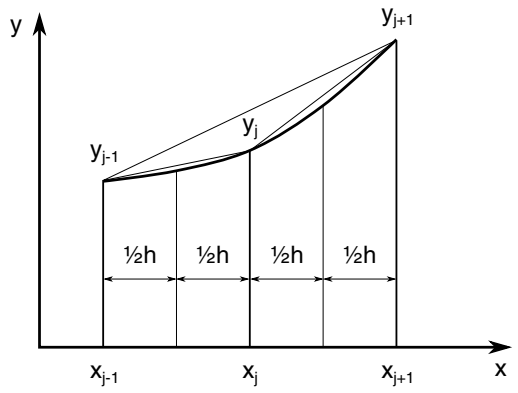

Figure 2.4: Illustration of how the discrete values may be used to estimate various orders of derivatives at location x_i.

To find the truncation error we proceed in a systematical manner by:

$$y'(x_j) = a \cdot y(x_{j-1}) + b \cdot y(x_j) + c \cdot y(x_{j+1}) + O(h^m) \tag{2.37}$$

where we shall determine the constants a, b and c together with the error term. For simplicity we use the notation $y_j \equiv y(x_j)$, $y'_j \equiv y'(x_j)$ and so on. From the Taylor series expansion in (2.27) and (2.28) we get

$$\begin{aligned} &a \cdot y_{j-1} + b \cdot y_j + c \cdot y_{j+1} = \\ &a \cdot \left[y_j - h y'_j + \frac{h^2}{2} y''_j - \frac{h^3}{6} y'''_j(\xi) \right] + b \cdot y_j + \\ &c \cdot \left[y_j + h y'_j + \frac{h^2}{2} y''_j + \frac{h^3}{6} y'''_j(\xi) \right] \end{aligned} \tag{2.38}$$

By collecting terms in Eq. (2.38) we get:

$$\begin{aligned} a \cdot y_{j-1} + b \cdot y_j + c \cdot y_{j+1} = \\ (a+b+c)\, y_j + (c-a)\, h\, y'_j + \\ (a+c)\, \frac{h^2}{2} y''_j + (c-a)\, \frac{h^3}{6}\, y'''(\xi) \end{aligned} \tag{2.39}$$

From Eq.(2.39) we may then find a, b and c such that y'_j gets as high accuracy as possible by :

$$\begin{aligned} a + b + c &= 0 \\ (c-a) \cdot h &= 1 \\ a + c &= 0 \end{aligned} \tag{2.40}$$

The solution to (2.40) is

$$a = -\frac{1}{2h},\ b = 0 \text{ and } c = \frac{1}{2h}$$

which when inserted in (2.37) gives

$$y'_j = \frac{y_{j+1} - y_{j-1}}{2h} - \frac{h^2}{6}y'''(\xi) \tag{2.41}$$

By comparison of (2.41) with (2.37) we see that the error term is $O(h^m) = -\frac{h^2}{6}y'''(\xi)$, which means that $m = 2$. As expected, (2.41) is identical to (2.32).

Let's use this method to find a forward difference expression for $y'(x_j)$ with accuracy of $O(h^2)$. Second order accuracy requires at least three unknown coefficients. Thus,

$$y'(x_j) = a \cdot y_j + b \cdot y_{j+1} + c \cdot y_{j+2} + O(h^m) \tag{2.42}$$

The procedure goes as in the previous example as follows,

$$\begin{aligned}
&a \cdot y_j + b \cdot y_{j+1} + c \cdot y_{j+2} = \\
&a \cdot y_j + b \cdot \left[y_j + hy'_j + \frac{h^2}{2}y''_j + \frac{h^3}{6}y'''(\xi) \right] + \\
&c \cdot \left[y_j + 2hy'_j + 2h^2y''_j + \frac{8h^3}{6}y'''_j(\xi) \right] \\
=&(a + b + c) \cdot y_j + (b + 2c) \cdot hy'_j \\
+&h^2 \left(\frac{b}{2} + 2c \right) \cdot y''_j + \frac{h^3}{6}(b + 8c) \cdot y'''(\xi)
\end{aligned}$$

We determine a, b and c such that y'_j becomes as accurate as possible. Then we get,

$$\begin{aligned}
a + b + c &= 0 \\
(b + 2c) \cdot h &= 1 \\
\frac{b}{2} + 2c &= 0
\end{aligned} \tag{2.43}$$

The solution of (2.43) is

$$a = -\frac{3}{2h},\ b = \frac{2}{h},\ c = -\frac{1}{2h}$$

which inserted in (2.42) gives

$$y'_j = \frac{-3y_j + 4y_{j+1} - y_{j+2}}{2h} + \frac{h^2}{3}y'''(\xi) \tag{2.44}$$

The error term $O(h^m) = \frac{h^2}{3}y'''(\xi)$ shows that $m = 2$.

Here follows some difference formulas derived with the procedure above:

Forward differences:

$$\frac{dy_i}{dx} = \frac{y_{i+1} - y_i}{\Delta x} + \frac{1}{2}y''(\xi)\Delta x$$
$$\frac{dy_i}{dx} = \frac{-3y_i + 4y_{i+1} - y_{i+2}}{2\Delta x} + \frac{1}{3}y'''(\xi)\cdot(\Delta x)^2$$
$$\frac{dy_i}{dx} = \frac{-11y_i + 18y_{i+1} - 9y_{i+2} + y_{i+3}}{6\Delta x} + \frac{1}{4}y^{(4)}(\xi)\cdot(\Delta x)^3$$
$$\frac{d^2y_i}{dx^2} = \frac{y_i - 2y_{i+1} + y_{i+2}}{(\Delta x)^2} + y'''(\xi)\cdot\Delta x$$
$$\frac{d^2y_i}{dx^2} = \frac{2y_i - 5y_{i+1} + 4y_{i+2} - y_{i+3}}{(\Delta x)^2} + \frac{11}{12}y^{(4)}(\xi)\cdot(\Delta x)^2$$

Backward differences:

$$\frac{dy_i}{dx} = \frac{y_i - y_{i-1}}{\Delta x} + \frac{1}{2}y''(\xi)\Delta x$$
$$\frac{dy_i}{dx} = \frac{3y_i - 4y_{i-1} + y_{i-2}}{2\Delta x} + \frac{1}{3}y'''(\xi)\cdot(\Delta x)^2$$
$$\frac{dy_i}{dx} = \frac{11y_i - 18y_{i-1} + 9y_{i-2} - y_{i-3}}{6\Delta x} + \frac{1}{4}y^{(4)}(\xi)\cdot(\Delta x)^3$$
$$\frac{d^2y_i}{dx^2} = \frac{y_i - 2y_{i-1} + y_{i-2}}{(\Delta x)^2} + y'''(\xi)\cdot\Delta x$$
$$\frac{d^2y_i}{dx^2} = \frac{2y_i - 5y_{i-1} + 4y_{i-2} - y_{i-3}}{(\Delta x)^2} + \frac{11}{12}y^{(4)}(\xi)\cdot(\Delta x)^2$$

Central differences:

$$\frac{dy_i}{dx} = \frac{y_{i+1} - y_{i-1}}{2\Delta x} - \frac{1}{6}y'''(\xi)(\Delta x)^2$$
$$\frac{dy_i}{dx} = \frac{-y_{i+2} + 8y_{i+1} - 8y_{i-1} + y_{i-2}}{12\Delta x} + \frac{1}{30}y^{(5)}(\xi)\cdot(\Delta x)^4$$
$$\frac{d^2y_i}{dx^2} = \frac{y_{i+1} - 2y_i + y_{i-1}}{(\Delta x)^2} - \frac{1}{12}y^{(4)}(\xi)\cdot(\Delta x)^2$$
$$\frac{d^2y_i}{dx^2} = \frac{-y_{i+2} + 16y_{i+1} - 30y_i + 16y_{i-1} - y_{i-2}}{12(\Delta x)^2} + \frac{1}{90}y^{(6)}(\xi)\cdot(\Delta x)^4$$
$$\frac{d^3y_i}{dx^3} = \frac{y_{i+2} - 2y_{i+1} + 2y_{i-1} - y_{i-2}}{2(\Delta x)^3} + \frac{1}{4}y^{(5)}(\xi)\cdot(\Delta x)^2$$

2.5.1 Example: Discretization of a diffusion term

This diffusion term $\frac{d}{dx}\left(p(x)\frac{d}{dx}u(x)\right)$ often appears in difference equations, and it may be beneficial to treat the term as it is instead of first execute the differentiation. Below we will illustrate how the diffusion term may be discretized with the three various difference approaches: forward, backward and central differences.

Central differences: We use central differences (recall Figure 2.3) as follows,

$$\begin{aligned}\frac{d}{dx}\left(p(x)\cdot\frac{d}{dx}u(x)\right)\Bigg|_i \approx& \frac{[p(x)\cdot u'(x)]|_{i+\frac{1}{2}} - [p(x)\cdot u'(x)]|_{i-\frac{1}{2}}}{h}\\ =& \frac{p(x_{i+\frac{1}{2}})\cdot u'(x_{i+\frac{1}{2}}) - p(x_{i-\frac{1}{2}})\cdot u'(x_{i-\frac{1}{2}})}{h}\end{aligned}$$

Using central differences again, we get

$$u'(x_{i+\frac{1}{2}}) \approx \frac{u_{i+1}-u_i}{h},\ u'(x_{i-\frac{1}{2}}) \approx \frac{u_i - u_{i-1}}{h},$$

which inserted in the previous equation gives the final expression

$$\frac{d}{dx}\left(p(x)\cdot\frac{d}{dx}u(x)\right)\Bigg|_i \approx \frac{p_{i-\frac{1}{2}}\cdot u_{i-1} - (p_{i+\frac{1}{2}}+p_{i-\frac{1}{2}})\cdot u_i + p_{i+\frac{1}{2}}\cdot u_{i+1}}{h^2} + \text{error term} \tag{2.45}$$

where

$$\text{error term} = -\frac{h^2}{24}\cdot\frac{d}{dx}\left(p(x)\cdot u'''(x) + [p(x)\cdot u'(x)]''\right) + O(h^3)$$

If $p(x_{1+\frac{1}{2}})$ and $p(x_{1-\frac{1}{2}})$ cannot be found directly, we use

$$p(x_{1+\frac{1}{2}}) \approx \frac{1}{2}(p_{i+1}+p_i),\ p(x_{1-\frac{1}{2}}) \approx \frac{1}{2}(p_i+p_{i-1}) \tag{2.46}$$

Note that for $p(x) = 1 =$ constant we get the usual expression

$$\left.\frac{d^2u}{dx^2}\right|_i = \frac{u_{i+1}-2u_i+u_{i-1}}{h^2} + O(h^2)$$

Forward differences: We start with

$$\begin{aligned}\frac{d}{dx}\left(p(x)\cdot\frac{du}{dx}\right)\Bigg|_i \approx& \frac{p(x_{i+\frac{1}{2}})\cdot u'(x_{i+\frac{1}{2}}) - p(x_i)\cdot u'(x_i)}{\frac{h}{2}}\\ \approx& \frac{p(x_{i+\frac{1}{2}})\cdot\left(\frac{u_{i+1}-u_i}{h}\right) - p(x_i)\cdot u'(x_i)}{\frac{h}{2}}\end{aligned}$$

which gives

$$\left.\frac{d}{dx}\left(p(x)\cdot\frac{du}{dx}\right)\right|_i = \frac{2\cdot[p(x_{i+\frac{1}{2}})\cdot(u_{i+1}-u_i)-h\cdot p(x_i)\cdot u'(x_i)]}{h^2} + \text{error term} \tag{2.47}$$

where

$$\text{error term} = -\frac{h}{12}[3p''(x_i)\cdot u'(x_i)+6p'(x_i)\cdot u''(x_i)+4p(x_i)\cdot u'''(x_i)]+O(h^2) \tag{2.48}$$

We have kept the term $u'(x_i)$ since (2.47) usually is used at the boundary, and $u'(x_i)$ may be prescribed there. For $p(x)=1=$ constant we get the expression

$$u_i'' = \frac{2\cdot[u_{i+1}-u_i-h\cdot u'(x_i)]}{h^2} - \frac{h}{3}u'''(x_i)+O(h^2) \tag{2.49}$$

Backward Differences: We start with

$$\begin{aligned}\left.\frac{d}{dx}\left(p(x)\frac{du}{dx}\right)\right|_i &\approx \frac{p(x_i)\cdot u'(x_i)-p(x_i)\cdot u'(x_{i-\frac{1}{2}})\cdot u'(x_{i-\frac{1}{2}})}{\frac{h}{2}}\\ &\approx \frac{p(x_i)\cdot u'(x_i)-p(x_{i-\frac{1}{2}})\left(\frac{u_i-u_{i-1}}{h}\right)}{\frac{h}{2}}\end{aligned}$$

which gives

$$\left.\frac{d}{dx}\left(p(x)\frac{du}{dx}\right)\right|_i = \frac{2\cdot[h\cdot p(x_i)u'(x_i)-p(x_{i-\frac{1}{2}})\cdot(u_i-u_{i-1})]}{h^2} + \text{error term} \tag{2.50}$$

where

$$\text{error term} = \frac{h}{12}[3p''(x_i)\cdot u'(x_i)+6p'(x_i)\cdot u''(x_i)+4p(x_i)\cdot u'''(x_i)]+O(h^2) \tag{2.51}$$

This is the same error term as in (2.48) except from the sign. Also here we have kept the term $u'(x_i)$ since (2.51) usually is used at the boundary where $u'(x_i)$ may be prescribed. For $p(x)=1=$ constant we get the expression

$$u_i'' = \frac{2\cdot[h\cdot u'(x_i)-(u_i-u_{i-1})]}{h^2} + \frac{h}{3}u'''(x_i)+O(h^2) \tag{2.52}$$

2.6 Euler's method

The ODE is given as

$$\frac{dy}{dx} = y'(x) = f(x,y) \tag{2.53}$$

$$y(x_0) = y_0 \tag{2.54}$$

By using a first order forward approximation (2.29) of the derivative in (2.53) we obtain:

$$y(x_{n+1}) = y(x_n) + h \cdot f(x_n, y(x_n)) + O(h^2)$$

or

$$y_{n+1} = y_n + h \cdot f(x_n, y_n) \tag{2.55}$$

(2.55) is a difference equation and the scheme is called **Euler's method** (1768). The scheme is illustrated graphically in Figure 2.5. Euler's method is a first order method, since the expression for $y'(x)$ is first order of h. The method has a global error of order h, and a local of order h^2.

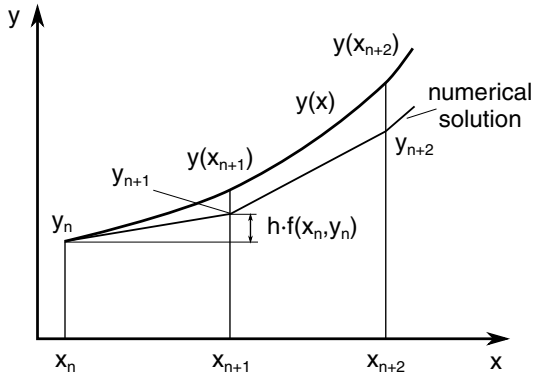

Figure 2.5: Graphical illustration of Euler's method.

2.6.1 Example: Eulers method on a simple ODE

```
# src-ch1/euler_simple.py

import numpy as np
import matplotlib.pylab as plt

""" example using eulers method for solving the ODE
    y'(x) = f(x, y) = y
    y(0) = 1

    Eulers method:
    y^(n + 1) = y^(n) + h*f(x, y^(n)), h = dx
    """

N = 30
x = np.linspace(0, 1, N + 1)
h = x[1] - x[0] # steplength
y_0 = 1 # initial condition
Y = np.zeros_like(x) # vector for storing y values
Y[0] = y_0 # first element of y = y(0)

for n in range(N):
```

```
    f = Y[n]
    Y[n + 1] = Y[n] + h*f

Y_analytic = np.exp(x)

# change default values of plot to make it more readable
LNWDT=2; FNT=15
plt.rcParams['lines.linewidth'] = LNWDT
plt.rcParams['font.size'] = FNT

plt.figure()
plt.plot(x, Y_analytic, 'b', linewidth=2.0)
plt.plot(x, Y, 'r--', linewidth=2.0)
plt.legend(['$e^x$', 'euler'], loc='best', frameon=False)
plt.xlabel('x')
plt.ylabel('y')
#plt.savefig('../fig-ch1/euler_simple.png', transparent=True)
plt.show()
```

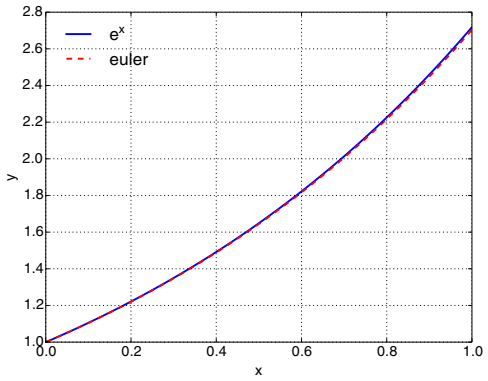

Figure 2.6: result from the code above

2.6.2 Example: Eulers method on the mathematical pendulum

```
# src-ch1/euler_pendulum.py

import numpy as np
import matplotlib.pylab as plt
from math import pi

""" example using eulers method for solving the ODE:
    theta''(t) + thetha(t) = 0
    thetha(0) = theta_0
    thetha'(0) = dtheta_0

    Reduction of higher order ODE:
    theta = y0
    theta' = y1
```

```
    theta'' = - theta = -y0

    y0' = y1
    y1' = -y0

    eulers method:
    y0^(n + 1) = y0^(n) + h*y1, h = dt
    y1^(n + 1) = y1^(n) + h*(-y0), h = dt
    """

N = 100
t = np.linspace(0, 2*pi, N + 1)
h = t[1] - t[0] # steplength
thetha_0 = 0.1
y0_0 = thetha_0 # initial condition
y1_0 = 0
Y = np.zeros((2, N + 1)) # 2D array for storing y values

Y[0, 0] = y0_0 # apply initial conditions
Y[1, 0] = y1_0

for n in range(N):
    y0_n = Y[0, n]
    y1_n = Y[1, n]

    Y[0, n + 1] = y0_n + h*y1_n
    Y[1, n + 1] = y1_n - h*y0_n

thetha = Y[0, :]
thetha_analytic = thetha_0*np.cos(t)

# change default values of plot to make it more readable
LNWDT=2; FNT=15
plt.rcParams['lines.linewidth'] = LNWDT
plt.rcParams['font.size'] = FNT

plt.figure()
plt.plot(t, thetha_analytic, 'b')
plt.plot(t, thetha, 'r--')

plt.legend([r'$\theta_0 \cdot cos(t)$', 'euler'], loc='best', frameon=False)
plt.xlabel('t')
plt.ylabel(r'$\theta$')
#plt.savefig('../fig-ch1/euler_pendulum.png', transparent=True)
plt.show()
```

2.6.3 Example: Generic euler implementation on the mathematical pendulum

```
# src-ch1/euler_pendulum_generic.py

import numpy as np
import matplotlib.pylab as plt
from math import pi

# define Euler solver
def euler(func, y_0, time):
    """ Generic implementation of the euler scheme for solution of systems of ODEs:
```

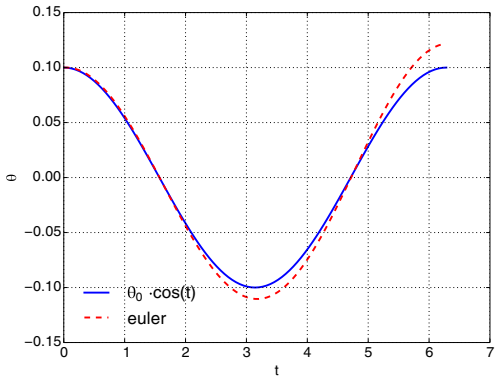

Figure 2.7: result from the code above

```
            y0' = y1
            y1' = y2
                .
                .
            yN' = f(yN-1,..., y1, y0, t)

            method:
            y0^(n+1) = y0^(n) + h*y1
            y1^(n+1) = y1^(n) + h*y2
                .
                .
            yN^(n + 1) = yN^(n) + h*f(yN-1, .. y1, y0, t)

        Args:
            func(function): func(y, t) that returns y' at time t; [y1, y2,...,f(yn-1, .. y1, y0, t)]
            y_0(array): initial conditions
            time(array): array containing the time to be solved for

        Returns:
            y(array): array/matrix containing solution for y0 -> yN for all timesteps"""

    y = np.zeros((np.size(time), np.size(y_0)))
    y[0,:] = y_0

    for i in range(len(time)-1):
        dt = time[i+1] - time[i]
        y[i+1,:]=y[i,:] + np.asarray(func(y[i,:], time[i]))*dt

    return y

def pendulum_func(y, t):
    """ function that returns the RHS of the mathematcal pendulum ODE:
        Reduction of higher order ODE:
        theta = y0
        theta' = y1
        theta'' = - theta = -y0

        y0' = y1
```

```
        y1' = -y0

        Args:
            y(array): array [y0, y1] at time t
            t(float): current time

        Returns:
            dy(array): [y0', y1'] = [y1, -y0]
        """

    dy = np.zeros_like(y)
    dy[:] = [y[1], -y[0]]
    return dy

N = 100
time = np.linspace(0, 2*pi, N + 1)
thetha_0 = [0.1, 0]

theta = euler(pendulum_func, thetha_0, time)
thetha = theta[:, 0]

thetha_analytic = thetha_0[0]*np.cos(time)

# change default values of plot to make it more readable
LNWDT=2; FNT=15
plt.rcParams['lines.linewidth'] = LNWDT
plt.rcParams['font.size'] = FNT

plt.figure()
plt.plot(time, thetha_analytic, 'b')
plt.plot(time, thetha, 'r--')

plt.legend([r'$\theta_0 \cdot cos(t)$', 'euler'], loc='best', frameon=False)
plt.xlabel('t')
plt.ylabel(r'$\theta$')
#plt.savefig('../fig-ch1/euler_pendulum.png', transparent=True)
plt.show()
```

2.6.4 Example: Sphere in free fall

Figure 2.8 illustrates a falling sphere in a gravitational field with a diameter d and mass m that falls vertically in a fluid. Use of Newton's 2nd law in the z-direction gives

$$m\frac{dv}{dt} = mg - m_f g - \frac{1}{2}m_f\frac{dv}{dt} - \frac{1}{2}\rho_f v\,|v|\,A_k C_D, \tag{2.56}$$

where the different terms are interpreted as follows: $m = \rho_k V$, where ρ_k is the density of the sphere and V is the sphere volume. The mass of the displaced fluid is given by $m_f = \rho_f V$, where ρ_f is the density of the fluid, whereas buoyancy and the drag coefficient are expressed by $m_f\, g$ and C_D, respectively. The projected area of the sphere is given by $A_k = \frac{\pi}{4}d^2$ and $\frac{1}{2}m_f$ is the hydro-dynamical mass (added mass). The expression for the hydro-dynamical mass is derived in White [16], page 539-540. To write Equation (2.56) on a more convenient form we

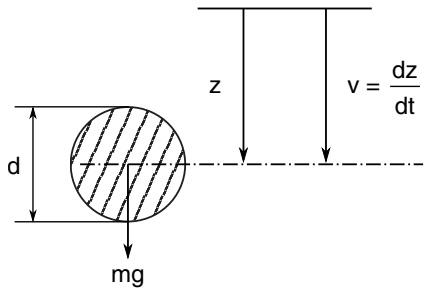

Figure 2.8: Falling sphere due to gravity.

introduce the following abbreviations:

$$\rho = \frac{\rho_f}{\rho_k},\ A = 1 + \frac{\rho}{2},\ B = (1-\rho)g,\ C = \frac{3\rho}{4d}. \tag{2.57}$$

in addition to the drag coefficient C_D which is a function of the Reynolds number $R_e = \dfrac{vd}{\nu}$, where ν is the kinematical viscosity. Equation (2.56) may then be written as

$$\frac{dv}{dt} = \frac{1}{A}(B - C \cdot v\,|v|\,C_d). \tag{2.58}$$

In air we may often neglect the buoyancy term and the hydro-dynamical mass, whereas this is not the case for a liquid. Introducing $v = \frac{dz}{dt}$ in Equation (2.58), we get a 2nd order ODE as follows

$$\frac{d^2z}{dt^2} = \frac{1}{A}\left(B - C \cdot \frac{dz}{dt}\left|\frac{dz}{dt}\right| C_d\right) \tag{2.59}$$

For Equation (2.59) two initial conditions must be specified, e.g. $v = v_0$ and $z = z_0$ for $t = 0$.

Figure 2.9 illustrates C_D as a function of Re. The values in the plot are not as accurate as the number of digits in the program might indicate. For example is the location and the size of the "valley" in the diagram strongly dependent of the degree of turbulence in the free stream and the roughness of the sphere. As the drag coefficient C_D is a function of the Reynolds number, it is also a function of the solution v (i.e. the velocity) of the ODE in Equation (2.58). We will use the function $C_D(Re)$ as an example of how functions may be implemented in Python.

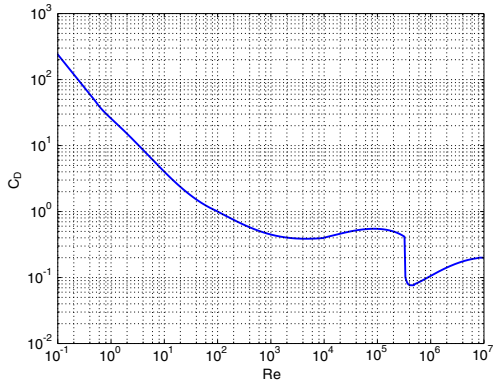

Figure 2.9: Drag coefficient C_D as function of the Reynold's number R_e.

2.6.5 Euler's method for a system

Euler's method may of course also be used for a system. Let's look at a simultaneous system of p equations

$$\begin{aligned} y_1' &= f_1(x, y_1, y_2, \dots y_p) \\ y_2' &= f_2(x, y_1, y_2, \dots y_p) \\ &. \\ &. \\ y_p' &= f_p(x, y_1, y_2, \dots y_p) \end{aligned} \tag{2.60}$$

with initial values

$$y_1(x_0) = a_1,\ y_2(x_0) = a_2, \dots,\ y_p(x_0) = a_p \tag{2.61}$$

Or, in vectorial format as follows,

$$\begin{aligned} \mathbf{y}' &= \mathbf{f}(x, \mathbf{y}) \\ \mathbf{y}(x_0) &= \mathbf{a} \end{aligned} \tag{2.62}$$

where $\mathbf{y}'$, $\mathbf{f}$, $\mathbf{y}$ and $\mathbf{a}$ are column vectors with p components.

The Euler scheme (2.55) used on (2.62) gives

$$\mathbf{y_{n+1}} = \mathbf{y_n} + h \cdot \mathbf{f}(x_n, \mathbf{y_n}) \tag{2.63}$$

For a system of three equations we get

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= y_3 \\ y_3' &= -\, y_1 y_3 \end{aligned} \tag{2.64}$$

In this case (2.63) gives

$$\begin{aligned}(y_1)_{n+1} &= (y_1)_n + h \cdot (y_2)_n \\ (y_2)_{n+1} &= (y_2)_n + h \cdot (y_3)_n \\ (y_3)_{n+1} &= (y_3)_n - h \cdot (y_1)_n \cdot (y_3)_n\end{aligned} \tag{2.65}$$

(2.66)

with $y_1(x_0) = a_1,\ y_2(x_0) = a_2,$ and $y_3(x_0) = a_3$

In section 2.4 we have seen how we can reduce a higher order ODE to a set of first order ODEs. In (2.67) and (2.68) we have the equation $\frac{d^2z}{dt^2} = g - \alpha \cdot \left(\frac{dz}{dt}\right)^2$ which we have reduced to a system as

$$\begin{aligned}\frac{dz}{dt} &= v \\ \frac{dv}{dt} &= g - \alpha \cdot v^2\end{aligned}$$

which gives an Euler scheme as follows,

$$\begin{aligned}z_{n+1} &= z_n + \Delta t \cdot v_n \\ v_{n+1} &= n_n + \Delta t \cdot [g - \alpha (v_n)^2] \\ &\text{med } z_0 = 0,\ v_0 = 0\end{aligned}$$

2.6.6 Example: Falling sphere with constant and varying drag

We write (2.58) and (2.59) as a system as follows,

$$\frac{dz}{dt} = v \tag{2.67}$$

$$\frac{dv}{dt} = g - \alpha v^2 \tag{2.68}$$

where

$$\alpha = \frac{3\rho_f}{4\rho_k \cdot d} \cdot C_D$$

The analytical solution with $z(0) = 0$ and $v(0) = 0$ is given by

$$z(t) = \frac{\ln(\cosh(\sqrt{\alpha g} \cdot t))}{\alpha} \tag{2.69}$$

$$v(t) = \sqrt{\frac{g}{\alpha}} \cdot \tanh(\sqrt{\alpha g} \cdot t) \tag{2.70}$$

The terminal velocity v_t is found by $\frac{dv}{dt} = 0$ which gives $v_t = \sqrt{\frac{g}{\alpha}}$.

We use data from a golf ball: $d = 41$ mm, $\rho_k = 1275$ kg/m^3, $\rho_k = 1.22$ kg/m^3, and choose $C_D = 0.4$ which gives $\alpha = 7 \cdot 10^{-3}$. The terminal velocity then becomes

$$v_t = \sqrt{\frac{g}{\alpha}} = 37.44$$

If we use Taylor's method from section 2.3 we get the following expression by using four terms in the series expansion:

$$z(t) = \frac{1}{2}gt^2 \cdot (1 - \frac{1}{6}\alpha g t^2) \tag{2.71}$$

$$v(t) = gt \cdot (1 - \frac{1}{3}\alpha g t^2) \tag{2.72}$$

By applying the Euler scheme (2.55) on (2.67) and (2.68)

$$z_{n+1} = z_n + \Delta t \cdot v_n \tag{2.73}$$

$$v_{n+1} = v_n + \Delta t \cdot (g - \alpha \cdot v_n^2),\ n = 0, 1, \ldots \tag{2.74}$$

with $z(0) = 0$ and $v(0) = 0$.

By adopting the conventions proposed in (2.23) and substituting z_0 for z and z_1 for v and we may render the system of equations in (2.67) and (2.68) as:

$$\frac{dz_0}{dt} = z_1$$

$$\frac{dz_1}{dt} = g - \alpha z_1^2$$

One way of implementing the integration scheme is given in the following function `euler()`:

```
def euler(func,z0, time):
    """The Euler scheme for solution of systems of ODEs.
    z0 is a vector for the initial conditions,
    the right hand side of the system is represented by func which returns
    a vector with the same size as z0 ."""

    z = np.zeros((np.size(time),np.size(z0)))
    z[0,:] = z0

    for i in range(len(time)-1):
        dt = time[i+1]-time[i]
        z[i+1,:]=z[i,:] + np.asarray(func(z[i,:],time[i]))*dt

    return z
```

The program **FallingSphereEuler.py** computes the solution for the first 10 seconds, using a time step of $\Delta t = 0.5$ s, and generates the plot in Figure 2.10. In addition to the case of constant drag coefficient, a solution for the case of varying C_D is included. To find C_D as function of velocity we use the function `cd_sphere()` that we implemented in (2.6.4). The complete program is as follows,

```
# src-ch1/FallingSphereEuler.py;DragCoefficientGeneric.py @ git@lrhgit/tkt4140/src/src-ch1/DragCoeff
from DragCoefficientGeneric import cd_sphere
from matplotlib.pyplot import *
import numpy as np

# change some default values to make plots more readable
LNWDT=2; FNT=11
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT


g = 9.81       # Gravity m/s^2
d = 41.0e-3    # Diameter of the sphere
rho_f = 1.22   # Density of fluid [kg/m^3]
rho_s = 1275   # Density of sphere [kg/m^3]
nu = 1.5e-5    # Kinematical viscosity [m^2/s]
CD = 0.4       # Constant drag coefficient

def f(z, t):
    """2x2 system for sphere with constant drag."""
    zout = np.zeros_like(z)
    alpha = 3.0*rho_f/(4.0*rho_s*d)*CD
    zout[:] = [z[1], g - alpha*z[1]**2]
    return zout

def f2(z, t):
    """2x2 system for sphere with Re-dependent drag."""
    zout = np.zeros_like(z)
    v = abs(z[1])
    Re = v*d/nu
    CD = cd_sphere(Re)
    alpha = 3.0*rho_f/(4.0*rho_s*d)*CD
    zout[:] = [z[1], g - alpha*z[1]**2]
    return zout

# define euler scheme
def euler(func,z0, time):
    """The Euler scheme for solution of systems of ODEs.
    z0 is a vector for the initial conditions,
    the right hand side of the system is represented by func which returns
    a vector with the same size as z0 ."""

    z = np.zeros((np.size(time),np.size(z0)))
    z[0,:] = z0

    for i in range(len(time)-1):
        dt = time[i+1]-time[i]
        z[i+1,:]=z[i,:] + np.asarray(func(z[i,:],time[i]))*dt

    return z

def v_taylor(t):
#    z = np.zeros_like(t)
    v = np.zeros_like(t)

    alpha = 3.0*rho_f/(4.0*rho_s*d)*CD
    v=g*t*(1-alpha*g*t**2)
    return v

# main program starts here

T = 10  # end of simulation
```

```
N = 20  # no of time steps
time = np.linspace(0, T, N+1)

z0=np.zeros(2)
z0[0] = 2.0

ze = euler(f, z0, time)      # compute response with constant CD using Euler's method
ze2 = euler(f2, z0, time)    # compute response with varying CD using Euler's method

k1 = np.sqrt(g*4*rho_s*d/(3*rho_f*CD))
k2 = np.sqrt(3*rho_f*g*CD/(4*rho_s*d))
v_a = k1*np.tanh(k2*time)    # compute response with constant CD using analytical solution

# plotting

legends=[]
line_type=['-',':','.','-.','--']

plot(time, v_a, line_type[0])
legends.append('Analytical (constant CD)')

plot(time, ze[:,1], line_type[1])
legends.append('Euler (constant CD)')

plot(time, ze2[:,1], line_type[3])
legends.append('Euler (varying CD)')

time_taylor = np.linspace(0, 4, N+1)

plot(time_taylor, v_taylor(time_taylor))
legends.append('Taylor (constant CD)')

legend(legends, loc='best', frameon=False)
font = {'size' : 16}
rc('font', **font)
xlabel('Time [s]')
ylabel('Velocity [m/s]')
#savefig('example_sphere_falling_euler.png', transparent=True)
show()
```

Python implementation of the drag coefficient function and how to plot it. The complete Python program **CDsphere.py** used to plot the drag coefficient in the example above is listed below. The program uses a function `cd_sphere` which results from a curve fit to the data of Evett and Liu [4]. In our setting we will use this function for two purposes, namely to demonstrate how functions and modules are implemented in Python and finally use these functions in the solution of the ODE in Equations (2.58) and (2.59).

```
# src-ch1/CDsphere.py

from numpy import logspace, zeros

# Define the function cd_sphere
def cd_sphere(Re):
    "This function computes the drag coefficient of a sphere as a function of the Reynolds number Re
    # Curve fitted after fig . A -56 in Evett and Liu: "Fluid Mechanics and Hydraulics"
```

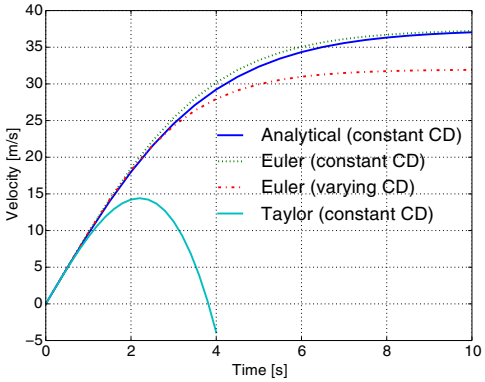

Figure 2.10: Euler's method with $\Delta t = 0.5$ s.

```
    from numpy import log10, array, polyval

    if Re <= 0.0:
        CD = 0.0
    elif Re > 8.0e6:
        CD = 0.2
    elif Re > 0.0 and Re <= 0.5:
        CD = 24.0/Re
    elif Re > 0.5 and Re <= 100.0:
        p = array([4.22, -14.05, 34.87, 0.658])
        CD = polyval(p, 1.0/Re)
    elif Re > 100.0 and Re <= 1.0e4:
        p = array([-30.41, 43.72, -17.08, 2.41])
        CD = polyval(p, 1.0/log10(Re))
    elif Re > 1.0e4 and Re <= 3.35e5:
        p = array([-0.1584, 2.031, -8.472, 11.932])
        CD = polyval(p, log10(Re))
    elif Re > 3.35e5 and Re <= 5.0e5:
        x1 = log10(Re/4.5e5)
        CD = 91.08*x1**4 + 0.0764
    else:
        p = array([-0.06338, 1.1905, -7.332, 14.93])
        CD = polyval(p, log10(Re))
    return CD

# Calculate drag coefficient
Npts = 500
Re = logspace(-1, 7, Npts, True, 10)
CD = zeros(Npts)
i_list = range(0, Npts-1)
for i in i_list:
    CD[i] = cd_sphere(Re[i])

# Make plot
from matplotlib import pyplot
# change some default values to make plots more readable
LNWDT=2; FNT=11
pyplot.rcParams['lines.linewidth'] = LNWDT; pyplot.rcParams['font.size'] = FNT
```

```
pyplot.plot(Re, CD, '-b')
pyplot.yscale('log')
pyplot.xscale('log')
pyplot.xlabel('$Re$')
pyplot.ylabel('$C_D$')
pyplot.grid('on', 'both', 'both')
#pyplot.savefig('example_sphere.png', transparent=True)
pyplot.show()
```

In the following, we will break up the program and explain the different parts. In the first code line,

```
from numpy import logspace, zeros
```

the functions `logspace` and `zeros` are imported from the package `numpy`. The `numpy` package (*NumPy* is an abbreviation for *Numerical Python*) enables the use of *array* objects. Using `numpy` a wide range of mathematical operations can be done directly on complete arrays, thereby removing the need for loops over array elements. This is commonly called *vectorization* and may cause a dramatic increase in computational speed of Python programs. The function `logspace` works on a logarithmic scale just as the function `linspace` works on a regular scale. The function `zeros` creates arrays of a certain size filled with zeros. Several comprehensive guides to the numpy package may be found at http://www.numpy.org.

In **CDsphere.py** a function `cd_sphere` was defined as follows:

```
def cd_sphere(Re):
    "This function computes the drag coefficient of a sphere as a function of the Reynolds number Re
    # Curve fitted after fig . A -56 in Evett and Liu: "Fluid Mechanics and Hydraulics"

    from numpy import log10, array, polyval

    if Re <= 0.0:
        CD = 0.0
    elif Re > 8.0e6:
        CD = 0.2
    elif Re > 0.0 and Re <= 0.5:
        CD = 24.0/Re
    elif Re > 0.5 and Re <= 100.0:
        p = array([4.22, -14.05, 34.87, 0.658])
        CD = polyval(p, 1.0/Re)
    elif Re > 100.0 and Re <= 1.0e4:
        p = array([-30.41, 43.72, -17.08, 2.41])
        CD = polyval(p, 1.0/log10(Re))
    elif Re > 1.0e4 and Re <= 3.35e5:
        p = array([-0.1584, 2.031, -8.472, 11.932])
        CD = polyval(p, log10(Re))
    elif Re > 3.35e5 and Re <= 5.0e5:
        x1 = log10(Re/4.5e5)
        CD = 91.08*x1**4 + 0.0764
    else:
        p = array([-0.06338, 1.1905, -7.332, 14.93])
        CD = polyval(p, log10(Re))
    return CD
```

The function takes `Re` as an argument and returns the value `CD`. All Python functions begin with `def`, followed by the function name, and then inside parentheses a comma-separated list of function arguments, ended with a colon. Here we have only one argument, `Re`. This argument acts as a standard variable inside the function. The statements to perform inside the function must be indented. At the end of a function it is common to use the `return` statement to return the value of the function.

Variables defined inside a function, such as `p` and `x1` above, are *local* variables that cannot be accessed outside the function. Variables defined outside functions, in the "main program", are *global* variables and may be accessed anywhere, also inside functions.

Three more functions from the `numpy` package are imported in the function. They are not used outside the function and are therefore chosen to be imported only if the function is called from the main program. We refer to the documentation of NumPy for details about the different functions.

The function above contains an example of the use of the `if-elif-else` block. The block begins with `if` and a boolean expression. If the boolean expression evaluates to `true` the *indented* statements following the `if` statement are carried out. If not, the boolean expression following the `elif` is evaluated. If none of the conditions are evaluated to `true` the statements following the `else` are carried out.

In the code block

```
Npts = 500
Re = logspace(-1, 7, Npts, True, 10)
CD = zeros(Npts)
i_list = range(0, Npts-1)
for i in i_list:
    CD[i] = cd_sphere(Re[i])
```

the function `cd_sphere` is called. First, the number of data points to be calculated are stored in the integer variable `Npts`. Using the `logspace` function imported earlier, `Re` is assigned an array object which has float elements with values ranging from 10^{-1} to 10^{7}. The values are uniformly distributed along a 10-logarithmic scale. `CD` is first defined as an array with `Npts` zero elements, using the `zero` function. Then, for each element in `Re`, the drag coefficient is calculated using our own defined function `cd_sphere`, in a `for` loop, which is explained in the following.

The function `range` is a built-in function that generates a list containing arithmetic progressions. The `for i in i_list` construct creates a loop over all elements in `i_list`. In each pass of the loop, the variable `i` refers to an element in the list, starting with `i_list[0]` (0 in this case) and ending with the last element `i_list[Npts-1]` (499 in this case). Note that element indices start at 0 in Python. After the colon comes a block of statements which does something useful with the current element; in this case, the return of the function call `cd_sphere(Re[i])` is assigned to `CD[i]`. Each statement in the block must be indented.

Lastly, the drag coefficient is plotted and the figure generated:

```
from matplotlib import pyplot
# change some default values to make plots more readable
LNWDT=2; FNT=11
pyplot.rcParams['lines.linewidth'] = LNWDT; pyplot.rcParams['font.size'] = FNT

pyplot.plot(Re, CD, '-b')
pyplot.yscale('log')
pyplot.xscale('log')
pyplot.xlabel('$Re$')
pyplot.ylabel('$C_D$')
pyplot.grid('on', 'both', 'both')
#pyplot.savefig('example_sphere.png', transparent=True)
pyplot.show()
```

To generate the plot, the package `matplotlib` is used. `matplotlib` is the standard package for curve plotting in Python. For simple plotting the `matplotlib.pyplot` interface provides a Matlab-like interface, which has been used here. For documentation and explanation of this package, we refer to http://www.matplotlib.org.

First, the curve is generated using the function `plot`, which takes the x-values and y-values as arguments (`Re` and `CD` in this case), as well as a string specifying the line style, like in Matlab. Then changes are made to the figure in order to make it more readable, very similarly to how it is done in Matlab. For instance, in this case it makes sense to use logarithmic scales. A png version of the figure is saved using the `savefig` function. Lastly, the figure is showed on the screen with the `show` function.

To change the font size the function `rc` is used. This function takes in the object `font`, which is a *dictionary* object. Roughly speaking, a dictionary is a list where the index can be a text (in lists the index must be an integer). It is best to think of a dictionary as an unordered set of `key:value` pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of `key:value` pairs within the braces adds initial `key:value` pairs to the dictionary. In this case the dictionary `font` contains one `key:value` pair, namely `'size' : 16`.

Descriptions and explanations of all functions available in `pyplot` may be found here.

2.7 Python functions with vector arguments and modules

For many numerical problems variables are most conveniently expressed by arrays containing many numbers (i.e. vectors) rather than single numbers (i.e. scalars). The function `cd_sphere` above takes a scalar as an argument and returns a scalar value too. For computationally intensive algorithms where variables are stored in arrays this is inconvenient and time consuming, as each of the array elements must be sent to the function independently. In the following, we will therefore show how to implement functions with vector arguments that also return vectors. This may be done in various ways. Some possibilities are presented in the

following, and, as we shall see, some are more time consuming than others. We will also demonstrate how the time consumption (or efficiency) may be tested.

A simple extension of the single-valued function `cd_sphere` is as follows:

```
def cd_sphere_py_vector(ReNrs):
    CD = zeros_like(ReNrs)
    counter = 0

    for Re in ReNrs:
        CD[counter] = cd_sphere(Re)
        counter += 1
    return CD
```

The new function `cd_sphere_py_vector` takes in an array `ReNrs` and calculates the drag coefficient for each element using the previous function `cd_sphere`. This does the job, but is not very efficient.

A second version is implemented in the function `cd_sphere_vector`. This function takes in the array `Re` and calculates the drag coefficient of all elements by multiple calls of the function `numpy.where`; one call for each condition, similarly as each `if` statement in the function `cd_sphere`. The function is shown here:

```
def cd_sphere_vector(Re):
    "Computes the drag coefficient of a sphere as a function of the Reynolds number Re."
    # Curve fitted after fig . A -56 in Evett and Liu: "Fluid Mechanics and Hydraulics"

    from numpy import log10, array, polyval, where, zeros_like
    CD = zeros_like(Re)

    CD = where(Re < 0, 0.0, CD)      # condition 1

    CD = where((Re > 0.0) & (Re <=0.5), 24/Re, CD) # condition 2

    p = array([4.22, -14.05, 34.87, 0.658])
    CD = where((Re > 0.5) & (Re <=100.0), polyval(p, 1.0/Re), CD) #condition 3

    p = array([-30.41, 43.72, -17.08, 2.41])
    CD = where((Re > 100.0)  & (Re <= 1.0e4), polyval(p, 1.0/log10(Re)), CD) #condition 4

    p = array([-0.1584, 2.031, -8.472, 11.932])
    CD = where((Re > 1.0e4)  &  (Re <= 3.35e5), polyval(p, log10(Re)), CD) #condition 5

    CD = where((Re > 3.35e5) & (Re <= 5.0e5), 91.08*(log10(Re/4.5e5))**4 + 0.0764, CD) #condition 6

    p  = array([-0.06338, 1.1905, -7.332, 14.93])
    CD = where((Re > 5.05e5)  &  (Re <= 8.0e6), polyval(p, log10(Re)), CD) #condition 7

    CD = where(Re > 8.0e6, 0.2, CD)  # condition 8
    return CD
```

A third approach we will try is using boolean type variables. The 8 variables `condition1` through `condition8` in the function `cd_sphere_vector_bool` are boolean variables of the same size and shape as `Re`. The elements of the boolean variables evaluate to either `True` or `False`, depending on if the corresponding element in `Re` satisfy the condition the variable is assigned.

```
def cd_sphere_vector_bool(Re):
    "Computes the drag coefficient of a sphere as a function of the Reynolds number Re."
```

```
    # Curve fitted after fig . A -56 in Evett and Liu: "Fluid Mechanics and Hydraulics"

    from numpy import log10, array, polyval, zeros_like

    condition1 = Re < 0
    condition2 = logical_and(0 < Re, Re <= 0.5)
    condition3 = logical_and(0.5 < Re, Re <= 100.0)
    condition4 = logical_and(100.0 < Re, Re <= 1.0e4)
    condition5 = logical_and(1.0e4 < Re, Re <= 3.35e5)
    condition6 = logical_and(3.35e5 < Re, Re <= 5.0e5)
    condition7 = logical_and(5.0e5 < Re, Re <= 8.0e6)
    condition8 = Re > 8.0e6

    CD = zeros_like(Re)
    CD[condition1] = 0.0

    CD[condition2] = 24/Re[condition2]

    p = array([4.22,-14.05,34.87,0.658])
    CD[condition3] = polyval(p,1.0/Re[condition3])

    p = array([-30.41,43.72,-17.08,2.41])
    CD[condition4] = polyval(p,1.0/log10(Re[condition4]))

    p = array([-0.1584,2.031,-8.472,11.932])
    CD[condition5] = polyval(p,log10(Re[condition5]))

    CD[condition6] = 91.08*(log10(Re[condition6]/4.5e5))**4 + 0.0764

    p  = array([-0.06338,1.1905,-7.332,14.93])
    CD[condition7] = polyval(p,log10(Re[condition7]))

    CD[condition8] = 0.2

    return CD
```

Lastly, the built-in function `vectorize` is used to automatically generate a vector-version of the function `cd_sphere`, as follows:

```
cd_sphere_auto_vector = vectorize(cd_sphere)
```

To provide a convenient and practical means to compare the various implementations of the drag function, we have collected them all in a file **DragCoefficientGeneric.py**. This file constitutes a Python module which is a concept we will discuss in section 2.8.

2.8 How to make a Python-module and some useful programming features

Python modules A module is a file containing Python definitions and statements and represents a convenient way of collecting useful and related functions, classes or Python code in a single file. A motivation to implement the drag coefficient function was that we should be able to import it in other programs to

solve e.g. the problems outlined in (2.6.4). In general, a file containing Python-code may be executed either as a main program (script), typically with `python filename.py` or imported in another script/module with `import filename`.

A module file should not execute a main program, but rather just define functions, import other modules, and define global variables. Inside modules, the standard practice is to only have functions and not any statements outside functions. The reason is that all statements in the module file are executed from top to bottom during import in another module/script [10], and thus a desirable behavior is no output to avoid confusion. However, in many situations it is also desirable to allow for tests or demonstration of usage inside the module file, and for such situations the need for a main program arises. To meet these demands Python allows for a fortunate construction to let a file act both as a module with function definitions only (i.e. no main program) and as an ordinary program we can run, with functions and a main program. The latter is possible by letting the main program follow an `if` test of the form:

```
if __name__ =='__main__':
    <main program statements>
```

The `__name__` variable is automatically defined in any module and equals the module name if the module is imported in another program/script, but when the module file is executed as a program, `__name__` equals the string `'__main__'`. Consequently, the `if` test above will only be true whenever the module file is executed as a program and allow for the execution of the `<main program statements>`. The `<main program statements>` is normally referred to as the *test block* of a module.

The module name is the file name without the suffix `.py` [10], i.e. the module contained in the module file `filename.py` has the module name `filename`. Note that a module can contain executable statements as well as function definitions. These statements are intended to initialize the module and are executed only the first time the module name is encountered in an import statement. They are also run if the file is executed as a script.

Below we have listed the content of the file `DragCoefficientGeneric.py` to illustrate a specific implementation of the module `DragCoefficientGeneric` and some other useful programming features in Python. The functions in the module are the various implementations of the drag coefficient functions from the previous section.

Python lists and dictionaries

- Lists hold a list of values and are initialized with empty brackets, e.g. `fncnames = []`. The values of the list are accessed with an index, starting from zero. The first value of `fncnames` is `fncnames[0]`, the second value of `fncnames` is `fncnames[1]` and so on. You can remove values from the list, and add new values to the end by `fncnames`. Example: `fncnames.append(name)` will append `name` as the last value of the list `fncnames`. In case it was empty prior to the append-operation, `name` will be the only element in the list.

- Dictionaries are similar to what their name suggests - a dictionary - and an empty dictionary is initialized with empty braces, e.g. `CD = {}`. In a dictionary, you have an 'index' of words or keys and the values accessed by their 'key'. Thus, the values in a dictionary aren't numbered or ordered, they are only accessed by the key. You can add, remove, and modify the values in dictionaries. For example with the statement `CD[name] = func(ReNrs)` the results of `func(ReNrs)` are stored in the list `CD` with key `name`.

To illustrate a very powerful feature of Python data structures allowing for lists of e.g. function objects we put all the function names in a list with the statement:

```
funcs = [cd_sphere_py_vector, cd_sphere_vector, cd_sphere_vector_bool, \
         cd_sphere_auto_vector]  # list of functions to test
```

which allows for convenient looping over all of the functions with the following construction:

```
for func in funcs:
```

Exception handling Python has a very convenient construction for testing of potential errors with try-except blocks:

```
try:
    <statements>
except ExceptionType1:
    <remedy for ExceptionType1 errors>
except ExceptionType2:
    <remedy for ExceptionType1 errors>
except:
    <remedy for any other errors>
```

In the `DragCoefficientGeneric` module, this feature is used to handle the function name for a function which has been vectorized automatically. For such a function `func.func_name` has no value and will return an error, and the name may be found by the statements in the exception block.

Efficiency and benchmarking The function `clock` in the module time, return a time expressed in seconds for the current statement and is frequently used for benchmarking in Python or timing of functions. By subtracting the time `t0` recored immediately before a function call from the time immediately after the function call, an estimate of the elapsed cpu-time is obtained. In our module `DragCoefficientGeneric` the efficiency is implemented with the codelines:

```
t0 = time.clock()
CD[name] = func(ReNrs)
exec_times[name] = time.clock() - t0
```

Sorting of dictionaries The computed execution times are for convenience stored in the dictionary `exec_time` to allow for pairing of the names of the functions and their execution time. The dictionary may be sorted on the values and the corresponding keys sorted are returned by:

```
exec_keys_sorted = sorted(exec_times, key=exec_times.get)
```

Afterwards the results may be printed with name and execution time, ordered by the latter, with the most efficient function at the top:

```
    for name_key in exec_keys_sorted:
        print name_key, '\t execution time = ', '%6.6f' % exec_times[name_key]
```

By running the module `DragCoefficientGeneric` as a script, and with 500 elements in the `ReNrs` array we got the following output:

```
cd_sphere_vector_bool   execution time = 0.000312
cd_sphere_vector        execution time = 0.000641
cd_sphere_auto_vector   execution time = 0.009497
cd_sphere_py_vector     execution time = 0.010144
```

Clearly, the function with the boolean variables was fastest, the straight forward vectorized version `cd_sphere_py_vector` was slowest and the built-in function `vectorize` was nearly as inefficient.

The complete module **DragCoefficientGeneric** is listed below.

```
# src-ch1/DragCoefficientGeneric.py

from numpy import linspace,array,append,logspace,zeros_like,where,vectorize,\
    logical_and
import numpy as np
from matplotlib.pyplot import loglog,xlabel,ylabel,grid,savefig,show,rc,hold,\
    legend, setp

from numpy.core.multiarray import scalar

# single-valued function
def cd_sphere(Re):
    "Computes the drag coefficient of a sphere as a function of the Reynolds number Re."
    # Curve fitted after fig . A -56 in Evett and Liu: "Fluid Mechanics and Hydraulics"

    from numpy import log10,array,polyval

    if Re <= 0.0:
        CD = 0.0
    elif Re > 8.0e6:
        CD = 0.2
    elif Re > 0.0 and Re <= 0.5:
        CD = 24.0/Re
    elif Re > 0.5 and Re <= 100.0:
        p = array([4.22,-14.05,34.87,0.658])
        CD = polyval(p,1.0/Re)
    elif Re > 100.0 and Re <= 1.0e4:
        p = array([-30.41,43.72,-17.08,2.41])
        CD = polyval(p,1.0/log10(Re))
    elif Re > 1.0e4 and Re <= 3.35e5:
        p = array([-0.1584,2.031,-8.472,11.932])
        CD = polyval(p,log10(Re))
    elif Re > 3.35e5 and Re <= 5.0e5:
        x1 = log10(Re/4.5e5)
        CD = 91.08*x1**4 + 0.0764
    else:
        p = array([-0.06338,1.1905,-7.332,14.93])
        CD = polyval(p,log10(Re))
    return CD
```

```
# simple extension cd_sphere
def cd_sphere_py_vector(ReNrs):
    CD = zeros_like(ReNrs)
    counter = 0

    for Re in ReNrs:
        CD[counter] = cd_sphere(Re)
        counter += 1
    return CD

# vectorized function
def cd_sphere_vector(Re):
    "Computes the drag coefficient of a sphere as a function of the Reynolds number Re."
    # Curve fitted after fig . A -56 in Evett and Liu: "Fluid Mechanics and Hydraulics"

    from numpy import log10, array, polyval, where, zeros_like
    CD = zeros_like(Re)

    CD = where(Re < 0, 0.0, CD)     # condition 1

    CD = where((Re > 0.0) & (Re <=0.5), 24/Re, CD) # condition 2

    p = array([4.22, -14.05, 34.87, 0.658])
    CD = where((Re > 0.5) & (Re <=100.0), polyval(p, 1.0/Re), CD) #condition 3

    p = array([-30.41, 43.72, -17.08, 2.41])
    CD = where((Re > 100.0)  & (Re <= 1.0e4), polyval(p, 1.0/log10(Re)), CD) #condition 4

    p = array([-0.1584, 2.031, -8.472, 11.932])
    CD = where((Re > 1.0e4)  &  (Re <= 3.35e5), polyval(p, log10(Re)), CD) #condition 5

    CD = where((Re > 3.35e5) & (Re <= 5.0e5), 91.08*(log10(Re/4.5e5))**4 + 0.0764, CD) #condition 6

    p  = array([-0.06338, 1.1905, -7.332, 14.93])
    CD = where((Re > 5.05e5)  &  (Re <= 8.0e6), polyval(p, log10(Re)), CD) #condition 7

    CD = where(Re > 8.0e6, 0.2, CD)  # condition 8
    return CD

# vectorized boolean
def cd_sphere_vector_bool(Re):
    "Computes the drag coefficient of a sphere as a function of the Reynolds number Re."
    # Curve fitted after fig . A -56 in Evett and Liu: "Fluid Mechanics and Hydraulics"

    from numpy import log10, array, polyval, zeros_like

    condition1 = Re < 0
    condition2 = logical_and(0 < Re, Re <= 0.5)
    condition3 = logical_and(0.5 < Re, Re <= 100.0)
    condition4 = logical_and(100.0 < Re, Re <= 1.0e4)
    condition5 = logical_and(1.0e4 < Re, Re <= 3.35e5)
    condition6 = logical_and(3.35e5 < Re, Re <= 5.0e5)
    condition7 = logical_and(5.0e5 < Re, Re <= 8.0e6)
    condition8 = Re > 8.0e6

    CD = zeros_like(Re)
    CD[condition1] = 0.0

    CD[condition2] = 24/Re[condition2]

    p = array([4.22,-14.05,34.87,0.658])
```

```
    CD[condition3] = polyval(p,1.0/Re[condition3])

    p = array([-30.41,43.72,-17.08,2.41])
    CD[condition4] = polyval(p,1.0/log10(Re[condition4]))

    p = array([-0.1584,2.031,-8.472,11.932])
    CD[condition5] = polyval(p,log10(Re[condition5]))

    CD[condition6] = 91.08*(log10(Re[condition6]/4.5e5))**4 + 0.0764

    p  = array([-0.06338,1.1905,-7.332,14.93])
    CD[condition7] = polyval(p,log10(Re[condition7]))

    CD[condition8] = 0.2

    return CD


if __name__ == '__main__':
#Check whether this file is executed (name==main) or imported as a module

    import time
    from numpy import mean

    CD = {} # Empty list for all CD computations


    ReNrs = logspace(-2,7,num=500)

    # make a vectorized version of the function automatically
    cd_sphere_auto_vector = vectorize(cd_sphere)

    # make a list of all function objects
    funcs = [cd_sphere_py_vector, cd_sphere_vector, cd_sphere_vector_bool, \
             cd_sphere_auto_vector]  # list of functions to test


    # Put all exec_times in a dictionary and fncnames in a list
    exec_times = {}
    fncnames = []
    for func in funcs:
        try:
            name = func.func_name
        except:
            scalarname = func.__getattribute__('pyfunc')
            name = scalarname.__name__+'_auto_vector'

        fncnames.append(name)

        # benchmark
        t0 = time.clock()
        CD[name] = func(ReNrs)
        exec_times[name] = time.clock() - t0

    # sort the dictionary exec_times on values and return a list of the corresponding keys
    exec_keys_sorted = sorted(exec_times, key=exec_times.get)

    # print the exec_times by ascending values
    for name_key in exec_keys_sorted:
        print name_key, '\t execution time = ', '%6.6f' % exec_times[name_key]
```

```
    # set fontsize prms
    fnSz = 16; font = {'size'   : fnSz}; rc('font',**font)

    # set line styles
    style = ['v-', '8-', '*-', 'o-']
    mrkevry = [30, 35, 40, 45]

    # plot the result for all functions
    i=0
    for name in fncnames:
        loglog(ReNrs, CD[name], style[i], markersize=10, markevery=mrkevry[i])
        hold('on')
        i+=1

    # use fncnames as plot legend
    leg = legend(fncnames)
    leg.get_frame().set_alpha(0.)
    xlabel('$Re$')
    ylabel('$C_D$')
    grid('on', 'both', 'both')
#    # savefig('example_sphere_generic.png', transparent=True) # save plot if needed
    show()
```

2.8.1 Example: Numerical error as a function of Δt

In this example we will assess how the error of our implementation of the Euler method depends on the time step Δt in a systematic manner. We will solve a problem with an analytical solution in a loop, and for each new solution we do the following:

- Divide the time step by two (or double the number of time steps)
- Compute the error
- Plot the error

Euler's method is a first order method and we expect the error to be $O(h) = O(\Delta t)$. Consequently if the time-step is divided by two, the error should also be divided by two. As errors normally are small values and are expected to be smaller and smaller for decreasing time steps, we normally do not plot the error itself, but rather the logarithm of the absolute value of the error. The latter we do due to the fact that we are only interested in the order of magnitude of the error, whereas errors may be both positive and negative. As the initial value is always correct we discard the first error at time zero to avoid problems with the logarithm of zero in `log_error = np.log2(abs_error[1:])`.

```
# coding: utf-8
# src-ch1/Euler_timestep_ctrl.py;DragCoefficientGeneric.py @ git@lrhgit/tkt4140/src/src-ch1/DragCoef
from DragCoefficientGeneric import cd_sphere
from matplotlib.pyplot import *
import numpy as np

# change some default values to make plots more readable
LNWDT=2; FNT=11
```

```
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT



g = 9.81      # Gravity m/s^2
d = 41.0e-3   # Diameter of the sphere
rho_f = 1.22  # Density of fluid [kg/m^3]
rho_s = 1275  # Density of sphere [kg/m^3]
nu = 1.5e-5   # Kinematical viscosity [m^2/s]
CD = 0.4      # Constant drag coefficient

def f(z, t):
    """2x2 system for sphere with constant drag."""
    zout = np.zeros_like(z)
    alpha = 3.0*rho_f/(4.0*rho_s*d)*CD
    zout[:] = [z[1], g - alpha*z[1]**2]
    return zout

# define euler scheme
def euler(func,z0, time):
    """The Euler scheme for solution of systems of ODEs.
    z0 is a vector for the initial conditions,
    the right hand side of the system is represented by func which returns
    a vector with the same size as z0 ."""

    z = np.zeros((np.size(time),np.size(z0)))
    z[0,:] = z0

    for i in range(len(time)-1):
        dt = time[i+1]-time[i]
        z[i+1,:]=z[i,:] + np.asarray(func(z[i,:],time[i]))*dt
    return z

def v_taylor(t):
#    z = np.zeros_like(t)
    v = np.zeros_like(t)
    alpha = 3.0*rho_f/(4.0*rho_s*d)*CD
    v=g*t*(1-alpha*g*t**2)
    return v

# main program starts here

T = 10  # end of simulation
N = 10  # no of time steps


z0=np.zeros(2)
z0[0] = 2.0

# Prms for the analytical solution
k1 = np.sqrt(g*4*rho_s*d/(3*rho_f*CD))
k2 = np.sqrt(3*rho_f*g*CD/(4*rho_s*d))

Ndts = 4  # Number of times to divide the dt by 2
legends=[]
error_diff = []

for i in range(Ndts+1):
    time = np.linspace(0, T, N+1)
    ze = euler(f, z0, time)     # compute response with constant CD using Euler's method
    v_a = k1*np.tanh(k2*time)   # compute response with constant CD using analytical solution
```

```
    abs_error=np.abs(ze[:,1] - v_a)
    log_error = np.log2(abs_error[1:])
    max_log_error = np.max(log_error)

    plot(time[1:], log_error)
    legends.append('Euler scheme: N ' + str(N) + ' timesteps' )
    N*=2
    if i > 0:
        error_diff.append(previous_max_log_err-max_log_error)

    previous_max_log_err = max_log_error

print 'Approximate order of scheme n =', np.mean(error_diff)
print 'Approximate error reuduction by dt=dt/2:', 1/2**(np.mean(error_diff))


# plot analytical solution
# plot(time,v_a)
# legends.append('analytical')

# fix plot
legend(legends, loc='best', frameon=False)
xlabel('Time [s]')
#ylabel('Velocity [m/s]')
ylabel('log2-error')
#savefig('example_euler_timestep_study.png', transparent=True)
show()
```

The plot resulting from the code above is shown in Figure (2.11). The difference or distance between the curves seems to be rather constant after an initial transient. As we have plotted the logarithm of the absolute value of the error ϵ_i, the difference d_{i+1} between two curves is $d_{i+1} = \log_2 \epsilon_i - \log_2 \epsilon_{i+1} = \log_2 \frac{\epsilon_i}{\epsilon_{i+1}}$. A rough visual inspection of Figure (2.11) yields $d_{i+1} \approx 1.0$, from which we may deduce the apparent order of the scheme:

$$n = \log_2 \frac{\epsilon_i}{\epsilon_{i+1}} \approx 1 \Rightarrow \epsilon_{i+1} \approx 0.488483620\,\epsilon_i \tag{2.75}$$

The print statement returns 'n=1.0336179048' and '0.488483620794', thus we see that the error is reduced even slightly more than the theoretically expected value for a first order scheme, i.e. $\Delta t_{i+1} = \Delta t_i/2$ yields $\epsilon_{i+1} \approx \epsilon_i/2$.

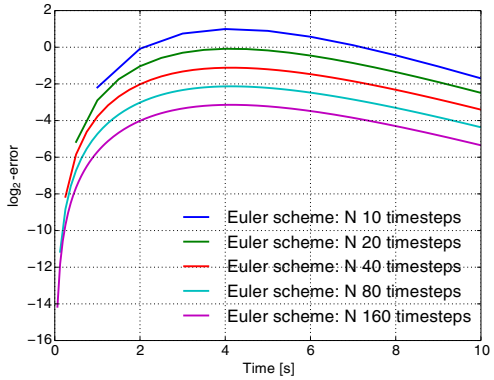

Figure 2.11: Plots for the logarithmic errors for a falling sphere with constant drag. The timestep Δt is reduced by a factor two from one curve to the one immediately below.

2.9 Heun's method

From (2.29) or (2.33) we have

$$y''(x_n, y_n) = f'\left(x_n, y(x_n, y_n)\right) \approx \frac{f(x_n + h) - f(x_n)}{h} \tag{2.76}$$

The Taylor series expansion (2.27) gives

$$y(x_n + h) = y(x_n) + hy'[x_n, y(x_n)] + \frac{h^2}{2} y''[x_n, y(x_n)] + O(h^3)$$

which, inserting (2.76), gives

$$y_{n+1} = y_n + \frac{h}{2} \cdot [f(x_n, y_n) + f(x_{n+1}, y(x_{n+1}))] \tag{2.77}$$

This formula is called the trapezoidal formula, since it reduces to computing an integral with the trapezoidal rule if $f(x, y)$ is only a function of x. Since y_{n+1} appears on both sides of the equation, this is an implicit formula which means that we need to solve a system of non-linear algebraic equations if the function $f(x, y)$ is non-linear. One way of making the scheme explicit is to use the Euler scheme (2.55) to calculate $y(x_{n+1})$ on the right side of (2.77). The resulting scheme is often denoted **Heun's method**.

The scheme for Heun's method becomes

$$y^p_{n+1} = y_n + h \cdot f(x_n, y_n) \tag{2.78}$$

$$y_{n+1} = y_n + \frac{h}{2} \cdot [f(x_n, y_n) + f(x_{n+1}, y^p_{n+1})] \tag{2.79}$$

Index p stands for "predicted". (2.78) is then the predictor and (2.79) is the corrector. This is a second order method. For more details, see [3]. Figure 2.12 is a graphical illustration of the method.

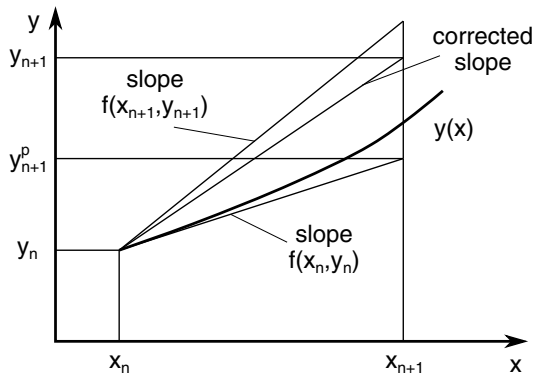

Figure 2.12: Illustration of Heun's method.

In principle we could make an iteration procedure where we after using the corrector use the corrected values to correct the corrected values to make a new predictor and so on. This will likely lead to a more accurate solution of the difference scheme, but not necessarily of the differential equation. We are therefore satisfied by using the corrector once. For a system, we get

$$\mathbf{y^p_{n+1}} = \mathbf{y_n} + h \cdot \mathbf{f}(x_n, \mathbf{y_n}) \tag{2.80}$$

$$\mathbf{y_{n+1}} = \mathbf{y_n} + \frac{h}{2} \cdot [\mathbf{f}(x_n, \mathbf{y_n}) + \mathbf{f}(x_{n+1}, \mathbf{y^p_{n+1}})] \tag{2.81}$$

Note that $\mathbf{y}^p_{n+1}$ is a temporary variable that is not necessary to store.

If we use (2.80) and (2.81) on the example in (2.64) we get

Predictor:

$$\begin{aligned}(y_1)^p_{n+1} &= (y_1)_n + h \cdot (y_2)_n \\ (y_2)^p_{n+1} &= (y_2)_n + h \cdot (y_3)_n \\ (y_3)^p_{n+1} &= (y_3)_n - h \cdot (y_1)_n \cdot (y_3)_n\end{aligned}$$

Corrector:

$$\begin{aligned}(y_1)_{n+1} &= (y_1)_n + 0.5h \cdot [(y_2)_n + (y_2)^p_{n+1}] \\ (y_2)_{n+1} &= (y_2)_n + 0.5h \cdot [(y_3)_n + (y_3)^p_{n+1}] \\ (y_3)_{n+1} &= (y_3)_n - 0.5h \cdot [(y_1)_n \cdot (y_3)_n + (y_1)^p_{n+1} \cdot (y_3)^p_{n+1}]\end{aligned}$$

2.9.1 Example: Newton's equation

Let's use Heun's method to solve Newton's equation from section 2.1,

$$y'(x) = 1 - 3x + y + x^2 + xy, \; y(0) = 0 \tag{2.82}$$

with analytical solution

$$y(x) = 3\sqrt{2\pi e} \cdot \exp\left(x\left(1+\frac{x}{2}\right)\right) \cdot \left[\operatorname{erf}\left(\frac{\sqrt{2}}{2}(1+x)\right) - \operatorname{erf}\left(\frac{\sqrt{2}}{2}\right)\right] + 4 \cdot \left[1 - \exp\left(x\left(1+\frac{x}{2}\right)\right)\right] - x \tag{2.83}$$

Here we have $f(x,y) = 1 - 3x + y + x^2 + xy = 1 + x(x-3) + (1+x)y$

The following program **NewtonHeun.py** solves this problem using Heun's method, and the resulting figure is shown in Figure 2.13.

```
# src-ch1/NewtonHeun.py
# Program Newton
# Computes the solution of Newton's 1st order equation (1671):
# dy/dx = 1-3*x + y + x^2 +x*y , y(0) = 0
# using Heun's method.

import numpy as np

xend = 2
dx = 0.1
steps = np.int(np.round(xend/dx, 0)) + 1
y, x = np.zeros((steps,1), float), np.zeros((steps,1), float)
y[0], x[0] = 0.0, 0.0

for n in range(0,steps-1):
    x[n+1] = (n+1)*dx
    xn = x[n]
    fn = 1 + xn*(xn-3) + y[n]*(1+xn)
    yp = y[n] + dx*fn
    xnp1 = x[n+1]
    fnp1 = 1 + xnp1*(xnp1-3) + yp*(1+xnp1)
    y[n+1] = y[n] + 0.5*dx*(fn+fnp1)

# Analytical solution
from scipy.special import erf
a = np.sqrt(2)/2
t1 = np.exp(x*(1+ x/2))
t2 = erf((1+x)*a)-erf(a)
ya = 3*np.sqrt(2*np.pi*np.exp(1))*t1*t2 + 4*(1-t1)-x

# plotting
import matplotlib.pylab as plt

# change some default values to make plots more readable
LNWDT=2; FNT=11
plt.rcParams['lines.linewidth'] = LNWDT; plt.rcParams['font.size'] = FNT

plt.plot(x, y, '-b.', x, ya, '-g.')
plt.xlabel('x')
plt.ylabel('y')

plt.title('Solution to Newton\'s equation')
plt.legend(['Heun', 'Analytical'], loc='best', frameon=False)
plt.grid()
#plt.savefig('newton_heun.png', transparent=True)
plt.show()
```

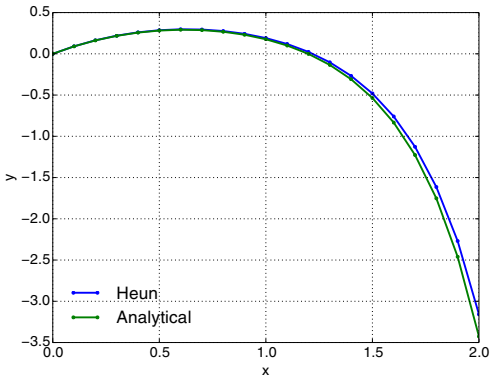

Figure 2.13: Velocity of falling sphere using Euler's and Heun's methods.

2.9.2 Example: Falling sphere with Heun's method

Let's go back to (2.6.6), and implement a new function `heun()` in the program FallingSphereEuler.py.

We recall the system of equations as

$$\begin{aligned} \frac{dz}{dt} &= v \\ \frac{dv}{dt} &= g - \alpha v^2 \end{aligned}$$

which by use of Heun's method in (2.80) and (2.81) becomes

Predictor:

$$\begin{aligned} z_{n+1}^p &= z_n + \Delta t v_n \\ v_{n+1}^p &= v_n + \Delta t \cdot (g - \alpha v_n^2) \end{aligned} \tag{2.84}$$

Corrector:

$$\begin{aligned} z_{n+1} &= z_n + 0.5\Delta t \cdot (v_n + v_{n+1}^p) \\ v_{n+1} &= v_n + 0.5\Delta t \cdot \left[2g - \alpha[v_n^2 + (v_{n+1}^p)^2\right] \end{aligned} \tag{2.85}$$

with initial values $z_0 = z(0) = 0,\ v_0 = v(0) = 0$. Note that we don't use the predictor z_{n+1}^p since it doesn't appear on the right hand side of the equation system.

One possible way of implementing this scheme is given in the following function named `heun()`, in the program **ODEschemes.py**:

```
def heun(func, z0, time):
    """The Heun scheme for solution of systems of ODEs.
    z0 is a vector for the initial conditions,
    the right hand side of the system is represented by func which returns
```

```
    a vector with the same size as z0 ."""

    z = np.zeros((np.size(time), np.size(z0)))
    z[0,:] = z0
    zp = np.zeros_like(z0)

    for i, t in enumerate(time[0:-1]):
        dt = time[i+1] - time[i]
        zp = z[i,:] + np.asarray(func(z[i,:],t))*dt   # Predictor step
        z[i+1,:] = z[i,:] + (np.asarray(func(z[i,:],t)) + np.asarray(func(zp,t+dt)))*dt/2.0 # Correc
```

Using the same time steps as in (2.6.6), we get the response plotted in Figure 2.14.

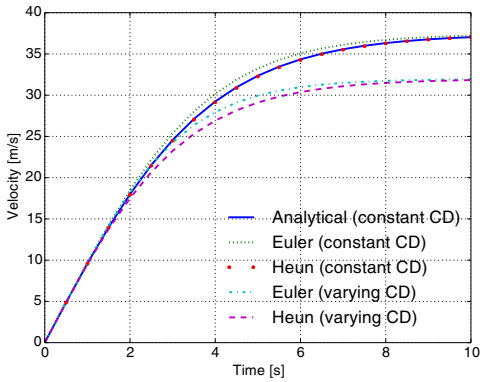

Figure 2.14: Velocity of falling sphere using Euler's and Heun's methods.

The complete program **FallingSphereEulerHeun.py** is listed below. Note that the solver functions `euler` and `heun` are imported from the script **ODEschemes.py**.

```
# src-ch1/FallingSphereEulerHeun.py;ODEschemes.py @ git@lrhgit/tkt4140/src/src-ch1/ODEschemes.py;
from DragCoefficientGeneric import cd_sphere
from ODEschemes import euler, heun
from matplotlib.pyplot import *
import numpy as np

# change some default values to make plots more readable
LNWDT=5; FNT=11
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT

g = 9.81       # Gravity m/s^2
d = 41.0e-3    # Diameter of the sphere
rho_f = 1.22   # Density of fluid [kg/m^3]
rho_s = 1275   # Density of sphere [kg/m^3]
nu = 1.5e-5    # Kinematical viscosity [m^2/s]
CD = 0.4       # Constant drag coefficient

def f(z, t):
    """2x2 system for sphere with constant drag."""
    zout = np.zeros_like(z)
```

```
    alpha = 3.0*rho_f/(4.0*rho_s*d)*CD
    zout[:] = [z[1], g - alpha*z[1]**2]
    return zout

def f2(z, t):
    """2x2 system for sphere with Re-dependent drag."""
    zout = np.zeros_like(z)
    v = abs(z[1])
    Re = v*d/nu
    CD = cd_sphere(Re)
    alpha = 3.0*rho_f/(4.0*rho_s*d)*CD
    zout[:] = [z[1], g - alpha*z[1]**2]
    return zout

# main program starts here

T = 10  # end of simulation
N = 20  # no of time steps
time = np.linspace(0, T, N+1)

z0=np.zeros(2)
z0[0] = 2.0

ze = euler(f, z0, time)      # compute response with constant CD using Euler's method
ze2 = euler(f2, z0, time)    # compute response with varying CD using Euler's method

zh = heun(f, z0, time)      # compute response with constant CD using Heun's method
zh2 = heun(f2, z0, time)    # compute response with varying CD using Heun's method

k1 = np.sqrt(g*4*rho_s*d/(3*rho_f*CD))
k2 = np.sqrt(3*rho_f*g*CD/(4*rho_s*d))
v_a = k1*np.tanh(k2*time)   # compute response with constant CD using analytical solution

# plotting

legends=[]
line_type=['-',':','.','-.','--']

plot(time, v_a, line_type[0])
legends.append('Analytical (constant CD)')

plot(time, ze[:,1], line_type[1])
legends.append('Euler (constant CD)')

plot(time, zh[:,1], line_type[2])
legends.append('Heun (constant CD)')

plot(time, ze2[:,1], line_type[3])
legends.append('Euler (varying CD)')

plot(time, zh2[:,1], line_type[4])
legends.append('Heun (varying CD)')

legend(legends, loc='best', frameon=False)

xlabel('Time [s]')
ylabel('Velocity [m/s]')
#savefig('example_sphere_falling_euler_heun.png', transparent=True)
show()
```

2.10 Generic second order Runge-Kutta method

We seek to derive a generic second order Runge-Kutta method for the solution of a generic ODE (2.2) on the form:

$$y_{n+1} = y_n + h\,(a_1\,K_1 + a_2\,K_2) \tag{2.86}$$

where a_1 and a_2 are some weights to be determined and K_1 and K_2 are derivatives on the form:

$$K_1 = f(x_n, y_n) \qquad \text{and} \qquad K_2 = f(x_n + p_1\,h, y_n + p_2\,K_1\,h) \tag{2.87}$$

By substitution of (2.87) in (2.86) we get:

$$y_{n+1} = y_n + a_1 h\ f(x_n, y_n) + a_2 h\ f(x_n + p_1\,h, y_n + p_2\,K_1\,h) \tag{2.88}$$

Now, we may find a Taylor-expansion of $f(x_n + p_1\,h, y_n + p_2\,K_1\,h)$:

$$\begin{aligned} f(x_n + p_1\,h, y_n + p_2\,K_1\,h) &= f + p_1 h\ f_x + p_2 K_1 h\ f_y + \text{h.o.t.} \\ &= f + p_1 h\ f_x + p_2 h\ f f_y + \text{h.o.t.} \end{aligned} \tag{2.89}$$

where we for convenience have adopted the common notation for partial derivatives

$$f_x \equiv \frac{\partial f}{\partial x} \qquad \text{and} \qquad f_y \equiv \frac{\partial f}{\partial y} \tag{2.90}$$

By substitution of (2.89) in (2.88) we eliminate the implicit dependency of y_{n+1}

$$\begin{aligned} y_{n+1} &= y_n + a_1 h\,f(x_n, y_n) + a_2 h\,(f + p_1 h\ f_x + p_2 h\ f f_y) \\ &= y_n + (a_1 + a_2)\ hf + (a_2 p_1\ f_x + a_2 p_2\ f f_y)\ h^2 \end{aligned} \tag{2.91}$$

Further, a second order derivative of the solution to (2.2) may be obtained by differentiation:

$$y'' = \frac{d^2 y}{dx^2} = \frac{df}{dx} = \partial f x\ \frac{dx}{dx} + \partial f y \frac{dy}{dx} = f_x + f f_y \tag{2.92}$$

which may be used in a second order Taylor expansion of the solution of (2.2):

$$y(x_n + h) = y_n + hy' + \frac{h^2}{2}\ y'' + O(h^3) \tag{2.93}$$

Substitution of (2.92) and (2.2) into (2.93) yields:

$$y(x_n + h) = y_n + hf + \frac{h^2}{2}\,(f_x + f f_y) + O(h^3) \tag{2.94}$$

Now the idea of the generic second order Runge-Kutta method is to select a_1, a_2, K_1, and K_2 in such a way that (2.91) approximates (2.94), which will be true if:

$$\begin{aligned} a_1 + a_2 &= 1 \\ a_2\, p_1 &= \frac{1}{2} \\ a_2\, p_2 &= \frac{1}{2} \end{aligned} \tag{2.95}$$

Note, that since we have 4 unknowns (a_1, a_2, K_1, and K_2) and only 3 equations in (2.95), several methods of the kind proposed in (2.86) are possible. 2.9 is retrieved by selecting $a_2 = 1/2$, from which we get $a_1 = 1/2$ and $p_1 = p_2 = 1$ from (2.95).

2.11 Runge-Kutta of 4th order

Euler's method and Heun's method belong to the Runge-Kutta family of explicit methods, and is respectively Runge-Kutta of 1st and 2nd order, the latter with one time use of corrector. Explicit Runge-Kutta schemes are single step schemes that try to copy the Taylor series expansion of the differential equation to a given order.

The classical Runge-Kutta scheme of 4th order (RK4) is given by

$$\begin{aligned} k_1 &= f(x_n, y_n) \\ k_2 &= f(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \\ k_3 &= f(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_2) \\ k_4 &= f(x_n + h, y_n + hk_3) \\ y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \tag{2.96}$$

We see that we are actually using Euler's method four times and find a weighted gradient. The local error is of order $O(h^5)$, while the global is of $O(h^4)$. We refer to [3].

Figure 2.15 shows a graphical illustration of the RK4 scheme.

In detail we have

1. In point (x_n, y_n) we know the gradient k_1 and use this when we go forward a step $h/2$ where the gradient k_2 is calculated.
2. With this gradient we start again in point (x_n, y_n), go forward a step $h/2$ and find a new gradient k_3.
3. With this gradient we start again in point (x_n, y_n), but go forward a complete step h and find a new gradient k_4.

4. The four gradients are averaged with weights 1/6, 2/6, 2/6 and 1/6. Using the averaged gradient we calculate the final value y_{n+1}.

Each of the steps above are Euler steps.

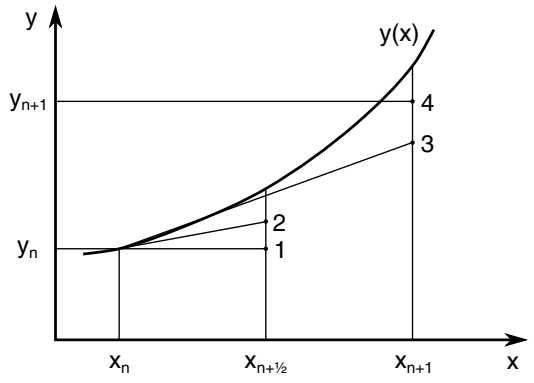

Figure 2.15: Illustration of the RK4 scheme.

Using (2.96) on the equation system in (2.64) we get

$$(y_1)_{n+1} = (y_1)_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$
$$(y_2)_{n+1} = (y_2)_n + \frac{h}{6}(l_1 + 2l_2 + 2l_3 + l_4) \tag{2.97}$$
$$(y_3)_{n+1} = (y_3)_n + \frac{h}{6}(m_1 + 2m_2 + 2m_3 + m_4)$$

(2.98)

where

$$
\begin{aligned}
k_1 &= y_2 \\
l_1 &= y_3 \\
m_1 &= -y_1 y_3
\end{aligned}
$$

$$
\begin{aligned}
k_2 &= (y_2 + h l_l/2) \\
l_2 &= (y_3 + h m_1/2) \\
m_2 &= -[(y_1 + h k_1/2)(y_3 + h m_1/2)]
\end{aligned}
$$

$$
\begin{aligned}
k_3 &= (y_2 + h l_2/2) \\
l_3 &= (y_3 + h m_2/2) \\
m_3 &= -[(y_1 + h k_2/2)(y_3 + h m_2/2)]
\end{aligned}
$$

$$
\begin{aligned}
k_4 &= (y_2 + h l_3) \\
l_4 &= (y_3 + h m_3) \\
m_4 &= -[(y_1 + h k_3)(y_3 + h m_3)
\end{aligned}
$$

2.11.1 Example: Falling sphere using RK4

Let's implement the RK4 scheme and add it to the falling sphere example. The scheme has been implemented in the function `rk4()`, and is given below

```
def rk4(func, z0, time):
    """The Runge-Kutta 4 scheme for solution of systems of ODEs.
    z0 is a vector for the initial conditions,
    the right hand side of the system is represented by func which returns
    a vector with the same size as z0 ."""

    z = np.zeros((np.size(time),np.size(z0)))
    z[0,:] = z0
    zp = np.zeros_like(z0)

    for i, t in enumerate(time[0:-1]):
        dt = time[i+1] - time[i]
        dt2 = dt/2.0
        k1 = np.asarray(func(z[i,:], t))                # predictor step 1
        k2 = np.asarray(func(z[i,:] + k1*dt2, t + dt2)) # predictor step 2
        k3 = np.asarray(func(z[i,:] + k2*dt2, t + dt2)) # predictor step 3
        k4 = np.asarray(func(z[i,:] + k3*dt, t + dt))   # predictor step 4
        z[i+1,:] = z[i,:] + dt/6.0*(k1 + 2.0*k2 + 2.0*k3 + k4) # Corrector step
```

Figure 2.16 shows the results using Euler, Heun and RK4. AS seen, RK4 and Heun are more accurate than Euler. The complete program **FallingSphereEulerHeunRK4.py** is listed below. The functions `euler`, `heun` and `rk4` are imported from the program **ODEschemes.py**.

```
# src-ch1/FallingSphereEulerHeunRK4.py;ODEschemes.py @ git@lrhgit/tkt4140/src/src-ch1/ODEschemes.py;
```

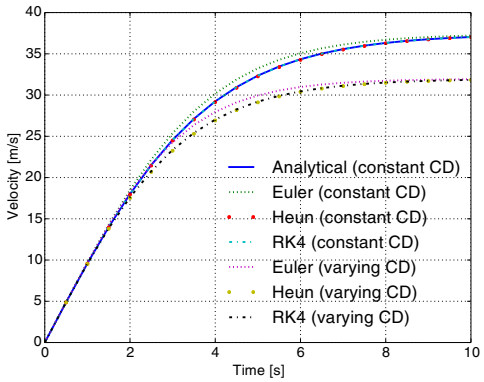

Figure 2.16: Velocity of falling sphere using Euler, Heun and RK4.

```
from DragCoefficientGeneric import cd_sphere
from ODEschemes import euler, heun, rk4
from matplotlib.pyplot import *
import numpy as np

# change some default values to make plots more readable
LNWDT=5; FNT=11
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT

g = 9.81       # Gravity m/s^2
d = 41.0e-3    # Diameter of the sphere
rho_f = 1.22   # Density of fluid [kg/m^3]
rho_s = 1275   # Density of sphere [kg/m^3]
nu = 1.5e-5    # Kinematical viscosity [m^2/s]
CD = 0.4       # Constant drag coefficient

def f(z, t):
    """2x2 system for sphere with constant drag."""
    zout = np.zeros_like(z)
    alpha = 3.0*rho_f/(4.0*rho_s*d)*CD
    zout[:] = [z[1], g - alpha*z[1]**2]
    return zout

def f2(z, t):
    """2x2 system for sphere with Re-dependent drag."""
    zout = np.zeros_like(z)
    v = abs(z[1])
    Re = v*d/nu
    CD = cd_sphere(Re)
    alpha = 3.0*rho_f/(4.0*rho_s*d)*CD
    zout[:] = [z[1], g - alpha*z[1]**2]
    return zout

# main program starts here

T = 10  # end of simulation
N = 20  # no of time steps
time = np.linspace(0, T, N+1)
```

```
z0=np.zeros(2)
z0[0] = 2.0

ze = euler(f, z0, time)      # compute response with constant CD using Euler's method
ze2 = euler(f2, z0, time)    # compute response with varying CD using Euler's method

zh = heun(f, z0, time)      # compute response with constant CD using Heun's method
zh2 = heun(f2, z0, time)    # compute response with varying CD using Heun's method

zrk4 = rk4(f, z0, time)      # compute response with constant CD using RK4
zrk4_2 = rk4(f2, z0, time)  # compute response with varying CD using RK4

k1 = np.sqrt(g*4*rho_s*d/(3*rho_f*CD))
k2 = np.sqrt(3*rho_f*g*CD/(4*rho_s*d))
v_a = k1*np.tanh(k2*time)    # compute response with constant CD using analytical solution

# plotting

legends=[]
line_type=['-',':','.','-.',':','.','-.']

plot(time, v_a, line_type[0])
legends.append('Analytical (constant CD)')

plot(time, ze[:,1], line_type[1])
legends.append('Euler (constant CD)')

plot(time, zh[:,1], line_type[2])
legends.append('Heun (constant CD)')

plot(time, zrk4[:,1], line_type[3])
legends.append('RK4 (constant CD)')

plot(time, ze2[:,1], line_type[4])
legends.append('Euler (varying CD)')

plot(time, zh2[:,1], line_type[5])
legends.append('Heun (varying CD)')

plot(time, zrk4_2[:,1], line_type[6])
legends.append('RK4 (varying CD)')

legend(legends, loc='best', frameon=False)


xlabel('Time [s]')
ylabel('Velocity [m/s]')
#savefig('example_sphere_falling_euler_heun_rk4.png', transparent=True)
show()
```

2.11.2 Example: Particle motion in two dimensions

In this example we will calculate the motion of a particle in two dimensions. First we will calculate the motion of a smooth ball with drag coefficient given by the previously defined function `cd_sphere()` (see (2.6.4)), and then of a golf ball with drag and lift. The problem is illustrated in the following figure:

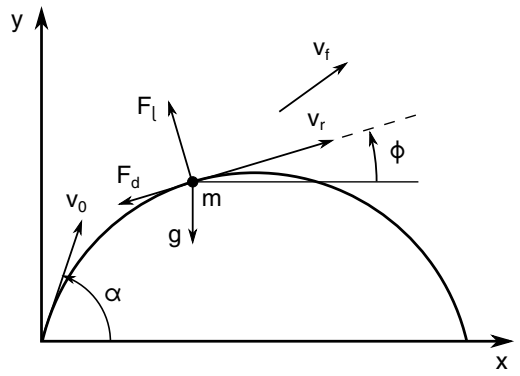

Figure 2.17: An illustration of a particle subjected to both gravity and drag in two dimensions.

where v is the absolute velocity, $v_f =$ is the velocity of the fluid, $v_r = v - v_f$ is the relative velocity between the fluid and the ball, α is the elevation angle, v_0 is the initial velocity and ϕ is the angle between the x-axis and v_r.

$\mathbf{F}_l$ is the lift force stemming from the rotation of the ball (the Magnus-effect) and is normal to v_r. With the given direction the ball rotates counter-clockwise (backspin). $\mathbf{F}_d$ is the fluids resistance against the motion and is parallel to v_r. These forces are given by

$$\mathbf{F}_d = \frac{1}{2}\rho_f A C_D v_r^2 \tag{2.99}$$

$$\mathbf{F}_l = \frac{1}{2}\rho_f A C_L v_r^2 \tag{2.100}$$

C_D is the drag coefficient, C_L is the lift coefficient, A is the area projected in the velocity direction and ρ_F is the density of the fluid.

Newton's law in x- and y-directions gives

$$\frac{dv_x}{dt} = -\rho_f \frac{A}{2m} v_r^2 (C_D \cdot \cos(\phi) + C_L \sin(\phi)) \tag{2.101}$$

$$\frac{dv_y}{dt} = \rho_f \frac{A}{2m} v_r^2 (C_L \cdot \cos(\phi) - C_D \sin(\phi)) - g \tag{2.102}$$

From the figure we have

$$\cos(\phi) = \frac{v_{rx}}{v_r}$$

$$\sin(\phi) = \frac{v_{ry}}{v_r}$$

We assume that the particle is a sphere, such that $C = \rho_f \frac{A}{2m} = \frac{3\rho_f}{4\rho_k d}$ as in (2.6.4). Here d is the diameter of the sphere and ρ_k the density of the sphere.

Now (2.101) and (2.102) become

$$\frac{dv_x}{dt} = -C \cdot v_r(C_D \cdot v_{rx} + C_L \cdot v_{ry}) \tag{2.103}$$

$$\frac{dv_y}{dt} = C \cdot v_r(C_L \cdot v_{rx} - C_D \cdot v_{ry}) - g \tag{2.104}$$

With $\frac{dx}{dt} = v_x$ and $\frac{dy}{dt} = v_y$ we get a system of 1st order equations as follows,

$$\begin{aligned}
\frac{dx}{dt} &= v_x \\
\frac{dy}{dt} &= v_y \\
\frac{dv_x}{dt} &= -C \cdot v_r(C_D \cdot v_{rx} + C_L \cdot v_{ry}) \\
\frac{dv_y}{dt} &= C \cdot v_r(C_L \cdot v_{rx} - C_D \cdot v_{ry}) - g
\end{aligned} \tag{2.105}$$

Introducing the notation $x = y_1$, $y = y_2$, $v_x = y_3$, $v_y = y_4$, we get

$$\begin{aligned}
\frac{dy_1}{dt} &= y_3 \\
\frac{dy_2}{dt} &= y_4 \\
\frac{dy_3}{dt} &= -C \cdot v_r(C_D \cdot v_{rx} + C_L \cdot v_{ry}) \\
\frac{dy_4}{dt} &= C \cdot v_r(C_L \cdot v_{rx} - C_D \cdot v_{ry}) - g
\end{aligned} \tag{2.106}$$

Here we have $v_{rx} = v_x - v_{fx} = y_3 - v_{fx}$, $v_{ry} = v_y - v_{fy} = y_4 - v_{fy}$, $v_r = \sqrt{v_{rx}^2 + v_{ry}^2}$

Initial conditions for $t = 0$ are

$$\begin{aligned}
y_1 &= y_2 = 0 \\
y_3 &= v_0 \cos(\alpha) \\
y_4 &= v_0 \sin(\alpha)
\end{aligned}$$

Let's first look at the case of a smooth ball. We use the following data (which are the data for a golf ball):

Diameter $d = 41\text{mm}$, mass $m = 46\text{g}$ which gives $\rho_k = \dfrac{6m}{\pi d^3} = 1275\text{kg/m}^3$

We use the initial velocity $v_0 = 50$ m/s and solve (2.106) using the Runge-Kutta 4 scheme. In this example we have used the Python package **Odespy** (ODE Software in Python), which offers a large collection of functions for solving ODE's. The RK4 scheme available in Odespy is used herein.

The right hand side in (2.106) is implemented as the following function:

```
def f(z, t):
    """4x4 system for smooth sphere with drag in two directions."""
    zout = np.zeros_like(z)
    C = 3.0*rho_f/(4.0*rho_s*d)
    vrx = z[2] - vfx
    vry = z[3] - vfy
    vr = np.sqrt(vrx**2 + vry**2)
    Re = vr*d/nu
    CD = cd_sphere(Re) # using the already defined function
    zout[:] = [z[2], z[3], -C*vr*(CD*vrx), C*vr*(-CD*vry) - g]
```

Note that we have used the function `cd_sphere()` defined in (2.6.4) to calculate the drag coefficient of the smooth sphere.

The results are shown for some initial angles in Figure 2.18.

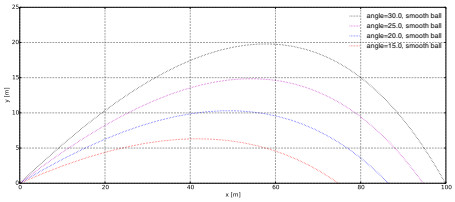

Figure 2.18: Motion of smooth ball with drag.

Now let's look at the same case for a golf ball. The dimension and weight are the same as for the sphere. Now we need to account for the lift force from the spin of the ball. In addition, the drag data for a golf ball are completely different from the smooth sphere. We use the data from Bearman and Harvey [1] who measured the drag and lift of a golf ball for different spin velocities in a vindtunnel. We choose as an example 3500 rpm, and an initial velocity of $v_0 = 50$ m/s.

The right hand side in (2.106) is now implemented as the following function:

```
def f3(z, t):
    """4x4 system for golf ball with drag and lift in two directions."""
    zout = np.zeros_like(z)
    C = 3.0*rho_f/(4.0*rho_s*d)
    vrx = z[2] - vfx
    vry = z[3] - vfy
    vr = np.sqrt(vrx**2 + vry**2)
    Re = vr*d/nu
    CD, CL = cdcl(vr, nrpm)
    zout[:] = [z[2], z[3], -C*vr*(CD*vrx + CL*vry), C*vr*(CL*vrx - CD*vry) - g]
```

The function `cdcl()` (may be downloaded here) gives the drag and lift data for a given velocity and spin.

The results are shown in Figure 2.19. The motion of a golf ball with drag but without lift is also included. We see that the golf ball goes much farther than the smooth sphere, due to less drag and the lift.

The complete program **ParticleMotion2D.py** is listed below.

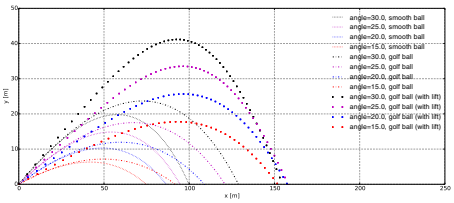

Figure 2.19: Motion of golf ball with drag and lift.

```
# src-ch1/ParticleMotion2D.py;DragCoefficientGeneric.py @ git@lrhgit/tkt4140/src/src-ch1/DragCoeffic

from DragCoefficientGeneric import cd_sphere
from cdclgolfball import cdcl
from matplotlib.pyplot import *
import numpy as np
import odespy

g = 9.81       # Gravity [m/s^2]
nu = 1.5e-5    # Kinematical viscosity [m^2/s]
rho_f = 1.20   # Density of fluid [kg/m^3]
rho_s = 1275   # Density of sphere [kg/m^3]
d = 41.0e-3    # Diameter of the sphere [m]
v0 = 50.0      # Initial velocity [m/s]
vfx = 0.0      # x-component of fluid's velocity
vfy = 0.0      # y-component of fluid's velocity

nrpm = 3500    # no of rpm of golf ball

# smooth ball
def f(z, t):
    """4x4 system for smooth sphere with drag in two directions."""
    zout = np.zeros_like(z)
    C = 3.0*rho_f/(4.0*rho_s*d)
    vrx = z[2] - vfx
    vry = z[3] - vfy
    vr = np.sqrt(vrx**2 + vry**2)
    Re = vr*d/nu
    CD = cd_sphere(Re) # using the already defined function
    zout[:] = [z[2], z[3], -C*vr*(CD*vrx), C*vr*(-CD*vry) - g]
    return zout

# golf ball without lift
def f2(z, t):
    """4x4 system for golf ball with drag in two directions."""
    zout = np.zeros_like(z)
    C = 3.0*rho_f/(4.0*rho_s*d)
    vrx = z[2] - vfx
    vry = z[3] - vfy
    vr = np.sqrt(vrx**2 + vry**2)
    Re = vr*d/nu
    CD, CL = cdcl(vr, nrpm)
    zout[:] = [z[2], z[3], -C*vr*(CD*vrx), C*vr*(-CD*vry) - g]
    return zout

# golf ball with lift
```

```
def f3(z, t):
    """4x4 system for golf ball with drag and lift in two directions."""
    zout = np.zeros_like(z)
    C = 3.0*rho_f/(4.0*rho_s*d)
    vrx = z[2] - vfx
    vry = z[3] - vfy
    vr = np.sqrt(vrx**2 + vry**2)
    Re = vr*d/nu
    CD, CL = cdcl(vr, nrpm)
    zout[:] = [z[2], z[3], -C*vr*(CD*vrx + CL*vry), C*vr*(CL*vrx - CD*vry) - g]
    return zout


# main program starts here

T = 7   # end of simulation
N = 60  # no of time steps
time = np.linspace(0, T, N+1)

N2 = 4
alfa = np.linspace(30, 15, N2)   # Angle of elevation [degrees]
angle = alfa*np.pi/180.0 # convert to radians

legends=[]
line_color=['k','m','b','r']
figure(figsize=(20, 8))
hold('on')
LNWDT=4; FNT=18
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT

# computing and plotting

# smooth ball with drag
for i in range(0,N2):
    z0 = np.zeros(4)
    z0[2] = v0*np.cos(angle[i])
    z0[3] = v0*np.sin(angle[i])
    solver = odespy.RK4(f)
    solver.set_initial_condition(z0)
    z, t = solver.solve(time)
    plot(z[:,0], z[:,1], ':', color=line_color[i])
    legends.append('angle='+str(alfa[i])+', smooth ball')

# golf ball with drag
for i in range(0,N2):
    z0 = np.zeros(4)
    z0[2] = v0*np.cos(angle[i])
    z0[3] = v0*np.sin(angle[i])
    solver = odespy.RK4(f2)
    solver.set_initial_condition(z0)
    z, t = solver.solve(time)
    plot(z[:,0], z[:,1], '-.', color=line_color[i])
    legends.append('angle='+str(alfa[i])+', golf ball')

# golf ball with drag and lift
for i in range(0,N2):
    z0 = np.zeros(4)
    z0[2] = v0*np.cos(angle[i])
    z0[3] = v0*np.sin(angle[i])
    solver = odespy.RK4(f3)
    solver.set_initial_condition(z0)
```

```
    z, t = solver.solve(time)
    plot(z[:,0], z[:,1], '.', color=line_color[i])
    legends.append('angle='+str(alfa[i])+', golf ball (with lift)')

legend(legends, loc='best', frameon=False)
xlabel('x [m]')
ylabel('y [m]')
axis([0, 250, 0, 50])
#savefig('example_particle_motion_2d_2.png', transparent=True)
show()
```

2.12 Basic notions on numerical methods for IVPs

In this section we define basic notions about numerical method that are needed in order to identify their essential properties.

Consider the first order ODE

$$y' = f(x, y), \tag{2.107}$$

where $f(x, y)$ is the source term. Moreover, the ODE is equipped with initial conditions

$$y(x_0) = y_0. \tag{2.108}$$

Finally, we define $y(x_n)$ as the solution of IVP defined by (2.107) and (2.108) evaluated at $x = x_n$, whereas y_n is a numerical approximation of $y(x_n)$ at the same location. We can now define an approximation **error** as

$$e_n = y(x_n) - y_n \tag{2.109}$$

and some useful variants such as the **absolute error**

$$|e_n| = |y(x_n) - y_n| \tag{2.110}$$

and the **relative error**

$$r_n = \frac{|y(x_n) - y_n|}{|y(x_n)|}. \tag{2.111}$$

We can define a **generic explicit numerical scheme** for Equation (2.107) as

$$y_{n+1} = y_n + h\phi(x_n, y_n, h), \tag{2.112}$$

where h is the discretization step for x, *i.e.* $h = x_{n+1} - x_n$, whereas $\phi(x_n, y_n, h)$ is the increment function. Note that one can define a generic implicit or multi-step scheme by changing the arguments of ϕ.

Having defined a generic numerical scheme, we can say that it is consistent if

$$\lim_{h \to 0} \phi(x, y, h) = \phi(x, y, 0) = f(x, y).$$

In short, the approximation produced by a consistent numerical scheme will converge to the original ODE as $h \to 0$.

Two additional useful definitions are the **exact differential operator**

$$L_e(y) = y' - f(x, y) = 0 \tag{2.113}$$

and the **approximate differential operator**

$$L_a(y_n) = y_{n+1} - [y_n + h\phi(x_n, y_n, h)] = 0. \tag{2.114}$$

Now we have introduced all necessary concepts to define the **local truncation error (LTE)**

$$\tau_n = \frac{1}{h} L_a(y(x_n)). \tag{2.115}$$

In practice, one applies the approximate differential operator, that is defined by the numerical scheme, to the exact solution of the problem at hand. The evaluation of the exact solution for different x around x_n, as required by L_a is performed using a Taylor series expansion.

Finally, we can state that a scheme is p-th order accurate by examining its LTE and observing its leading term

$$\tau_n = Ch^p + H.O.T., \tag{2.116}$$

where C is a constant, independent of h and $H.O.T.$ are the higher order terms of the LTE.

Example: LTE for Euler's scheme

Consider the IVP defined by

$$y' = \lambda y, \tag{2.117}$$

with initial condition

$$y(0) = 1. \tag{2.118}$$

The approximation operator (2.114) for Euler's scheme is

$$L_a^{euler} = y_{n+1} - [y_n + h\lambda y_n], \tag{2.119}$$

whereas the LTE can be computed by inserting $y(x)$ in (2.119)

$$\tau_n = \frac{1}{h}\left\{L_a(y(x_n))\right\} = \frac{1}{h}\left\{y(x_{n+1}) - [y(x_n) + h\lambda y(x_n)]\right\}, \tag{2.120}$$

$$= \frac{1}{h}\left\{y(x_n) + hy'(x_n) + \frac{h^2}{2}y''(x_n) + \ldots + \frac{1}{p!}h^p y^{(p)}(x_n) - y(x_n) - h\lambda y(x_n)\right\} \tag{2.121}$$

$$= \frac{1}{2}hy''(x_n) + \ldots + \frac{1}{p!}h^{p-1}y^{(p)}(x_n) \tag{2.122}$$

$$\approx \frac{1}{2}hy''(x_n). \tag{2.123}$$

2.13 Variable time stepping methods

Recall the local truncation error of the Euler scheme applied to the model exponential decay equation (2.117), computed in (2.123)

$$\tau_n^{euler} = \frac{1}{2}hy''(x_n).$$

The error of the numerical approximation depends on h in a linear fashion, as expected for a first order scheme. However, the approximation error does also depend on the solution itself, more precisely on its second derivative. This is a property of the problem and we have no way of intervening on it. If one wants to set a maximum admissible error, then the only parameter left is the time step h. The wisest manner to use this freedom is to adopt a variable time-stepping strategy, in which the time step is adapted according to estimates of the error as long as one advances in x. Here we briefly present one possible approach, namely the 5-th order Runge-Kutta-Fehlberg scheme with variable time step (RKF45), as presented for example in [3]. Schematically, the main steps for implementing such scheme are listed bellow:

1. Assuming that the solution at x_n is available, compute a fourth and a fifth order RKF solution at time x_{n+1}
2. Compute an estimate of the error: $e_{n+1} \approx |y_{n+1}^{RKF5} - y_{n+1}^{RKF4}|$
3. Increase or decrease h accoring to pre-defined maximum and minimum admissible errors.

An example of how to implement RKF45 is given bellow. Solutions obtained using this method and a first order Euler fixed-step scheme are shown in Figure 2.20. Moreover, this figures shows how time step varies as one moves along the x axis.

```
# src-ch1/fixed_vs_variable.py;

import numpy as np
import matplotlib.pyplot as plt


# Solve
# y' = beta * y
# y(0) = 1.
# with explicit Euler (fixed h) and RKF45

# ode coefficient
beta = -10.
# ode initial condition
y0 = 1.
# time step for Euler's scheme
h = 0.02
# domain
xL = 0.
xR = 1.
# number of intervals for Euler's scheme
N = int((xR-xL)/h)
# correct h for Euler's scheme
h = (xR-xL)/float(N)
# independent variable array for Euler's scheme
x = np.linspace(xL,xR,N)
# unknown variable array for Euler's scheme
y = np.zeros(N)
# x for exact solution
xexact = np.linspace(xL,xR,max(1000.,100*N))

# ode function
def f(x,y):
    return beta * y

# exact solution
def exact(x):
    return y0*np.exp(beta*x)

# Euler's method increment
def eulerIncrementFunction(x,yn,h,ode):
    return ode(x,yn)

# RKF45 increment
def rkf45step(x,yn,h,ode):
    # min and max time step
    hmin = 1e-5
    hmax = 5e-1
    # min and max errors
    emin = 1e-7
    emax = 1e-5
    # max number of iterations
    nMax = 100
    # match final time
    if x+h > xR:
        h = xR-x
    # loop to find correct error (h)
    update = 0
    for i in range(nMax):
        k1 = ode(x,yn)
        k2 = ode(x+h/4.,yn+h/4.*k1)
```

```
        k3 = ode(x+3./8.*h,yn+3./32.*h*k1+9./32.*h*k2)
        k4 = ode(x+12./13.*h,yn+1932./2197.*h*k1-7200./2197.*h*k2+7296./2197.*h*k3)
        k5 = ode(x+h,yn+439./216.*h*k1-8.*h*k2+3680./513.*h*k3-845./4104.*h*k4)
        k6 = ode(x+h/2.,yn-8./27.*h*k1+2.*h*k2-3544./2565.*h*k3+1859./4140.*h*k4-11./40.*h*k5)
        # 4th order solution
        y4 = yn + h * (25./216*k1 + 1408./2565.*k3+2197./4104.*k4-1./5.*k5)
        # fifth order solution
        y5 = yn + h * (16./135.*k1 + 6656./12825.*k3 + 28561./56430.*k4 - 9./50.*k5 +2./55.*k6)
        # error estimate
        er = np.abs(y5-y4)

        if er < emin:
            # if error small, enlarge h, but match final simulation time
            h = min(2.*h,hmax)
            if x+h > xR:
                h = xR-x
                break
        elif er > emax:
            # if error big, reduce h
            h = max(h/2.,hmin)
        else:
            # error is ok, take this h and y5
            break

    if i==nMax-1:
        print "max number of iterations reached, check parameters"

    return x+h, y5, h , er

# time loop for euler
y[0] = y0
for i in range(N-1):
    y[i+1] = y[i] + h * eulerIncrementFunction(x[i],y[i],h,f)


# time loop for RKF45
nMax = 1000

xrk = np.zeros(1)
yrk = y0*np.ones(1)
hrk = np.zeros(1)
h = 0.5

for i in range(nMax):
    xloc , yloc, h , er = rkf45step(xrk[-1],yrk[-1],h,f)
    xrk = np.append(xrk,xloc)
    yrk = np.append(yrk,yloc)
    if i==0:
        hrk[i] = h
    else:
        hrk = np.append(hrk,h)
    if xrk[-1]==xR:
        break

plt.subplot(211)
plt.plot(xexact,exact(xexact),'b-',label='Exact')
plt.plot(x,y,'rx',label='Euler')
plt.plot(xrk,yrk,'o', markersize=7,markeredgewidth=1,markeredgecolor='g',markerfacecolor='None',label
plt.xlabel('x')
plt.ylabel('y(x)')
plt.legend()
```

```
plt.subplot(212)
plt.plot(xrk[1:],hrk,'o', markersize=7,markeredgewidth=1,markeredgecolor='g',markerfacecolor='None',l
plt.ylabel('h')
plt.legend(loc='best')
#plt.savefig('fixed_vs_variable.png')
plt.show()
```

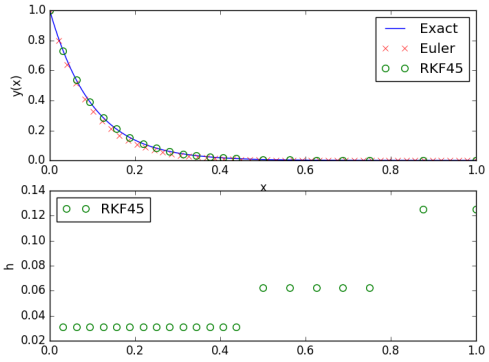

Figure 2.20: Numerical solution for the model exponential decay equation (2.117) using the Euler scheme and the RKF45 variable time step scheme (top panel) and time step used at each iteration by the variable time stepping scheme (bottom panel).

2.14 Numerical error as a function of Δt for ODE-schemes

To investigate whether the various ODE-schemes in our module 'ODEschemes.py' have the expected, theoretical order, we proceed in the same manner as outlined in (2.8.1). The complete code is listed at the end of this section but we will highlight and explain some details in the following.

To test the numerical order for the schemes we solve a somewhat general linear ODE:

$$\begin{aligned} u'(t) &= a\,u + b \\ u(t_0) &= u_0 \end{aligned} \tag{2.124}$$

which has the analytical solutions:

$$u = \begin{cases} \left(u_0 + \frac{b}{a}\right)\ e^{a\,t} - \frac{b}{a}, & a \neq 0 \\ u_0 + b\,t, & a = 0 \end{cases} \tag{2.125}$$

The right hand side defining the differential equation has been implemented in function `f3` and the corresponding analytical solution is computed by `u_nonlin_analytical`:

```
def f3(z, t, a=2.0, b=-1.0):
    """ """
    return a*z + b

def u_nonlin_analytical(u0, t, a=2.0, b=-1.0):
    from numpy import exp
    TOL = 1E-14
    if (abs(a)>TOL):
        return (u0 + b/a)*exp(a*t)-b/a
    else:
        return u0 + b*t
```

— The basic idea for the convergence test in the function `convergence_test` is that we start out by solving numerically an ODE with an analytical solution on a relatively coarse grid, allowing for direct computations of the error. We then reduce the timestep by a factor two (or double the grid size), repeatedly, and compute the error for each grid and compare it with the error of previous grid.

The Euler scheme (2.55) is $O(h)$, whereas the Heun scheme (2.78) is $O(h^2)$, and Runge-Kutta (2.96) is $O(h^4)$, where the h denote a generic step size which for the current example is the timestep Δt. The order of a particular scheme is given exponent n in the error term $O(h^n)$. Consequently, the Euler scheme is a first oder scheme, Heun is second order, whereas Runge-Kutta is fourth order.

By letting ϵ_{i+1} and ϵ_i denote the errors on two consecutive grids with corresponding timesteps $\Delta t_{i+1} = \dfrac{\Delta t_i}{2}$. The errors ϵ_{i+1} and ϵ_i for a scheme of order n are then related by:

$$\epsilon_{i+1} = \frac{1}{2^n}\epsilon_i \tag{2.126}$$

Consequently, whenever ϵ_{i+1} and ϵ_i are known from consecutive simulations an estimate of the order of the scheme may be obtained by:

$$n \approx \log_2 \frac{\epsilon_i}{\epsilon_{i+1}} \tag{2.127}$$

The theoretical value of n is thus $n = 1$ for Euler's method, $n = 2$ for Heun's method and $n = 4$ for RK4.

In the function `convergence_test` the schemes we will subject to a convergence test is ordered in a list `scheme_list`. This allows for a convenient loop over all schemes with the clause: `for scheme in scheme_list:`. Subsequently, for each scheme we refine the initial grid (`N=30`) `Ndts` times in the loop `for i in range(Ndts+1):` and solve and compute the order estimate given by (2.127) with the clause `order_approx.append(previous_max_log_err - max_log_err)`. Note that we can not compute this for the first iteration (`i=0`), and that we use a an initial empty list `order_approx` to store the approximation of the order `n` for each grid refinement. For each grid we plot $\log_2(\epsilon)$ as a function of time with: `plot(time[1:], log_error, linestyles[i]+colors[iclr], markevery=N/5)` and for each plot we construct the corresponding legend by appending a new element to the legends-list `legends.append(scheme.func_name +': N = ' + str(N))`. This construct produces a string with both the scheme name and the number of

elements N. The plot is not reproduced below, but you may see the result by downloading and running the module yourself.

Having completed the given number of refinements `Ndts` for a specific scheme we store the `order_approx` for the scheme in a dictionary using the name of the scheme as a key by `schemes_orders[scheme.func_name] = order_approx`. This allows for an illustrative plot of the order estimate for each scheme with the clause:

```
for key in schemes_orders:
            plot(N_list, (np.asarray(schemes_orders[key])))
```

and the resulting plot is shown in Figure 2.21, and we see that our numerical approximations for the orders of our schemes approach the theoretical values as the number of timesteps increase (or as the timestep is reduced by a factor two consecutively).

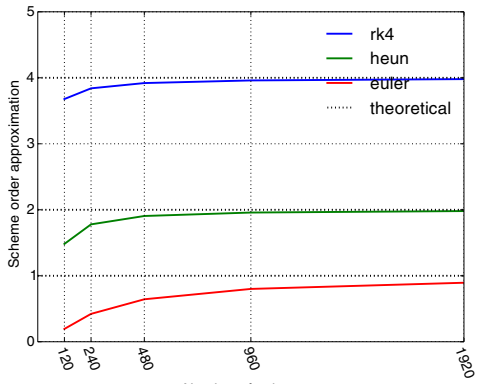

Figure 2.21: The convergence rate for the various ODE-solvers a function of the number of timesteps.

The complete function `convergence_test` is a part of the module `ODEschemes` and is isolated below:

```
def convergence_test():
    """ Test convergence rate of the methods """
    from numpy import linspace, size, abs, log10, mean, log2
    figure()
    tol = 1E-15
    T = 8.0   # end of simulation
    Ndts = 5 # Number of times to refine timestep in convergence test

    z0 = 2

    schemes =[euler, heun, rk4]
    legends=[]
    schemes_order={}

    colors = ['r', 'g', 'b', 'm', 'k', 'y', 'c']
```

```
        linestyles = ['-', '--', '-.', ':', 'v--', '*-.']
        iclr = 0
        for scheme in schemes:
            N = 30    # no of time steps
            time = linspace(0, T, N+1)

            order_approx = []

            for i in range(Ndts+1):
                z = scheme(f3, z0, time)
                abs_error = abs(u_nonlin_analytical(z0, time)-z[:,0])
                log_error = log2(abs_error[1:]) # Drop 1st elt to avoid log2-problems (1st elt is ze
                max_log_err = max(log_error)
                plot(time[1:], log_error, linestyles[i]+colors[iclr], markevery=N/5)
                legends.append(scheme.func_name +': N = ' + str(N))
                hold('on')

                if i > 0: # Compute the log2 error difference
                    order_approx.append(previous_max_log_err - max_log_err)
                previous_max_log_err = max_log_err

                N *=2
                time = linspace(0, T, N+1)

            schemes_order[scheme.func_name] = order_approx
            iclr += 1

        legend(legends, loc='best')
        xlabel('Time')
        ylabel('log(error)')
        grid()

        N = N/2**Ndts
        N_list = [N*2**i for i in range(1, Ndts+1)]
        N_list = np.asarray(N_list)

        figure()
        for key in schemes_order:
            plot(N_list, (np.asarray(schemes_order[key])))

        # Plot theoretical n for 1st, 2nd and 4th order schemes
        axhline(1.0, xmin=0, xmax=N, linestyle=':', color='k')
        axhline(2.0, xmin=0, xmax=N, linestyle=':', color='k')
        axhline(4.0, xmin=0, xmax=N, linestyle=':', color='k')
        xticks(N_list, rotation=-70)
        legends = schemes_order.keys()
        legends.append('theoretical')
        legend(legends, loc='best', frameon=False)
        xlabel('Number of unknowns')
        ylabel('Scheme order approximation')
        axis([0, max(N_list), 0, 5])
#        savefig('ConvergenceODEschemes.png', transparent=True)

    def manufactured_solution():
        """ Test convergence rate of the methods, by using the Method of Manufactured solutions.
            The coefficient function f is chosen to be the normal distribution
            f = (1/(sigma*sqrt(2*pi)))*exp(-((t-mu)**2)/(2*sigma**2)).
            The ODE to be solved is than chosen to be: f''' + f''*f + f' = RHS,
            leading to to f''' = RHS - f''*f - f
         """
        from numpy import linspace, size, abs, log10, mean, log2
```

```
from sympy import exp, symbols, diff, lambdify
from math import sqrt, pi

print "solving equation f''' + f''*f + f' = RHS"
print "which lead to f''' = RHS - f''*f - f"
t = symbols('t')
sigma=0.5 # standard deviation
mu=0.5 # mean value
Domain=[-1.5, 2.5]
t0 = Domain[0]
tend = Domain[1]

f = (1/(sigma*sqrt(2*pi)))*exp(-((t-mu)**2)/(2*sigma**2))
dfdt = diff(f, t)
d2fdt = diff(dfdt, t)
d3fdt = diff(d2fdt, t)
RHS = d3fdt + dfdt*d2fdt + f

f = lambdify([t], f)
dfdt = lambdify([t], dfdt)
d2fdt = lambdify([t], d2fdt)
RHS = lambdify([t], RHS)

def func(y,t):
    yout = np.zeros_like(y)
    yout[:] = [y[1], y[2], RHS(t) -y[0]- y[1]*y[2]]

    return yout

z0 = np.array([f(t0), dfdt(t0), d2fdt(t0)])

figure()
tol = 1E-15
Ndts = 5 # Number of times to refine timestep in convergence test
schemes =[euler, heun, rk4]
legends=[]
schemes_order={}

colors = ['r', 'g', 'b', 'm', 'k', 'y', 'c']
linestyles = ['-', '--', '-.', ':', 'v--', '*-.']
iclr = 0
for scheme in schemes:
    N = 100    # no of time steps
    time = linspace(t0, tend, N+1)
    fanalytic = np.zeros_like(time)
    k = 0
    for tau in time:
        fanalytic[k] = f(tau)
        k = k + 1

    order_approx = []

    for i in range(Ndts+1):
        z = scheme(func, z0, time)
        abs_error = abs(fanalytic-z[:,0])
        log_error = log2(abs_error[1:]) # Drop 1st elt to avoid log2-problems (1st elt is ze
        max_log_err = max(log_error)
        plot(time[1:], log_error, linestyles[i]+colors[iclr], markevery=N/5)
        legends.append(scheme.func_name +': N = ' + str(N))
        hold('on')
```

```
                if i > 0: # Compute the log2 error difference
                    order_approx.append(previous_max_log_err - max_log_err)
                previous_max_log_err = max_log_err

                N *=2
                time = linspace(t0, tend, N+1)
                fanalytic = np.zeros_like(time)
                k = 0
                for tau in time:
                    fanalytic[k] = f(tau)
                    k = k + 1

            schemes_order[scheme.func_name] = order_approx
            iclr += 1

        legend(legends, loc='best')
        xlabel('Time')
        ylabel('log(error)')
        grid()

        N = N/2**Ndts
        N_list = [N*2**i for i in range(1, Ndts+1)]
        N_list = np.asarray(N_list)

        figure()
        for key in schemes_order:
            plot(N_list, (np.asarray(schemes_order[key])))

        # Plot theoretical n for 1st, 2nd and 4th order schemes
        axhline(1.0, xmin=0, xmax=N, linestyle=':', color='k')
        axhline(2.0, xmin=0, xmax=N, linestyle=':', color='k')
        axhline(4.0, xmin=0, xmax=N, linestyle=':', color='k')
        xticks(N_list, rotation=-70)
        legends = schemes_order.keys()
        legends.append('theoretical')
        legend(legends, loc='best', frameon=False)
        title('Method of Manufactured Solution')
        xlabel('Number of unknowns')
        ylabel('Scheme order approximation')
        axis([0, max(N_list), 0, 5])
#         savefig('MMSODEschemes.png', transparent=True)
    # test using MMS and solving a set of two nonlinear equations to find estimate of order
    def manufactured_solution_Nonlinear():
        """ Test convergence rate of the methods, by using the Method of Manufactured solutions.
            The coefficient function f is chosen to be the normal distribution
            f = (1/(sigma*sqrt(2*pi)))*exp(-((t-mu)**2)/(2*sigma**2)).
            The ODE to be solved is than chosen to be: f''' + f''*f + f' = RHS,
            leading to f''' = RHS - f''*f - f
         """
        from numpy import linspace, abs
        from sympy import exp, symbols, diff, lambdify
        from math import sqrt, pi
        from numpy import log, log2

        t = symbols('t')
        sigma=  0.5 # standard deviation
        mu = 0.5 # mean value
        #### Perform needed differentiations based on the differential equation ####
        f = (1/(sigma*sqrt(2*pi)))*exp(-((t-mu)**2)/(2*sigma**2))
        dfdt = diff(f, t)
        d2fdt = diff(dfdt, t)
```

```
d3fdt = diff(d2fdt, t)
RHS = d3fdt + dfdt*d2fdt + f
#### Create Python functions of f, RHS and needed differentiations of f ####
f = lambdify([t], f, np)
dfdt = lambdify([t], dfdt, np)
d2fdt = lambdify([t], d2fdt)
RHS = lambdify([t], RHS)

def func(y,t):
    """ Function that returns the dfn/dt of the differential equation f + f''*f + f''' = RHS
        as a system of 1st order equations; f = f1
                f1' = f2
                f2' = f3
                f3' = RHS - f1 - f2*f3

        Args:
            y(array): solutian array [f1, f2, f3] at time t
            t(float): current time

        Returns:
            yout(array): differantiation array [f1', f2', f3'] at time t
    """
    yout = np.zeros_like(y)
    yout[:] = [y[1], y[2], RHS(t) -y[0]- y[1]*y[2]]

    return yout

t0, tend = -1.5, 2.5
z0 = np.array([f(t0), dfdt(t0), d2fdt(t0)]) # initial values

schemes = [euler, heun, rk4] # list of schemes; each of which is a function
schemes_error = {} # empty dictionary. to be filled in with lists of error-norms for all sch
h = [] # empty list of time step

Ntds = 4 # number of times to refine dt

fig, ax = subplots(1, len(schemes), sharey = True, squeeze=False)

for k, scheme in enumerate(schemes):
    N = 20    # initial number of time steps
    error = [] # start of with empty list of errors for all schemes
    legendList = []

    for i in range(Ntds + 1):
        time = linspace(t0, tend, N+1)

        if k==0:
            h.append(time[1] - time[0]) # add this iteration's dt to list h
        z = scheme(func, z0, time) # Solve the ODE by calling the scheme with arguments. e.g
        fanalytic = f(time) # call analytic function f to compute analytical solutions at ti

        abs_error = abs(z[:,0]- fanalytic) # calculate infinity norm of the error
        error.append(max(abs_error))

        ax[0][k].plot(time, z[:,0])
        legendList.append('$h$ = ' + str(h[i]))

        N *=2 # refine dt

    schemes_error[scheme.func_name] = error # Add a key:value pair to the dictionary. e.g: "
```

```
            ax[0][k].plot(time, fanalytic, 'k:')
            legendList.append('$u_m$')
            ax[0][k].set_title(scheme.func_name)
            ax[0][k].set_xlabel('time')

        ax[0][2].legend(legendList, loc = 'best', frameon=False)
        ax[0][0].set_ylabel('u')
        setp(ax, xticks=[-1.5, 0.5,  2.5], yticks=[0.0, 0.4 , 0.8, 1.2])

#        #savefig('../figs/normal_distribution_refinement.png')
        def Newton_solver_sympy(error, h, x0):
            """ Function that solves for the nonlinear set of equations
                error1 = C*h1^p --> f1 = C*h1^p - error1 = 0
                error2 = C*h2^p --> f2 = C h2^p - error 2 = 0
                where C is a constant h is the step length and p is the order,
                with use of a newton rhapson solver. In this case C and p are
                the unknowns, whereas h and error are knowns. The newton rhapson
                method is an iterative solver which take the form:
                xnew = xold - (J^-1)*F, where J is the Jacobi matrix and F is the
                residual funcion.
                    x = [C, p]^T
                    J = [[df1/dx1  df2/dx2],
                         [df2/dx1  df2/dx2]]
                    F = [f1, f2]
                This is very neatly done with use of the sympy module

                Args:
                    error(list): list of calculated errors [error(h1), error(h2)]
                    h(list): list of steplengths corresponding to the list of errors
                    x0(list): list of starting (guessed) values for x

                Returns:
                    x(array): iterated solution of x = [C, p]

            """
            from sympy import Matrix
            #### Symbolic computiations: ####
            C, p = symbols('C p')
            f1 = C*h[-2]**p - error[-2]
            f2 = C*h[-1]**p - error[-1]
            F = [f1, f2]
            x = [C, p]

            def jacobiElement(i,j):
                return diff(F[i], x[j])

            Jacobi = Matrix(2, 2, jacobiElement) # neat way of computing the Jacobi Matrix
            JacobiInv = Jacobi.inv()
            #### Numerical computations: ####
            JacobiInvfunc = lambdify([x], JacobiInv)
            Ffunc = lambdify([x], F)
            x = x0

            for n in range(8): #perform 8 iterations
                F = np.asarray(Ffunc(x))
                Jinv = np.asarray(JacobiInvfunc(x))
                xnew = x - np.dot(Jinv, F)
                x = xnew
                #print "n, x: ", n, x
            x[0] = round(x[0], 2)
```

```
            x[1] = round(x[1], 3)
            return x

        ht = np.asarray(h)
        eulerError = np.asarray(schemes_error["euler"])
        heunError = np.asarray(schemes_error["heun"])
        rk4Error = np.asarray(schemes_error["rk4"])


        [C_euler, p_euler] = Newton_solver_sympy(eulerError, ht, [1,1])
        [C_heun, p_heun] = Newton_solver_sympy(heunError, ht, [1,2])
        [C_rk4, p_rk4] = Newton_solver_sympy(rk4Error, ht, [1,4])

        from sympy import latex
        h = symbols('h')
        epsilon_euler = C_euler*h**p_euler
        epsilon_euler_latex = '$' + latex(epsilon_euler) + '$'
        epsilon_heun = C_heun*h**p_heun
        epsilon_heun_latex = '$' + latex(epsilon_heun) + '$'
        epsilon_rk4 = C_rk4*h**p_rk4
        epsilon_rk4_latex = '$' + latex(epsilon_rk4) + '$'

        print epsilon_euler_latex
        print epsilon_heun_latex
        print epsilon_rk4_latex

        epsilon_euler = lambdify(h, epsilon_euler, np)
        epsilon_heun = lambdify(h, epsilon_heun, np)
        epsilon_rk4 = lambdify(h, epsilon_rk4, np)

        N = N/2**(Ntds + 2)
        N_list = [N*2**i for i in range(1, Ntds + 2)]
        N_list = np.asarray(N_list)
        print len(N_list)
        print len(eulerError)
        figure()
        plot(N_list, log2(eulerError), 'b')
        plot(N_list, log2(epsilon_euler(ht)), 'b--')
        plot(N_list, log2(heunError), 'g')
        plot(N_list, log2(epsilon_heun(ht)), 'g--')
        plot(N_list, log2(rk4Error), 'r')
        plot(N_list, log2(epsilon_rk4(ht)), 'r--')
        LegendList = ['${\epsilon}_{euler}$', epsilon_euler_latex, '${\epsilon}_{heun}$', epsilon_heu
        legend(LegendList, loc='best', frameon=False)
        xlabel('-log(h)')
        ylabel('-log($\epsilon$)')

#         #savefig('../figs/MMS_example2.png')
```

The complete module `ODEschemes` is listed below and may easily be downloaded in your Eclipse/LiClipse IDE:

```
# src-ch1/ODEschemes.py

import numpy as np
from matplotlib.pyplot import plot, show, legend, hold,rcParams,rc, figure, axhline, close,\
    xticks, title, xlabel, ylabel, savefig, axis, grid, subplots, setp

# change some default values to make plots more readable
LNWDT=3; FNT=10
```

```
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT


# define Euler solver
def euler(func, z0, time):
    """The Euler scheme for solution of systems of ODEs.
    z0 is a vector for the initial conditions,
    the right hand side of the system is represented by func which returns
    a vector with the same size as z0 ."""

    z = np.zeros((np.size(time), np.size(z0)))
    z[0,:] = z0

    for i in range(len(time)-1):
        dt = time[i+1] - time[i]
        z[i+1,:]=z[i,:] + np.asarray(func(z[i,:], time[i]))*dt

    return z


# define Heun solver
def heun(func, z0, time):
    """The Heun scheme for solution of systems of ODEs.
    z0 is a vector for the initial conditions,
    the right hand side of the system is represented by func which returns
    a vector with the same size as z0 ."""

    z = np.zeros((np.size(time), np.size(z0)))
    z[0,:] = z0
    zp = np.zeros_like(z0)

    for i, t in enumerate(time[0:-1]):
        dt = time[i+1] - time[i]
        zp = z[i,:] + np.asarray(func(z[i,:],t))*dt   # Predictor step
        z[i+1,:] = z[i,:] + (np.asarray(func(z[i,:],t)) + np.asarray(func(zp,t+dt)))*dt/2.0 # Correc

    return z


# define rk4 scheme
def rk4(func, z0, time):
    """The Runge-Kutta 4 scheme for solution of systems of ODEs.
    z0 is a vector for the initial conditions,
    the right hand side of the system is represented by func which returns
    a vector with the same size as z0 ."""

    z = np.zeros((np.size(time),np.size(z0)))
    z[0,:] = z0
    zp = np.zeros_like(z0)

    for i, t in enumerate(time[0:-1]):
        dt = time[i+1] - time[i]
        dt2 = dt/2.0
        k1 = np.asarray(func(z[i,:], t))                # predictor step 1
        k2 = np.asarray(func(z[i,:] + k1*dt2, t + dt2)) # predictor step 2
        k3 = np.asarray(func(z[i,:] + k2*dt2, t + dt2)) # predictor step 3
        k4 = np.asarray(func(z[i,:] + k3*dt, t + dt))   # predictor step 4
        z[i+1,:] = z[i,:] + dt/6.0*(k1 + 2.0*k2 + 2.0*k3 + k4) # Corrector step

    return z
```

```
if __name__ == '__main__':
    a = 0.2
    b = 3.0
    u_exact = lambda t: a*t   +  b

    def f_local(u,t):
        """A function which returns an np.array but less easy to read
        than f(z,t) below. """
        return np.asarray([a + (u - u_exact(t))**5])

    def f(z, t):
        """Simple to read function implementation """
        return [a + (z - u_exact(t))**5]


    def test_ODEschemes():
        """Use knowledge of an exact numerical solution for testing."""
        from numpy import linspace, size

        tol = 1E-15
        T = 2.0  # end of simulation
        N = 20  # no of time steps
        time = linspace(0, T, N+1)


        z0 = np.zeros(1)
        z0[0] = u_exact(0.0)

        schemes  = [euler, heun, rk4]

        for scheme in schemes:
            z = scheme(f, z0, time)
            max_error = np.max(u_exact(time) - z[:,0])
            msg = '%s failed with error = %g' % (scheme.func_name, max_error)
            assert max_error < tol, msg

    # f3 defines an ODE with analytical solution in u_nonlin_analytical
    def f3(z, t, a=2.0, b=-1.0):
        """ """
        return a*z + b

    def u_nonlin_analytical(u0, t, a=2.0, b=-1.0):
        from numpy import exp
        TOL = 1E-14
        if (abs(a)>TOL):
            return (u0 + b/a)*exp(a*t)-b/a
        else:
            return u0 + b*t


    # Function for convergence test
    def convergence_test():
        """ Test convergence rate of the methods """
        from numpy import linspace, size, abs, log10, mean, log2
        figure()
        tol = 1E-15
        T = 8.0   # end of simulation
        Ndts = 5 # Number of times to refine timestep in convergence test

        z0 = 2
```

```
        schemes =[euler, heun, rk4]
        legends=[]
        schemes_order={}

        colors = ['r', 'g', 'b', 'm', 'k', 'y', 'c']
        linestyles = ['-', '--', '-.', ':', 'v--', '*-.']
        iclr = 0
        for scheme in schemes:
            N = 30    # no of time steps
            time = linspace(0, T, N+1)

            order_approx = []

            for i in range(Ndts+1):
                z = scheme(f3, z0, time)
                abs_error = abs(u_nonlin_analytical(z0, time)-z[:,0])
                log_error = log2(abs_error[1:]) # Drop 1st elt to avoid log2-problems (1st elt is ze
                max_log_err = max(log_error)
                plot(time[1:], log_error, linestyles[i]+colors[iclr], markevery=N/5)
                legends.append(scheme.func_name +': N = ' + str(N))
                hold('on')

                if i > 0: # Compute the log2 error difference
                    order_approx.append(previous_max_log_err - max_log_err)
                previous_max_log_err = max_log_err

                N *=2
                time = linspace(0, T, N+1)

            schemes_order[scheme.func_name] = order_approx
            iclr += 1

        legend(legends, loc='best')
        xlabel('Time')
        ylabel('log(error)')
        grid()

        N = N/2**Ndts
        N_list = [N*2**i for i in range(1, Ndts+1)]
        N_list = np.asarray(N_list)

        figure()
        for key in schemes_order:
            plot(N_list, (np.asarray(schemes_order[key])))

        # Plot theoretical n for 1st, 2nd and 4th order schemes
        axhline(1.0, xmin=0, xmax=N, linestyle=':', color='k')
        axhline(2.0, xmin=0, xmax=N, linestyle=':', color='k')
        axhline(4.0, xmin=0, xmax=N, linestyle=':', color='k')
        xticks(N_list, rotation=-70)
        legends = schemes_order.keys()
        legends.append('theoretical')
        legend(legends, loc='best', frameon=False)
        xlabel('Number of unknowns')
        ylabel('Scheme order approximation')
        axis([0, max(N_list), 0, 5])
#        savefig('ConvergenceODEschemes.png', transparent=True)

    def manufactured_solution():
        """ Test convergence rate of the methods, by using the Method of Manufactured solutions.
```

```
    The coefficient function f is chosen to be the normal distribution
    f = (1/(sigma*sqrt(2*pi)))*exp(-((t-mu)**2)/(2*sigma**2)).
    The ODE to be solved is than chosen to be: f''' + f''*f + f' = RHS,
    leading to to f''' = RHS - f''*f - f
 """
from numpy import linspace, size, abs, log10, mean, log2
from sympy import exp, symbols, diff, lambdify
from math import sqrt, pi

print "solving equation f''' + f''*f + f' = RHS"
print "which lead to f''' = RHS - f''*f - f"
t = symbols('t')
sigma=0.5 # standard deviation
mu=0.5 # mean value
Domain=[-1.5, 2.5]
t0 = Domain[0]
tend = Domain[1]

f = (1/(sigma*sqrt(2*pi)))*exp(-((t-mu)**2)/(2*sigma**2))
dfdt = diff(f, t)
d2fdt = diff(dfdt, t)
d3fdt = diff(d2fdt, t)
RHS = d3fdt + dfdt*d2fdt + f

f = lambdify([t], f)
dfdt = lambdify([t], dfdt)
d2fdt = lambdify([t], d2fdt)
RHS = lambdify([t], RHS)

def func(y,t):
    yout = np.zeros_like(y)
    yout[:] = [y[1], y[2], RHS(t) -y[0]- y[1]*y[2]]

    return yout

z0 = np.array([f(t0), dfdt(t0), d2fdt(t0)])

figure()
tol = 1E-15
Ndts = 5 # Number of times to refine timestep in convergence test
schemes =[euler, heun, rk4]
legends=[]
schemes_order={}

colors = ['r', 'g', 'b', 'm', 'k', 'y', 'c']
linestyles = ['-', '--', '-.', ':', 'v--', '*-.']
iclr = 0
for scheme in schemes:
    N = 100    # no of time steps
    time = linspace(t0, tend, N+1)
    fanalytic = np.zeros_like(time)
    k = 0
    for tau in time:
        fanalytic[k] = f(tau)
        k = k + 1

    order_approx = []

    for i in range(Ndts+1):
        z = scheme(func, z0, time)
        abs_error = abs(fanalytic-z[:,0])
```

```
                log_error = log2(abs_error[1:]) # Drop 1st elt to avoid log2-problems (1st elt is ze
                max_log_err = max(log_error)
                plot(time[1:], log_error, linestyles[i]+colors[iclr], markevery=N/5)
                legends.append(scheme.func_name +': N = ' + str(N))
                hold('on')

                if i > 0: # Compute the log2 error difference
                    order_approx.append(previous_max_log_err - max_log_err)
                previous_max_log_err = max_log_err

                N *=2
                time = linspace(t0, tend, N+1)
                fanalytic = np.zeros_like(time)
                k = 0
                for tau in time:
                    fanalytic[k] = f(tau)
                    k = k + 1

            schemes_order[scheme.func_name] = order_approx
            iclr += 1

        legend(legends, loc='best')
        xlabel('Time')
        ylabel('log(error)')
        grid()

        N = N/2**Ndts
        N_list = [N*2**i for i in range(1, Ndts+1)]
        N_list = np.asarray(N_list)

        figure()
        for key in schemes_order:
            plot(N_list, (np.asarray(schemes_order[key])))

        # Plot theoretical n for 1st, 2nd and 4th order schemes
        axhline(1.0, xmin=0, xmax=N, linestyle=':', color='k')
        axhline(2.0, xmin=0, xmax=N, linestyle=':', color='k')
        axhline(4.0, xmin=0, xmax=N, linestyle=':', color='k')
        xticks(N_list, rotation=-70)
        legends = schemes_order.keys()
        legends.append('theoretical')
        legend(legends, loc='best', frameon=False)
        title('Method of Manufactured Solution')
        xlabel('Number of unknowns')
        ylabel('Scheme order approximation')
        axis([0, max(N_list), 0, 5])
#         savefig('MMSODEschemes.png', transparent=True)
    # test using MMS and solving a set of two nonlinear equations to find estimate of order
    def manufactured_solution_Nonlinear():
        """ Test convergence rate of the methods, by using the Method of Manufactured solutions.
            The coefficient function f is chosen to be the normal distribution
            f = (1/(sigma*sqrt(2*pi)))*exp(-((t-mu)**2)/(2*sigma**2)).
            The ODE to be solved is than chosen to be: f''' + f''*f + f' = RHS,
            leading to f''' = RHS - f''*f - f
         """
        from numpy import linspace, abs
        from sympy import exp, symbols, diff, lambdify
        from math import sqrt, pi
        from numpy import log, log2

        t = symbols('t')
```

```
sigma=  0.5 # standard deviation
mu = 0.5 # mean value
#### Perform needed differentiations based on the differential equation ####
f = (1/(sigma*sqrt(2*pi)))*exp(-((t-mu)**2)/(2*sigma**2))
dfdt = diff(f, t)
d2fdt = diff(dfdt, t)
d3fdt = diff(d2fdt, t)
RHS = d3fdt + dfdt*d2fdt + f
#### Create Python functions of f, RHS and needed differentiations of f ####
f = lambdify([t], f, np)
dfdt = lambdify([t], dfdt, np)
d2fdt = lambdify([t], d2fdt)
RHS = lambdify([t], RHS)

def func(y,t):
    """ Function that returns the dfn/dt of the differential equation f + f''*f + f''' = RHS
        as a system of 1st order equations; f = f1
                f1' = f2
                f2' = f3
                f3' = RHS - f1 - f2*f3

        Args:
            y(array): solutian array [f1, f2, f3] at time t
            t(float): current time

        Returns:
            yout(array): differantiation array [f1', f2', f3'] at time t
    """
    yout = np.zeros_like(y)
    yout[:] = [y[1], y[2], RHS(t) -y[0]- y[1]*y[2]]

    return yout

t0, tend = -1.5, 2.5
z0 = np.array([f(t0), dfdt(t0), d2fdt(t0)]) # initial values

schemes = [euler, heun, rk4] # list of schemes; each of which is a function
schemes_error = {} # empty dictionary. to be filled in with lists of error-norms for all sch
h = [] # empty list of time step

Ntds = 4 # number of times to refine dt

fig, ax = subplots(1, len(schemes), sharey = True, squeeze=False)

for k, scheme in enumerate(schemes):
    N = 20    # initial number of time steps
    error = [] # start of with empty list of errors for all schemes
    legendList = []

    for i in range(Ntds + 1):
        time = linspace(t0, tend, N+1)

        if k==0:
            h.append(time[1] - time[0]) # add this iteration's dt to list h
        z = scheme(func, z0, time) # Solve the ODE by calling the scheme with arguments. e.g
        fanalytic = f(time) # call analytic function f to compute analytical solutions at ti

        abs_error = abs(z[:,0]- fanalytic) # calculate infinity norm of the error
        error.append(max(abs_error))

        ax[0][k].plot(time, z[:,0])
```

```
                legendList.append('$h$ = ' + str(h[i]))

                N *=2 # refine dt

            schemes_error[scheme.func_name] = error # Add a key:value pair to the dictionary. e.g: "

            ax[0][k].plot(time, fanalytic, 'k:')
            legendList.append('$u_m$')
            ax[0][k].set_title(scheme.func_name)
            ax[0][k].set_xlabel('time')

        ax[0][2].legend(legendList, loc = 'best', frameon=False)
        ax[0][0].set_ylabel('u')
        setp(ax, xticks=[-1.5, 0.5,  2.5], yticks=[0.0, 0.4 , 0.8, 1.2])

#        #savefig('../figs/normal_distribution_refinement.png')
        def Newton_solver_sympy(error, h, x0):
            """ Function that solves for the nonlinear set of equations
                error1 = C*h1^p --> f1 = C*h1^p - error1 = 0
                error2 = C*h2^p --> f2 = C h2^p - error 2 = 0
                where C is a constant h is the step length and p is the order,
                with use of a newton rhapson solver. In this case C and p are
                the unknowns, whereas h and error are knowms. The newton rhapson
                method is an iterative solver which take the form:
                xnew = xold - (J^-1)*F, where J is the Jacobi matrix and F is the
                residual funcion.
                    x = [C, p]^T
                    J = [[df1/dx1  df2/dx2],
                         [df2/dx1  df2/dx2]]
                    F = [f1, f2]
                This is very neatly done with use of the sympy module

                Args:
                    error(list): list of calculated errors [error(h1), error(h2)]
                    h(list): list of steplengths corresponding to the list of errors
                    x0(list): list of starting (guessed) values for x

                Returns:
                    x(array): iterated solution of x = [C, p]

            """
            from sympy import Matrix
            #### Symbolic computiations: ####
            C, p = symbols('C p')
            f1 = C*h[-2]**p - error[-2]
            f2 = C*h[-1]**p - error[-1]
            F = [f1, f2]
            x = [C, p]

            def jacobiElement(i,j):
                return diff(F[i], x[j])

            Jacobi = Matrix(2, 2, jacobiElement) # neat way of computing the Jacobi Matrix
            JacobiInv = Jacobi.inv()
            #### Numerical computations: ####
            JacobiInvfunc = lambdify([x], JacobiInv)
            Ffunc = lambdify([x], F)
            x = x0

            for n in range(8): #perform 8 iterations
```

```
                F = np.asarray(Ffunc(x))
                Jinv = np.asarray(JacobiInvfunc(x))
                xnew = x - np.dot(Jinv, F)
                x = xnew
                #print "n, x: ", n, x
            x[0] = round(x[0], 2)
            x[1] = round(x[1], 3)
            return x

        ht = np.asarray(h)
        eulerError = np.asarray(schemes_error["euler"])
        heunError = np.asarray(schemes_error["heun"])
        rk4Error = np.asarray(schemes_error["rk4"])


        [C_euler, p_euler] = Newton_solver_sympy(eulerError, ht, [1,1])
        [C_heun, p_heun] = Newton_solver_sympy(heunError, ht, [1,2])
        [C_rk4, p_rk4] = Newton_solver_sympy(rk4Error, ht, [1,4])

        from sympy import latex
        h = symbols('h')
        epsilon_euler = C_euler*h**p_euler
        epsilon_euler_latex = '$' + latex(epsilon_euler) + '$'
        epsilon_heun = C_heun*h**p_heun
        epsilon_heun_latex = '$' + latex(epsilon_heun) + '$'
        epsilon_rk4 = C_rk4*h**p_rk4
        epsilon_rk4_latex = '$' + latex(epsilon_rk4) + '$'

        print epsilon_euler_latex
        print epsilon_heun_latex
        print epsilon_rk4_latex

        epsilon_euler = lambdify(h, epsilon_euler, np)
        epsilon_heun = lambdify(h, epsilon_heun, np)
        epsilon_rk4 = lambdify(h, epsilon_rk4, np)

        N = N/2**(Ntds + 2)
        N_list = [N*2**i for i in range(1, Ntds + 2)]
        N_list = np.asarray(N_list)
        print len(N_list)
        print len(eulerError)
        figure()
        plot(N_list, log2(eulerError), 'b')
        plot(N_list, log2(epsilon_euler(ht)), 'b--')
        plot(N_list, log2(heunError), 'g')
        plot(N_list, log2(epsilon_heun(ht)), 'g--')
        plot(N_list, log2(rk4Error), 'r')
        plot(N_list, log2(epsilon_rk4(ht)), 'r--')
        LegendList = ['${\epsilon}_{euler}$', epsilon_euler_latex, '${\epsilon}_{heun}$', epsilon_heu
        legend(LegendList, loc='best', frameon=False)
        xlabel('-log(h)')
        ylabel('-log($\epsilon$)')

#        #savefig('../figs/MMS_example2.png')


    def plot_ODEschemes_solutions():
        """Plot the solutions for the test schemes in schemes"""
        from numpy import linspace
        figure()
        T = 1.5  # end of simulation
```

```
    N = 50  # no of time steps
    time = linspace(0, T, N+1)

    z0 = 2.0

    schemes  = [euler, heun, rk4]
    legends = []

    for scheme in schemes:
        z = scheme(f3, z0, time)
        plot(time, z[:,-1])
        legends.append(scheme.func_name)

    plot(time, u_nonlin_analytical(z0, time))
    legends.append('analytical')
    legend(legends, loc='best', frameon=False)

manufactured_solution_Nonlinear()
#test_ODEschemes()
#convergence_test()
#plot_ODEschemes_solutions()
#manufactured_solution()
show()
```

2.15 Absolute stability of numerical methods for ODE IVPs

To investigate stability of numerical methods for the solution of intital value ODEs on the generic form (2.2), let us consider the following model problem:

$$y'(x) = \lambda \cdot y(x) \tag{2.128}$$

$$y(0) = 1 \tag{2.129}$$

which has the analytical solution:

$$y(x) = e^{\lambda x},\ \lambda\,, \tag{2.130}$$

with λ being a constant.

The solution $y(x)$ of (2.128) is illustrated in Figure 2.22 for positive ($\lambda = 1$) and negative ($\lambda = -2$) growth factors.

We illustrate the stability concept for the numerical solution of ODEs by two examples below, namely, the Euler scheme and the Heun scheme. In the latter example we will also allude to how the analysis may be applied for higher order RK-methods.

2.15.1 Example: Stability of Euler's method

By applying the generic 2.6 On our particular model problem given by (2.128) we obtain the following generic scheme:

$$y_{n+1} = (1 + \lambda h)\, y_n, \qquad h = \Delta x, \quad n = 0, 1, 2, \ldots \tag{2.131}$$

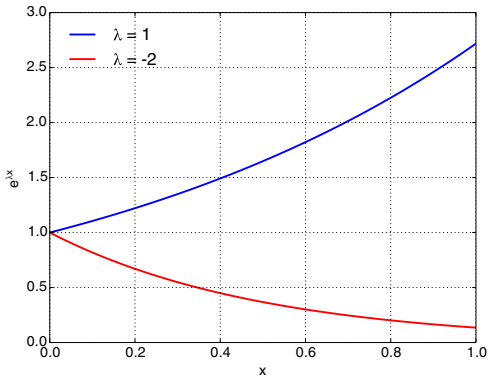

Figure 2.22: Solution of ODE for exponential growth with positive and negative growth factor λ.

The numerical soltion y_n at the n-th iteration may be related with the initial solution y_0 by applying the scheme (2.131) iteratively:

$$\begin{aligned} y_1 &= (1+\lambda h)\, y_0 \\ y_2 &= (1+\lambda h)\, y_1 = (1+\lambda h)^2\, y_0 \\ &\vdots \end{aligned}$$

which yields:

$$y_n = (1+\lambda h)^n\, y_0, \qquad n = 1, 2, 3, \ldots \tag{2.132}$$

To investigate stability we first introduce the analytic amplification factor G_a as

$$G_a = \frac{y(x_{n+1})}{y(x_n)} \tag{2.133}$$

Note that $y(x_n)$ represents the analytical solution of (2.128) at x_n, whereas y_n denotes the numerical approximation. For the model problem at hand we see that G_a reduces to:

$$G_a = \frac{y(x_{n+1})}{y(x_n)} = \exp[\lambda(x_{n+1} - x_n)] = e^{\lambda h} = 1 + \lambda h + (\lambda h)^2/2 \ldots \tag{2.134}$$

Exponential growth $\lambda > 0$.

Consider first the case of a positive $\lambda > 0$, corresponding to an exponential growth, as given by (2.130). We observe that our scheme will also exhibit exponential growth (2.133), and will thus have no concerns in terms of stability. Naturally, the choice of h will affect the accuracy and numerical error of the solution.

Exponential decay $\lambda < 0$.

In case of exponential decay with a negative $\lambda < 0$, we adopt the following convention for convenience:

$$\lambda = -\alpha, \qquad \alpha > 0 \tag{2.135}$$

and (2.132) may be recasted to:

$$y_n = (1 - \alpha h)^n \, y_0 \tag{2.136}$$

If y_n is to decrease as n increases we must have

$$|1 - \alpha h| < 1 \qquad \Rightarrow -1 < 1 - \alpha h < 1 \tag{2.137}$$

which yields the following criterion for selection of h:

$$0 < \alpha h < 2, \ \alpha > 0 \tag{2.138}$$

The criterion may also be formulated by introducing the numerical amplification factor G as:

$$G = \frac{y_{n+1}}{y_n} \tag{2.139}$$

which for the current example (2.131) reduces to:

$$G = \frac{y_{n+1}}{y_n} = 1 + \lambda h = 1 - \alpha h \tag{2.140}$$

For the current example $G > 1$ for $\lambda > 0$ and $G < 1$ for $\lambda < 0$. Compare with the expression for the analytical amplification factor G_a (2.134).

Numerical amplification factor Consider IVP (2.128). Then, the numerical amplification factor is defined as

$$G(z) = \frac{y_{n+1}}{y_n}, \tag{2.141}$$

with $z = h\lambda$.

As we shall see later on, the numerical amplification factor $G(z)$ is some function of $z = h\lambda$, commonly a polynomial for an explicit method and a rational function for implicit methods. For consistent methods $G(z)$ will be an approximation to e^z near $z = 0$, and if the method is p-th order accurate, then $G(z) - e^z = \mathcal{O}(z^{p+1})$ as $z \to 0$.

Absolute stability The numerical scheme is said to be stable when:

$$|G(z)| \leq 1 \tag{2.142}$$

Region of absolute stability The region of absolute stability for a one-step method is given by

$$\mathcal{S} = \{z \in \mathbb{C} : |G(z)| \leq 1\}. \tag{2.143}$$

For example, in the case under examination, when $\alpha = 100$ we must choose $h < 0.02$ to produce an stable numerical solution.

By the introduction of the G in (2.140) we may rewrite (2.132) as:

$$y_n = G^n\, y_0, \qquad n = 0, 1, 2, \ldots \tag{2.144}$$

From (2.144) we observe that a stable, exponentially decaying solution ($\lambda < 0$) the solution will oscillate if $G < 0$, as $G^n < 0$ when n is odd and $G^n > 0$ when n is even. As this behavior is qualitatively different from the behaviour of the analytical solution, we will seek to avoid such behavior.

The Euler scheme is conditionally stable for our model equation when

$$0 < \alpha h < 2,\ \alpha > 0 \tag{2.145}$$

but oscillations will occur when

$$1 < \alpha h < 2,\ \alpha > 0 \tag{2.146}$$

Consequently, the Euler scheme will be stable and free of spurious oscillations when

$$0 < \alpha h < 1, \qquad \alpha > 0 \tag{2.147}$$

For example the numerical solution with $\alpha = 10$ will be stable in the intervall $0 < h < 0.2$, but exhibit oscillations in the interval $0.1 < h < 0.2$. Note, however, that the latter restriction on avoiding spurious oscillations will not be of major concerns in pratice, as the demand for descent accuracy will dictate a far smaller h.

2.15.2 Example: Stability of Heun's method

In this example we will analyze the stability of 2.9 when applied on our model ODE (2.128) with $\lambda = -\alpha,\ \alpha > 0$.

The predictor step reduces to:

$$y_{n+1}^p = y_n + h \cdot f_n = (1 - \alpha h) \cdot y_n \tag{2.148}$$

for convenience the predictor y_{n+1}^p may be eliminated in the corrector step:

$$y_{n+1} = y_n + \frac{h}{2} \cdot (f_n + f_{n+1}^p) = (1 - \alpha h + \frac{1}{2}(\alpha h)^2) \cdot y_n \tag{2.149}$$

which shows that Heun's method may be represented as a one-step method for this particular and simple model problem. From (2.149) we find the numerical amplification factor G (ref<eq:1618a);

$$G = \frac{y_{n+1}}{y_n} = 1 - \alpha h + \frac{1}{2}(\alpha h)^2 \tag{2.150}$$

Our taks is now to investigate for which values of αh the condition for stability (2.142) may be satisfied.

First, we may realize that (2.150) is a second order polynomial in αh and that $G = 1$ for $\alpha h = 0$ and $\alpha h = 2$. The numerical amplification factor has an extremum where:

$$\frac{dG}{d\alpha h} = \alpha h - 1 = 0 \Rightarrow \alpha h = 1$$

and since

$$\frac{d^2 G}{d(\alpha h)^2} > 0$$

this extremum is a minimum and $G_{\min} = G(\alpha h = 1) = 1/2$. Consequently we may conclude that 2.9 will be conditionally stable for (2.128) in the following αh-range (stability range):

$$0 < \alpha h < 2 \tag{2.151}$$

Note that this condition is the same as for the 2.15.1 above, but as $G > 0$ always, the scheme will not produce spurious oscillations.

The 2.6 and 2.9 are RK-methods of first and second order, respectively. Even though the αh-range was not expanded by using a RK-method of 2 order, the quality of the numerical predictions were improved as spurios oscillations will vanish.

2.15.3 Stability of higher order RK-methods

A natural questions to raise is whether the stability range may be expanded for higher order RK-methods, so let us now investigate their the stability range.

The analytical solution of our model ODE (2.128) is:

$$y(x) = e^{\lambda x} = 1 + \lambda x + \frac{(\lambda x)^2}{2!} + \frac{(\lambda x)^3}{3!} + \cdots + \frac{(\lambda x)^n}{n!} + \cdots \tag{2.152}$$

By substitution of λx with λh, we get the solution of the difference equation corresponding to Euler's method (RK1) with the two first terms, Heun's method (RK2) the three first terms RK3 the four first terms, and finally RK4 with the first five terms:

$$G = 1 + \lambda h + \frac{(\lambda h)^2}{2!} + \frac{(\lambda h)^3}{3!} + \cdots + \frac{(\lambda h)^n}{n!}. \tag{2.153}$$

We focus on decaying solutions with $\lambda < 0$ and we have previously show from (2.138) that $0 < \alpha h < 2, \alpha > 2$ or:

$$-2 < \lambda h < 0 \tag{2.154}$$

for RK1 and RK3, but what about RK3? In this case:

$$G = 1 + \lambda h + \frac{(\lambda h)^2}{2!} + \frac{(\lambda h)^3}{3!} \tag{2.155}$$

The limiting values for the G-polynomial are found for $G = \pm 1$ which for real roots are:

$$-2.5127 < \lambda h < 0 \tag{2.156}$$

For RK4 we get correspondingly

$$G = 1 + \lambda h + \frac{(\lambda h)^2}{2!} + \frac{(\lambda h)^3}{3!} + \frac{(\lambda h)^4}{4!} \tag{2.157}$$

which for $G = 1$ has the real roots:

$$-2.7853 < \lambda h < 0 \tag{2.158}$$

In conclusion, we may argue that with respect to stability there is not much to gain by increasing the order for RK-methods. We have assumed λ to be real in the above analysis. However, λ may me complex for higer order ODEs and for systems of first order ODEs. If complex roots are included the regions of stability for RK-methods become as illstrated in figure 2.23.

Naturally, we want numerical schemes which are stable in the whole left half-plane. Such a scheme would be denoted *strictly absolutely stable* or simply L-stable (where L alludes to *left half-plane*). Regretfully, there are no existing, explicit methods which are L-stable. For L-stable methods one has to restort to implicit methods which are to be discussed in 2.15.4.

2.15.4 Stiff equations

While there is no precise definition of **stiff ODEs** in the literature, these are some generic characteristics for ODEs/ODE systems that are often called as such:

- ODE solution has different time scales. For example, the ODE solution can be written as the sum of a *fast* and a *slow* component as:

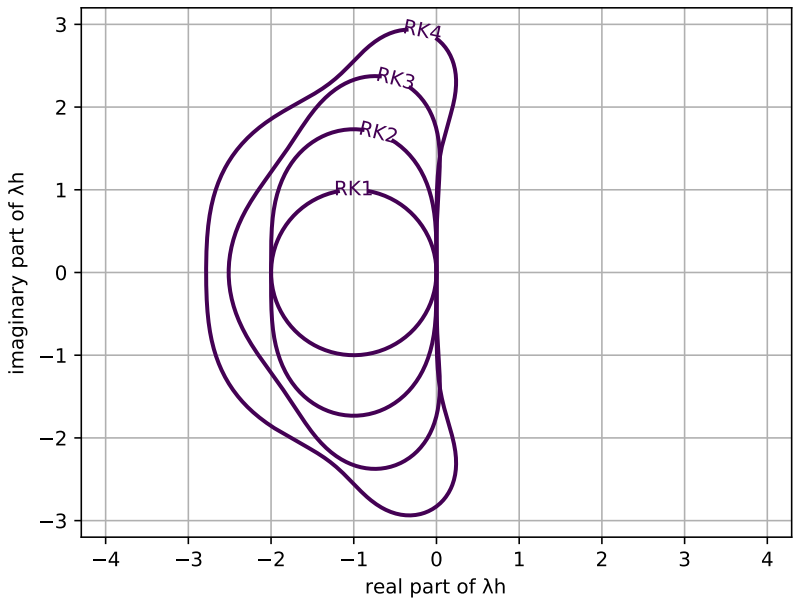

Figure 2.23: Regions of stability for RK-methods order 1-4 for complex roots.

$$y(x) = y_{slow}(x) + y_{fast}(x).$$

- For a first ODE system

$$\boldsymbol{y}'(x) = \boldsymbol{f}(x, \boldsymbol{y}),$$

the above statement is equivalent to the case in which the eigenvalues of the Jacobian $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{y}}$ differ greatly in magnitude.

- The definition of h for stiff ODEs is limited by stability concerns rather than by accuracy.

This last point is related to the definition of absolute stability regions for explicit schemes, which is inversely proportional to λ. Note that λ is in fact the (only) eigenvalue of the Jacobian for ODE (2.128)!

Example for stiff ODE: Consider the IVP

$$\left.\begin{aligned} y' &= \lambda y - \lambda sin(x) + cos(x), \qquad \lambda < 0, \\ y(0) &= 1. \end{aligned}\right\} \tag{2.159}$$

The analytical solution of problem (2.15.4) is given by

$$y(x) = sin(x) + e^{\lambda x},$$

from which we can identify a *slow* component

$$y_{slow}(x) = sin(x)$$

and a *fast* component

$$y_{fast}(x) = e^{\lambda x},$$

for $|\lambda|$ sufficiently large.

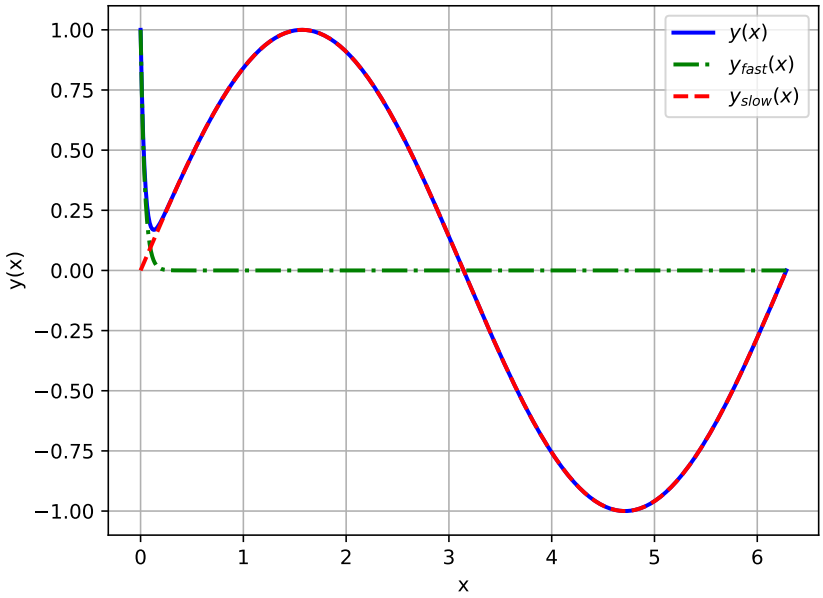

Figure 2.24: Solution (2.15.4) of stiff ODE IVP (2.15.4) for $\lambda = -25$.

A way to avoid having to use excessively small time steps, which can be even harmful for the accuracy of the approximation, due to error accumulation, is to use numerical schemes which have a larger region of absolute stability than explicit schemes.

2.15.5 Example: Stability of implicit Euler's method

The most natural, and widely used, candidate as a replacement of an explicit first order scheme is the **implicit** Euler's scheme. Equation (2.128) discretized using this scheme reads

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

and after solving for y_{n+1}

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

Therefore, the amplification factor for this scheme is

$$G = \frac{1}{1-h\lambda} = \frac{1}{1+h\alpha}, \quad \lambda = -\alpha, \quad \alpha > 0.$$

It is evident that $G < 1$ for any h (recall that h is positive by definition). This means that the interval of absolute stability for the implicit Euler's scheme includes $\mathbb{R}^-$. If this analysis was carried out allowing for λ to be a complex number with negative real part, then the region of absolute stability for the implicit Euler's scheme would have included the entire complex half-plane with negative real part $\mathbb{C}^-$.

2.15.6 Example: Stability of trapezoidal rule method

Equation (2.128) discretized by the trapezoidal rule scheme reads

$$y_{n+1} = y_n + \frac{h}{2}(f_n + f_{n+1}) = y_n + \frac{h}{2}(\lambda y_n + \lambda y_{n+1}),$$

and after solving for y_{n+1}

$$y_{n+1} = \frac{1 + \frac{\lambda h}{2}}{1 - \frac{\lambda h}{2}} y_n.$$

Therefore, the amplification factor for this scheme is

$$G = \frac{1 + \frac{\lambda h}{2}}{1 - \frac{\lambda h}{2}} = \frac{2 - \alpha h}{2 + \alpha h}, \quad \lambda = -\alpha, \quad \alpha > 0.$$

As for the implicit Euler's scheme, also in this case we have that $G < 1$ for any h. Therefore, interval of absolute stability for the implicit Euler's scheme includes $\mathbb{R}^-$. In this case, the analysis for λ complex number with negative real part, would have shown that the region of absolute stability for this scheme is precisely the entire complex half-plane with negative real part $\mathbb{C}^-$.

The above results motivate the following definition:

A-stable scheme A numerical scheme for (2.128) is said to be A-stable when its region of absolute stability includes the entire complex half-plane with negative real part $\mathbb{C}^-$.

This important definition was introduced in the '60s by Dahlquist, who also proved that A-stable schemes can be most second order accurate. A-stable schemes are *unconditionally* stable and therefore robust, since no stability problems will be observed. However, the user must always be aware of the fact that even if one is allowed to take a large value for h, the error must be contained within acceptable limits.

2.16 Exercises

Exercise 1: Solving Newton's first differential equation using euler's method

One of the earliest known differential equations, which Newton solved with series expansion in 1671 is:

$$y'(x) = 1 - 3x + y + x^2 + xy,\ y(0) = 0 \tag{2.160}$$

Newton gave the following solution:

$$y(x) \approx x - x^2 + \frac{x^3}{3} - \frac{x^4}{6} + \frac{x^5}{30} - \frac{x^6}{45} \tag{2.161}$$

Today it is possible to give the solution on closed form with known functions as follows,

$$y(x) = 3\sqrt{2\pi e} \cdot \exp\left[x\left(1+\frac{x}{2}\right)\right] \cdot \left[\operatorname{erf}\left(\frac{\sqrt{2}}{2}(1+x)\right) - \operatorname{erf}\left(\frac{\sqrt{2}}{2}\right)\right] + 4 \cdot \left[1 - \exp[x\left(1+\frac{x}{2}\right)\right] - x \tag{2.162}$$

Note the combination $\sqrt{2\pi e}$.

a) Solve Eq. (2.160) using Euler's method. Plot and compare with Newton's solution Eq. (2.161) and the analytical solution Eq. (2.162). Plot between $x = 0$ and $x = 1.5$

Hint 1. The function **scipy.special.erf** will be needed. See `http://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.special.erf.html`.

Hint 2. Figure should look something like this:

Exercise 2: Solitary wave

This exercise focuses on a solitary wave propagating on water.

The differential equation for a solitary wave can be written as:

$$\frac{d^2Y}{dX^2} = \frac{3Y}{D^2}\left(\frac{A}{D} - \frac{3}{2D}Y\right) \tag{2.163}$$

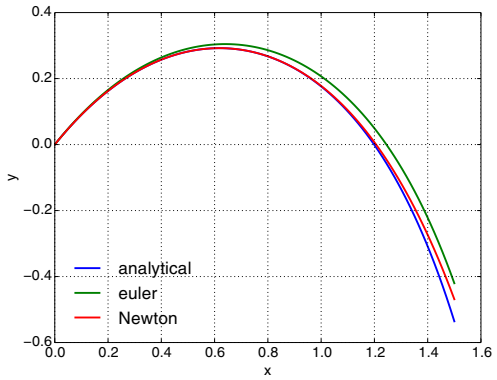

Figure 2.25: Problem 1 with 101 samplepoints between $x = 0$ and $x = 1.5$.

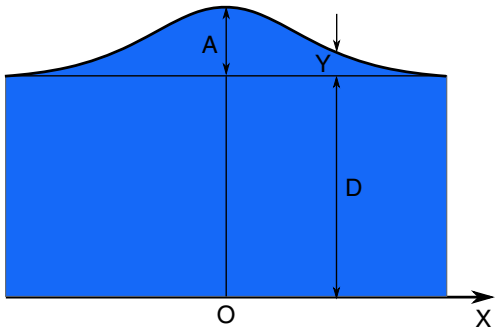

Figure 2.26: Solitary wave.

where D is the middle depth, $Y(X)$ is the wave height above middle depth and A is the wave height at $X = 0$. The wave is symmetric with respect to $X = 0$. See Figure 2.26. The coordinate system follows the wave.

By using dimensionless variables: $x = \frac{X}{D}$, $a = \frac{A}{D}$, $y = \frac{Y}{A}$, Eq. (2.163) can be written as:

$$y''(x) = a\,3\,y(x)\left(1 - \frac{3}{2}y(x)\right) \tag{2.164}$$

initial conditions: $y(0) = 1$, $y'(0) = 0$. Use $a = \frac{2}{3}$

Pen and paper

The following problems should be done using pen and paper:

a) Calculate y(0.6), and y'(0.6) using euler's method with $\Delta x = 0.2$

b) Solve **a**) using Heuns's method.
c) Perform a taylor expansion series around $x = 0$ (include 3 parts) on Eq. (2.164), and compare results with **a**) and **b**).
d) Solve Eq. (2.164) analytically for $a = \frac{2}{3}$, given:

$$\int \frac{dy}{y\sqrt{1-y}} = -2\,arctanh\left(\sqrt{1-y}\right)$$

Compare with solutions in **a**), **b**) and **c**).

Programing: Write a program that solve **a**), **b**) and **c**) numerically, and compare with the analytical solution found in **d**). Solve first with $\Delta x = 0.2$, and experiment with different values.

Hint 1. Solutions:
a) $y(0.6) = 0.88$, $y'(0.6) = -0.569$.
b) $y(0.6) = 0.8337$, $y'(0.6) = -0.4858$.
c) $y \approx 1 - \frac{x^2}{2} + \frac{x^4}{6}$, $y' \approx -x + \frac{2}{3}x^3$
d $y = \frac{1}{cosh^2\left(x/\sqrt{2}\right)} = \frac{1}{1+cosh\left(\sqrt{2}\cdot x\right)}$

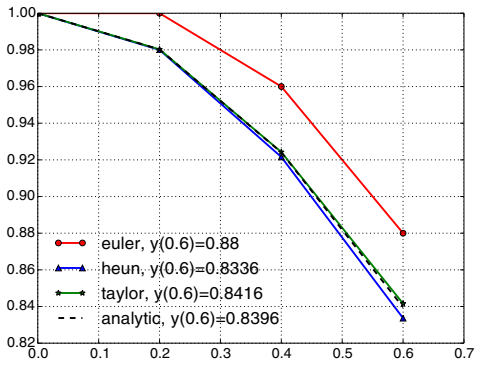

Figure 2.27: Plot should look something like this.

Hint 2. If you want you can use this template and fill in the lines where it's indicated.

```
#import matplotlib; matplotlib.use('Qt4Agg')
import matplotlib.pylab as plt
#plt.get_current_fig_manager().window.raise_()
import numpy as np

#### set default plot values: ####
LNWDT=3; FNT=15
plt.rcParams['lines.linewidth'] = LNWDT; plt.rcParams['font.size'] = FNT
```

```
""" This script solves the problem with the solitary wave:

        y'' = a*3*y*(1-y*3/2)

        y(0) = 1, y'(0) = 0

    or as a system of first order differential equations (y0 = y, y1 = y'):

        y0' = y'
        y1' = a*3*y0*(1-y0*3/2)

        y0(0) = 1, y1(0) = 0

"""
a = 2./3
h = 0.2 # steplength dx
x_0, x_end = 0, 0.6

x = np.arange(x_0, x_end + h, h) # allocate x values

#### solution vectors: ####
Y0_euler = np.zeros_like(x) # array to store y values
Y1_euler = np.zeros_like(x) # array to store y' values

Y0_heun = np.zeros_like(x)
Y1_heun = np.zeros_like(x)

#### initial conditions: ####
Y0_euler[0] = 1 # y(0) = 1
Y1_euler[0] = 0 # y'(0) = 0

Y0_heun[0] = 1
Y1_heun[0] = 0


#### solve with euler's method ####

for n in range(len(x) - 1):
    y0_n = Y0_euler[n] # y at this timestep
    y1_n = Y1_euler[n] # y' at this timestep

    "Fill in lines below"
    f0 =
    f1 =
    "Fill in lines above"

    Y0_euler[n + 1] = y0_n + h*f0
    Y1_euler[n + 1] = y1_n + h*f1

#### solve with heun's method: ####

for n in range(len(x) - 1):
    y0_n = Y0_heun[n] # y0 at this timestep (y_n)
    y1_n = Y1_heun[n] # y1 at this timestep (y'_n)

    "Fill in lines below"
    f0 =
    f1 =

    y0_p =
```

```
    y1_p =

    f0_p =
    f1_p =
    "Fill in lines above"

    Y0_heun[n + 1] = y0_n + 0.5*h*(f0 + f0_p)
    Y1_heun[n + 1] = y1_n + 0.5*h*(f1 + f1_p)


Y0_taylor = 1 - x**2/2 + x**4/6
Y1_taylor = -x + (2./3)*x**3

Y0_analytic = 1./(np.cosh(x/np.sqrt(2))**2)
```

Exercise 3: Mathematical pendulum

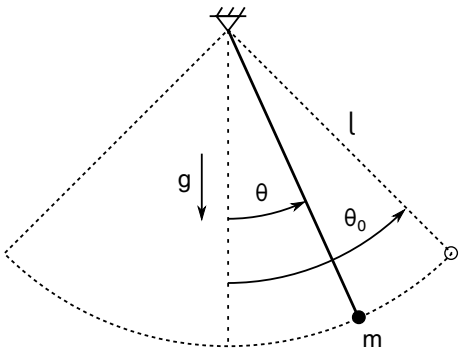

Figure 2.28: Mathematical pendulum.

Figure 2.28 shows a mathematical pendulum where the motion is described by the following ODE:

$$\frac{d^2\theta}{d\tau^2} + \frac{g}{l}\sin\theta = 0 \tag{2.165}$$

with initial conditions

$$\theta(0) = \theta_0 \tag{2.166}$$

$$\frac{d\theta}{d\tau}(0) = \dot{\theta}_0 \tag{2.167}$$

We introduce a dimensionless time $t = \sqrt{\frac{g}{l}}t$ such that (2.165) may be written as

$$\ddot{\theta}(t) + \sin\theta(t) = 0 \tag{2.168}$$

with initial conditions

$$\theta(0) = \theta_0 \tag{2.169}$$
$$\dot{\theta}(0) = \dot{\theta}_0 \tag{2.170}$$

Assume that the pendulum wire is a massless rod, such that $-\pi \leq \theta_0 \leq \pi$.

The total energy (potential + kinetic), which is constant, may be written in dimensionless form as

$$\frac{1}{2}(\dot{\theta})^2 - \cos\theta = \frac{1}{2}(\dot{\theta}_0)^2 - \cos\theta_0 \tag{2.171}$$

We define an energy function $E(\theta)$ from (2.171) as

$$E(\theta) = \frac{1}{2}[(\dot{\theta})^2 - (\dot{\theta}_0)^2] + \cos\theta_0 - \cos\theta \tag{2.172}$$

We see that this function should be identically equal to zero at all times. But, when it is computed numerically, it will deviate from zero due to numerical errors.

Movie 1: `mov-ch1/pendulum.mp4`

a) Write a Python program that solves the ODE in (2.168) with the specified initial conditions using Heun's method, for given values of θ_0 and $\dot{\theta}_0$. Set for instance $\theta_0 = 85^o$ and $\dot{\theta}_0 = 0$. (remember to convert use rad in ODE) Experiment with different time steps Δt, and carry out the computation for at least a whole period. Plot both the amplitude and the energy function in (2.172) as functions of t. Plot in separate figures.

b) Solve **a**) using Euler's method.

c) Solve the linearized version of the ODE in (2.168):

$$\ddot{\theta}(t) + \theta(t) = 0 \tag{2.173}$$
$$\theta(0) = \theta_0,\ \dot{\theta}(0) = 0 \tag{2.174}$$

using both euler and Heuns method. Plot all four solutions (Problem 2, 3a and b) in the same figure. Experiment with different timesteps and values of θ_0.

Hint 1. Euler implementations of the linearized version of this problem may be found in the digital compendium. See `http://folk.ntnu.no/leifh/teaching/tkt4140/._main007.html`.

Hint 2. Figure should look something like this:

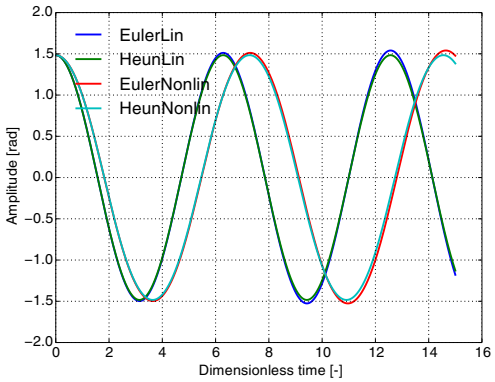

Figure 2.29: Problem 3 with 2500 samplepoints and $\theta_0 = 85^o$.

Exercise 4: Comparison of 2nd order RK-methods

In this exercise we seek to compare several 2nd order RK-methods as outlined in 2.10.

a) Derive the Midpoint method $a_2 = 1$ and Ralston's method $a_2 = 2/3$ from the generic second order RK-method in (2.86).

b) Implement the Midpoint method $a_2 = 1$ and Ralston's method $a_2 = 2/3$ and compare their numerical predictions with the predictions of Heun's method for a model problem.

Chapter 3

Shooting Methods for Boundary Value Problems

3.1 Shooting methods for boundary value problems with linear ODEs

Shooting methods are developed to transform boundary value probelms (BVPs) for ordinary differential equations to an equivalent initial value problem (IVP). The term "shooting method" is inspired by the problem illustrated in Figure 3.1, where the problem is to "shoot" a ballistic object in the field of gravity, aming hit a target at a given length L. The initial angle α is then changed in an repeated fasion, based on observed lengths, until the target at L is hit.

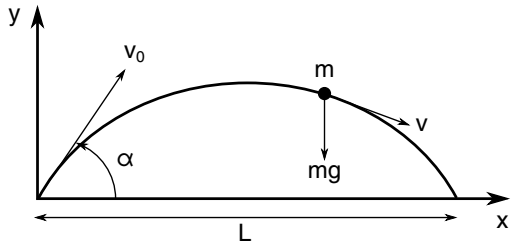

Figure 3.1: The trajectory of a ballistic object launched with an inital angle α.

The problem is a boundary value problem with one condition given at $x = 0$ and another at $x = L$. It might seems obvious that the problem of varying α until the condition at $x = L$ is satisfied has a solution. We will use this approach on problems which has nothing to do with ballistic trajectories.

Let us consider the following example:

$$y'' = y(x) \tag{3.1}$$

which is a second order, linear ODE with initial conditions:

$$y(0) = 0,\ y'(0) = s \tag{3.2}$$

and consequently an initial value problem which can be shown to have the following analytical solution:

$$y(x) = s \cdot \sinh(x) \tag{3.3}$$

For each choice of the initial value $y'(0) = s$ we will get a new solution $y(x)$, and in Figure 3.2 we see the solutions for $s = 0.2$ og 0.7.

The problem we really is (3.1) with the following boundary conditions:

$$y(0) = 0,\ y(1) = 1 \tag{3.4}$$

which is what we call a boundary value problem.

From (3.3) we realize that the boundary problem may be solve by selecting $s = s^*$ such that $y(1) = s^* \cdot \sinh(1)$ or

$$s^* = \frac{1}{\sinh(1)} \tag{3.5}$$

For this particular case we are able to find the analytical solution of both the initial problem and the boundary value problem. When this is not the case our method of approach is a *shooting method* where we select values of s until the condition $y(1) = 1$ is fulfilled. For arbitrary values of s the boundary value $y(1)$ becomes a function of s, which is linear if the ODE is linear and nonlinear if the ODE is nonlinear.

To illustrate the shooting method, consider a somewhat general boundary problem for a second order linear ODE:

$$y''(x) = p(x) \cdot y'(x) + q(x) \cdot y(x) + r(x) \tag{3.6}$$

where $p(x)$, $q(x)$ are arbritary functions and $r(x)$ may be considered as a source term. The boundary values may be prescribed as:

$$y(a) = \alpha,\ y(b) = \beta \tag{3.7}$$

The second order ODE (3.6) may be reduced to a system of two ODEs (see 2.4):

$$\begin{aligned} y'(x) &= g(x) \\ g'(x) &= p(x) \cdot g(x) + q(x) \cdot y(x) + r(x) \end{aligned} \tag{3.8}$$

with the boundary conditions given in (3.7).

We start the shooting method by chosing the given boundary value $y(a) = \alpha$ as an initial condition. As we need an initial condition for $y'(\alpha) \equiv g(\alpha)$ to solve (3.8) as an initial value problem, we have to guess and initial value in a way such that the boundary value $y(b) = \beta$ is satisfied. As (3.6) is a linear ODE it suffices to guess two values of $s = g(\alpha) \equiv y'(a)$. The correct value for the initial value of s which gives the correct value of the solution $y(b) = \beta$, may then be found by linear interpolation. Note that $y(x)$ is always proportional to s when the ODE is linear.

To quantify the goodness of how well the boundary value resulting from our initial guess $s = g(a) \equiv y'(a)$, we introduce a *boundary value error function* ϕ:

$$\phi(s) = y(b; s) - \beta \tag{3.9}$$

The correct initial guess $s = s^*$ is found when the boundary value error function is zero:

$$\phi(s^*) = 0 \tag{3.10}$$

The procedure may be outlined as follows:

The shooting method for linear boundary value problems.

1. Guess two initial values s^0 og s^1
2. Compute the corresponding values for the error function ϕ^0 og ϕ^1 by solving the initial value problem in (3.8)
3. Find the correct initial value s^* by linear interpolation.
4. Compute the solution to the boundary value problem by solving the initial value problem in (3.8) with $g(a) = y'(a) = s^*$.

To see how we find s^* from lineariztion, consider first ϕ as a linear fuction of s:

$$\phi = k_a \cdot s + k_b \tag{3.11}$$

where k_a is the slope and k_b is the offset when $s = 0$. We easily see from (3.11) that $\phi = 0$ for $s^* = -\dfrac{k_b}{k_a}$. When we have two samples ϕ^0 og ϕ^1 of ϕ at s^0 og s^1, both k_a and k_b may be found from (3.11) to be:

$$k_a = \frac{\phi^1 - \phi^0}{s^1 - s^0}, \qquad k_b = \frac{s^1 \cdot \phi^0 - \phi^1 \cdot s^0}{s^1 - s^0} \tag{3.12}$$

and concequently we may find s^* as:

$$s^* = \frac{\phi^1 s^0 - \phi^0 s^1}{\phi^1 - \phi^0} \tag{3.13}$$

which also may be presented on an incremental form as:

$$s^* = s^1 + \Delta s, \qquad \Delta s = -\phi^1 \cdot \left(\frac{s^1 - s^0}{\phi^1 - \phi^0} \right) \tag{3.14}$$

Let us go back to our model boundary value example (3.1) with boundary values as prescribed in (3.4) which may be presented as reduced system of first order ODEs:

$$\begin{aligned} y'(x) &= g(x) \\ g'(x) &= y(x) \end{aligned} \tag{3.15}$$

For the given values of b and β and $s = y'(0) = g(0)$ the boundary value error function in (3.9) takes the form:

$$\phi(s) = y(1; s) - 1 \tag{3.16}$$

In accordance with the shooting method we choose two initial values $s^0 = 0.2$ og $s^1 = 0.7$ and solve (3.15) as an initial value problem with e.g. and RK4-solver with $\Delta x = 0.1$ and obtain the following results:.

m	s^m	$\phi(s^m)$
0	0.2	-0.7650
1	0.7	-0.1774

Note that we use m as a superindex for iterations, which will be used in case of nonlinear equations. Substitution into (3.13) gives the $s^* = 0.8510$. Subsequent use of the s^*-value in the RK4-solver yields $\phi(0.8510) = 0.0001$. We observe a good approximation to the correct value for the initial value may be found from the analytical solution as: $y'(0) = \dfrac{1}{\sinh(1)} = 0.8509$

The presentation above with the shooting method is chosen as it can easily be generalized to the solution of nonlinear ODEs.

Alternative approach for linear second order ODEs. For linear second order ODEs the solution may be found in somewhat simpler fashion, by solving the following sub-problems:

Sub-problem 1

$$y_0''(x) = p(x) \cdot y_0'(x) + q(x) \cdot y_0(x) + r(x), \qquad y_0(a) = \alpha, \quad y_0'(a) = 0 \tag{3.17}$$

and

Sub-problem 2

$$y_1''(x) = p(x) \cdot y_1'(x) + q(x) \cdot y_1(x), \qquad y_1(a) = 0, \quad y_1'(a) = 1 \tag{3.18}$$

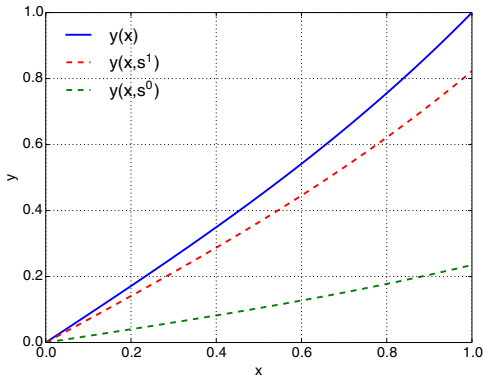

Figure 3.2: The solution $y(x)$ of a boundary value problem resulting from two initial guesses $y(x; s^0)$ and $y(x; s^1)$.

That is, the two sub-problems differ only by the source term $r(x)$ and the boundary conditions. Notice that the condition $y'(\alpha) = 0$ in (3.17) corresponds to $s^0 = 0$ and the condition $y'(\alpha) = 1$ i (3.18) corresponds to $s^1 = 1$.

Let $y_0(x)$ represent the solution to (3.17) and $y_1(x)$ be the solution to (3.18). The complete solution of the boundary value problem in (3.6) with the boundary conditions (3.7) may then be shown to be:

$$y(x) = y_0(x) + \left[\frac{\beta - y_0(b)}{y_1(b)}\right] \cdot y_1(x) = y_0(x) - \left[\frac{\phi^0}{\phi^1 + \beta}\right] \cdot y_1(x) \tag{3.19}$$

with $s^0 = 0$ og $s^1 = 1$. Note that the solution of (3.17) corresponds to a particular solution of (3.1) and (3.4), and that the solution of (3.18) corresponds to the homogenous solution of (3.1) and (3.4) as the source term is missing. From general theory on ODEs we know that the full solution may be found by the sum of a particular solution and the homogenous solution, given that the boundary conditions are satisfied.

3.1.1 Example: Couette-Poiseuille flow

In a Couette-Poiseuille flow, we consider fluid flow constrained between two walls, where the upper wall is moving at a prescribed velocity U_0 and at a distance $Y = L$ from the bottom wall. Additionally, the flow is subjected to a prescribed pressure gradient $\frac{\partial p}{\partial x}$.

We assume that the vertical velocity component $V = 0$ and concequently we get from continuity: $\frac{\partial U}{\partial X} = 0$ which implies that $U = U(Y)$, i.e. the velocity in the streamwise X-direction depends on the cross-wise Y-direction only.

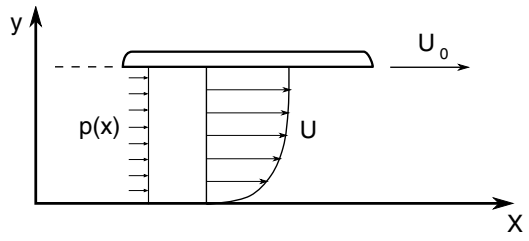

Figure 3.3: An illustration of Couette flow driven by a pressure gradient and a moving upper wall.

The equation of motion in the Y-direction simplifies to $\frac{\partial p}{\partial Y} = -\rho g$, whereas the equation of motion in the X-direction has the form:

$$\rho U \cdot \frac{\partial U}{\partial X} = -\frac{\partial p}{\partial X} + \mu \left(\frac{\partial^2 U}{\partial X^2} + \frac{\partial^2 U}{\partial Y^2} \right)$$

which due to the above assumptions and ramifications reduces to

$$\frac{d^2 U}{dY^2} = \frac{1}{\mu} \frac{dp}{dX} \tag{3.20}$$

with no-slip boundary conditions: $U(0) = 0,\ U(L) = U_0$. To render equation (3.20) on a more appropriate and generic form we introduce dimesionless variables: $u = \frac{U}{U_0},\ y = \frac{Y}{L},\ P = -\frac{1}{U_0}(\frac{dp}{dX})\frac{L^2}{\mu}$, which yields:

$$\frac{d^2 u}{dy^2} = -P \tag{3.21}$$

with corresponding boundary conditions:

$$u = 0 \,\text{for}\, y = 0, \qquad u = 1 \,\text{for}\, y = 1 \tag{3.22}$$

An analytical solution of equation (3.21) with the corresponding boundary conditions (3.22) may be found to be:

$$u = y \cdot \left[1 + \frac{P}{2}(1 - y) \right] \tag{3.23}$$

Observe that for $P \leq -2$ we will get negative velocities for some values of y. In Figure 3.4 velocity profiles are illustrated for a range of non-dimensional pressure gradients P.

To solve (3.21) numerically, we represent is as a system of equations:

$$\begin{aligned} u'(y) &= u_1(y) \\ u_1'(y) &= -P \end{aligned} \tag{3.24}$$

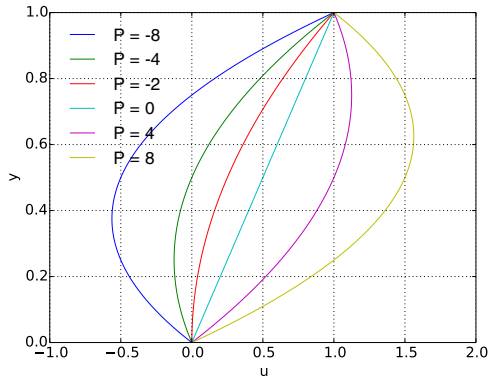

Figure 3.4: Velocity profiles for Couette-Poiseuille flow with various pressure gradients.

with corresponding boundary conditions:

$$u(0) = 0, \; u(1) = 1 \tag{3.25}$$

To solve this boundary value problem with a shooting method, we must find $s = u'(0) = u_1(0)$ such that the boundary condition $u(1) = 1$ is satisfied. We can express this condition in Dette kan uttrykkes på følgende måte:

$$\phi(s) = u(1; s) - 1, \qquad \text{such that} \qquad \phi(s) = 0 \qquad \text{when} \qquad s = s^*$$

We guess two values s^0 og s^1 and compute the correct s by linear interpolation due to the linearity of system of ODEs (3.24). For the linear interpolation see equation (3.14).

The shooting method is implemented in the python-code `Poiseuille_shoot.py` and results are computed and plotted for a range of non-dimensional pressure gradients and along with the analytical solution.

```
# src-ch2/Couette_Poiseuille_shoot.py;ODEschemes.py @ git@lrhgit/tkt4140/src/src-ch2/ODEschemes.py;

from ODEschemes import euler, heun, rk4
import numpy as np
from matplotlib.pyplot import *

# change some default values to make plots more readable
LNWDT=5; FNT=11
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT


N=200
L = 1.0
y = np.linspace(0,L,N+1)
```

```
def f(z, t):
    """RHS for Couette-Posieulle flow"""
    zout = np.zeros_like(z)
    zout[:] = [z[1], -dpdx]
    return zout

def u_a(y,dpdx):
    return y*(1.0 + dpdx*(1.0-y)/2.0);

beta=1.0 # Boundary value at y = L


# Guessed values
s=[1.0, 1.5]

z0=np.zeros(2)

dpdx_list=[-5.0, -2.5, -1.0, 0.0, 1.0,2.5, 5.0]
legends=[]

for dpdx in dpdx_list:
    phi = []
    for svalue in s:
        z0[1] = svalue
        z = rk4(f, z0, y)
        phi.append(z[-1,0] - beta)

    # Compute correct initial guess
    s_star = (s[0]*phi[1]-s[1]*phi[0])/(phi[1]-phi[0])
    z0[1] = s_star

    # Solve the initial value problem which is a solution to the boundary value problem
    z = rk4(f, z0, y)

    plot(z[:,0],y,'-.')
    legends.append('rk4: dp='+str(dpdx))

    # Plot the analytical solution
    plot(u_a(y, dpdx),y,':')
    legends.append('exa: dp='+str(dpdx))

# Add the labels
legend(legends,loc='best',frameon=False) # Add the legends
xlabel('u/U0')
ylabel('y/L')
show()
```

3.1.2 Example: Simply supported beam with constant cross-sectional area

Figure 3.5 illustrates a simply supported beam subjected to an evenly distributed load q per length unit and a horizontal force P.

The differential equation for the deflection $U(X)$ is given by:

$$\frac{d^2U}{dX^2} + \frac{P}{EI}U = -\frac{q}{2EI}(L^2 - X^2),\ P > 0 \tag{3.26}$$

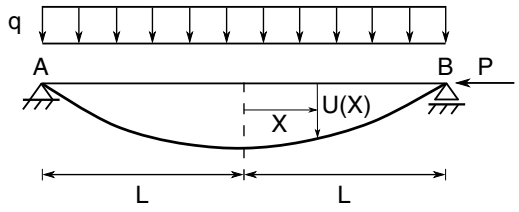

Figure 3.5: Simply supported beam with an evenly distributed load q per length unit.

with the following boundary conditions:

$$U(-L) = U(L) = 0 \tag{3.27}$$

where EI denotes the flexural rigidity. The equation (3.26) was linearized by assuming small deflections. Alternatively we may assume $\frac{dU}{dX}(0) = 0$ due to the symmetry.

For convenience we introduce dimensionless variables:

$$x = \frac{x}{L},\ u = \frac{P}{qL^2} \cdot U,\ \theta^2 = \frac{PL^2}{EI} \tag{3.28}$$

which by substitution in (3.26) and (3.27) yield:

$$\frac{d^2u}{dx^2} + \theta^2 \cdot u = \theta^2 \frac{(1-x^2)}{2}, \qquad -1 < x < 1 \tag{3.29}$$

with the corresponding boundary conditions:

$$u(-1) = 0, \quad u(1) = 0 \tag{3.30}$$

Equation (3.29) with boundary conditions (3.30) have the analytixcal solution:

$$u(x) = \frac{1}{\theta^2} \cdot \left[\frac{\cos(\theta x)}{\cos(\theta)} - 1\right] - \frac{(1-x^2)}{2} \tag{3.31}$$

The buckling load for this case is given by $P_k = \dfrac{\pi EI}{4L^2}$ such that

$$0 \leq \theta \leq \frac{\pi}{2} \tag{3.32}$$

Numerical solution. We wish to solve the second order linear ODE in (3.29) by means of shooting methods and choose the alternative approach as outlined in equation (3.19).

Two similar sub-problems are then two be solved; they differ only with the source term and the boundary conditions.

Sub-problem 1

$$u_0''(x) = -\theta^2\, u_0(x) + \theta^2\, \frac{(1-x^2)}{2}, \qquad u_0(-1) = 0, \quad u_0'(-1) = 0 \tag{3.33}$$

$$\tag{3.34}$$

Sub-problem 2

$$u_1''(x) = -\theta^2 \cdot u_1(x) \qquad u_1(-1) = 0, \quad u_1'(-1) = 1 \tag{3.35}$$

$$\tag{3.36}$$

From the superposition principle for linear equation given in equation n (3.19) a complete solution is obtained as a combination of the solutions to the two sub-problems:

$$u(x) = u_0(x) - \frac{u_0(1)}{u_1(1)} \cdot u_1(x) \tag{3.37}$$

Now, to solve the equation numerically we write (3.33) and (3.35) as a system of first order ODEs:

Sub-problem 1 as a system of first order ODEs

For convenience we introduce the notation $u_0 = y_1$ and $u_o' = y_2$:

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= -\theta^2 \cdot \left(y_1 + \frac{(1-x^2)}{2} \right) \end{aligned} \tag{3.38}$$

with the corresponding initial conditions

$$y_1(-1) = 0, \qquad y_2(-1) = 0 \tag{3.39}$$

Sub-problem 2 as a system of first order ODEs

With $u_1 = y_1$ og $u_1' = y_2$:

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= -\theta^2 y_1 \end{aligned} \tag{3.40}$$

and initial conditions

$$y_1(-1) = 0, \qquad y_2(-1) = 1 \tag{3.41}$$

Both sub-problems must be integrated from $x = -1$ to $x = 1$ such that $u_0(1)$ and $u_1(1)$ may be found and the solution to the original problem in equation (3.29) may be constructed according to (3.37). A python-code `beam_deflect_shoot_constant.py` for the problem is listed below. Here the resluting solution is compared with the analytical solution too.

```
# src-ch2/beam_deflect_shoot_constant.py;ODEschemes.py @ git@lrhgit/tkt4140/src/src-ch2/ODEschemes.py

from ODEschemes import euler, heun, rk4
from numpy import cos, sin
import numpy as np
from matplotlib.pyplot import *

# change some default values to make plots more readable
LNWDT=2; FNT=15; FMLY='helvetica';
rcParams['lines.linewidth'] = LNWDT
rcParams['font.size'] = FNT
rcParams['font.family'] = FMLY


def SubProblem1(y, x):
    """ system 1 of Governing differential equation on beam bending with constant cross section
    Args:
        y(array): an array containg y and its derivatives up to second order. (RHS)
        x(array): an array
    Returns:
        yout(array): dydx RHS of reduced system of 1st order equations
    """
    yout = np.zeros_like(y)
    yout[:] = [y[1],-theta2*(y[0]+0.5*(1-x**2))]

    return yout

def SubProblem2(y, x):
    """ system 2 of Governing differential equation on beam bending with constant cross section
    Args:
        y(array): an array containg y and its derivatives up to second order. (RHS)
        x(array): an array
    Returns:
        yout(array): dydx RHS of reduced system of 1st order equations
    """
    yout = np.zeros_like(y)
    yout[:] = [y[1], -theta2*y[0]]
    return yout

# === main program starts here ===

N = 20 # number of elements
L = 1.0 # half the length of the beam
x = np.linspace(-L, L, N + 1) # allocate space
theta = 1 # PL**2/EI
theta2 = theta**2

solverList = [euler, heun, rk4]
solver = solverList[2]

s = [0, 1] # guessed values
# === shoot ===
y0Sys1 = [0, s[0]] # initial values of u and u'
y0Sys2 = [0, s[1]]

u0 = solver(SubProblem1, y0Sys1,x)
u1 = solver(SubProblem2, y0Sys2,x)

u0 = u0[:,0] # extract deflection from solution data
u1 = u1[:,0]
```

```
u = u0 -(u0[-1]/u1[-1])*u1 # interpolate to find correct solution
ua = (1/theta2)*(cos(theta*x)/cos(theta) - 1)-(1 - x**2)/2 # analytical solution

legendList=[] # empty list to append legends as plots are generated

plot(x,u,'y',)
plot(x,ua,'r:')
legendList.append('shooting technique')
legendList.append('analytical solution')
## Add the labels
legend(legendList,loc='best',frameon=False)
ylabel('u')
xlabel('x')
grid(b=True, which='both', color='0.65',linestyle='-')
#savefig('../fig-ch2/beam_deflect_shoot_constant.png', transparent=True)
#savefig('../fig-ch2/beam_deflect_shoot_constant.pdf', transparent=True)
#sh
show()
```

The output of `beam_deflect_shoot_constant.py` is shown in Figure 3.6.

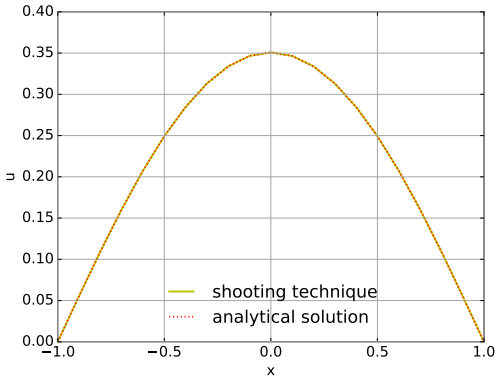

Figure 3.6: The anyalytical and numerical solutions for simply supported beam with an evenly distributed load q per length unit. The results are produced with a shooting method implemented in `beam_deflect_shoot_constant.py`.

3.1.3 Example: Simply supported beam with varying cross-sectional area

In this example we will study a simply supported beam with varying cross-sectional area and refer to the figure in our previous example (3.1.2). The difference with the previous example is that the second area momentis a function of X due to the varying cross-sectional area.

We denote the second area moment at $X = 0$ as I_0, and let the second area moment I vary in the following manner:

$$I(X) = \frac{I_0}{1 + (X/L)^n}, \ n = 2, 4, 6, \ldots \tag{3.42}$$

Consider a rectangular cross-sectional area with a constant width b and a varying height h:

$$h(X) = \frac{h_0}{[1 + (X/L)^n]^{1/3}} \tag{3.43}$$

where h_0 denotes h at $X = 0$. From equation (3.26) in example (3.1.2) we have:

$$\frac{d^2U}{dX^2} + \frac{P}{EI}U = -\frac{q}{2EI}(L^2 - X^2) \tag{3.44}$$

$$U(-L) = U(L) = 0, \quad \frac{dU(0)}{dX} = 0 \tag{3.45}$$

We start by computing the moment distribution $M(X)$ in the beam, which is related with the displaceent by the following expression:

$$\frac{d^2U}{dX^2} = -\frac{M}{EI} \tag{3.46}$$

By substitution of equation (3.46) in (3.44) and two esubsequent differentiations we get:

$$\frac{d^2M}{dX^2} + \frac{P}{EI}@, M = -q \tag{3.47}$$

To represent the resulting second order ODE on a more convenient and generic form we introduce the dimensionless variables:

$$x = \frac{X}{L}, \ m = \frac{M}{qL^2} \tag{3.48}$$

and let

$$P = \frac{EI_0}{L^2} \tag{3.49}$$

such that equation (3.47) becomes:

$$m''(x) + (1 + x^n) \cdot m(x) = -1 \tag{3.50}$$

with the boundary conditions:

$$m(-1) = m(1) = 0, \ m'(0) = 0 \tag{3.51}$$

To our knowledge, no analytical solution exists for equation (3.50), even if it is linear and analytical solutions exists for the previous example in (3.1.2).

Once the moment distribution has been computed, the dimensionless displacement $u(x)$ may retrieved from:

$$u(x) = m(x) - \frac{1}{2}(1 - x^2) \tag{3.52}$$

The displacement in physical dimensions U may subsequently computed from the dimensionless u:

$$U = \frac{qL^4}{EI_0}u \tag{3.53}$$

Further, the differential equaiton for $u(x)$ may be found by substitution of equation (3.52) into (3.50):

$$u''(x) + (1 + x^n) \cdot u(x) = -\frac{1}{2}(1 - x^2) \cdot (1 + x^n) \tag{3.54}$$

By expoliting the symmetry in the problem, we solve equation (3.50) numerically with a shooting method. The boundary conditions are:

$$m(1) = 0, \qquad m'(0) = 0 \tag{3.55}$$

The dimensionless displacement $u(x)$ is subsequently computed from equation (3.52). We have implemented the outlined approach in `beam_deflect_shoot_varying.py` in such a way that one easily may choose between various RK-solvers (`euler`, `heun`, og `rk4`) from the module `ODEschemes`.

We represent the second order ODE (3.50) as a system of first order ODEs by introducing the following conventions $m(x) = y_1(x)$ og $m'(x) = y_2(x)$:

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= -\left(1 + (1 + x^n) \cdot y_1\right) \end{aligned} \tag{3.56}$$

and the boundary conditions become:

$$y_2(0) = y_1(1) = 0 \tag{3.57}$$

In this case $y_2(0) \equiv m'(0)$ is given, which leaves $m(0) \equiv y_1(0)$ to be guessed such that the boundary condition at the other end $m(1) \equiv y_1(1) = 0$ is fulfilled. To express the approach algorithmically as previously, we let $s = y_1(0)$ and let the bondary value error function be $\phi(s) = y_1(1; s)$.

Simplified shooting method for linear ODEs

1. Let the two initial values $s^0 = 0$ og $s^1 = 1$ for simplicity
2. Compute the corresponding values for the error function ϕ^0 og ϕ^1 by solving the initial value problem (in the current example (3.56)).
3. Find the correct initial value s^* by linear interpolation:

$$s^* = \frac{\phi^0}{\phi^0 - \phi^1} \tag{3.58}$$

The simplfied expression for s in equation (3.58) may be found from (3.14) by setting $s^0 = 0$ og $s^1 = 1$.

One should be aware of that the choice in (3.58) is not necessarily always a good choice even though the ODE is linear. If the solution is of the kind $e^{\alpha x}$ when both α and x are large, we may end up outside the allowable range even for double precision which is approximately 10^{308}. In our example above this is not a problem and we may use (3.58) and we have implemented the approach in `beam_deflect_shoot_varying.py` as shown below where both the moment and deflection is computed for $n = 2$.

```
# src-ch2/beam_deflect_shoot_varying.py;ODEschemes.py @ git@lrhgit/tkt4140/src/src-ch2/ODEschemes.py

from ODEschemes import euler, heun, rk4
from numpy import cos, sin
import numpy as np
from matplotlib.pyplot import *

# change some default values to make plots more readable
LNWDT=3; FNT=11
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT

def f(y, x):
    """Governing differential equation on beam bending as a system of 1. order equations.
    Args:
        y(array): an array containg y and its derivatives up to second order. (RHS)
        x(array): space array
    Returns:
        dydx(array): dydx. RHS of reduced system of 1st order equations
    """
    yout = np.zeros_like(y)
    yout[:] = [y[1],-(1+(1+x**n)*y[0])]

    return yout


# === main program starts here ===

N = 10 # number of elements
L = 1.0 # half the length of the beam
x = np.linspace(0,L,N+1) # vector of
theta = 1 # PL**2/EI
theta2 = theta**2
h0=1 # height of beam at x = 0
n=2 # polynomial order to go into h(x)
h=h0/(1+(x/L)**n)**(1/3.) # height of beam assuming constant width

solvers = [euler, heun, rk4]
solver = solvers[2]

s = [0, 1] # guessed values
# === shoot ===
y01 = [s[0], 0] # initial values
```

```
y02 = [s[1], 0]

m0 = solver(f, y01, x)
m1 = solver(f, y02, x)

phi0 = m0[-1,0]
phi1 = m1[-1,0]

sfinal = phi0/(phi0 - phi1) # find correct initial value of moment m(x=0) using secant method

y03 = [sfinal, 0]

mfinal = solver(f,y03,x)
m = mfinal[:, 0] # extract moment from solution data

u = m -0.5*(1 - x**2) # final solution of dimensionless deflection
ua = (1/theta2)*(cos(theta*x)/cos(theta)-1)-(1-x**2)/2 #analytical solution with constant stifness

legendList=[] # empty list to append legends as plots are generated

plot(x, m, 'b',)
legendList.append('m')
plot(x, u, 'r',)
legendList.append('u')
## Add the labels
legend(legendList,loc='best',frameon=False)
ylabel('m, u')
xlabel('x')
grid(b=True, which='both', axis='both',linestyle='-')
show()
```

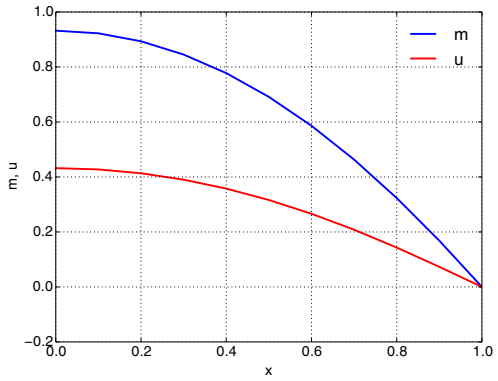

Figure 3.7: Moment m and deflection u .

3.2 Shooting methods for boundary value problems with nonlinear ODEs

As model examaple for a non-linear boundary value problems we will investigate:

$$\begin{aligned} y''(x) &= \tfrac{3}{2}y^2 \\ y(0) = 4,\ & y(1) = 1 \end{aligned} \tag{3.59}$$

Our model problem (3.59) may be proven to have two solutions.

Solution I:

$$y_I = \frac{4}{(1+x)^2} \tag{3.60}$$

Solution II: Can be expressed by elliptic Jacobi-functions (see G.3 in Numeriske Beregninger)

Both solutions are illustrated in Figure 3.8 .

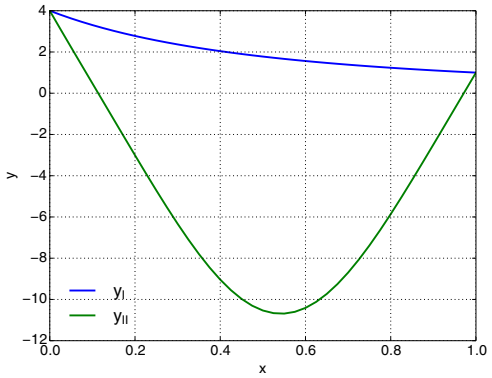

Figure 3.8: The two solutions y_I og y_{II} of the non-linear ODE (3.59).

Our model equation (3.59) may be reduced to a system of ODEs in by introducing the common notation $y \to y_0$ og $y' = y_0' = y_1$ and following the procedure in 2.4:

$$\begin{aligned} y_0'(x) &= y_1(x) \\ y_1'(x) &= \tfrac{3}{2}[y_0(x)]^2 \end{aligned} \tag{3.61}$$

with the corresponding boundary conditions:

$$y_0(0) = 4,\ y_0(1) = 1 \tag{3.62}$$

We will use the shooting method to find $s = y'(0) = y_1(0)$ such that the boundary condition $y_0(1) = 1$ is satisfied. Our boundary value error function becomes:

$$\phi(s^m) = y_0(1; s^m) - 1, \ m = 0, 1, \ldots \text{such that } y_0(1) \to 1 \text{ for } \phi(s^m) \to 0, \ m \to \infty \tag{3.63}$$

As our model equation is non-linear our error function $\phi(s)$ will also be non-linear, and we will have to conduct an iteration process as outlined in (3.63). Our task will then be to find the values s^* so that our boundary value error function $\phi(s)$ becomes zero $\phi(s^*) = 0$. For this purpose we choose the *secant method* (See [2], section 3.3).

Two initial guesses s^0 and s^1 are needed to start the iteration process. Eq. (3.61) is then solved twice (using s^0 and s^1). From these solutions two values for the error function (3.63) may be calculated, namely ϕ^0 and ϕ^1. The next value of s (s^2) is then found at the intersection of the gradient (calculated by the secant method) with the $s - axis$.

For an arbitrary iteration, illustrated in Figure 3.9 we find it more convenient to introduce the Δs:

$$s^{m+1} = s^m + \Delta s \tag{3.64}$$

where

$$\Delta s = s^{m+1} - s^m = -\phi(s^m) \cdot \left[\frac{s^m - s^{m-1}}{\phi(s^m) - \phi(s^{m-1})} \right], \ m = 1, 2, \ldots$$

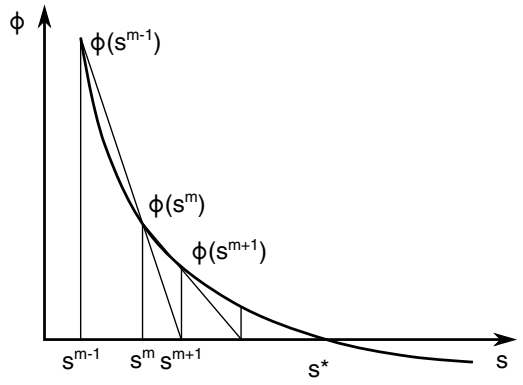

Figure 3.9: An illustration of the usage of the secant method label to find the zero for a non-linear boundary value error function.

Given that s^{m-1} og s^m may be taken as known quantities the iteration process process may be outlined as follows:

Iteration process

1. Compute $\phi(s^{m-1})$ og $\phi(s^m)$ from numerical solution of (3.61) and (3.63).
2. Compute Δs and s^{m+1} from (3.64)
3. Update
   - $s^{m-1} \leftarrow s^m$
   - $s^m \leftarrow s^{m+1}$
   - $\phi(s^{m-1}) \leftarrow \phi(s^m)$
4. Repeat 1-3 until convergence

Examples on convergence criteria:

Control of absolute error:

$$|\Delta s| < \varepsilon_1 \tag{3.65}$$

Control of relative error:

$$\left| \frac{\Delta s}{s^{m+1}} \right| < \varepsilon_2 \tag{3.66}$$

The crietria (3.65) and (3.66) are frequently used in combination with:

$$|\phi(s^{m+1})| < \varepsilon_3 \tag{3.67}$$

We will now use this iteration process to solve (3.61). However, a frequent problem is to find appropriate initial guesses, in particular in when ϕ is a non-linear function and may have multiple solutions. A simple way to assess ϕ is to make implement a program which plots ϕ for a wide range of s-values and then vary this range until the zeros are in the range. An example for such a code is given below for our specific model problem:

```
# src-ch2/phi_plot_non_lin_ode.py;ODEschemes.py @ git@lrhgit/tkt4140/src/src-ch2/ODEschemes.py;

from ODEschemes import euler, heun, rk4
from numpy import cos, sin
import numpy as np
from matplotlib.pyplot import *

# change some default values to make plots more readable
LNWDT=2; FNT=11
```

```
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT
font = {'size' : 16}; rc('font', **font)


N=20
L = 1.0
x = np.linspace(0,L,N+1)

def f(z, t):
    zout = np.zeros_like(z)
    zout[:] = [z[1],3.0*z[0]**2/2.0]
    return zout

beta=1.0 # Boundary value at x = L

solvers = [euler, heun, rk4] #list of solvers
solver=solvers[2] # select specific solver

smin=-45.0
smax=1.0
s_guesses = np.linspace(smin,smax,20)

# Guessed values
#s=[-5.0, 5.0]

z0=np.zeros(2)
z0[0] = 4.0

z = solver(f,z0,x)
phi0 = z[-1,0] - beta

nmax=10
eps = 1.0e-3
phi = []
for s in s_guesses:
    z0[1] = s
    z = solver(f,z0,x)
    phi.append(z[-1,0] - beta)


legends=[] # empty list to append legends as plots are generated

plot(s_guesses,phi)
ylabel('phi')
xlabel('s')
grid(b=True, which='both', color='0.65',linestyle='-')

show()
close()
```

Based on our plots of ϕ we may now provided qualified inital guesses for and choose $s^0 = -3.0$ og $s^1 = -6.0$. The complete shooting method for the our boundary value problem may be found here:

```
# src-ch2/non_lin_ode.py;ODEschemes.py @ git@lrhgit/tkt4140/src/src-ch2/ODEschemes.py;

from ODEschemes import euler, heun, rk4
from numpy import cos, sin
import numpy as np
from matplotlib.pyplot import *
```

```
# change some default values to make plots more readable
LNWDT=3; FNT=16
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT



N=40
L = 1.0
x = np.linspace(0,L,N+1)

def dsfunction(phi0,phi1,s0,s1):
    if (abs(phi1-phi0)>0.0):
        return    -phi1 *(s1 - s0)/float(phi1 - phi0)
    else:
        return 0.0


def f(z, t):
    zout = np.zeros_like(z)
    zout[:] = [z[1],3.0*z[0]**2/2.0]
    return zout


def y_analytical(x):
    return 4.0/(1.0+x)**2


beta=1.0 # Boundary value at x = L

solvers = [euler, heun, rk4] #list of solvers
solver=solvers[2] # select specific solver

# Guessed values
# s=[-3.0,-9]
s=[-40.0,-10.0]

z0=np.zeros(2)
z0[0] = 4.0
z0[1] = s[0]

z = solver(f,z0,x)
phi0 = z[-1,0] - beta

nmax=10
eps = 1.0e-3
for n in range(nmax):
    z0[1] = s[1]
    z = solver(f,z0,x)
    phi1 = z[-1,0] - beta
    ds = dsfunction(phi0,phi1,s[0],s[1])
    s[0]  = s[1]
    s[1] +=  ds
    phi0 = phi1
    print 'n = {}  s1 = {} and ds = {}'.format(n,s[1],ds)

    if (abs(ds)<=eps):
        print 'Solution converged for eps = {} and s1 ={} and ds = {}. \n'.format(eps,s[1],ds)
        break

legends=[] # empty list to append legends as plots are generated
```

```
plot(x,z[:,0])
legends.append('y')

plot(x,y_analytical(x),':^')
legends.append('y analytical')

# Add the labels
legend(legends,loc='best',frameon=False) # Add the legends
ylabel('y')
xlabel('x/L')
show()
```

After selection $\Delta x = 0.1$ and using the RK4-solver, the following iteration output may be obtained:

m	s^{m-1}	$\phi(s^{m-1})$	s^m	$\phi(s^m)$	s^{m+1}	$\phi(s^{m+1})$
1	-3.0	26.8131	-6.0	6.2161	-6.9054	2.9395
2	-6.0	6.2161	-6.9054	2.9395	-7.7177	0.6697
3	-6.9054	2.9395	-7.7177	0.6697	-7.9574	0.09875
4	-7.7177	0.6697	-7.9574	0.09875	-7.9989	0.0004

After four iterations $s = y'(0) = -7.9989$, while the analytical value is -8.0. The code *non_lin_ode.py* illustrates how our non-linear boundary value problem may be solve with a shooting method and offer graphical comparison of the numerical and analytical solution.

The secant method is simple and efficient and does not require any knowledge of the analytical expressions for the derivatives for the function for which the zeros are sought, like e.g. for Newton-Raphsons method. Clearly a drawback is that two initial guesses are mandatory to start the itertion process. However, by using some physical insight for the problemn and ploting the ϕ-function for a wide range of s-values, the problem is normally possible to deal with.

3.2.1 Example: Large deflection of a cantilever

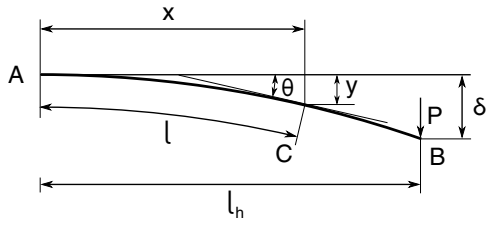

Figure 3.10: Large deflection of a beam subjected to a vertical load.

Consider a cantilever (see figure 3.10) which is anchored at one end A to a vertical support from which it is protruding. The cantilever is subjected to

a structural load P at the other end B. When subjected to a structural load, the cantilever carries the load to the support where it is forced against by a moment and shear stress [8]. In this example we will allow for large deflections and for that reason we introduce the arc length l and the angle of inclination θ as variables.

All lengths are rendered dimensionless by division with the cantilever length L. The arch length l ranges consequently from $l = 0$ in A to $l = 1$ in B. Without further ado we present the differential equation for the elastic cantilever as:

$$\kappa = \frac{d\theta}{L \cdot dl} = \frac{M}{EI} \tag{3.68}$$

where κ denotes the curvature, M the moment and EI flexural rigidity. The balance of moments in C may by computed as C: $M = P \cdot L(l_h - x)$ which by substitution in (3.68) results in:

$$\frac{d\theta}{dl} = \frac{PL^2}{EI}(l_h - x) \tag{3.69}$$

From figure 3.10 we may deduce the following geometrical relations:

$$\frac{dx}{dl} = \cos\theta, \ \frac{dy}{dl} = \sin\theta \tag{3.70}$$

Further, differentiation of (3.69) with respect to the arch length l and substitution of (3.70) yields:

$$\frac{d^2\theta}{dl^2} + \frac{PL^2}{EI}\cos\theta = 0 \tag{3.71}$$

For convenience we introduce the parameter α which is defined as:

$$\alpha^2 = \frac{PL^2}{EI} \tag{3.72}$$

As a result we must solve the following differential equations:

$$\frac{d^2\theta}{dl^2} + \alpha^2\cos\theta = 0 \tag{3.73}$$

$$\frac{dy}{dl} = \sin\theta \tag{3.74}$$

with the following boundary conditions:

$$y(0) = 0, \ \theta(0) = 0, \ \frac{d\theta}{dl}(1) = 0 \tag{3.75}$$

The first two boundary conditions are due the anchoring in A, whereas the latter is due to a vanishing moment in B. The analytical solution of this problem may be found in appendix G, section G.3 in Numeriske Beregninger.

Numerical solution

First, we need to represent (3.73) and (3.74) as a system of first order differntial equations. By introducing the conventions $\theta = z_0$, $\theta' = z_1$ and $y = z_2$ we get:

$$\begin{aligned} z_0' &= z_1 \\ z_1' &= -\alpha^2 \cos z_0 \\ z_2' &= \sin z_0 \end{aligned} \tag{3.76}$$

with the boundary conditions

$$z_0(0) = 0,\ z_1(1) = 0,\ z_2(0) = 0 \tag{3.77}$$

We have to guess the initial value $\theta'(0)$ such that the condition $\frac{d\theta}{dl}(1) = 0$ is satisfied.

To do so, we let $s = \theta'(0) = z_1(0)$ and $\phi(s) = \theta'(1; s) - 0 = z_1(1)$. Consequently, we have to find $s = s^*$ such that $\phi(s^*) = 0$, which with z-variables takes the form: $s = z_1(0)$.

With the introduction of these convendtions the task at hand becomes to find $s = s^*$ such that:

$$\phi(s^*) = z_1(1; s^*) = 0 \tag{3.78}$$

Additionally we find:

$$l_h = \frac{s^*}{\alpha^2} \tag{3.79}$$

Due to the nonlinear nature of (3.76) the function (3.78) will be nonlinear too and we will use the secant method to find s^*. To start the secant iterations we need two intial guesses s^0 og s^1. Suitable initial guesses may be found by first looking at $\phi(s)$ graphically. The python-code `phi_plot_beam_shoot.py` produces Figure 3.11.

```
# src-ch2/phi_plot_beam_shoot.py;ODEschemes.py @ git@lrhgit/tkt4140/src/src-ch2/ODEschemes.py;

from ODEschemes import euler, heun, rk4
from numpy import cos, sin
import numpy as np
from matplotlib.pyplot import *

# change some default values to make plots more readable
LNWDT=3; FNT=11
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT



N=20
L = 1.0
y = np.linspace(0,L,N+1)
```

```
def f(z, t):
    """RHS for deflection of beam"""
    zout = np.zeros_like(z)
    zout[:] = [z[1],-alpha2*cos(z[0]),sin(z[0])]
    return zout

solvers = [euler, heun, rk4] #list of solvers
solver=solvers[2] # select specific solver

alpha2 = 5.0
beta=0.0 # Boundary value at y = L

N_guess = 30
s_guesses=np.linspace(1,5,N_guess)

z0=np.zeros(3)

phi = []
for s_guess in s_guesses:
    z0[1] = s_guess
    z = solver(f,z0,y)
    phi.append(z[-1,1] - beta)

legends=[] # empty list to append legends as plots are generated
plot(s_guesses,phi)

# Add the labels
legend(legends,loc='best',frameon=False) # Add the legends
title('alpha2 = ' + str(alpha2))
ylabel('phi')
xlabel('s')
grid(b=True, which='both')
show()
```

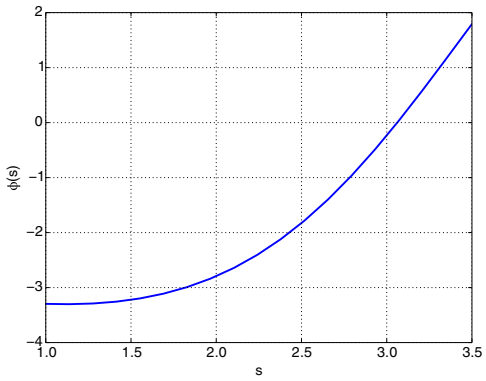

Figure 3.11: A plot of $\phi(s)$ for $\alpha^2 = 5$ for identification of zeros.

From visual inspection of Figure 3.11 we find that ϕ as a zero at approximately $s^* \approx 3.05$. Note, that the zeros depend on the values of α.

Given this approximation of the zero we may provide two initial guesses $s^0 = 2.5$ and $s^1 = 5.0$ which are 'close enough' for the secant method to converge. An example for how the solution may be found and presented is given in `beam_deflect_shoot.py`.

```
# src-ch2/beam_deflect_shoot.py;ODEschemes.py @ git@lrhgit/tkt4140/src/src-ch2/ODEschemes.py;

from ODEschemes import euler, heun, rk4
from numpy import cos, sin
import numpy as np
from matplotlib.pyplot import *

# change some default values to make plots more readable
LNWDT=3; FNT=11
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT



N=20
L = 1.0
y = np.linspace(0,L,N+1)

def dsfunction(phi0,phi1,s0,s1):
    if (abs(phi1-phi0)>0.0):
        return    -phi1 *(s1 - s0)/float(phi1 - phi0)
    else:
        return 0.0


def f(z, t):
    """RHS for deflection of beam"""
    zout = np.zeros_like(z)
    zout[:] = [z[1],-alpha2*cos(z[0]),sin(z[0])]
    return zout

alpha2 = 5.0
beta=0.0 # Boundary value at y = L

solvers = [euler, heun, rk4] #list of solvers
solver=solvers[2] # select specific solver

# Guessed values
s=[2.5, 5.0]

z0=np.zeros(3)

z0[1] = s[0]
z = solver(f,z0,y)
phi0 = z[-1,1] - beta

nmax=10
eps = 1.0e-10
for n in range(nmax):
    z0[1] = s[1]
    z = solver(f,z0,y)
    phi1 = z[-1,1] - beta
    ds = dsfunction(phi0,phi1,s[0],s[1])
    s[0]  = s[1]
    s[1] +=  ds
    phi0 = phi1
```

```
    print 'n = {}  s1 = {} and ds = {}'.format(n,s[1],ds)

    if (abs(ds)<=eps):
        print 'Solution converged for eps = {} and s1 ={} and ds = {}. \n'.format(eps,s[1],ds)
        break

legends=[] # empty list to append legends as plots are generated

plot(y,z[:,0])
legends.append(r'$\theta$')

plot(y,z[:,1])
legends.append(r'$d\theta/dl$')

plot(y,z[:,2])
legends.append(r'$y$')


# Add the labels
legend(legends,loc='best',frameon=False) # Add the legends
ylabel(r'$\theta, d\theta/dl, y$')
xlabel('y/L')
#grid(b=True, which='both', axis='both',linestyle='-')
grid(b=True, which='both', color='0.65',linestyle='-')

show()
```

And the resulting output of solutions is illustrated in Figure 3.12

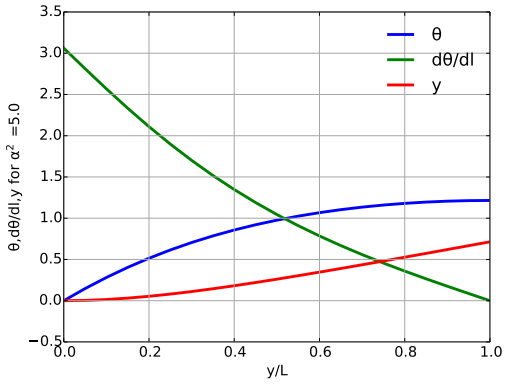

Figure 3.12: Solutions generated with a shooting method for large deflections of a cantilever subjected to a point load.

3.3 Notes on similarity solutions

The transient one dimensional heat equation may be represented:

$$\frac{\partial T}{\partial \tau} = \alpha \frac{\partial^2 T}{\partial X^2} \tag{3.80}$$

where τ and X denote time and spatial coordinates, respectively. The temperature T is a function of time and space $T = T(X, \tau)$, and α is the thermal diffusivity. (See appendix B in Numeriske Beregninger for a derivation)

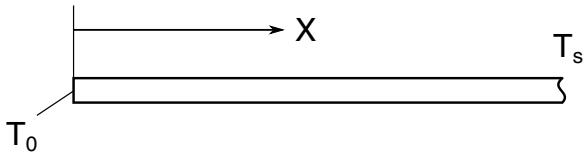

Figure 3.13: Beam in the right half-space in one-dimension.

In Figure 3.13 a one-dimensional beam in the right half-space ($0 \leq X < \infty$) is illustrated. The beam has initially a temperature T_s, but at time $\tau = 0$, the temperature at the left end $X = 0$ is abruptly set to T_0, and kept constant thereafter.

We wish to compute the temperature distribution in the beam as a function of time τ. The partial differential equation describing this problem is given by equation (3.80), and to model the time evolution of the temperature we provide the following initial condition:

$$T(X, \tau) = T_s, \ \tau < 0 \tag{3.81}$$

along with the boundary conditions which do not change in time:

$$T(0, \tau) = T_0, \ T(\infty, \tau) = T_s \tag{3.82}$$

Before we solve the problem numerically, we scale equation (3.80) by the introduction of the following dimensionless variables:

$$u = \frac{T - T_0}{T_s - T_0}, \qquad x = \frac{X}{L}, \qquad t = \frac{\tau \cdot \alpha}{L^2} \tag{3.83}$$

where L is a characteristic length. By substitution of the dimensonless variables in equation (3.83) in (3.80), we get the following:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}, \qquad 0 < x < \infty \tag{3.84}$$

accompanied by the dimensionless initial condition:

$$u(x,t) = 1, \quad t < 0 \tag{3.85}$$

and dimensionless boundary conditions:

$$u(0,t) = 0, \qquad u(\infty,t) = 1 \tag{3.86}$$

The particular choice of the time scale in t (3.83) has been made to make the thermal diffusivity vanish and to present the governing partial on a canonical form (3.84) which has many analytical solutions and a wide range of applications. The dimensionless time in (3.83) is a dimensionless number, which is commonly referred to as the Fourier-number.

We will now try to transform the partial differential equation (3.84) with boundary conditions (3.85) to a simpler ordinary differential equations. We will do so by introducing some appropriate scales for the time and space coordinates:

$$\bar{x} = a\,x \qquad \text{and} \qquad \bar{t} = b\,t \tag{3.87}$$

where a and b are some positive constants. Substitution of equation (3.87) into equation (3.84) yields the following equation:

$$\frac{\partial u}{\partial \bar{t}} = \frac{a^2}{b}\frac{\partial^2 u}{\partial \bar{x}^2} \tag{3.88}$$

We chose $b = a^2$ to bring the scaled equation (3.88) on the canonical, dimensionless form of equation (3.84) with the boundary conditions:

$$u(x,t) = u(\bar{x},\bar{t}) = u(ax, a^2 t), \qquad \text{with} \quad b = a^2 \tag{3.89}$$

For (3.89) to be independent of $a > 0$, the solution $u(x,t)$ has to be on the form:

$$u(x,t) = f\left(\frac{x}{\sqrt{t}}\right),\ g\left(\frac{x^2}{t}\right),\ \text{etc.} \tag{3.90}$$

and for convenience we choose the first alternative:

$$u(x,t) = f\left(\frac{x}{\sqrt{t}}\right) = f\left(\eta\right) \tag{3.91}$$

where we have introduced a new similarity variable η defined as:

$$\eta = \frac{x}{2\sqrt{t}} \tag{3.92}$$

and the factor 2 has been introduced to obtain a simpler end result only. By introducing the similariy variable η in equation (3.91), we transfor the solution (and the differential equation) from being a function of x and t, to only depend on one variable, namely η. A consequence of this transformation is that one profile $u(\eta)$, will define the solution for all x and t, i.e. the solutions will be similar and is denoted a similarity solution for that reason and (3.92) a similarity transformation.

The Transformation in equation (3.92) is often referred to as the Boltzmann-transformation.

The original PDE in equation (3.84) has been transformed to an ODE, which will become clearer from the following. We have introduced the following variables:

$$t = \frac{\tau \cdot \alpha}{L^2}, \qquad \eta = \frac{x}{2\sqrt{t}} = \frac{X}{2\sqrt{\tau\alpha}} \tag{3.93}$$

Let us now solve equation (3.84) analytically and introduce:

$$u = f(\eta) \tag{3.94}$$

with the boundary conditions:

$$f(0) = 0, \; f(\infty) = 1 \tag{3.95}$$

Based on the mathematical representation of the solution in equation (3.94) we may express the partial derivatives occuring in equation (3.84) as:

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial \eta}\left(\frac{\partial n}{\partial t}\right) = f'(\eta) \cdot \left(-\frac{x}{4t\sqrt{t}}\right) = -f'(\eta)\frac{\eta}{2t}$$
$$\frac{\partial u}{\partial x} = \frac{\partial u}{\partial \eta}\left(\frac{\partial \eta}{\partial x}\right) = f'(\eta)\frac{1}{2\sqrt{t}}, \qquad \frac{\partial^2 u}{\partial x^2} = \frac{\partial}{\partial x}\left(f'(\eta)\frac{1}{2\sqrt{t}}\right) = f''(\eta)\frac{1}{4t}$$

By substituting the expressions for $\dfrac{\partial u}{\partial t}$ and $\dfrac{\partial^2 u}{\partial x^2}$ in equation (3.84) we transform the original PDE in equation (3.84) to an ODE in equation (3.96) as stated above:

$$f''(\eta)\frac{1}{4t} + f'(\eta)\frac{\eta}{2t} = 0$$

or equivalently:

$$f''(\eta) + 2\eta f'(\eta) = 0 \tag{3.96}$$

The ODE in equation (3.96) may be solved directely by integration. First, we rewrite equation (3.96) as:

$$\frac{f''(\eta)}{f'(\eta)} = -2\eta$$

which may be integrated to yield:

$$\ln f'(\eta) = -\eta^2 + \ln C_1 \tag{3.97}$$

which may be simplified by and exponential transformation on both sides:

$$f'(\eta) = C_1 e^{-\eta^2}$$

and integrated once again to yield:

$$f(\eta) = C_1 \int_0^{\eta} e^{-t^2}\, dt \tag{3.98}$$

where we have used the boundary condition $f(0) = 0$ from (3.95).

The integral in (3.98) is related with the error function:

$$\int_0^x e^{-t^2} dt = \frac{\sqrt{\pi}}{2}\text{erf(x)}$$

where the error function $erf(x)$ is defined as:

$$\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \tag{3.99}$$

Substitution of equation (3.99) in equation (3.98) yields:

$$f(\eta) = C_1 \frac{\sqrt{\pi}}{2}\text{erf}(\eta) \tag{3.100}$$

As the error function has the property $\text{erf}(\eta) \to 1$ for $\eta \to \infty$ we get: $C_1 = \dfrac{2}{\sqrt{\pi}}$, and subsequent substitution of equation (3.100) yields:

A similarity solution for the one-dimensional heat equation

$$u(x,t) = \text{erf}(\eta) = \text{erf}\left(\frac{x}{2\sqrt{t}}\right) \tag{3.101}$$

If we wish to express $u(x,t)$ by means of the original variables we have from equation (3.83):

$$\frac{T(X,\tau) - T_0}{T_s - T_0} = \text{erf}\left(\frac{X}{2\sqrt{\tau \cdot \alpha}}\right) \tag{3.102}$$

In Figure 3.14 the error function $\text{erf}(\eta)$, which is a solution of the one-dimensional heat equation (3.84), is plotted along with the complementary error function $\text{erfc}(\eta)$ defined as:

$$\text{erfc}(\eta) = 1 - \text{erf}(\eta) = \frac{2}{\sqrt{\pi}} \int_{\eta}^{\infty} e^{-t^2}\, dt \tag{3.103}$$

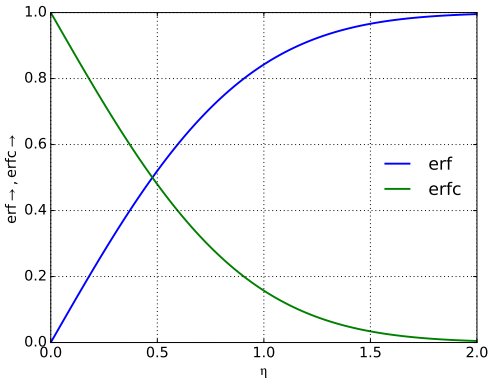

Figure 3.14: The error function is a similarity solution of the one-dimensional heat equation expressed by the similarity variable η.

3.3.1 Example: Freezing of a waterpipe

A dry layer of soil at initial temperature $20°C$, is in a cold period exposed to a suface temperature of $-15°C$ in 30 days and nigths. The question of concern is how deep the waterpipe must be located in order to avoid that the water in the pipe starts freezing?

The thermal diffusivity is $\alpha = 5 \cdot 10^{-4}\ m^2/hour$. With reference to (3.81) we have $T_0 = -15°C$ and $T_s = 20°C$, and $\tau = 30$ days and nigths $= 720$ hours, and water freezes at $0°C$.

From (3.83):

$$u = \frac{T - T_0}{T_s - T_0} = \frac{0 - (-15)}{20 - (-15)} = \frac{3}{7} = 0.4286$$

Some values for erf(η) are tabulated below :

x	erf(η)	η	erf(η)
0.00	0.00000	1.00	0.84270
0.20	0.22270	1.20	0.91031
0.40	0.42839	1.40	0.95229
0.60	0.60386	1.60	0.97635
0.80	0.74210	1.80	0.98909

From the tabulated values for $erf(x)$ and (3.101) we find erf(η) $= 0.4286 \rightarrow$ $\eta \approx 0.4$ which according to (3.93) yields:

$$X = 0.4 \cdot 2\sqrt{\tau \cdot \alpha} = 0.8 \cdot \sqrt{720 \cdot 5 \cdot 10^{-4}} = 0.48\text{m}$$

While we have used a constant value for the diffusivity α, in can vary in the range $\alpha \in [3\,10^{-4} \ldots 10^{-3}]\ m^2/s$, and the soil will normaly contain some humidity.

3.3.2 Example: Stokes' first problem: flow over a suddenly started plate

Analytical solutions of the Navier-Stokes equations may be found only for situations for which some simplifying assumptions are made for the flow field, geometry etc. Several solutions are known for laminar flow due to moving boundaries and these are often used to illustrate the viscous boundary layer behaviour of such flow regimes.

In this example we illustrate Stokes' first problem flow over a suddenly started plate. Consider a quiescent fluid at time $\tau < 0$ resting on a plate parallel to the X-axis (see 3.15).

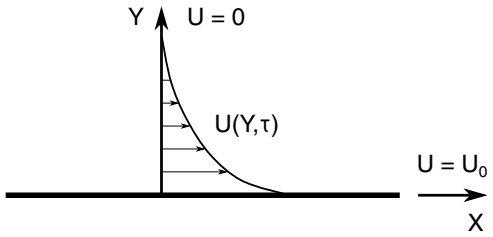

Figure 3.15: Stokes' first problem: flow over a suddenly started plate.

At time $\tau = 0$ the plate is accelerated up to a constant velocity $U = U_0$, which allows for a parallel-flow assumption of $V = 0$, $W = 0$, for the velocity components orthogonal to U.

$$\frac{\partial U}{\partial \tau} + U\frac{\partial U}{\partial X} + V\frac{\partial U}{\partial Y} = -\frac{1}{\rho}\frac{\partial p}{\partial X} + \nu\left(\frac{\partial^2 U}{\partial X^2} + \frac{\partial^2 U}{\partial Y^2}\right) \tag{3.104}$$

A more detailed derivation is provided in appendix B in Numeriske Beregninger). Sufficiently far downstream one may further assume that the flow field is independent of the streamwise coordinate X too, ie. $U = U(Y, \tau)$, and the (3.104) reduces to:

$$\frac{\partial U}{\partial \tau} = \nu\frac{\partial^2 U}{\partial Y^2},\ 0 < Y < \infty \tag{3.105}$$

The assumption of a suddenly started plate implies the initial condition:

$$U(U, 0) = 0 \tag{3.106}$$

and the assumptions of no slip at the plate and a quiescent fluid far away from the wall implies the following boundary conditions:

$$U(0, \tau) = U_0 \quad U(\infty, \tau) = 0 \tag{3.107}$$

To present the problem given by (3.105), (3.106), and (3.107) in an even more convenient manner we introduce the following dimensionless variables:

$$u = \frac{U}{U_0},\ y = \frac{Y}{L},\ t = \frac{\tau \cdot \nu}{L^2} \tag{3.108}$$

where U_0, L are characteristic velocity and length, respectively. Further, substitution of (3.108) in (3.105) yields the following equation:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial y^2},\ 0 < y < \infty \tag{3.109}$$

with corresponding initial condition:

$$u(y, 0) = 0 \tag{3.110}$$

and boundary conditions

$$\begin{aligned} u(0, t) &= 1 \\ u(\infty, t) &= 0 \end{aligned} \tag{3.111}$$

We now realize that the Stokes' first problems is the same as the on given by (3.84) and (3.85), save for a substitution of 0 and 1 in the boundary conditions. . The solution of (3.109) and (3.110) and the appropriate boundary conditions (3.111) is:

$$u(y, t) = 1 - \text{erf}(\eta) = \text{erfc}(\eta) = \text{erfc}\left(\frac{y}{2\sqrt{t}}\right) \tag{3.112}$$

which has the following expression in physical variables:

$$U(Y, \tau) = U_0 \text{erfc}\left(\frac{Y}{2\sqrt{\tau \cdot \nu}}\right) \tag{3.113}$$

3.3.3 Example: The Blasius equation

The Blasius equation is a nonlinear ODE which may be derived as a simplification of the Navier-Stokes equations for the particular case of stationary, incompressible boundary layer flows.

Detailed derivations of the Blasius equation may be found in numerous text in fluid mechanics concerned with boundary layer flow (appendix C, section C.2 in Numeriske Beregninger). However, for the sake of completeness we provide a brief derivation of the Blasius equation in the follwing. A stationary, incompressible boundary layer flow is governed by a simplifed version of the Navier-Stokes equations (see equation (C.1.10), appendix C in Numeriske Beregninger).

For this particular flow regime, conservation of mass corresponds to:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{3.114}$$

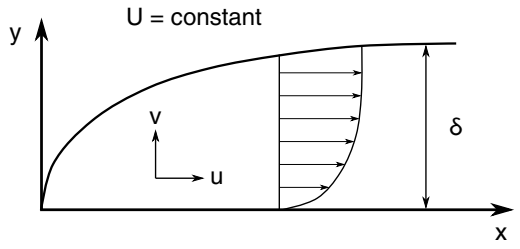

Figure 3.16: Boundary layer development over a horizontal plate. fig:216

whereas balance of linear momentum is ensured by:

$$u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{1}{\rho}\frac{dp}{dx} + \nu\frac{\partial^2 u}{\partial y^2} \tag{3.115}$$

For the case of a constant free stream velocity $U = U_0$ we may show that the streamwise pressure gradient $\frac{dp}{dx} = 0$. (see equation (C.1.8) in appendix C in Numeriske Beregninger), and the equation eq:23028) reduces to:

$$u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = \nu\frac{\partial^2 u}{\partial y^2} \tag{3.116}$$

with corresponding boundary conditions:

$$u(0) = 0 \tag{3.117}$$

$$u \to U_0 \text{ for } y \to \delta \tag{3.118}$$

In two dimensions, the mass conservation (3.114) may conveniently be satisfied by the introduction of a stream function $\psi(x, y)$, defined as:

$$u = \frac{\partial \psi}{\partial y},\ v = -\frac{\partial \psi}{\partial x} \tag{3.119}$$

The introduction of a similarity variable η and the non-dimensional stream function f

$$\eta = \sqrt{\frac{U_0}{2\nu x}} \cdot y \qquad \text{og} \qquad f(\eta) = \frac{\psi}{\sqrt{2U_o\nu x}} \tag{3.120}$$

we may transform the PDE in equation (3.120) to an ODE, which is the famous Blasius equation, for f:

$$f'''(\eta) + f(\eta) \cdot f''(\eta) = 0 \tag{3.121}$$

where we use the conventional notation $\frac{df}{d\eta} \equiv f'(\eta)$ and similar for higher order derivatives.

The physical velocity components may be derived from the stream function by:

$$\frac{u}{U_0} = f'(\eta),\ \frac{v}{U_0} = \frac{\eta \cdot f'(\eta) - f(\eta)}{\sqrt{2Re_x}} \tag{3.122}$$

where the Reynolds number has been introduced as

$$Re_x = \frac{U_0 x}{\nu} \tag{3.123}$$

The no slip condition $u = 0$ for $y = 0$ will concequently correspond to $f'(0) = 0$ ad $f'(\eta) = \frac{u}{U_0}$ in equation (3.122).

In abscence of suction and blowing at the boundary for $\eta = 0$, the other no slip condition, i.e $v = 0$ for $\eta = 0$, correponds to $f(0) = 0$ from equation (3.122). Further, the condition for the free stream of $u \to U_0$ from equation (3.118), corresponds to $f'(\eta) \to 1$ for $\eta \to \infty$.

Consequently, the boundary conditions for the boundary layer reduce to:

$$f(0) = f'(0) = 0,\ f'(\eta_\infty) = 1 \tag{3.124}$$

The shear stress at the wall may be computed from: Skjærspenningen:

$$\tau_{xy} = \mu U_0\, f''(\eta)\sqrt{\frac{U_0}{2\nu x}} \tag{3.125}$$

Numerical solution

The ODE in equation (3.121) with the boundary conditions in (3.124) represent a boundary value proble which we intend to solve with a shooting technique. We start by writing the third order, nonlinear ODE as a set of three, first order ODEs:

$$\begin{aligned} f_0' &= f_1 \\ f_1' &= f_2 \\ f_2' &= -f_0\, f_2 \end{aligned} \tag{3.126}$$

where $f_0 = f$ and the corresponding boundary values are represented as

$$\begin{aligned} f_0(0) = f_1(0) &= 0 \\ f_1(\eta_\infty) &= 1 \end{aligned} \tag{3.127}$$

As $f''(0)$ is unknown we must find the correct value with the shooting method, $f''(0) = f_2(0) = s$. Note that s, which for the Blasius equaiton corresponds to the wall shear stress, must be chosen such that the boundary condition $f_1(\eta_\infty) = 1$ is satisfied. We formulate this condition mathematically by introducing the boundary value error function ϕ:

$$\phi(s) = f_1(\eta_\infty; s) - 1 \tag{3.128}$$

Vi velger å bruke sekantmetoden til nullpunktsbestemmelsen i

As the system of ODEs in equation (3.126) is nonlinear, we follow in the procedure as outlined in 3.2, and use the secant method to find the correct initial value $s = s^*$ such that the boundary value error function is zero $\phi(s^*) = 0$.

Iteration proceduce

The boundary value error function ϕ in (3.128) is a nonlinear function of s (due to the nonlinear ODE it is derived from), and therefore, we must itereate to find the correct value.

Two initial guesses s^0 and s^1 are needed to start the iteration process, where the superscript is an iteration counter. Consequently, s^m referst to to s-value after the m-th iteration. In accordance with the generic procedure in (3.64) we find it more convenient to introduce the Δs:

$$s^{m+1} = s^m + \Delta s, \qquad \Delta s = -\phi(s^m) \cdot \left[\frac{s^m - s^{m-1}}{\phi(s^m) - \phi(s^{m-1})} \right], \; m = 1, 2. \ldots \tag{3.129}$$

To formulate the iteration procedure we assume two values s^{m-1} s^m to be know, which intitially correspond to s^0 and s^1. The interation procedure the becomes:

1. Compute $\phi(s^{m-1})$ and $\phi(s^m)$ by solving (3.126).
2. Compute Δs og s^{m+1} from (3.129)
3. Update
   - $s^{m-1} \leftarrow s^m$
   - $s^m \leftarrow s^{m+1}$
   - $\phi(s^{m-1}) \leftarrow \phi(s^m)$
4. Repeat 1-3 until convergence

For this particular problem, the Blasisus equation, the correct value is $s^* = 0.46960....$

Note that the convergence criteria is $\Delta s < \varepsilon$, i.e. a test on the absolute value of Δs, whereas a relative metric would be $\left| \frac{\Delta s}{s} \right|$.

For some problems it may be difficult to guess appropriate initial guesses s^0 og s^1 to start the iteration procedure.

A very simple procedure to get an overview of the zeros for a generic function is to plot it graphically. In Figure 3.17 we have plotted $\phi(s)$ for a very wide range $s \in [0.05, 5.0]$.

The computations of ϕ were performeed with the code `phi_plot_blasius_shoot_v2.py`, and by visual inspection we observe that the zero is in the range $[0.45, 0.5]$, i.e. much more narrow that our initial guess.

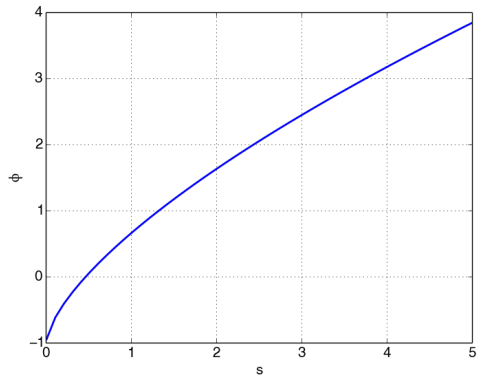

Figure 3.17: Exploration of the error function ϕ as a function of s for the Blasius equation.

```
# src-ch2/phi_plot_blasius_shoot_v2.py;ODEschemes.py @ git@lrhgit/tkt4140/src/src-ch2/ODEschemes.py;


from ODEschemes import euler, heun, rk4
from matplotlib.pyplot import *
# Change some default values to make plots more readable on the screen
LNWDT=3; FNT=20
matplotlib.rcParams['lines.linewidth'] = LNWDT; matplotlib.rcParams['font.size'] = FNT

def fblasius(y, x):
    """ODE-system for the Blasius-equation"""
    return [y[1],y[2], -y[0]*y[2]]

solvers = [euler, heun, rk4] #list of solvers
solver=solvers[2] # select specific solver


from numpy import linspace, exp, abs
xmin = 0
xmax = 5.75

N = 50  # no x-values
x = linspace(xmin, xmax, N+1)

# Guessed values
#s=[0.1,0.8]
s_guesses=np.linspace(0.01,5.0)

z0=np.zeros(3)

beta=1.0 #Boundary value for eta=infty

phi = []
for s_guess in s_guesses:
    z0[2] = s_guess
    u = solver(fblasius, z0, x)
    phi.append(u[-1,1] - beta)
```

```
plot(s_guesses,phi)

title('Phi-function for the Blasius equation')
ylabel('phi')
xlabel('s')
grid(b=True, which='both')
show()
```

Once approriate values for s^0 and s^1 have been found we are ready to find the solution of the Blasius equation with the shooting method as outlined in `blasius_shoot_v2.py`.

```
# src-ch2/blasius_shoot_v2.py;ODEschemes.py @ git@lrhgit/tkt4140/src/src-ch2/ODEschemes.py;

from ODEschemes import euler, heun, rk4
from matplotlib.pyplot import *
# Change some default values to make plots more readable on the screen
LNWDT=3; FNT=20
matplotlib.rcParams['lines.linewidth'] = LNWDT; matplotlib.rcParams['font.size'] = FNT

def fblasius(y, x):
    """ODE-system for the Blasius-equation"""
    return [y[1],y[2], -y[0]*y[2]]


def dsfunction(phi0,phi1,s0,s1):
    if (abs(phi1-phi0)>0.0):
        return    -phi1 *(s1 - s0)/float(phi1 - phi0)
    else:
        return 0.0

solvers = [euler, heun, rk4] #list of solvers
solver=solvers[0] # select specific solver


from numpy import linspace, exp, abs
xmin = 0
xmax = 5.750

N = 400  # no x-values
x = linspace(xmin, xmax, N+1)

# Guessed values
s=[0.1,0.8]


z0=np.zeros(3)
z0[2] = s[0]


beta=1.0 #Boundary value for eta=infty

## Compute phi0

u = solver(fblasius, z0, x)
phi0 = u[-1,1] - beta

nmax=10
eps = 1.0e-3
```

```
for n in range(nmax):
    z0[2] = s[1]
    u = solver(fblasius, z0, x)
    phi1 = u[-1,1] - beta
    ds = dsfunction(phi0,phi1,s[0],s[1])
    s[0]  = s[1]
    s[1]  += ds
    phi0 = phi1
    print 'n = {}  s1 = {} and ds = {}'.format(n,s[1],ds)

    if (abs(ds)<=eps):
        print 'Solution converged for eps = {} and s1 ={} and ds = {}. \n'.format(eps,s[1],ds)
        break


plot(u[:,1],x,u[:,2],x)
xlabel('u og u\'')
ylabel('eta')


legends=[]
legends.append('velocity')
legends.append('wall shear stress')
legend(legends,loc='best',frameon=False)
title('Solution of the Blaisus eqn with '+str(solver.func_name)+'-shoot')
show()
close() #Close the window opened by show()
```

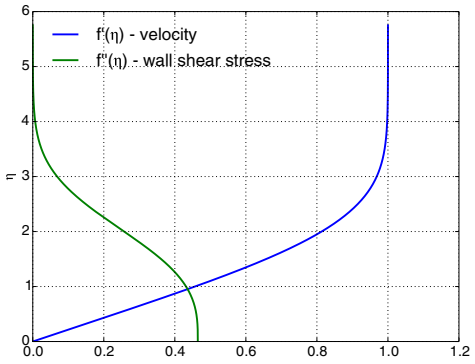

Figure 3.18: Solutions of the Blasius equation: dimensionles velocity $f'(\eta)$ and shear stress $f''(\eta)$.

In Figure 3.18 the dimensionless velocity $f'(\eta)$ and shear stress $f''(\eta)$ are plotted. The maxial $f''(\eta)$ is inntreffer for $\eta = 0$, which means that the maximal shear stress is at the wall.

Notice further that $f'''(0) = 0$ means $f''(\eta)$ has a vertical tanget at the wall.

3.4 Shooting method for linear ODEs with two unknown initial conditions

In the follwing we will extend the shoothing methods we we derived in (3.1) to problems with two initial conditions.

For such problemsn we must guess to initial values r and s with the corresponding boundary value error functions ϕ og ψ, respectively.

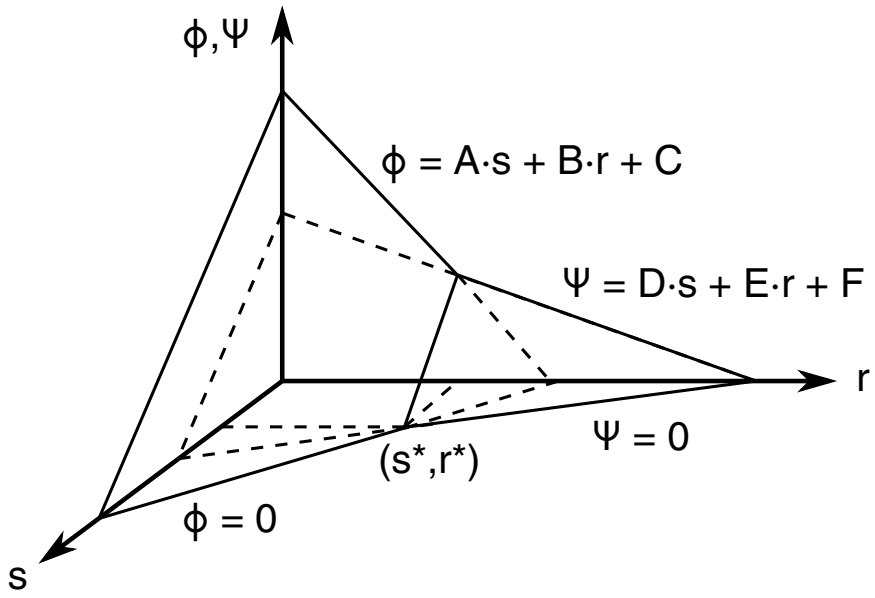

Figure 3.19: The boundary value error functions for boundary value problems with two unknown initial conditions.

As the ODE to be solved is linear, we know that the resulting boundary value error functions also will be linear. As ϕ and ψ will be functions of the two initial guesses r and s, the will express two planes which may be represented:

$$\begin{aligned} \phi &= A \cdot s + B \cdot r + C \\ \psi &= D \cdot s + E \cdot r + F \end{aligned} \tag{3.130}$$

where $A, B, C, \cdots, F$ are constants to be determined. The correct values r^* og s^* satisfy $\phi(r^*, s^*) = 0$ and $\psi(r^*, s^*) = 0$. (See Figure 3.19.)

Consequently we have six constants which may be found from the equations; three equations for ϕ:

$$\begin{aligned}\phi^0 &= A \cdot s^0 + B \cdot r^0 + C \\ \phi^1 &= A \cdot s^1 + B \cdot r^1 + C \\ \phi^{(2)} &= A \cdot s^{(2)} + B \cdot r^{(2)} + C\end{aligned} \tag{3.131}$$

and three equations for ψ:

$$\begin{aligned}\psi^0 &= D \cdot s^0 + E \cdot r^0 + F \\ \psi^1 &= D \cdot s^1 + E \cdot r^1 + F \\ \psi^{(2)} &= D \cdot s^{(2)} + E \cdot r^{(2)} + F\end{aligned} \tag{3.132}$$

where $s^0,\ s^1,\ s^{(2)},\ r^0,\ r^1$ og $r^{(2)}$ are initial guesses for the unknown initial values. For convenience we choose the following pairs of initial guesses:

$$\begin{aligned}s^0 &= 0, \qquad & r^0 &= 0 \\ s^1 &= 0, \qquad & r^1 &= 1 \\ s^{(2)} &= 1, \qquad & r^{(2)} &= 0\end{aligned} \tag{3.133}$$

which results in the following expressions for the constants:

$$\begin{aligned}A = \phi^{(2)} - \phi^0, \quad B = \phi^1 - \phi^0, \quad C = \phi^0, \\ D = \psi^{(2)} - \psi^0, \quad E = \psi^1 - \psi^0, \quad F = \psi^0\end{aligned} \tag{3.134}$$

The correct values r^* og s^* which satisfies $\phi(r^*,\ s^*) = 0$ and $\psi(r^*,\ s^*) = 0$, i.e. $A \cdot s^* + B \cdot r^* + C = D \cdot s^* + E \cdot r^* + C = 0$ may be expressed by the coefficients:

$$s^* = \frac{E \cdot C - B \cdot F}{D \cdot B - A \cdot E}, \qquad r^* = \frac{A \cdot F - D \cdot C}{D \cdot B - A \cdot E} \tag{3.135}$$

which after substitution of $A, B, C, \ldots, F$ results in:

$$\begin{aligned}s^* &= \frac{\psi^1 \cdot \phi^0 - \phi^1 \cdot \psi^0}{(\psi^{(2)} - \psi^0) \cdot (\phi^1 - \phi^0) - (\phi^{(2)} - \phi^0) \cdot (\psi^1 - \psi^0)} \\ r^* &= \frac{\phi^{(2)} \cdot \psi^0 - \psi^{(2)} \cdot \phi^0}{(\psi^{(2)} - \psi^0) \cdot (\phi^1 - \phi^0) - (\phi^{(2)} - \phi^0) \cdot (\psi^1 - \psi^0)}\end{aligned} \tag{3.136}$$

The shooting method solution procedure for a linear ODE boundary value problem with two initial guesses then becomes:

Shooting method for two unknown initial values

1. Solve the ODE for the three pairs of initial guesses r og s as given in (3.133)
2. Find the correspondning values of ϕ and ψ.
3. Compute the correct initial values r^* og s^* from equation (3.136)).

We will illustrate the usage of this procedure in example (3.4.1).

3.4.1 Example: Liquid in a cylindrical container

In this example we consider a cylindrical container filled with a liquid to the height H for two different geometries, namely constant container wall thickness and with a container with at wall thickness which varies linearly (See Figure 3.20). We will look at the two geometries separately below.

Constant wall thickness. We denote radial displacement with W. In the zoomed in part of Figure 3.20, V denotes shear force per unit length while M denotes moment per unit length.

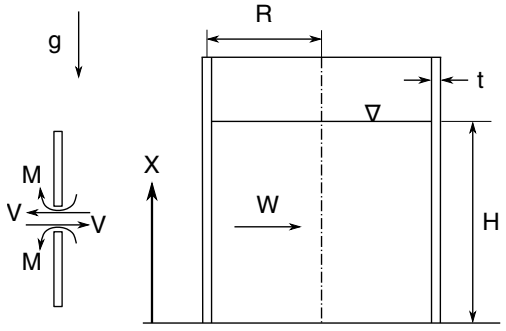

Figure 3.20: A cylindrical container with a liquid of heigth H. The arrows denote positive direction.

The differential equation for the displacement W is given by:

$$\frac{d^4W}{dX^4} + B \cdot W = -\gamma \frac{H - X}{D} \tag{3.137}$$

where:

$$B = \frac{12(1-\nu^2)}{R^2t^2}, \qquad D = \frac{Et^3}{12(1-\nu^2)}, \qquad \gamma = \rho g$$

and ν is Poisson's ratio, E the E-modulus and ρ the density of the fluid. For convenience we introduce dimensionless variables:

$$x = \frac{X}{H} \qquad \text{and} \qquad w = \frac{E}{\gamma H t}\left(\frac{t}{R}\right)^2 W \tag{3.138}$$

which after substitution in equation (3.137) results in the following dimensionless, fourth order ODE:

$$\frac{d^4w}{dx^4} + 4\,\beta^4\,w = -4\,\beta^4\,(1-x) \qquad \text{where} \qquad \beta^4 = \frac{3(1-\nu^2)H^4}{R^2t^2} \tag{3.139}$$

Boundary conditions

To solve the ODE in equation (3.139) we need to supply some boundary conditions at both ends. At $x = 0$:

$$W = 0, \qquad \text{and} \qquad \frac{dW}{dX} = 0 \text{ (fixed beam)}$$

At the other end, for $X = H$ both the moment and the shear force must be zero, which mathematically corresponds to:

$$M = -D\frac{d^2W}{dX^2} = 0,\ V = -D\frac{d^3W}{dX^3} = 0$$

The dimensionless expression for the moment is $m(x) = -\dfrac{d^2w}{dx^2}$, while the dimemsionless shear force has the expression: $v(x) = -\dfrac{d^3w}{dx^3}$.

In dimensionless variables the boundary conditions takes the form:

$$w = 0, \qquad \frac{dw}{dx} = 0 \qquad \text{for} \quad x = 0 \tag{3.140}$$

$$\frac{d^2w}{dx^2} = 0, \qquad \frac{d^3w}{dx^3} = 0 \qquad \text{for} \quad x = 1 \tag{3.141}$$

For our example we select a container with the following dimensions $R = 8.5m$, $H = 7.95m$, $t = 0.35m$, $\gamma = 9810N/m^3$ (water), $\nu = 0.2$, and the E-modulus $E = 2 \cdot 10^4$MPa. For these values the nondimensional parameter $\beta = 6.0044$.

Numerical solution with the shooting method for two unknown initial values

First we need to reformulate the fourth order ODE in (3.139) as a system of first order ODEs by following the procedure outlined in 2.4.

By using the following auxilliary variables $w = y_0,\ w' = y_0' = y_1,\ w'' = y_1' = y_2,\ w''' = y_2' = y_3$ we may write (3.137) as a system of four, first order ODEs:

$$\begin{aligned} y_0' &= y_1 \\ y_1' &= y_2 \\ y_2' &= y_3 \\ y_3' &= -4\beta^4(y_0 + 1 - x) \end{aligned} \tag{3.142}$$

with the following boundary conditions:

$$y_0(0) = 0,\ y_1(0) = 0,\ y_2(1) = 0,\ y_3(1) = 0 \tag{3.143}$$

We intend to solve the boundary value problem defined by equations (3.142) and (3.143) with a shooting method approach, i.e. we will use and intital value solver and guess the unknown initital values. The intention with the shooting method is to find the unknown initial values $s = w''(0) = y_2(0)$ and $r = w'''(0) = y_3(0)$ such that the boundary values $y_2(1) = 0$ and $y_3(1) = 0$.

We introduce the boundary value error function as $\phi(r, s) = y_2(1; r, s)$ and $\psi(r, s) = y_3(1; r, s)$ and the correct values for our intitial guesses r^* og s^* are found when $\phi(r^*, s^*) = 0$.

To find the correct initial guesses $s = w''(0) = y_2(0)$ and $r = w'''(0)$ we do the following:

- Solve (3.142) three times, always with $y_0(0)$ and $y_1(0) = 0$. The values of r and s for each solution is taken from (3.133).
- Compute the correct values for r^* and s^* from (3.136).

Below you find a python implementation of the procedure. Rune the code and check for yourself if the boundary conditions are satisfied.

```
# src-ch2/tank1.py;ODEschemes.py @ git@lrhgit/tkt4140/src/src-ch2/ODEschemes.py;

from ODEschemes import euler, heun, rk4
from numpy import cos, sin
import numpy as np
from matplotlib.pyplot import *

# change some default values to make plots more readable
LNWDT=3; FNT=11
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT
font = {'size' : 16}; rc('font', **font)


def tank1(y, x):
    """Differential equation for the displacement w in a cylindrical tank with constant wall-thickne
    Args:
        y(array): an array containg w and its derivatives up to third order.
        x(array): independent variable
    Returns:
        dydx(array): RHS of the system of first order differential equations
    """
    dydx = np.zeros_like(y)
```

```
    dydx[0] = y[1]
    dydx[1] = y[2]
    dydx[2] = y[3]
    dydx[3] = -4*beta4*(y[0]+1-x)

    return dydx

# === main program ===
R = 8.5 # Radius [m]
H = 7.95 # height [m]
t = 0.35 # thickness [m]
ny = 0.2 # poissons number
beta = H*(3*(1 - ny**2)/(R*t)**2)**0.25 # angle
beta4 = beta**4
N = 100
X = 1.0
x = np.linspace(0,X,N + 1)

solverList = [euler, heun, rk4]
solver = solverList[2]
#shoot:
s = np.array([0, 0, 1])
r = np.array([0, 1, 0])
phi = np.zeros(3)
psi = np.zeros(3)


# evaluate the boundary value error functions for the initial guesses in s and r
for k in range(3):
    y0 = np.array([0, 0, s[k], r[k]])
    y = solver(tank1, y0, x)
    phi[k] = y[-1, 2]
    psi[k]=y[-1, 3]

# calculate correct r and s
denominator = (psi[2] - psi[0])*(phi[1] - phi[0]) - (phi[2] - phi[0])*(psi[1] - psi[0])

rstar = (phi[2]*psi[0] - psi[2]*phi[0])/denominator
sstar = (psi[1]*phi[0] - phi[1]*psi[0])/denominator
print 'rstar', rstar, 'sstar', sstar

# compute the correct solution with the correct initial guesses
y0 = np.array([0, 0, sstar, rstar])
y = solver(tank1, y0, x)

legendList=[]

plot(x,-y[:,3]/beta**2)
plot(x,-y[:,2]/beta)
legendList.append(r'$v(x)/\beta^2$ ')
legendList.append(r'$m(x)/\beta$ ')
# Add the labels
legend(legendList,loc='best',frameon=False)
ylabel('v, m')
xlabel('x')
grid(b=True, which='both', color='0.65',linestyle='-')
#savefig('../figs/tank1.pdf')
show()
```

Note that the ODE in equation (3.139) may be classified as singular as the highest order derivative is multiplied with a potentially very small number $\varepsilon = \frac{1}{4\beta^4}$, when rearranged. We observe that $\varepsilon \to 0$ as $\beta \to \infty$ and that the nature of equation (3.139) will go from an ODE to an algebraic equation. The analytical solution may be shown to have terms of the kind $e^{\beta x}\sin\beta x$ og $e^{\beta x}\cos\beta x$ (see equation (D.1.6) and (D.1.8), part D.1, appendix D in Numeriske Beregninger.

Varying wall thickness. In the following we will modify the example above of liquid in a cylindrical container by allowing for a wall thickness which varies linearly from t_0 at the bottom to t_1 at the top.

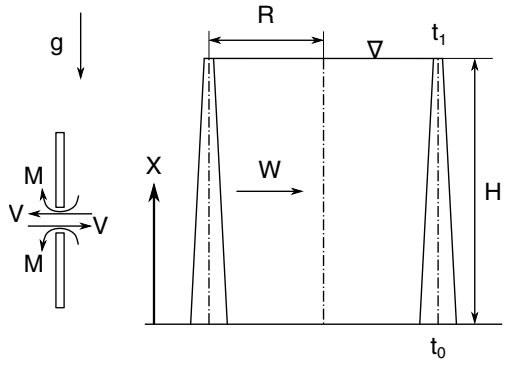

Figure 3.21: A cylindrical container with a varying wall thickenss containing a liquid of heigth H.

Based on t_0 and t_1, we introduce $\alpha = \dfrac{t_0 - t_1}{t_0}$ the steepness of the wall thickness, which allows for the thickness t to be represented as:

$$t = \left(1 - \alpha\frac{X}{H}\right)\cdot t_0,\ 0 \le X \le H \tag{3.144}$$

The differential equaiton for the displacement W for this situation is given by:

$$\frac{d^2}{dX^2}\left(D\frac{d^2W}{dX^2}\right) + E\frac{tW}{R^2} = -\gamma(H - X) \tag{3.145}$$

where $D = \dfrac{Et^3}{12(1-\nu^2)}$, $\gamma = \rho g$, with the same meaning as in the previous part of the example.

Dimensionless form

$$x = \frac{X}{H},\quad w = \frac{E}{\gamma H t_0}\left(\frac{t_0}{R}\right)^2 W,\quad \beta^4 = \frac{3(1-\nu^2)}{R^2 t_0^2}\,H^4 \tag{3.146}$$

By substitution of (3.146) in (3.145) we get the following nondimensional ODE:

$$\frac{d^2}{dx^2}\left[(1-\alpha x)^3\frac{d^2w}{dx^2}\right]+4\beta^4(1-\alpha x)\cdot w=-4\beta^4(1-x) \tag{3.147}$$

which may be differentiated to yield:

$$\frac{d^4w(x)}{dx^4}-\frac{6\alpha}{(1-\alpha x)}\frac{d^5w(x)}{dx^5}+\frac{6\alpha^2}{(1-\alpha x)^2}\frac{d^2w(x)}{dx^2}+\frac{4\beta^4}{(1-\alpha x)^2}w(x)=-\frac{4\beta^4(1-x)}{(1-\alpha x)^5} \tag{3.148}$$

This problem has also an analytical, but more complicated, solution (see appendix D, part D.2 in Numeriske Beregninger) expressed by means of Kelvin functions, a particular kind of Bessel functions. However, the numerical solution remains the same as for the case with the constant wall thickness.

We choose physical parameters as for the previous example:

$$\begin{gathered}R=8.5m,\ H=7.95m,\ t_0=0.35m,\ t_1=0.1m\\ \gamma=9810N/m^3(\text{vann}),\ \nu=0.2,\ E=2\cdot10^4\text{MPa}\end{gathered} \tag{3.149}$$

and $\alpha=\frac{t_0-t_1}{t_0}=\frac{5}{7}$ and as previously we get $\beta=6.0044$.

Numerical solution

We proceed as before with $w=y_0,\ w'=y_0'=y_1,\ w''=y_1'=y_2,\ w'''=y_2'=y_3,\ z=1-\alpha x$ and write (3.144) as a system of four first order ODEs:

$$\begin{aligned}y_0'&=y_1\\ y_1'&=y_2\\ y_2'&=y_3\\ y_3'&=\frac{6\alpha}{z}\,y_3-\frac{6\alpha^2}{z^2}\,y_2-\frac{4\beta^4}{z^2}\,y_0-\frac{4\beta^4(1-x)}{z^3}\end{aligned} \tag{3.150}$$

with the following boundary conditions:

$$y_0(0)=0,\ y_1(0)=0,\ y_2(1)=0,\ y_3(1)=0 \tag{3.151}$$

Only the last equation in the system of ODEs (3.150) differs from the previous part of the example 3.4.1).

The procedure will be as for 3.4.1, except for the we now are introduced the additional parameter α.

```
# src-ch2/tank2.py;ODEschemes.py @ git@lrhgit/tkt4140/src/src-ch2/ODEschemes.py;

from ODEschemes import euler, heun, rk4
from numpy import cos, sin
import numpy as np
from matplotlib.pyplot import *

# change some default values to make plots more readable
```

```
LNWDT=3; FNT=11
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT
font = {'size' : 16}; rc('font', **font)


def tank2(y, x):
    """Differential equation for the displacement w in a cylindrical tank with linearly varying wall
    Args:
        y(array): an array containg w and its derivatives up to third order.
        x(array): independent variable
    Returns:
        dydx(array): RHS of the system of first order differential equations
    """
    z = 1-alpha*x
    dydx = np.zeros_like(y)
    dydx[0] = y[1]
    dydx[1] = y[2]
    dydx[2] = y[3]
    temp = (6*alpha/z)*y[3]- (6*alpha**2/z**2)*y[2]
    dydx[3] = temp - 4*beta4*y[0]/z**2 - 4*beta4*(1-x)/z**3

    return dydx

R = 8.5 # radius [m]
H = 7.95 # height [m]
t0 = 0.35 # thickness [m]
t1 = 0.1 # thickness [m]
ny = 0.2 # poissons number
beta = H*(3*(1-ny**2)/(R*t0)**2)**0.25
beta4 = beta**4
alpha = (t0-t1)/t0
N = 100
print "beta: ", beta, "alpha", alpha

X = 1.0
x = np.linspace(0,X,N + 1)

solverList = [euler, heun, rk4] #list of solvers
solver = solverList[2] # select specific solver

# shoot:
s = np.array([0, 0, 1])
r = np.array([0, 1, 0])
phi = np.zeros(3)
psi = np.zeros(3)
for k in range(3):
    y0 = np.array([0,0,s[k],r[k]])
    y = solver(tank2,y0,x)
    phi[k] = y[-1, 2]
    psi[k]=y[-1, 3]

# calculate correct r and s
denominator = (psi[2] - psi[0])*(phi[1] - phi[0]) - (phi[2] - phi[0])*(psi[1] - psi[0])
rstar = (phi[2]*psi[0] - psi[2]*phi[0])/denominator
sstar = (psi[1]*phi[0] - phi[1]*psi[0])/denominator

print 'rstar', rstar, 'sstar', sstar

# compute the correct solution with the correct initial guesses
y0 = np.array([0, 0, sstar, rstar])
y = solver(tank2,y0,x)
```

```
legends=[] # empty list to append legends as plots are generated
plot(x,-y[:,3]/beta**2)
plot(x,-y[:,2]/beta)
legends.append(r'$v(x)/\beta^2$')
legends.append(r'$m(x)/\beta$')

# Add the labels
legend(legends,loc='best',frameon=False) # Add the legends
ylabel('v, m')
xlabel('x')
grid(b=True, which='both', color='0.65',linestyle='-')
show()
```

3.5 Exercises

Exercise 5: Stokes first problem for a non-Newtonian fluid

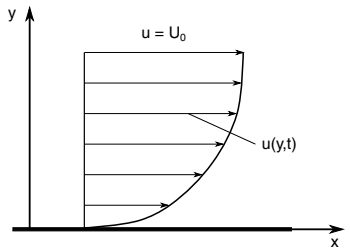

Figure 3.22: Stokes first problem for a non-Newtonian fluid: schematic representation of the velocity profile.

A non-Newtonian fluid flows in the plane with constant velocity U_0, paralell with the x-axis. At time $t = 0$ we insert a thin plate into the flow, paralell to the x-axis. See Figure 3.22 (The plate must be concidered as infinitesimal thin, with infinite extent in th x-directions.) We are interested in finding the velocity field $u(y, t)$ which developes because of the adherence between the fluid and the plate. The equation of motion for the problem may be stated as:

$$\rho \frac{\partial u}{\partial t} = \frac{\partial \tau_{xy}}{\partial y} \tag{3.152}$$

The relation between the shear stress τ_{xy} and the velocitygradient $\frac{\partial u}{\partial y}$ is given by:

$$\tau_{xy} = K \cdot \left| \frac{\partial u}{\partial y} \right|^{\frac{1}{2}} \frac{\partial u}{\partial y} \tag{3.153}$$

where K is a positive constant.

We introduce the following transformation which reduces the system to an ordinary differential equation:

$$\eta = C \cdot \frac{y}{t^{\frac{2}{5}}}, \quad f(\eta) = \frac{u}{U_0} \tag{3.154}$$

where C is a positive constant, $y \geq 0$ and $t \geq 0$. By inserting Eq. (3.153) into Eq. (3.152) and using (3.154), we get:

$$f''(\eta) + 2\,\eta\sqrt{f'(\eta)} = 0 \tag{3.155}$$

With boundaryconditions:

$$f(0) = 0, f(\delta) = 1 \tag{3.156}$$

where δ is the boundary-layer thickness.

Movie 2: `mov-ch2/stokes.mp4`

Pen and paper: The following problems should be done using pen and paper:

a) We are first gonna solve Eq. (3.155) as an initial-value problem. In this problem we thus set $f'(0) = 1.25$. Calculate f, and f' for $\eta = 0.6$ using Euler's method. Use $\Delta\eta = 0.2$

b) Write Eq (3.155) as a system of equations. Explain how you could solve the problem using shooting technique, when $f(0) = 0$, $f(\delta) = 1$, and δ is considered known.

c) During the solution of the system above we find that $f'(\delta) = 0$. This information makes it possible to determine δ as a part of the solution of the system in **b**). Explain how.

d) Solve Eq. (3.155) when $f_1(0) = f'(0)$ is given. Show that the boundary condition $f(\delta) = 1$ give $f_1(0) = \frac{225}{128} = 1.25311\ldots$

Programming:

Write a python program that solves the problem defined in **b**. Use the information about δ obtained in **c**.

Hint 1. Pen and paper:

a) Solutions: $f(0.6) = 0.732$, $f'(0.6) = 0.988$.

c) Perform the differentiation $\frac{d}{d\eta}\left[\sqrt{f'(\eta)}\right]$, and integrate Eq. (3.155) once. Solution: $\delta = \sqrt{2\sqrt{f'(0)}}$

Hint 2. Programming:
You may use the template script below:

```
# src-ch2/stokes.py;ODEschemes.py @ git@lrhgit/tkt4140/src/src-ch2/ODEschemes.py;

#import matplotlib; matplotlib.use('Qt4Agg')
import matplotlib.pylab as plt
#plt.get_current_fig_manager().window.raise_()
import numpy as np

from ODEschemes import euler, heun, rk4
from math import sqrt
from scipy.interpolate import splev, splrep

#### set default plot values: ####
LNWDT=2; FNT=15
plt.rcParams['lines.linewidth'] = LNWDT; plt.rcParams['font.size'] = FNT

#### a: ####
def func(f, eta):

    f0 = f[0]
    f1 = f[1]
    if f1<0:
        # f1 should not be negative
        f1=0
    df0 = ?
    df1 = ?

    dF = np.array([df0, df1])

    return dF


N = 5

Eta = np.linspace(0, 1, N + 1)
d_eta = Eta[1] - Eta[0]

f_0 = ?
df_0 = ?

F_0 = [f_0, df_0]

F = euler(func, F_0, Eta)

print "f(0.6) = {0}, f'(0.6) = {1}".format(F[N*3/5, 0], F[N*3/5, 1])

#### b/c: ####

N = 100 # using more gridpoints

d_eta = Eta[1] - Eta[0]

f_0 = 0

s0, s1 = 1., 1.1
delta = ?

Eta = np.linspace(0, delta, N + 1)
```

```
F_0 = [?, ?]

F = euler(func, F_0, Eta)

phi0 = ?

# Solve using shooting technique:
for n in range(4):
    F_0 = [?, ?]
    delta = ?
    Eta = np.linspace(0, delta, N + 1)

    F = euler(func, F_0, Eta)

    phi1 = ?

    s = ?

    s1, s0 = s, s1
    print "n = {0}, s = {1}, ds = {2}, delta = {3} ".format(n, s0, s1 - s0, delta)


# Transform to time-space variables:

C = 1.
U0 = 1
dt = 0.05
t0, tend = dt, 2.

dy = 0.01
y0, yend = 0, 2.

time = np.arange(dt, tend + dt, dt)

Y = np.arange(y0, yend + dy, dy)

Usolutions = np.zeros((len(time), len(Y)))

f = F[:, 0]*U0
tck = splrep(Eta, f) # create interpolation splines


for n, t in enumerate(time):
    Eta_n = C*Y/(t**(2./5))

    Eta_n = np.where(Eta_n>delta, delta, Eta_n) # if eta>delta set eta to delta

    Usolutions[n, :] = splev(Eta_n, tck)

from Visualization import myAnimation
myAnimation(Y, Usolutions, time)
```

You also need this file in order to run simulation:

```
# src-ch2/Visualization.py

import matplotlib.pylab as plt
import numpy as np
from matplotlib import animation

def myAnimation(Y, Usolutions, time):
```

```
    fig = plt.figure()
    ax = plt.axes(xlim=(0, 1.1), ylim=(0, Y[-1]))

    lines=[]     # list for plot lines for solvers and analytical solutions
    legends=[]   # list for legends for solvers and analytical solutions

    line, = ax.plot([], [])
    time_text = ax.text(0.1, 0.95, '', transform=ax.transAxes)
    dt = time[1]-time[0]
    plt.xlabel('u')
    plt.ylabel('y')


    # initialization function: plot the background of each frame
    def init():
        time_text.set_text('')
        line.set_data([], [])
        return lines,

    # animation function.  This is called sequentially
    def animate(i):
        time = i*dt
        time_text.set_text('time = %.4f' % time)
        U = Usolutions[i, :]
        line.set_data(U, Y)
        return lines,



    # call the animator.  blit=True means only re-draw the parts that have changed.
    anim = animation.FuncAnimation(fig, animate, frames=len(time), init_func=init, blit=False)
#    Writer = animation.writers['ffmpeg']
#    writer = Writer(fps=15, metadata=dict(artist='Me'), bitrate=1800)
#    anim.save('../mov/stokes.mov', writer=writer)

    plt.show()
```

Exercise 6: The Falkner-Skan equation

Blasius equation is a special case of a more general equation called the Falkner-Skan equation. The Falkner-Skan Transformation (1930) allows to write the original equation in a similarity form given by:

$$f''' + ff'' + \beta \cdot [1 - (f')^2] = 0. \tag{3.157}$$

It can be shown that the velocity $U(x)$ is defined as:

$$U(x) = U_0 x^m, \ m = \text{constant}. \tag{3.158}$$

(3.157) is sometimes called the wedge flow equation because parameter β has the geometric meaning as shown in Figure 3.23:

We see that half the wedge angle is $\beta \cdot \frac{\pi}{2}$. For $\beta = 0$, we get a flat plate and (3.157) becomes Blasius equation.

A detailed derivation can be found in Appendix C.3 in the Numeriske Beregninger. Here we reproduce some of the derivations for the sake of completeness.

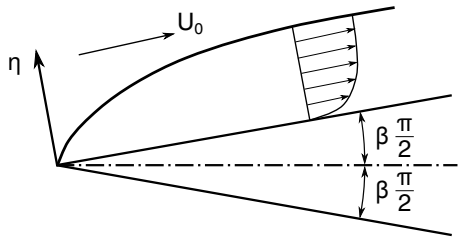

Figure 3.23: Schematic representation of the Falkner-Skan equation reference frame and problem setup.

Connection between m and β: $\beta = \frac{2m}{m+1}$ and $m = \frac{\beta}{2-\beta}$. We see that $\beta = 0$ results in $U(x) = U_0$.

Falkner-Skan transformation is given by:

$$\eta = \sqrt{\frac{U}{(2-\beta)\nu x}} \cdot y, \; f(\eta) = \frac{\psi}{\sqrt{(2-\beta)U\nu x}}, \tag{3.159}$$

where ψ is a potential function such that

$$u = \frac{\partial \psi}{\partial y}, \; v = \frac{\partial \psi}{\partial x}$$

If we choose the same boundary conditions as in the example (3.3.3), we get:

$$f''' + ff'' + \beta \cdot [1 - (f')^2] = 0, \tag{3.160}$$

with boundary conditions:

$$\begin{gathered} f(0) = f'(0) = 0, \\ f'(\eta_\infty) = 1. \end{gathered} \tag{3.161}$$

Chapter 4

Finite difference methods for ordinary differential equtions

4.1 Introduction

Consider the following boundary value problem

$$y''(x) = f(x)\,, \text{ for } 0 < x < 1\,, \tag{4.1}$$

with boundary conditions

$$y(0) = \alpha\,, \quad y(1) = \beta\,. \tag{4.2}$$

Our goal is to solve this problem using finite differences. First we discretize the independent variable $x_i = ih$, with $h = 1/(N+1)$. Having defined a grid, automatically defines our unknowns, i.e. $y_0, y_1, \ldots y_N, y_{N+1}$. From boundary conditions we know that $y_0 = \alpha$ and $y_{N+1} = \beta$, so that the number of true unknowns is N. Next we approximate $y''(x)$ by a centered finite difference approximation

$$y''(x_i) \approx \frac{1}{h^2}(y_{i-1} - 2y_i + y_{i+1})\,, \tag{4.3}$$

obtaining the following algebraic equations

$$\frac{1}{h^2}(y_{i-1} - 2y_i + y_{i+1}) = f(x_i) \tag{4.4}$$

for $i = 1, 2, \ldots, N$. It is important to note that the first equation ($i = 1$) involves the boundary value $y_0 = \alpha$ and that the last equation ($i = N$) involves the value

$y_{N+1} = \beta$. It can be easily noted that the set of equations (4.4) is as a linear system of N equations for N unknowns, which can be written as

$$AY = F\,, \tag{4.5}$$

where Y is the vector of unknowns $Y = [y_1, y_2, \ldots, y_N]^T$ and

$$A = \frac{1}{h^2}\begin{bmatrix} -2 & 1 & & & & & & \\ 1 & -2 & 1 & & & & & \\ & \cdot & \cdot & \cdot & & & & \\ & & \cdot & \cdot & \cdot & & & \\ & & & \cdot & \cdot & \cdot & & \\ & & & & & 1 & -2 & 1 \\ & & & & & & 1 & -2 \end{bmatrix}, F = \begin{bmatrix} f(x_1) - \alpha/h^2 \\ f(x_2) \\ \cdot \\ \cdot \\ \cdot \\ f(x_{N-1}) \\ f(x_N) - \beta/h^2 \end{bmatrix} \tag{4.6}$$

This is a nonsingular tridiagonal system, that can be solved for Y from any right-hand side F.

4.2 Errors and stability

In what follows we refer to the numerical method introduced in the previous section. This section is based in [12], the reader is referred to that book for further details.

In order to discuss about the approximation error introduced by a numerical scheme, the first step is to define what one means by error. Let us denote $\hat{Y}$ the vector of exact solutions at grid points, i.e.

$$\hat{Y} = [y(x_1), y(x_2), \ldots, y(x_N)]^T \tag{4.7}$$

and the error function E

$$E = Y - \hat{Y}\,. \tag{4.8}$$

Once the error has been define we will concentrate in determining *how big* this error is. We can do that by using *some* norm for vector E. For example, we can use the max-norm

$$\|E\|_\infty = \max_{1\le i\le N} |e_i| = \max_{1\le i\le N} |y_i - y(x_i)|\,, \tag{4.9}$$

the 1-norm

$$\|E\|_1 = h\sum_{i=1}^{N} |e_i| \tag{4.10}$$

or the 2-norm

$$\|E\|_2 = \left(h\sum_{i=1}^{N} |e_i|^2\right)^{1/2}. \tag{4.11}$$

Now our goal is to obtain a bound on the magnitude of the error vector E, so that, for example, if we can show that $\|E\|_\infty = \mathcal{O}(h^2)$, then it follows that the error in each point of the grid must be $\mathcal{O}(h^2)$ as well. In order to achive this goal, we will first define the *local truncation error* (LTE) of the numerical scheme and then, via stability, show that the global error can be bounded by the LTE.

4.2.1 Local truncation error

We have already study the LTE for initial value problems in Section 2.12. The LTE for the finite difference scheme (4.4) is obtained by replacing y_i with the true solution $y(x_i)$. In general, the exact solution $y(x_i)$ will not satisfy the finite difference equation. This difference is called LTE and denoted as

$$\tau_i = \frac{1}{h^2}(y(x_{i-1}) - 2y(x_i) + y(x_{i+1})) - f(x_i) \tag{4.12}$$

for $i = 1, 2, \ldots, N$. Since we in practice do not know the exact solution. However, if we assume that it is smooth, we can then use Taylor series expansions, which results in

$$\tau_i = (y''(x_i) + \frac{1}{12}h^2 y''''(x_i) + \mathcal{O}(h^4)) - f(x_i)\,. \tag{4.13}$$

Furthermore, by using the original differential equation (4.1), the LTE becomes

$$\tau_i = \frac{1}{12}h^2 y''''(x_i) + \mathcal{O}(h^4)\,. \tag{4.14}$$

Finally, we note that even if y'''' is in general unknown, it is some fixed function independent of h, so that $\tau_i = \mathcal{O}(h^2)$ as $h \to 0$.

Finally, we note that we can define a vector form of the LTE as

$$\tau = A\hat{Y} - F\,, \tag{4.15}$$

and so

$$A\hat{Y} = F + \tau\,. \tag{4.16}$$

4.2.2 Global error

We obtain a relation between LTE and global error by subtracting (4.16) from (4.5), obtaining

$$AE = -\tau\,. \tag{4.17}$$

This is just a matrix form of the system of equations

$$\frac{1}{h^2}(e_{i-1} - 2e_i + e_{i+1}) = -\tau_i \text{ for } i = 1, 2, \ldots, N \tag{4.18}$$

with boundary conditions $e_0 = e_{N+1} = 0$, because we are prescribing the solution at both ends. We can interpret (4.18) as the discretization of tne ODE

$$e''(x) = -\tau(x) \tag{4.19}$$

with boundary conditions $e(0) = 0$ and $e(1) = 0$. Since $\tau(x) \approx \frac{1}{12}h^2 y''''(x)$, integrating twice shows that the global error should be roughly

$$e(x) \approx -\frac{1}{12}h^2 y''(x) + \frac{1}{12}h^2(y''(0) + x(y''(1) - y''(0))) \tag{4.20}$$

and then the error should be $\mathcal{O}(h^2)$.

4.2.3 Stability

In the above made considerations we have assumed that solving the difference equations results in a decent approximation to the solution of the underlying differential equations. But since this is exactly what we are trying to prove, so that the reasoning is rather circular. Let us pursue a different approach and look at the discrete system

$$A^h E^h = -\tau^h\,, \tag{4.21}$$

where we have used the superscript h to indicate that these quantities depend on h. If we define $(A^h)^{-1}$ as the inverse of A^h, then we can write

$$E^h = -(A^h)^{-1}\tau^h \tag{4.22}$$

and taking the norms gives

$$\|E^h\| = \|(A^h)^{-1}\tau^h\| \tag{4.23}$$

$$\leq \|(A^h)^{-1}\|\|\tau^h\|. \tag{4.24}$$

We already know that $\|\tau^h\| = \mathcal{O}(h^2)$ and we want to verify that this is also true for $\|E^h\|$. Therefore, we need that $\|(A^h)^{-1}\|$ is bounded by some constant independent of h as $h \to 0$, in other words

$$\|(A^h)^{-1}\| \leq C \tag{4.25}$$

for all h sufficiently small. If that is true, we will then have that

$$\|E^h\| \leq C\|\tau^h\| \tag{4.26}$$

with the consequence that $\|E^h\|$ goes to zero at least as fast as $\|\tau^h\|$.

4.2.4 Consistency

A method is said to be *consistent* with the differential equation and boundary conditions if

$$\|\tau^h\| \to 0 \text{ as } h \to 0\,. \tag{4.27}$$

4.2.5 Convergence

We are finally in the position of giving a proper definition of convergence. We say that a method is *convergent* if $\|E^h\| \to 0$ as $h \to 0$. Combining the concepts introduced above, we can say that

$$\text{consistency} \ + \ \text{stability} \Longrightarrow \text{convergence}. \tag{4.28}$$

Lax equivalence theorem A *consistent* finite difference method for a well-posed linear BVP is *convergent* if and only if it is *stable*.

4.2.6 Stability - Example in 2-norm

We conclude this section with an example on how to verify the stability of a finite difference scheme for BVP. We consider the scheme used until now as example, i.e. (4.5).Since A in (4.5) is symmetric, the 2-norm of A is equal to its spectral radius

$$\|A\|_2 = \rho(A) = \max_{1\leq p\leq N} |\lambda_p| \,, \tag{4.29}$$

where λ_p refers to the p-th eigenvalue A. We further note that A^{-1} is also symmetric, so that its eigenvalues are the inverses of the eigenvalues of A

$$\|A^{-1}\|_2 = \rho(A^{-1}) = \max_{1\leq p\leq N} |(\lambda_p|)^{-1} = \left(\min_{1\leq p\leq N} |\lambda_p|\right)^{-1} . \tag{4.30}$$

These considerations tell us that in order to verify the stability of the numerical scheme under investigation we just have to compute the eigenvalues of A and show that they are bounded away from zero as $h \to 0$. In general this can be very trick, since we have an infinite set of matrices A (because A changes with h). However, in this case the structure is so simple that we can obtain a general expression for the eigenvalues.

We consider a single value of $h = 1/(N+1)$. Then the N eigenvalues of A are

$$\lambda_p = \frac{2}{h^2}(cos(p\pi h) - 1) \,, \tag{4.31}$$

for $p = 1, 2, \ldots, N$. Moreover, the eigenvector y^p, corresponding to λ_p has components y_i^p for $i = 1, 2, \ldots, N$ given by

$$y_i^p = sin(p\pi i h) \,. \tag{4.32}$$

You can verify this by checking that $Ay^p = \lambda_p y^p$.

It can be easily seen that the smallest eigenvalue of A, in magnitude, is

$$\lambda_1 = \frac{2}{h^2}(cos(\pi h) - 1) \tag{4.33}$$

$$= \frac{2}{h^2}(-\frac{1}{2}\pi^2 h^2 + \frac{1}{24}\pi^4 h^4 + \mathcal{O}(h^6)) \tag{4.34}$$

$$= -\pi^2 + \mathcal{O}(h^2)\,. \tag{4.35}$$

This is clearly bounded away from zero for $h \to 0$, which allows us to conclude that the scheme is stable in the 2-norm.

4.3 Tridiagonal systems of algebraic equations

When finite difference discretization methods are applied on ordinary of partial differential equations, the resulting equation systems will normally have a distinct *band structure*. In particular, a tridiagonal coefficient matrix will normally be the result when a second order ODE is solved. The tridiagonal matrix has only three diagonals with non-zero elements (hence the name); one main diagonal with the two other on each side of this. Such a tridiagonal system may be represented mathematically as:

$$\begin{aligned} &b_1 x_1 + c_1 x_2 = d_1 \\ &\quad \cdots \\ &a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i \\ &\quad \cdots \\ &a_N x_{N-1} + b_N x_N = d_N \\ &i = 1, 2, ...N \ , a_1 = c_N = 0 \end{aligned} \tag{4.36}$$

or more convenient in a matrix representation:

$$\begin{bmatrix} b_1 & c_1 & & & & & & & \\ a_2 & b_2 & c_2 & & & & & & \\ & \cdot & \cdot & \cdot & & & & & \\ & & \cdot & \cdot & \cdot & & & & \\ & & & \cdot & \cdot & \cdot & & & \\ & & & & & a_{N-1} & b_{N-1} & c_{N-1} \\ & & & & & & a_N & b_N \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_{N-1} \\ x_N \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \cdot \\ \cdot \\ \cdot \\ d_{N-1} \\ d_N \end{bmatrix} \tag{4.37}$$

The linear algebraic equation system (4.36) may be solved by Gauss-elimination, which has a particular simple form for tridiagonal matrices and is often referred to as the Thomas-algorithm. More details on the derivation may be found in appendix I of Numeriske beregninger (in Norwegian).

The Thomas-algorithm has two steps; an elimination step and a substitution step. In the elimination step, sub-diagonal elements are eliminated by summation of two consecutive appropriate scaled rows in the matrix, starting at the first row at the top. By completion of the elimination step, the last unknown x_N may the be computed as $x_N := \frac{d_N}{b_N}$.

In the equation for x_{N-1} immediately above the last, there are only two unknows, namely x_N and x_{N-1}. But since we already have calculated x_N in the elimination step, we substitute its value in this equation and may thereby find x_{N-1}. By moving on to the equation for x_{N-2}, there will be only two unknowns as well, x_{N-2} and x_{N-1}, and since we have already calculated x_{N-1}, it may be substituted into this equation such that x_{N-2} may be found. This procedure of back-substitution may be repeated until all the unknows for the tridiagolnal equation system is known. This step of the procedure is known as *back substitution* or simply *substitution*.

The algorithms for the two steps are listed outlined below in mathematical terms:

Elimination:

$$\begin{aligned} q_j &:= \frac{a_j}{b_{j-1}} \qquad \text{for} \quad j = 2, 3 \cdots, N \\ b_j &:= b_j - q_j \cdot c_{j-1} \\ d_j &:= d_j - q_j \cdot d_{j-1} \end{aligned} \tag{4.38}$$

Back substitution:

$$x_N := \frac{d_N}{b_N} \tag{4.39}$$

$$x_j := \frac{d_j - c_j \cdot x_{j+1}}{b_j} \qquad \text{for} \quad j = N-1, N-2, \cdots, 1 \tag{4.40}$$

The Python-code for (4.38) and (4.39) is implemented in **tdma**, which corresponds to **tri** in [2] , section 6.3 (see below).

```
def tdma(a, b, c, d):
    """Solution of a linear system of algebraic equations with a
        tri-diagonal matrix of coefficients using the Thomas-algorithm.

    Args:
        a(array): an array containing lower diagonal (a[0] is not used)
        b(array): an array containing main diagonal
        c(array): an array containing lower diagonal (c[-1] is not used)
        d(array): right hand side of the system
    Returns:
        x(array): solution array of the system

    """

    n = len(b)
    x = np.zeros(n)

    # elimination:

    for k in range(1,n):
```

```
        q = a[k]/b[k-1]
        b[k] = b[k] - c[k-1]*q
        d[k] = d[k] - d[k-1]*q

    # backsubstitution:

    q = d[n-1]/b[n-1]
    x[n-1] = q

    for k in range(n-2,-1,-1):
        q = (d[k]-c[k]*q)/b[k]
        x[k] = q


    return x
```

Stability of (4.38) and (4.39) is satisfied subject to the following conditions:

$$\begin{array}{l} \mid b_1 \mid > \mid c_1 \mid > 0 \\ \mid b_i \mid \geq \mid a_i \mid + \mid c_i \mid, a_i \cdot c_i \neq 0 \ , i = 2, 3, \cdots, N-1 \\ \mid b_n \mid > \mid a_N \mid > 0 \end{array} \tag{4.41}$$

Matrices satisfying (4.41) are called *diagonally dominant* for obvious reasons, and strictly diagonally in case $\geq$ may be substitued with $>$ in (4.41). Pivoting during Gauss-elimination is not necessary for diagonally dominant matrices, and thus the band structure is preserved and the algorithm becomes less CPU-intensive. See appendix I in Numeriske beregninger for a proof of (4.41).

Notice that all coefficients in each *row* of (4.37) has the same index. However, the linear algebraic equation system (4.36) may also be presented:

$$\begin{array}{l} b_1 x_1 + c_2 x_2 = d_1 \\ \qquad \cdots \\ a_{i-1} x_{i-1} + b_i x_i + c_{i+1} x_{i+1} = d_i \\ \qquad \cdots \\ a_{N-1} x_{N-1} + b_N x_N = d_N \\ \\ i = 1, 2, ...N \ , a_1 = c_N = 0 \end{array} \tag{4.42}$$

or in matrix form:

$$\begin{bmatrix} b_1 & c_2 & & & & & & \\ a_1 & b_2 & c_3 & & & & & \\ & \cdot & \cdot & \cdot & & & & \\ & & \cdot & \cdot & \cdot & & & \\ & & & \cdot & \cdot & \cdot & & \\ & & & & & a_{N-2} & b_{N-1} & c_N \\ & & & & & & a_{N-1} & b_N \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_{N-1} \\ x_N \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \cdot \\ \cdot \\ \cdot \\ d_{N-1} \\ d_N \end{bmatrix} \tag{4.43}$$

We notice that in this notation the coefficients of each *column* in the matrix (4.43) has the same index.

The version in (4.43) may be deduced from (4.37) by subtraction of 1 from the a-inddices and addition of 1 for the c-indices. Commercial codes like Matlab store tridiagonal matrices on the form given in (4.43). We have implemented (4.43) in **tridiag**.

```
def tripiv(a, b, c, d):
    """Solution of a linear system of algebraic equations with a
        tri-diagonal matrix of coefficients using the Thomas-algorithm with pivoting.

    Args:
        a(array): an array containing lower diagonal (a[0] is not used)
        b(array): an array containing main diagonal
        c(array): an array containing lower diagonal (c[-1] is not used)
        d(array): right hand side of the system
    Returns:
        x(array): solution array of the system

    """

    n = len(b)
    x = np.zeros(n)
    fail = 0

    # reordering

    a[0] = b[0]
    b[0] = c[0]
    c[0] = 0

    # elimination:

    l = 0

    for k in range(0,n):
        q = a[k]
        i = k

        if l < n-1:
            l = l + 1

        for j in range(k+1,l+1):
            q1 = a[j]
            if (np.abs(q1) > np.abs(q)):
                q = q1
                i = j
        if q == 0:
            fail = -1

        if i != k:
            q = d[k]
            d[k] = d[i]
            d[i] = q
            q = a[k]
            a[k] = a[i]
            a[i] = q
            q = b[k]
            b[k] = b[i]
            b[i] = q
            q = c[k]
            c[k] =c[i]
```

```
            c[i] = q
        for i in range(k+1,l+1):
            q = a[i]/a[k]
            d[i] = d[i]-q*d[k]
            a[i] = b[i]-q*b[k]
            b[i] = c[i]-q*c[k]
            c[i] = 0


    # backsubstitution

    x[n-1] = d[n-1]/a[n-1]
    x[n-2] = (d[n-2]-b[n-2]*x[n-1])/a[n-2]

    for i in range(n-3,-1,-1):

        q = d[i] - b[i]*x[i+1]
        x[i] = (q - c[i]*x[i+2])/a[i]

    return x

emfs /src-ch3/ #python #package TRIdiagonalSolvers.py @ git@lrhgit/tkt4140/src/src-ch3/TRIdiagonalSo
```

4.4 Examples

4.4.1 Example: Heat exchanger with constant cross-section

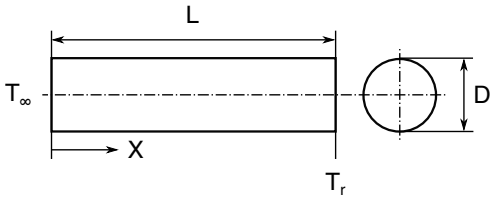

Figure 4.1: Schematic illustration of a heat exchanger.

A circular cylindrical rod of length L, environmental temperature T_∞, and a constant temperature T_r at $X = L$ is illustrated in Figure 4.1. The physical paramters relevant for the heat conduction are the *heat transfer coefficient* $\bar{h} = 100W/m^2\ ^\circ C$ and the *thermal conductivity* $k = 200W/m/^\circ C$.

From equation (B.0.22), appendix B in Numeriske Beregninger we find that the governing equation for the heat transfer in the circular cylindrical rod in Figure 4.1 is governed by:

$$\frac{d}{dX}\left[A(X)\,\frac{d(T-T_\infty}{dX}\right] = \frac{\bar{h}P}{k}(T-T_\infty) \tag{4.44}$$

which for rod with constant cross-section, i.e. $A(x) = A$ reduces to:

$$\frac{d^2}{dX^2}(T - T_\infty) = \frac{\bar{h}P}{kA}(T - T_\infty) \tag{4.45}$$

For convenience we introduce dimensonless variables:

$$x = \frac{X}{L} \ , \ \theta = \frac{T - T_\infty}{T_r - T_\infty} \ , \ \beta^2 = \frac{\bar{h}P}{kA}L^2 \tag{4.46}$$

where L is a characteristic length (see Figure B.5 i Numeriske Beregninger). A relative, dimensionless temperature may be constructed by using a reference temperature T_r and temperature and the temperature T_∞ in combination. The parameteren β^2 is frequently referred to as the Biot-number, $Bi = \beta^2$.

Given the assumptions and prerequesites above (4.44) may be presented in the more convenient and generic form:

$$\frac{d^2\theta}{dx^2} - \beta^2\theta = 0 \tag{4.47}$$

which is a second order ODE with the following generic analytical solution:

$$\theta(x) = A\sinh(\beta x) = -B\cosh(\beta x) \tag{4.48}$$

where the constants A and B must be determined from the boundary condtions.

In the following we will investigate the numerical solution to two different boundary conditions commonly denoted as *prescribed boundary conditions* (which for this particular corresponds to prescribed temperatures) and *mixed boundary conditions*.

Prescribed temperatures:

$$\begin{aligned} T &= T_\infty && \text{for } X = 0 \\ T &= T_r && \text{for } X = L \end{aligned}$$

which may be presented on dimensionless form as:

$$\begin{aligned} \theta &= 0 && \text{for } x = 0 \\ \theta &= 1 && \text{for } x = 1 \end{aligned} \tag{4.49}$$

The analytical solution (4.48) for these particular boundary conditions reduces to:

$$\theta(x) = \frac{\sinh(\beta x)}{\sinh(\beta)} \ , \ \frac{d\theta}{dx} = \beta\,\frac{\cosh(\beta x)}{\sinh(\beta)} \tag{4.50}$$

With $D = 0.02m$ and $L = 0.2m$ the Biot-number $\beta^2 = 4$. By doubling the length the Biot-number quadurples. For eksemplet ovenfor blir Biot-tallet:

$$\beta^2 = \frac{\bar{h}P}{kA}\,L^2 = \frac{2}{D}\,L^2 \tag{4.51}$$

Mixed boundary conditions:

$$\begin{aligned} Q_x = 0 = \frac{dT}{dX} \qquad & \text{for } X = 0 \\ T = T_r \qquad & \text{for } X = L \end{aligned}$$

A zero temperature gradient at $X = 0$ corresponds to an isolated rod at that location. The dimensionless representation of this boundary conditions is:

$$\frac{d\theta}{dx} = 0 \qquad \text{for } x = 0 \tag{4.52}$$

$$\theta = 1 \qquad \text{for } x = 1 \tag{4.53}$$

The analytical solution (4.48) reduces for the mixed boundary conditions to:

$$\theta(x) = \frac{\cosh(\beta x)}{\cosh(\beta)} \ , \ \frac{d\theta}{dx} = \beta \, \frac{\sinh(\beta x)}{\cosh(\beta)} \tag{4.54}$$

Numerical solution. We will use central differences for the term $\dfrac{d^2\theta}{dx^2}$ and with $\left.\dfrac{d^2\theta}{dx^2}\right|_i \approx \dfrac{\theta_{i-1} - 2\theta_i + \theta_{i+1}}{h^2}$ we get the following difference equation:

$$\theta_{i-1} - (2 + \beta^2 h^2) \, \theta_i + \theta_{i+1} = 0 \tag{4.55}$$

Prescribed temperatures
We enumerate the nodes for the unkowns as shown in Figure 4.2:

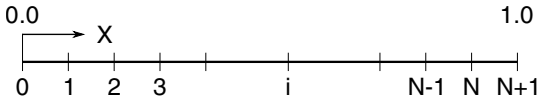

Figure 4.2: Enumeration of nodes for prescribed boundary temperatures.

The x-coordinates can be denoted in a compact manner by:: $x_i = i \ h$, $i = 0, 1, ..., N + 1$ where $h = \frac{1}{N+1}$, i.e. the prescribed temperatures at the boundaries are denoted θ_0 and θ_{N+1}, respectively.

A good practice is to apply the generic scheme (4.55) for nodes in the immediate vincinity of the boundaries, and first we take a look at $i = 1$

$$\theta_0 - (2 + \beta^2 h^2) \, \theta_1 + \theta_2 = 0 \rightarrow (2 + \beta^2 h^2) \, \theta_1 + \theta_2 = 0$$

which may be simplified by substitution of the prescribed boundary value $\theta(0) = \theta_0 = 0$

$$(2+\beta^2 h^2)\,\theta_1 + \theta_2 = 0$$

For the other boundary at $i = N$ the generic scheme (4.55) is:

$$\theta_{N-1} - (2+\beta^2 h^2)\,\theta_N + \theta_{N+1} = 0$$

which by substitution of the prescribed value for $\theta_{N+1} = 1$ yields:

$$\theta_{N-1} - (2+\beta^2 h^2)\,\theta_N = -1$$

A complete system of equations may finally be obtained from (4.4.1), (4.55), and (4.4.1):

$$\begin{aligned} i = 1: &\quad -(2+\beta^2 h^2)\,\theta_1 + \theta_2 = 0 \\ i = 2,3,...,N-1: &\quad \theta_{i-1} - (2+\beta^2 h^2)\,\theta_i + \theta_{i+1} = 0 \\ i = N: &\quad \theta_{N-1} - (2+\beta^2 h^2)\,\theta_N = -1 \end{aligned} \tag{4.56}$$

See appendix A, section A.5, example A.15 in Numeriske Beregninger, this example is treated in more detail. The following system of coefficents may be obtained by comparison with (4.36):

$$\begin{aligned} a_i &= 1\ , & i &= 2,3,...,N \\ b_i &= -(2+\beta^2 h^2)\ , & i &= 1,2,...,N \\ c_i &= 1\ , & i &= 1,2,...,N-1 \\ d_i &= 0\ , & i &= 1,2,...,N-1 \\ d_N &= -1 \end{aligned} \tag{4.57}$$

In the program **ribbe1.py** below the linear, tri-diagonal equation system (4.57) resulting from the discretization in (4.55) is solved by using two SciPy modules, the generic `scipy.linalg.solve` and the computationally more efficient `scipy.sparse.linalg.spsolve`.

SciPy is built using the optimized ATLAS LAPACK and BLAS libraries and has has very fast linear algebra capabilities. Allegedly, all the raw lapack and blas libraries are available for even greater speed. However, the sparse linear algebra module of scipy offers easy-to-use python interfaces to these routines. Numpy has also a linalg-module, however, an advantage of using scipy.linalg over numpy.linalg is that it is always compiled with BLAS/LAPACK support, while this is optional for numpy. Note that the scipy.linalg algorithms do not exploit the sparse nature of the matrix as they use direct solvers for dense matrices.

However, SciPy offers a sparse matrix package scipy.sparse. The `spdiags` function may be used to construct a sparse matrix from diagonals. Note that all the diagonals must have the same length as the dimension of their sparse matrix - consequently some elements of the diagonals are not used. The first k elements

are not used of the k super-diagonal, whereas the last k elements are not used of the $-k$ sub-diagonal. For a quick tutorial of the usage of these sparse solvers see SciPy sparse examples.

We have implemented a simple means to compare the two solution procedures with a `tic-toc` statements. You may experiment with various problems sizes (by varying the element size h) to assess the impact on the computational speed of the two procedures. The analytical solution is given by (4.50).

```
# src-ch3/section321/ribbe1.py

import numpy as np
import scipy as sc
import scipy.linalg
import scipy.sparse
import scipy.sparse.linalg
import time
from math import sinh

#import matplotlib.pyplot as plt
from matplotlib.pyplot import *
# Change some default values to make plots more readable on the screen
LNWDT=3; FNT=20
matplotlib.rcParams['lines.linewidth'] = LNWDT; matplotlib.rcParams['font.size'] = FNT

# Set simulation parameters
beta = 5.0
h = 0.001               # element size
L =1.0                  # length of domain
n = int(round(L/h)) -1  # number of unknowns, assuming known boundary values
x=np.arange(n+2)*h      # x includes min and max at boundaries were bc are imposed.


#Define useful functions

def tri_diag_setup(a, b, c, k1=-1, k2=0, k3=1):
    return np.diag(a, k1) + np.diag(b, k2) + np.diag(c, k3)

def theta_analytical(beta,x):
    return np.sinh(beta*x)/np.sinh(beta)

#Create matrix for linalg solver
a=np.ones(n-1)
b=-np.ones(n)*(2+(beta*h)**2)
c=a
A=tri_diag_setup(a,b,c)

#Create matrix for sparse solver
diagonals=np.zeros((3,n))
diagonals[0,:]= 1                       #all elts in first row is set to 1
diagonals[1,:]= -(2+(beta*h)**2)
diagonals[2,:]= 1
A_sparse = sc.sparse.spdiags(diagonals, [-1,0,1], n, n,format='csc') #sparse matrix instance

#Crete rhs array
d=np.zeros(n)
d[n-1]=-1

#Solve linear problems
tic=time.clock()
```

```
theta = sc.sparse.linalg.spsolve(A_sparse,d) #theta=sc.linalg.solve_triangular(A,d)
toc=time.clock()
print 'sparse solver time:',toc-tic

tic=time.clock()
theta2=sc.linalg.solve(A,d,)
toc=time.clock()
print 'linalg solver time:',toc-tic

# Plot solutions
plot(x[1:-1],theta,x[1:-1],theta2,'-.',x,theta_analytical(beta,x),':')
legend(['sparse','linalg','analytical'])
show()
close()
print 'done'
```

The numerical predictions are presented and compared with analytical values in the table below:

x	numerical	analytical	rel.err
0.0	0.00000	0.00000	
0.1	0.05561	0.05551	0.00176
0.2	0.11344	0.11325	0.00170
0.3	0.17582	0.17554	0.00159
0.4	0.24522	0.24487	0.00144
0.5	0.32444	0.32403	0.00126
0.6	0.41663	0.41619	0.00105
0.7	0.52548	0.52506	0.00082
0.8	0.65536	0.65499	0.00056
0.9	0.81145	0.81122	0.00029
1.0	1.00000	1.00000	0.00000

Mixed boundary conditions. Version 1

For mixed boundary conditions the nodes with uknown temperatures are enumerated as shown in Fig. (4.3), to make the first unknown temperature we compute have index 1. The x-coordinates are given by: $x_i = (i-1)\ h,\ i = 1, 2, ..., N+1$ der $h = \frac{1}{N}$ and we wil use second order central differences as an approximation of the zero-gradient boundary condition $\frac{d\theta}{dx} = 0$ at the left boundary where $x = 0$.

Figure 4.3: Enumeration of nodes for mixed boundary conditions.

For a generic node 'i' the central difference approximation may be denoted:

$$\left.\frac{d\theta}{dx}\right|_i \approx \frac{\theta_{i+1} - \theta_{i-1}}{2h} \tag{4.58}$$

which for node 1 takes the form:

$$\frac{\theta_2 - \theta_0}{2h} = 0 \tag{4.59}$$

and we observe that this strategy requires a value of the temperature θ_0 at a node 0 which is not included in Figure 4.3. However, this is not a problem since θ_0 may be eliminated using the identity obtained from (4.59):

$$\theta_0 = \theta_2 \tag{4.60}$$

Due to the zero gradient boundary condition, the first of the N equations, represented on generic form by (4.55), takes the particular form:

$$-(2 + \beta^2 h^2)\, \theta_1 + 2\theta_2 = 0 \tag{4.61}$$

This first equation (4.61) is the only equation which differs from the resulting equation system for prescibed boundary conditions in (4.56). All the coefficients a_i, b_i, and d_i are the same as in for the prescribed temperature version in (4.57), except for c_1:

$$c_1 = 2, \qquad c_i = 1, \qquad i = 2, ...N - 1 \tag{4.62}$$

Mixed boundary conditions. Version 2

An alternative version 2 for implementation of the zero-gradient boundary condition may be obtained by using a forward approximation for the gradient as given by (4.55) for a generic node i:

$$\left.\frac{d\theta}{dx}\right|_i \approx \frac{-3\theta_i + 4\theta_{i+1} - \theta_{i+2}}{2h} \tag{4.63}$$

which takes the following form for node 1 where is should evaluate to zero:

$$\left.\frac{d\theta}{dx}\right|_1 \approx \frac{-3\theta_1 + 4\theta_2 - \theta_3}{2h} = 0 \tag{4.64}$$

From equation (4.64) we see that θ_3 may be eliminated by:

$$\theta_3 = 4\theta_2 - 3\theta_1 \tag{4.65}$$

The first difference equation (4.55) in which θ_3 occurs, is the one for node 2

$$\theta_1 - (2 + \beta^2 h^2)\, \theta_2 + \theta_3 = 0 \tag{4.66}$$

and we may eliminate θ_3 from equation (4.66) by substitution of (4.65):

$$2\,\theta_1 - (2 + \beta^2 h^2)\,\theta_2 = 0 \tag{4.67}$$

This is the first equation in the system of equations and the only one which differs from (4.56). Rather than (4.57), we get the following equations for the coefficients:

$$b_1 = 2 \qquad b_i = (2 + \beta^2 h^2) \qquad i = 2, ..., N \tag{4.68}$$

$$c_1 = -(2 - \beta^2 h^2) \qquad c_i = 1 \qquad i = 2, ..., N-1 \tag{4.69}$$

For convenience we summarise the resulting system of equations by:

$$\begin{aligned} &2\,\theta_1 - (2 - \beta^2 h^2)\,\theta_2 = 0, && i = 1 \\ &\theta_{i-1} - (2 + \beta^2 h^2)\,\theta_i + \theta_{i+1} = 0, && i = 2, 3, ..., N-1 \\ &\theta_{N-1} - (2 + \beta^2 h^2)\,\theta_N = -1, && i = N \end{aligned} \tag{4.70}$$

Note that we in this case, with a forward difference apporoximation of a gradient boundary condition, had to combine the approximation of the gradient with the difference equation to get a resulting tri-diagonal system. The the program **ribbe2** we solve the mixed boundary conditions with both Version 1 and Version 2.

```
# src-ch3/section321/ribbe2.py

import numpy as np
import scipy as sc
import scipy.linalg
import scipy.sparse
import scipy.sparse.linalg
import time
from numpy import cosh

#import matplotlib.pyplot as plt
from matplotlib.pyplot import *
# Change some default values to make plots more readable on the screen
LNWDT=3; FNT=20
matplotlib.rcParams['lines.linewidth'] = LNWDT; matplotlib.rcParams['font.size'] = FNT

# Set simulation parameters
beta = 3.0
h = 0.001               # element size
L =1.0               # length of domain
n = int(round(L/h))  # # of unknowns, assuming known bndry values at outlet
x=np.arange(n+1)*h      # x includes min and max at boundaries were bc are imposed.


#Define useful functions

def tri_diag_setup(a, b, c, k1=-1, k2=0, k3=1):
    return np.diag(a, k1) + np.diag(b, k2) + np.diag(c, k3)
```

```
def theta_analytical(beta,x):
    return np.cosh(beta*x)/np.cosh(beta)

#Create matrix for linalg solver
a=np.ones(n-1)                  # sub-diagonal
b=-np.ones(n)*(2+(beta*h)**2)   # diagonal
c=np.ones(n-1)                  # sub-diagonal
#c=a.copy()                      # super-diagl, copy as elts are modified later
#c=a
# particular diagonal values due to derivative bc
version=2
if (version==1):
    c[0]=2.0
else:
    b[0]=2.0
    c[0]=-(2-(beta*h)**2)
    print 'version 2'

A=tri_diag_setup(a,b,c)

#Create matrix for sparse solver
diagonals=np.zeros((3,n))
diagonals[0,:]= 1.0               # all elts in first row is set to 1
diagonals[0,0]= 1.0
diagonals[1,:]= -(2+(beta*h)**2)
diagonals[2,:]=1.0

if (version==1):
    diagonals[2,1]= 2.0               # index 1 as the superdiagonal of spdiags is not used,
else:
    diagonals[1,0]=2.0                # Sets the first element in the main diagonal
    diagonals[2,1]= -(2+(beta*h)**2)  # index 1 as the superdiagonal of spdiags is not used,
 super-diagonal

A_sparse = sc.sparse.spdiags(diagonals, [-1,0,1], n, n,format='csc') #sparse matrix instance

#Crete rhs array
d=np.zeros(n)
d[-1]=-1

#Solve linear problems
tic=time.clock()
theta = sc.sparse.linalg.spsolve(A_sparse,d) #theta=sc.linalg.solve_triangular(A,d)
toc=time.clock()
print 'sparse solver time:',toc-tic

tic=time.clock()
theta2=sc.linalg.solve(A,d,)
toc=time.clock()
print 'linalg solver time:',toc-tic

# Plot solutions
plot(x[0:-1],theta,x[0:-1],theta2,'-.',x,theta_analytical(beta,x),':')
xlabel('x')
ylabel(r'Dimensionless temperature $\mathregular{\theta}$')
legend(['sparse','linalg','analytical'])
show()
close()
print 'done'
```

The relative error is computed from $\varepsilon_{rel} = |(\theta_{num} - \theta_{analyt})/\theta_{analyt}|$, and the results of the computations are given in the table below:

x	Anlytical	Ctr.diff	Rel.err	Fwd.diff	Rel. err
0.0	0.26580	0.26665	0.00320	0.26613	0.00124
0.1	0.27114	0.27199	0.00314	0.27156	0.00158
0.2	0.28735	0.28820	0.00295	0.28786	0.00176
0.3	0.31510	0.31594	0.00267	0.31567	0.00180
0.4	0.35549	0.35632	0.00232	0.35610	0.00171
0.5	0.41015	0.41095	0.00194	0.41078	0.00153
0.6	0.48128	0.48202	0.00154	0.48189	0.00128
0.7	0.57171	0.57237	0.00114	0.57228	0.00098
0.8	0.68510	0.68561	0.00075	0.68555	0.00067
0.9	0.82597	0.82628	0.00037	0.82625	0.00034
1.0	1.00000	1.00000	0.00000	1.00000	0.00000

We observe that the two versions for zero-gradient boundary condition yields approximately the same result except close to $x = 0$, where the forward difference is somewhat better. In section (6.3.1) we will take a closer look at the accuracy of gradient boundary conditions.

Mixed boundary conditions. Version 3

Rather than the enumeration in Figure (4.3), we may use the enumeration in Fig. (4.2) with $x_i = i\,h\ ,\ i = 0, 1, ..., N+1$ where $h = \frac{1}{N+1}$ such that we must take $i = 1$ in (4.55) to get:

$$\theta_0 - (2 + \beta^2 h^2)\,\theta_1 + \theta_2 = 0$$

The boundary condition in (4.63) the becomes:

$$\left.\frac{d\theta}{dx}\right|_0 = \frac{-3\theta_0 + 4\theta_1 - \theta_2}{2h} = 0$$

from which an elimination equation for θ_0 may be obtained:

$$\theta_0 = 4(\theta_1 - \theta_2)/3 \tag{4.71}$$

Concequently, equation (4.71) may be used to eliminate θ_0 from the first difference equation:

$$-(2 + 3\beta^2 h^2)\,\theta_1 + 2\theta_2 = 0 \tag{4.72}$$

With approach θ_0 is not solved with the equation system, but retrieved subsequently from equation (4.71).

4.4.2 Cooling rib with variable cross-section

In Figure 4.4 trapezoidal cooling rib with length L and width b is illustrated. The thickness vaires from d at $X = 0$ to D at $X = L$. The cooling rib has a

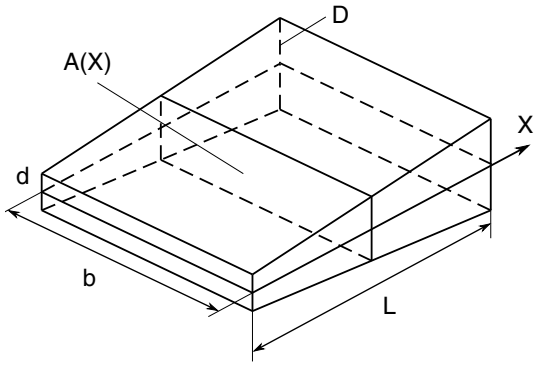

Figure 4.4: A cooling rib of length L, width b and a varying cross-sectional area $A(X)$ (trapezoidal).

heat loss at $X = 0$ and a prescribed temperature T_L at X_L, with a constant surounding temperature T_∞ .

In Figure 4.5 the upper half of the cooling rib is illustrated for a better quantitative demonstration of the geometry, where d and D denote the diameters of the cooling rib at the inlet and outlet, respectively, whereas L denotes the length of the rib.

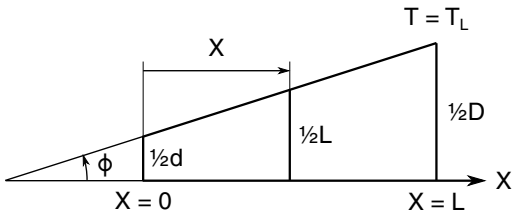

Figure 4.5: A cooling rib with a trapezoidal cross-section.

The following geometrical relations may be deducted:

$$\tan(\phi) = \frac{D/2 - d/2}{L} = \frac{d^*/2 - d/2}{X}$$

Further, the spatial dependent diameter d^* may be isolated and presented as:

$$d^*(X) = d + \left(\frac{D-d}{L}\right) X \tag{4.73}$$

We assume that the temperature mostly varies in the X-direction such that the derivation in appendix B in Numeriske metoder is valid. We assume accordingly that D $\ll$ L and that $0 < \frac{d}{D} < 1$.

In Numeriske metoder the quasi one-dimensional differential equation for heat conduction is derived, and for stationary conditions is reduces to equation (B.0.22):

$$\frac{d}{dX}\left[A(X)\,\frac{d(T-T_\infty)}{dX}\right] = \frac{\bar{h}P(X)}{k}(T-T_\infty) \tag{4.74}$$

with corresponding boundary conditions:

$$\frac{dT(0)}{dX} = \frac{\bar{h}_0}{k}[T(0)-T_\infty]\ ,\ T(L) = T_L \tag{4.75}$$

For the trapezoidal cross-section in Fig. (4.4):

$$P(X) = 2(b+d^*) \approx 2b \qquad \text{for } d^* \ll b \qquad \text{and} A(X) = b\,d^*$$

For convenience we introduce dimensionless quantities for temperature $\theta = \frac{T-T_\infty}{T_L-T_\infty}$, length $x = \frac{X}{L}$, and the diameter ratio $\alpha = \frac{d}{D}$, $0 < \alpha < 1$ and the Biot numbers:

$$\beta^2 = \frac{2\bar{h}\,L^2}{D\,k} \qquad \text{and} \qquad \beta_0^2 = \frac{\bar{h}_0}{k}\,L \tag{4.76}$$

Equipped with these dimensionless quantities we may render equation (4.74) as:

$$\frac{d}{dx}\left[\{\alpha+(1-\alpha)\,x\}\frac{d\theta(x)}{dx}\right] - \beta^2\theta(x) = 0 \tag{4.77}$$

and corresponding boundary conditions become:

$$\frac{d\theta}{dx}(0) = \beta_0^2\,\theta(0)\ ,\ \theta(1) = 1 \tag{4.78}$$

As $d \to 0$ the trapezoidal profile approaches a triangular profile and $\alpha \to 0$, and consequently equation (4.77) takes the form:

$$x\frac{d^2\theta}{dx^2} + \frac{d\theta}{dx} - \beta^2\theta(x) = 0 \tag{4.79}$$

For the left boundary at $x = 0$ we get:

$$\frac{d\theta}{dx}(0) - \beta^2\theta(0) = 0 \tag{4.80}$$

An analytical solution of equation (4.77) and (4.79) is outlined in G.5 of Numeriske Beregninger.

4.4.3 Example: Numerical solution for specific cooling rib

In the current example we will look at the solution for a specific cooling rib based on the generic representation a bove. We consider the a rib with the following geometry:

$$L = 0.1m, \quad D = 0.01m, \quad d = 0.005m, \tag{4.81}$$

and physical constants:

$$\bar{h} = 80W/m^2/^\circ C, \bar{h}_0 = 200W/m^2/^\circ C, \text{og } k = 40W/m/^\circ C \tag{4.82}$$

Together, these particular values of geometrical and physical variables correspond to $\beta^2 = 4.0$, $\alpha = \frac{1}{2}$ and $\beta_0^2 = 0.5$ and equation (4.77) will consequently take the form:

$$\frac{d}{dx}\left[(1+x)\,\frac{d\theta}{dx}\right] - 8\,\theta(x) = 0 \tag{4.83}$$

Analytical solution An analytical solution of the resulting differential equation (4.83) for this example may be found in appendix G.5 in Numeriske metoder and a corresponding python snippet is shown below along with the resulting plot:

```
# coding: utf-8
# src-ch3/trapes.py;

import numpy as np
from matplotlib.pyplot import *
from numpy import sqrt as sqrt
from scipy.special import kv as besselk
from scipy.special import iv as besseli


# change some default values to make plots more readable
LNWDT=3; FNT=11
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT
font = {'size' : 16}; rc('font', **font)

def trapes_analytical(h):

    N=int(round(1.0/h))      # number of unknowns, assuming the RHS boundary value is known
    x=np.arange(N+1)*h
#    x = np.linspace(0, 1,  N)
    z0 = 4*sqrt(2)
    z1 = 8
    g = sqrt(2)/8

    k1z0, i0z1 = besselk(1, z0), besseli(0, z1)
    k0z1, i1z0 = besselk(0, z1), besseli(1, z0)
    k0z0, i0z0 = besselk(0, z0), besseli(0, z0)

    J = k1z0*i0z1 + k0z1*i1z0 + g*(k0z0*i0z1 - k0z1*i0z0)
    A = (k1z0 + g*k0z0)/J
    B = (i1z0 - g*i0z0)/J
```

```
    z = sqrt(32.*(1 + x))
    theta = A* besseli(0, z) + B* besselk(0, z)
    dtheta = 16*(A*besseli(1, z) - B*besselk(1, z))/z
    return x, theta, dtheta


if __name__ == '__main__':
    legends=[] # empty list to append legends as plots are generated

    h=0.2
    x, theta, dtheta = trapes_analytical(h)

    plot(x,theta,'b',)
    legends.append(r'$\theta$')
    plot(x,dtheta,'r')
    legends.append(r'$\theta^{\prime}$')

    ## Add the labels
    legend(legends,loc='best',frameon=False) # Add the legends
    ylabel(r'$\theta$ and $\theta^{\prime}$')
    xlabel('x')



    show()

    close()
```

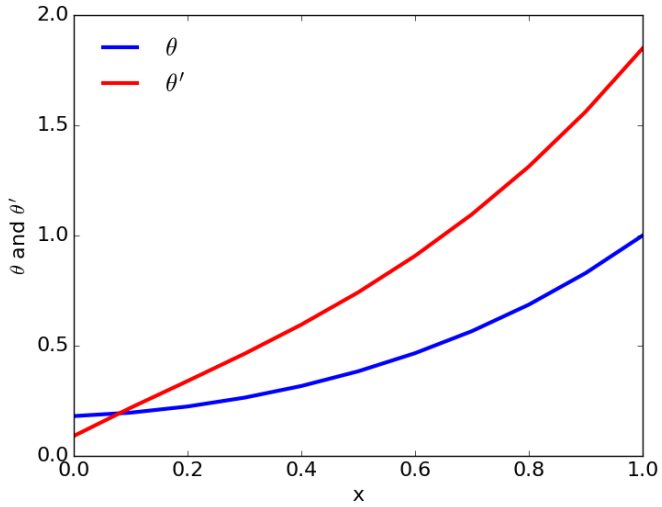

Figure 4.6: The dimensionless temperature θ and the spatial derivative θ' as a function of of the spatial variable x.

Numerical solution

We will solve equation (4.83) numerically by central differences, and first rewrite it as:

$$(1+x)\frac{d^2\theta}{dx^2} + \frac{d\theta}{dx} - 8\,\theta(x) = 0 \tag{4.84}$$

with the boundary conditions:

$$\frac{d\theta}{dx}(0) = \beta_0^2\,\theta(0) = \frac{\theta(0)}{2}, \qquad \theta(1) = 1 \tag{4.85}$$

The procedure for this example will follow the along the lines of the procedure in section (4.4.1) for mixed boundary conditions.

Figure 4.7: Discretization of a bar of unit length.

The x-coordinates are discretized by: $x_i = i\ h\ ,\ i = 0, 1, 2, \ldots, N$ where $h = \frac{1}{N}$ and a central difference approximation of equation (4.84) leads to the following discretized equation:

$$(1+x_i)\ \frac{\theta_{i+1} - 2\theta_i + \theta_{i-1}}{h^2} + \frac{\theta_{i+1} - \theta_{i-1}}{2h} - 8\theta_i = 0$$

which after collection of terms may be presented as for generic point x_i along the unit length bar in Figure 4.7:

$$-(1-\gamma_i)\ \theta_{i-1} + 2\ (1+8h\ \gamma_i)\ \theta_i - (1+\gamma_i)\ \theta_{i+1} = 0 \tag{4.86}$$

where

$$\gamma_i = \frac{h}{2\ (1+x_i)} = \frac{h}{2\ (1+i\ h)}, \qquad i = 1, \ldots, N-2 \tag{4.87}$$

We observe that equation (4.86) form a tridiagonal equation system as only the immediate neigbors of θ_i, namely θ_{i-1} and θ_{i-1}, are involved for the $i-th$ equation.

At the left boundary we need equation (4.85) to be satisfied, which has the second order discretized equivalent $\frac{\theta_2 - \theta_0}{2h} = \frac{\theta_1}{2}$ which yields $\theta_0 = \theta_2 - \theta_1\ h$ and after substitution in equation (4.86) for $i = 0$ we get the following discretized version at the boundary:

$$[2 + h\ (1 + 15\ \gamma_0)]\ \theta_0 - 2\ \theta_1 = 0 \tag{4.88}$$

Note that the discretization $\theta_0 = \theta_2 - \theta_1\ h$ is equally valid, it is just that the implementation of the boundary condtion would then violate the nice tridiagonal structure of the discretized equation system, and thus we prefer equation (4.88) which fits very nicely in the tri-diagonal structure.

For $i = N-1$ we make use of the fact that $\theta_N = 1$ is given :

$$-(1-\gamma_{N-1})\ \theta_{N-2} + 2\ (1+8h\ \gamma_{N-1})\ \theta_{N-1} = (1+\gamma_{N-1})\ \theta_N = 1+\gamma_{N-1} \quad (4.89)$$

Together, the equations (4.86), (4.88),and (4.89) form a tridiagonal, linear equation system which we will solve using the sparse library `scipy.sparse` from SciPy. An example of how to use the sparse library may be found in our Introdcution to Scientific Python programming.

The python code for the implementation of the algorithm is shown below. We compare the solution with the analytical solution and again we observe the striking difference between the very complex analytical solution (involving Bessel-functions of various kinds) and the relatively simple numerical solution involving only the solution of a tridiagonal linear equation system.

As we have discretized equation (4.84) using central differences, we expect the numerical solution to be second order accurate and have added a code segment to approximate the order of the scheme. You may test the code an see whether our implementation has the expected order. Optionally, we have also provided the possibility to plot the solution as a function of the mesh size.

```
# coding: utf-8
# src-ch3/trapes_numerical.py;

import numpy as np
from matplotlib.pyplot import *
import scipy as sc
import scipy.sparse
import scipy.sparse.linalg
import time
from trapes import trapes_analytical
from Carbon.Aliases import false



# change some default values to make plots more readable
LNWDT=3; FNT=11
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT
font = {'size' : 16}; rc('font', **font)


def solve_trapes_numerically(h):
    L=1.0                    # length of domain
    n=int(round(L/h))        # number of unknowns, assuming the RHS boundary value is known
    x=np.arange(n+1)*h       # x includes xmin and xmax at boundaries
    gamma=h/(2*(1+x[0:-1])) # uses all but the last x-value for gamma

    subdiag=gamma[:-1]-1.0  #the subdiagonal has one less element than the diagonal

    diag=2.0*(1+8.0*h*gamma)
    diag[0]= 2.0+h*(1+15.0*gamma[0])
```

```
    superdiag=np.zeros(n-1)
    superdiag[0]=-2
    superdiag[1:]=-(gamma[0:-2]+1.0)

    A_diags=[subdiag,diag,superdiag] #note the offdiags are one elt shorter than the diag
    A = sc.sparse.diags(A_diags, [-1,0,1], format='csc')

    #Create rhs
    d=np.zeros(n)
    d[n-1]=1+gamma[-1]

    #Solve linear problem
    tic=time.clock()
    theta = sc.sparse.linalg.spsolve(A,d) #theta=sc.linalg.solve_triangular(A,d)
    toc=time.clock()
    print 'sparse solver time:',toc-tic

    return x,theta

if __name__ == '__main__':
    h=0.2
    xa, atheta, adtheta = trapes_analytical(h)
    x, theta=solve_trapes_numerically(h)

    # Plot solutions
    figure('Solutions')
    plot(xa,atheta,)
    plot(x[:-1],theta,'-.')

    legends=[] # empty list to append legends as plots are generated
    legends.append(r'$\theta$-analytical')
    legends.append(r'$\theta$')

    ## Add the labels
    legend(legends,loc='best',frameon=False) # Add the legends
    ylabel(r'$\theta$')
    xlabel('x')

    # Approximate the order of the scheme
    h=0.2
    h_range=[h/2.**(d) for d in np.arange(0,6)]

    error=[]
    for h in h_range:
        xa, atheta, adtheta = trapes_analytical(h)
        x,theta=solve_trapes_numerically(h)
        error.append(np.sqrt(np.sum(((atheta[:-1]-theta)/atheta[:-1])**2)))


    log_error = np.log2(error)
    dlog_err=log_error[0:-1]-log_error[1:]
    print('Approximate order:{0:.3f}'.format(1./np.mean(dlog_err)))

    plot_error=0
    if (plot_error):
        figure('error')
        ax = plot(h_range,error)
        yscale('log',basey=2)
        xlabel(r'meshsize $h$')
        ylabel(r'$\frac{\epsilon}{\theta_a}$')
```

```
show()
close()
```

4.5 Linearization of nonlinear algebraic equations

In this section we will present methods for linearization of nonlinear algebraic equations and will use the problem in section (4.4.2) as an illustrative example:

$$y''(x) = \frac{3}{2}y^2 \tag{4.90}$$

with the following boundary conditions:

$$y(0) = 4, \qquad y(1) = 1 \tag{4.91}$$

From (4.4.2) we know that one of the two analytical solutions is given by:

$$y = \frac{4}{(1+x)^2} \tag{4.92}$$

We use central differences for the term $y''(x)$ to discretize equation (4.90):

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} = \frac{3}{2}\, y_i^2$$

which after collection of terms may be presented as:

$$y_{i-1} - \left(2 + \frac{3}{2}h^2\, y_i\right)\, y_i + y_{i+1} = 0 \tag{4.93}$$

We have discretized the unit interval $[0, 1]$ in N parts with a corresponding mesh size $h = 1/N$ and discrete coordinates denoted by $x_i = h \cdot i$ for $i = 0, 1, \ldots, N+1$ (see Fig 4.8)

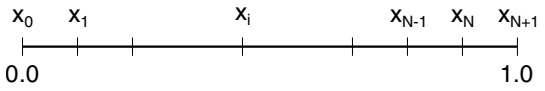

Figure 4.8: Discretization one-dimensional boundary value problem on a unit interval.

By substituting the known boundary values $y_0 = 4$ og $y_{N+1} = 1$ in equation (4.93) we get the following system of equations:

$$
\begin{aligned}
-\left(2+\frac{3}{2}h^2\ y_1\right)y_1 + y_2 &= -4 \\
&\vdots \\
y_{i-1} - \left(2+\frac{3}{2}h^2\ y_i\right)y_i + y_{i+1} &= 0 \\
&\vdots \\
y_{N-1} - \left(2+\frac{3}{2}h^2\ y_N\right)y_N &= -1
\end{aligned}
\tag{4.94}
$$

where $i = 1, 2, \ldots, N-1$. The coefficient matrix is tridiagonal, but the system is nonlinear. And as we have no direct analytical solutions for such nonlinear algebraic systems, we must linearize the system and develop an iteration strategy which hopefully will converge to a solution to the nonlinear problem. In the following we will present two methods for linearization.

4.5.1 Picard linearization

Due to the nonlinearities in equation (4.93), we must develop an iteration strategy and to do so we let y_i^{m+1} og y_i^m be the solution of the discretized equation (4.94) at iterations $m+1$ and m, respectively.

With Picard linearization we linearize the nonlinear terms simply by replacement of dependent terms at iteration $m+1$ with corresponding terms at iteration m until only linear terms are left. In equation (4.94) the nonlinearity is caused by the y^2-term and the correpondings terms need to be linearized.

By using the iteration nomenclature, equation (4.94) takes the form:

$$
\begin{aligned}
-\left(2+\frac{3}{2}y_1^{m+1}h^2\right)\ y_1^{m+1} + y_2^{m+1} &= -4 \\
&\vdots \\
y_{i-1}^{m+1} - \left(2+\frac{3}{2}y_i^{m+1}h^2\right)\ y_i^{m+1} + y_{i+1}^{m+1} &= 0 \\
&\vdots \\
y_{N-1}^{m+1} - \left(2+\frac{3}{2}y_N^{m+1}h^2\right)\ y_N^{m+1} &= -1
\end{aligned}
\tag{4.95}
$$

where $i = 2, 3, ...N-1$, and the iteration counter $m = 0, 1, 2, \ldots$. The linearization is performed by replacing $\frac{3}{2}h^2y_1^{m+1}$, $\frac{3}{2}h^2y_i^{m+1}$ and $\frac{3}{2}h^2y_N^{m+1}$ by $\frac{3}{2}h^2y_1^m$, $\frac{3}{2}h^2y_i^m$ and $\frac{3}{2}h^2y_N^m$ such that the following *linear system* is obtained:

$$
\begin{aligned}
-\left(2+\frac{3}{2}y_1^m h^2\right)\ y_1^{m+1} + y_2^{m+1} &= -4 \\
&\vdots \\
y_{i-1}^{m+1} - \left(2+\frac{3}{2}y_i^m h^2\right)\ y_i^{m+1} + y_{i+1}^{m+1} &= 0 \\
&\vdots \\
y_{N-1}^{m+1} - \left(2+\frac{3}{2}y_N^m h^2\right)\ y_N^{m+1} &= -1
\end{aligned}
\tag{4.96}
$$

We have with the procedure above discretized our analytical problem in equation (4.93) to a *linear*, tridiagonal system which may readly be solved with sparse solvers like the ones we introduced in 4.4.1.

Note that all values at iteration level m, i.e. y_i^m are *known* and the only unknowns in (4.96) are y_i^{m-1}. Thus, to start the iteration procedure we need an initial value for y_i^0. Unless a well informed initial guess is available, it is common to start with $y_i^0 = 0$, $i = 1, 2, ...N$. Naturally, initial values *close* to the numerical solution will result in faster convergence.

Additionally, we must test for *diagonal dominance* of the iteration scheme, which for equation (4.96) means:

$$
\left|2+\frac{3}{2}y_i^m h^2\right| \geq 2\ ,\ \ i = 1, 2, ..., N \tag{4.97}
$$

We observe from equation (4.97) that we have a diagonally dominant iteration matrix when $y_i^m > 0$. By inspection the analytical solutions in Figure 3.8, we observe that this condition is fulfilled for only one of the two solutions. In cases when the condition of diagonal dominance is *not* fulfilled, make sure your solver allows for pivotation, e.g. like `scipy.sparse.linalg` as demonstrated in 4.4.1. However, for demonstration purposes we have in this example implemented a version of the classical tri-diagonal solver `tdma` along with a version with pivotation `tripiv` in the module `TRIdiagonalSolvers`.

In cases where the one has no experience with the itereation process, e.g. while impelmenting, one may find it beneficial to use the number of iterations as a stopping critera, rather than an error/residual based stop criterion. However, after you have gained some experience with the particular problem at hand you may use other stop critera as demonstrated in section 4.5.4.

In the python-code `delay34.py` below we have implemented the procedure of Picard linearization as outlined above.

```
# src-ch3/delay34.py;TRIdiagonalSolvers.py @ git@lrhgit/tkt4140/src/src-ch3/TRIdiagonalSolvers.py;

import numpy as np
from matplotlib.pyplot import *
```

```
from TRIdiagonalSolvers import tdma
from TRIdiagonalSolvers import tripiv


# change some default values to make plots more readable
LNWDT=3; FNT=11
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT
font = {'size' : 16}; rc('font', **font)

h = 0.05 # steplength
n = int(round(1./h)-1) # number of equations
fac = (3./2.)*h**2
Nmax = 30 # max number of iterations

# iterative solution with Nmax iterations
#initilizing:
ym = np.zeros(n) # initial guess of y. ym = np.zeros(n) will converge to y1. ym = -20*x*(1-x) for in

for m in range(Nmax):
    """iteration process of linearized system of equations using delay method"""
    # need to set a, b, c and d vector inside for loop since indices are changed in tdma solver
    a = np.ones(n)
    c = np.ones(n)
    b = -(np.ones(n)*2. + fac*ym)
    d = np.zeros(n)
    d[n-1] = -1.
    d[0] = -4.

    ym1 = tdma(a,b,c,d) # solution

    if(m>0):
        max_dym  = np.max(np.abs((ym1 - ym)/ym))
        print('m = {} \t  max dym = {:6.4g} '.format(m, max_dym))

    ym = ym1


# compute analytical solution
xv = np.linspace(h, 1 - h, n)
ya = 4./(1 + xv)**2
error = np.abs((ym1 - ya)/ya)

print('\nThe max relative error after {} iterations is: {:4.3g}'.format(Nmax,np.max(error)))
```

The maximal relative deviation between consecutive iterations is stored in the variable `max_dym`. We observe that `max_dym` decreases for each iteration as the solution slowly converges to the analytical solution $y_I = \frac{4}{(1+x)^2}$ which was demonstrated in (3.8) in section (3.2).

After `Nmax` iterations we compute the `error` by :

```
error = np.abs((ym1 - ya)/ya)
```

Note that `error` is a vector by construction, since `ym1` and `ya` are vectors. Thus, to print a scalar measure of the error after Nmax iterations we issue the following command:

```
print('\nThe max relative error after {} iterations is: {:4.3g}'.format(Nmax,np.max(error)))
```

The method of Picard linearization.

The advantage with the method of Picard linearization, is that the linearization process is very simple. However, the simplicity comes at a prize, since it converges slowly and often requires a relatively good initial guess.

Once we have gained some experience on how the iteration procedure develops, we may make use of a stop criterion based on relative changes in the iterative solutions as shown in the python-snippet below:

```
# iterative solution with max_dym stop criterion

m=0; RelTol=1.0e-8
max_dym=1.0

ym = np.zeros(n) # initial guess of y. ym = np.zeros(n) will converge to y1. ym = -20*x*(1-x) for in

a = np.ones(n)
c = np.ones(n)

while (m<Nmax and max_dym>RelTol):

    d = np.zeros(n)
    d[n-1] = -1.
    d[0] = -4.

    b = -(np.ones(n)*2. + fac*ym)

    ym1 = tdma(a,b,c,d) # solution

    if(m>0):
        max_dym  = np.max(np.abs((ym1 - ym)/ym))
        print('m = {} \t  max dym = {:6.4g} '.format(m, max_dym))

    m+=1
    ym = ym1


error = np.abs((ym1 - ya)/ya)
print('\nThe max relative after {} iterations is: {:4.3g}'.format(m,np.max(error)))

# print results nicely with pandas
print_results=0
if (print_results):
    import pandas as pd
    data=np.column_stack((xv,ym1,ya))
    data_labeled = pd.DataFrame(data, columns=['xv','ym','ya'])
    print(data_labeled.round(3).to_string(index=False))

# plotting:
legends=[] # empty list to append legends as plots are generated
# Add the labels
plot(xv,ya,"r") # plot analytical solution
legends.append('analytic')
plot(xv,ym1,"b--") # plot numerical solution
legends.append('delayed linearization')
```

```
legend(legends,loc='best',frameon=False) # Add the legends
ylabel('y')
xlabel('x')
#grid(b=True, which='both', axis='both',linestyle='-')
grid(b=True, which='both', color='0.65',linestyle='-')

show()
```

Notice that we must set `max_dym>RelTol` prior to the `while-loop` to start the iterative process.

You may experiment with the code above to find that for a fixed mesh size `h=0.05`, the relative error will not be smaller than $\approx 5.6 \cdot 10^{-4}$, regardless of how low you set `RelTol`, i.e. after a converged solution is obtained for the given mesh size. Naturally, the error will be reduced if you reduce the mesh size `h`. Observe also that the number of iterations needed to obtain a converged solution for a given mesh size of `h=0.05` is smaller than the `Nmax=30`, which was our original upper limit.

4.5.2 Newton linearization

Motivating example of linearization Let us look at a motivating example which is simple to use for nonlinearities in products. For convenience we introduce δy_i as the the increment between two consecutive y_i-values:

$$y_i^{m+1} = y_i^m + \delta y_i \tag{4.98}$$

By assuming δy_i to be small, the nonlinear term in equation (4.93) may then be linearized using (4.98) in the following way:

$$\left(y_i^{m+1}\right)^2 = (y_i^m + \delta y_i)^2 = (y_i^m)^2 + 2y_i^m\,\delta y_i + (\delta y_i^m)^2 \approx (y_i^m)^2 + 2\,y_i^m\,\delta y_i = y_i^m\,\left(2y_i^{m+1} - y_i^m\right) \tag{4.99}$$

That is, quadratic expression in equation (4.99) is linearized by neglecting the quadratic terms $(\delta y)^2$ which are small compared to the other terms in (4.99).

Substitution of the linearizition (4.99) in (4.94) results in the following system of equations:

$$\begin{aligned}
-(2+3y_1^m h^2)\,y_1^{m+1} + y_2^{m+1} &= -\tfrac{3}{2}\,(y_i^m h)^2 - 4 \\
&\vdots \\
y_{i-1}^{m+1} - (2+3y_i^m h^2)\,y_i^{m+1} + y_{i+1}^{m+1} &= -\tfrac{3}{2}\,(y_i^m h)^2 \\
&\vdots \\
y_{N-1}^{m+1} - (2+3y_N^m h^2)\,y_N^{m+1} &= -\tfrac{3}{2}\,(y_N^m h)^2 - 1
\end{aligned} \tag{4.100}$$

where $i = 1, 2, \ldots, N-1$ and $m = 0, 1, \ldots$.

The resulting equation system (4.100), is a tridiagonal system which may be solved with the tmda-solver as for the previous example. An implementation is given below:

```
# src-ch3/taylor34.py;TRIdiagonalSolvers.py @ git@lrhgit/tkt4140/src/src-ch3/TRIdiagonalSolvers.py;


import numpy as np
from matplotlib.pyplot import *
from TRIdiagonalSolvers import tdma
from TRIdiagonalSolvers import tripiv
# change some default values to make plots more readable
LNWDT=3; FNT=11
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT
font = {'size' : 16}; rc('font', **font)

h = 0.05 # steplength
n = int(round(1./h)-1) # number of equations
fac = (3.)*h**2

#initilizing:
x = np.linspace(h,1-h,n)
ym = -20*x*(1-x) # initial guess of y.

#ym = np.zeros(n) # will converge to y1. ym = -20*x*(1-x) for instance will converge to y2


it, itmax, dymax, RelTol = 0, 15, 1., 10**-10
legends=[]
while (dymax > RelTol) and (it < itmax):
    """iteration process of linearized system of equations using taylor
    """
    plot(x,ym) # plot ym for iteration No. it
    legends.append(str(it))
    it = it + 1
    a = np.ones(n)
    c = np.ones(n)
    b = -(np.ones(n)*2. + fac*ym)
    d = -(fac*0.5*ym**2)
    d[n-1] = d[n-1]-1
    d[0] = d[0]-4

    ym1 = tdma(a,b,c,d) # solution
    dymax  = np.max(np.abs((ym1-ym)/ym))
    ym = ym1

    print('it = {},  dymax = {} '.format(it, dymax))

legend(legends,loc='best',frameon=False)


ya = 4./(1+x)**2
feil = np.abs((ym1-ya)/ya)
print "\n"

for l in range(len(x)):
    print 'x = {},  y = {},  ya = {}'.format(x[l], ym1[l], ya[l])


figure()

# plotting:
legends=[] # empty list to append legends as plots are generated
# Add the labels
plot(x,ya,"r") # plot analytical solution
```

```
legends.append('y1 analytic')
plot(x,ym1,"b--") # plot numerical solution
legends.append('y2 taylor linearization')

legend(legends,loc='best',frameon=False) # Add the legends
ylabel('y')
xlabel('x')
#grid(b=True, which='both', axis='both',linestyle='-')
grid(b=True, which='both', color='0.65',linestyle='-')

show()
```

You may experiment with by changing the initial, guessed solution, and observe what happens with the resulting converged solution. Which solution does the numerical solution converge to?

Observe that the iteration process converges much faster than for the Picard linearization approach. Notably, somewhat slow in the first iterations, but after some iterations quadratic convergence is obtained.

Generalization: Newton linearization The linearization in (4.98) and (4.99) may seen as truncated Taylor expansions of the dependent variable at iteration m, where only the first two terms are retained.

To generalize, we let F denote the nonlinear term to be linearized and assume F to be a function of the *dependent* variable z at the discrete point x_i (or i for short) at iteration m.

We will linearize $F(z_i)\big|^{m+1} \equiv F(z_i)_{m+1}$, and use the latter notation in the following. A Taylor expansion may then be expressed at the point i and iteration m

$$F(z_i)_{m+1} \approx F(z_i)_m + \delta z_i \left(\frac{\partial F}{dz_i}\right)_m \tag{4.101}$$

where $\delta z_i = z_i^{m+1} - z_i^m$.

Our simple example in equation (4.99) may be rendered in this slightly more generic representation:

$$\delta z_i \to \delta y_i, \qquad z_i^{m+1} \to y_i^{m+1}, \qquad z_i^m \to y_i^m, \qquad F(y_i)_{m+1} = (y_i^{m+1})^2 \ , \ F(y_i)_m = (y_i^m)^2$$

which after substitution in (4.101) yields: $(y_i^{m+1})^2 \approx (y_i^m)^2 + 2y_i^m \cdot \delta y_i$ which is identical to what was achieved in equaiton (4.99) .

The linearization in equation (4.101) is often referred to as *Newton-linearization*.

In many applications, the nonlinear term is composed z at various discrete, spatial locations and this has to be accounted for in the linearization. Assume to begin with that we have terms involving both z_i og z_{i+1}. The Taylor expansion will then take the form:

$$F(z_i, z_{i+1})_{m+1} \approx F(z_i, z_{i+1})_m + \left(\frac{\partial F}{\partial z_i}\right)_m \delta z_i + \left(\frac{\partial F}{\partial z_{i+1}}\right)_m \delta z_{i+1} \tag{4.102}$$

where:

$$\delta z_i = z_i^{m+1} - z_i^m, \qquad \delta z_{i+1} = z_{i+1}^{m+1} - z_{i+1}^m$$

When three spatial indices, e.g. z_{i-1}, z_i og z_{i+1}, are invoved, the Taylor expansion becomes:

$$F(z_{i-1}, z_i, z_{i+1})_{m+1} \approx F(z_{i-1}, z_i, z_{i+1})_m + \left(\frac{\partial F}{\partial z_{i-1}}\right)_m \delta z_{i-1} + \left(\frac{\partial F}{\partial z_i}\right)_m \delta z_i + \left(\frac{\partial F}{\partial z_{i+1}}\right)_m \delta z_{i+1} \tag{4.103}$$

where

$$\delta z_{i-1} = z_{i-1}^{m+1} - z_{i-1}^m, \qquad \delta z_i = z_i^{m+1} - z_i^m, \qquad \delta z_{i+1} = z_{i+1}^{m+1} - z_{i+1}^m$$

Naturally, the same procedure may be expaned whenever the nonlinear term is composed of instances of the dependent variable at more spatial locations.

4.5.3 Example: Newton linearization of various nonlinear ODEs

Nonlinear ODE 1 In this example of a nonlinear ODE we consider:

$$y''(x) + y(x)\sqrt{y(x)} = 0 \tag{4.104}$$

which after discretization with central differences may be presented:

$$y_{i+1}^{m+1} - 2y_i^{m+1} + y_{i-1}^{m+1} + h^2 y_i^{m+1}\sqrt{y_i^{m+1}} = 0 \tag{4.105}$$

In equation (4.105) it is the nonlinear term $y_i^{m+1}\sqrt{y_i^{m+1}}$ which must be linearized. As we in this example only have one index we make use of the simplest Taylor expansion (4.101) with $z_i \to y_i$ and $F(y_i) = y_i^{\frac{3}{2}}$:

$$F(y_i)_{m+1} \approx F(y_i)_m + \delta y_i \left(\frac{\partial F}{\partial y_i}\right)_m = (y_i^m)^{\frac{3}{2}} + \delta y_i \, \frac{3}{2} \, (y_i^m)^{\frac{1}{2}} \tag{4.106}$$

where $\delta y_i = y_i^{m+1} - y_i^m$. Equivalently by using the squareroot-symbols:

$$y_i^{m+1}\sqrt{y_i^{m+1}} \approx y_i^m \sqrt{y_i^m} + \frac{3}{2}\sqrt{y_i^m}\,\delta y_i \tag{4.107}$$

Observe that we have succeeded in the linearization process as all occurences of y_i^{m+1} in the two equivalent discretization in equation (4.106) and (4.107) are of power one.

After substition in equaiton (4.105) the following *linear* difference equation results:

$$y_{i-1}^{m+1} + \left(\frac{3}{2}h^2\sqrt{y_i^m} - 2\right) y_i^{m+1} + y_{i+1}^{m+1} = \frac{h^2}{2} y_i^m \sqrt{y_i^m} \tag{4.108}$$

Nonlinear ODE 2 As a second example of a nonlinear ODE consider:

$$y''(x) + \sin\left(y(x)\right) = 0 \tag{4.109}$$

which after discretization by central differences may be rendered:

$$y_{i+1}^{m+1} - 2y_i^{m+1} + y_{i-1}^{m+1} + h^2 \sin\left(y_i^{m+1}\right) = 0 \tag{4.110}$$

In equation (4.110) we identify $\sin\left(y_i^{m+1}\right)$ as the nonlinear term in need of linearization. Againg, we have only one index and may make use of (4.101) with $z_i \to y_i$ and $F(y_i) = \sin(y_i)$:

$$F(y_i)_{m+1} = \sin(y_i^{m+1}) \approx F(y_i)_m + \delta y_i \left(\frac{\partial F}{\partial y_i}\right)_m = \sin(y_i^m) + \delta y_i \ \cos(y_i^m)$$

which after substitution in equation (4.110) leads to the following difference equation:

$$y_{i-1}^{m+1} + (h^2\cos(y_i^m) - 2)y_i^{m+1} + y_{i+1}^{m+1} = h^2(y_i^m\cos(y_i^m) - \sin(y_i^m)) \tag{4.111}$$

Nonlinear ODE 3 Consider the ODE $y''(x) + y(x)\sqrt{y'(x)} = 0$ which after discretization with central differences may be presented:

$$y_{i+1}^{m+1} - 2y_i^{m+1} + y_{i-1}^{m+1} + \alpha\ y_i^{m+1}\sqrt{y_{i+1}^{m+1} - y_{i-1}^{m+1}} = 0\ ,\ \text{der}\ \alpha = h\sqrt{h/2} \tag{4.112}$$

The term $y_i^{m+1}\sqrt{y_{i+1}^{m+1} - y_{i-1}^{m+1}}$ is nonlinear and must be linearized. As three indices are involved in the term we utilize (4.103) with $z_{i-1} \to y_{i-1},\ \ z_i \to y_i$, $z_{i+1} \to y_{i+1}$, and finally $F(y_{i-1}, y_i, y_{i+1})_m = y_i\sqrt{y_{i+1} - y_{i-1}}$.

We present the individual terms in equation(4.103) :

$$F(y_{i-1}, y_i, y_{i+1})_m = y_i^m\sqrt{y_{i+1}^m - y_{i-1}^m}, \quad \left(\frac{\partial F}{\partial y_{i-1}}\right)_m = -\frac{y_i^m}{2\sqrt{y_{i+1}^m - y_{i-1}^m}}$$
$$\left(\frac{\partial F}{\partial y_i}\right)_m = \sqrt{y_{i+1}^m - y_{i-1}^m}, \qquad \left(\frac{\partial F}{\partial y_{i+1}}\right)_m = \frac{y_i^m}{2\sqrt{y_{i+1}^m - y_{i-1}^m}}$$

By collecting terms we get:

$$y_i^{m+1}\sqrt{y_{i+1}^{m+1} - y_{i-1}^{m+1}} \approx y_i^m\sqrt{y_{i+1}^m - y_{i-1}^m} - \frac{y_i^m}{2\sqrt{y_{i+1}^m - y_{i-1}^m}}\ \delta y_{i-1} + \sqrt{y_{i+1}^m - y_{i-1}^m}\ \delta y_i + \frac{y_i^m}{2\sqrt{y_{i+1}^m - y_{i-1}^m}}\ \delta y_{i+1} \tag{4.113}$$

Note that equation (4.113) is linear as y_{i-1}^{m+1}, y_i^{m+1} and y_{i+1}^{m+1} are only in first power. By substitution in equaiton (4.112) we get the linear disicretized equation:

$$\begin{aligned}\left(1-\alpha\tfrac{y_i^m}{2g^m}\right)\ y_{i-1}^{m+1}-(2-\alpha g^m)\ y_i^{m+1}+\left(1+\alpha\tfrac{y_i^m}{2g^m}\right)\ y_{i+1}^{m+1}\\ =\alpha\tfrac{y_i^m}{2g^m}(y_{i+1}^m-y_{i-1}^m)\\ \text{der } g^m=\sqrt{y_{i+1}^m-y_{i-1}^m}\end{aligned}\tag{4.114}$$

In the next section we will show how we can operate diretly with the incremental quantities δy_{i-1}^{m+1}, δy_i^{m+1} and δy_{i+1}^{m+1}.

Comparison with the method of Picard linearization

How would the examples above be presented with the method of Picard linearization?

Equation (4.105) takes the following form the Picard linearization:

$$y_{i-1}^{m+1}+(h^2\sqrt{y_i^{m+1}}-2)\ y_i^{m+1}+y_{i-1}^{m+1}=0\tag{4.115}$$

The coefficient for the y_i^{m+1}-term contains $\sqrt{y_i^{m+1}}$ which will be replaced by $\sqrt{y_i^m}$ such that:

$$y_{i-1}^{m+1}+(h^2\sqrt{y_i^m}-2)\ y_i^{m+1}+y_{i-1}^{m+1}=0\tag{4.116}$$

For Picard linearization equation (4.110) become:

$$y_{i-1}^{m+1}-2y_i^{m+1}+y_{i+1}^{m+1}=-h^2\sin(y_i^m)=0\tag{4.117}$$

whereas equation (4.112) looks like:

$$y_{i-1}^{m+1}+(\alpha\ \sqrt{y_{i+1}^{m+1}-y_{i-1}^{m+1}}-2)\ y_i^{m+1}+y_{i+1}^{m+1}=0\tag{4.118}$$

with Picard linearization. The coefficient in front of the y_i^{m+1}-term contains $\sqrt{y_{i+1}^{m+1}-y_{i-1}^{m+1}}$ which is replaced by $\sqrt{y_{i+1}^m-y_{i-1}^m}$ such that the equation becomes:

$$y_{i-1}^{m+1}+(\alpha\ \sqrt{y_{i+1}^m-y_{i-1}^m}-2)\ y_i^{m+1}+y_{i+1}^{m+1}=0$$

When we use the method of Picard linearization, the system first has to be presented in a form which identifies the coefficients in the equation system. If the coefficients have dependent variable at iteration $m+1$, the are to be replaced with the corresponding value at iteration m. Often this procedure is identical to a Taylor expansion truncated after the first term.

Thus, we may expect that the method of Picard linearization will converge more slowly compared to methods which use more terms in the Taylor expansion.

The method of Picard linearization is most frequently used for partial differential equations where e.g. the material parameters are functions of the dependent variables, such as temperature dependent heat conduction.

4.5.4 Various stop criteria

We let δy_i denote the increment between two consecutive iterations:

$$\delta y_i = y_i^{m+1} - y_i^m \ , \ i = 1, 2, ..., N, \qquad m = 0, 1, ... \tag{4.119}$$

where N denotes the number of physical grid points. Two classes of stop criteria based on absolute and relative metrics, may then be formulated as outlined in the following.

Stop criteria based on absolute values.

$$\max(|\delta y_i|) < \varepsilon_a \tag{4.120}$$

$$\frac{1}{n}\sum_{i=1}^{N} |\delta y_i| < \varepsilon_a \tag{4.121}$$

$$\frac{1}{n}\sqrt{\sum_{i=1}^{N} (\delta y_i)^2} < \varepsilon_a \tag{4.122}$$

where ε_a is a suitable small number.

Stop criteria based on relative values

$$\max\left(\left|\frac{\delta y_i}{y_i^{m+1}}\right|\right) < \varepsilon_r, \qquad y_i^{m+1} \neq 0 \tag{4.123}$$

$$\frac{\sum_{i=1}^{N} |\delta y_i|}{\sum_{i=1}^{N} |y_i^{m+1}|} < \varepsilon_r \tag{4.124}$$

$$\frac{\max(|\delta y_i|)}{\max(|y_i^{m+1}|)} < \varepsilon_r \tag{4.125}$$

Note that if the quantity of interest, i.e. y_i, is of order 1, the absolute and the relative criteria are equivalent. Normally we will prefer a relative criterion as these comply with the expression *number of correct* digits. If cases where the values of y_i are small in the whole computational domain (i.e. for all i) we may resort to a absolute value crterion

4.5.5 Example: Usage of the various stop criteria

In this example we will demonstrate the use of the various stop criteria outlined in the previous section on the problem in section (3.2), where we will compute the second solution y_{II}) as illustrated in Figure (3.8).

For convenience we reformulate the ODE (4.90) :

$$y''(x) = \frac{3}{2}y^2 \tag{4.126}$$

with the boundary condtions:

$$y(0) = 4, \qquad y(1) = 1 \tag{4.127}$$

We let our initial guess be expressed by the parabola $y_s = 20(x - x^2)$, $0 < x < 1$, which is plotted along with the solution y_{II} , in Figure (4.9).

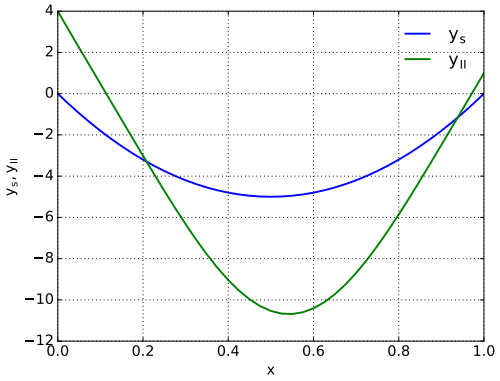

Figure 4.9: Intial parabolic guess and solution y_{II}.

For this example it is natural to use a relative stop criterion as we know that the solution has values in the range $[4, -10.68]$. After nine iterations the following table results: tabell:

Iter.nr.	t_{r2}	t_{r3}
1	$7.25 \cdot 10^{-1}$	$7.55 \cdot 10^{-1}$
2	$8.70 \cdot 10^{-1}$	$8.20 \cdot 10^{-1}$
3	$8.49 \cdot 10^{-1}$	$6.65 \cdot 10^{-1}$
4	$4.96 \cdot 10^{-1}$	$5.85 \cdot 10^{-1}$
5	$2.10 \cdot 10^{-1}$	$2.80 \cdot 10^{-1}$
6	$4.62 \cdot 10^{-2}$	$6.07 \cdot 10^{-2}$
7	$2.24 \cdot 10^{-3}$	$2.70 \cdot 10^{-3}$
8	$4.68 \cdot 10^{-6}$	$5.79 \cdot 10^{-6}$
9	$2.26 \cdot 10^{-11}$	$2.55 \cdot 10^{-11}$

The iteration evolution is representative for Newton-algorithms as the initial values are far away from the correct solution. The errors decreases slowly for the first six iterations, but afterwards we observe quadratic convergence. The two criteria do not differ significantly for this case.

Nedenfor vises listing av programmet **avvikr**.

A python code using a relative criteria is shown here:

```
# src-ch3/avvikr.py;TRIdiagonalSolvers.py @ git@lrhgit/tkt4140/src/src-ch3/TRIdiagonalSolvers.py;


import numpy as np
from matplotlib.pyplot import *
from TRIdiagonalSolvers import tdma
from TRIdiagonalSolvers import tripiv
# change some default values to make plots more readable
LNWDT=3; FNT=11
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT
font = {'size' : 16}; rc('font', **font)

# numerical parameters
h = 0.05 # steplength.
n = round(1./h)-1 # number of equations. h should be set so that n is an integer
fac = (3.)*h**2

#initilizing:
x = np.linspace(h,1-h,n)
ym = -20*x*(1-x) # initial guess of y. ym = np.zeros(n) will converge to y1. ym = -20*x*(1-x) for in

itmax = 15

for it in range(itmax):
    """iteration process of linearized system of equations"""
    # need to set a, b, c and d vector inside for loop since indices are changed in tdma solver
    a = np.ones(n)
    c = np.ones(n)
    b = -(np.ones(n)*2. + fac*ym)
    d = -(np.ones(n)*fac*0.5*ym**2)
    d[n-1] = -1.
    d[0] = -4.

    ym1 = tdma(a,b,c,d) # solution
    dy = np.abs(ym1-ym)
    tr2 = np.max(dy)/np.max(np.abs(ym1))
    tr3 = np.sum(dy)/np.sum(np.abs(ym1))
    ym = ym1

    print 'it = {},  tr2 = {},  tr3 = {} '.format(it, tr2, tr3)

ya = 4./(1+x)**2
feil = np.abs((ym1-ya)/ya)


# plotting:
legends=[] # empty list to append legends as plots are generated
# Add the labels
plot(x,ym1,"b") # plot numerical solution
legends.append('y2 numerical')
plot(x,ya,"r") # plot analytical solution
legends.append('y1 analytic')
```

```
legend(legends,loc='best',frameon=False) # Add the legends
ylabel('y')
xlabel('x')
grid(b=True, which='both', color='0.65',linestyle='-')

show()
```

A code where the absolute criteria is used is almost identical, except for the substitution of t_{r2} og t_{r3}

```
   ta1 = max(dy);
   ta2 = sum(dy)/n;
   ta3 = sqrt(dot(dy,dy))/n;
```

```
# src-ch3/avvika.py;TRIdiagonalSolvers.py @ git@lrhgit/tkt4140/src/src-ch3/TRIdiagonalSolvers.py;


import numpy as np
from matplotlib.pyplot import *
from TRIdiagonalSolvers import tdma
from TRIdiagonalSolvers import tripiv
# change some default values to make plots more readable
LNWDT=3; FNT=11
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT
font = {'size' : 16}; rc('font', **font)

# numerical parameters
h = 0.05 # steplength.
n = round(1./h)-1 # number of equations. h should be set so that n is an integer
fac = (3.)*h**2

#initilizing:
x = np.linspace(h,1-h,n)
ym = -20*x*(1-x) # initial guess of y. ym = np.zeros(n) will converge to y1. ym = -20*x*(1-x) for in

itmax = 15

for it in range(itmax):
    """iteration process of linearized system of equations"""
    # need to set a, b, c and d vector inside for loop since indices are changed in tdma solver
    a = np.ones(n)
    c = np.ones(n)
    b = -(np.ones(n)*2. + fac*ym)
    d = -(np.ones(n)*fac*0.5*ym**2)
    d[n-1] = -1.
    d[0] = -4.

    ym1 = tdma(a,b,c,d) # solution
    dy = np.abs(ym1-ym)
    ta1 = np.max(dy) # abs stop critertia 1
    ta2 = np.sum(dy)/n # abs stop critertia 2
    ta3 = np.sqrt(np.dot(dy,dy))/n # abs stop critertia 3

    ym = ym1

    print 'it = {},  ta2 = {},  ta3 = {} '.format(it, ta2, ta3)


xv = np.linspace(h,1-h,n)
ya = 4./(1+xv)**2
```

```
# plotting:
legends=[] # empty list to append legends as plots are generated
# Add the labels
plot(xv,ym1,"b")
legends.append('y2 numerical')
#plot(xv,ya,"r")
#legends.append('y1 analytic')
legend(legends,loc='best',frameon=False) # Add the legends
ylabel('y')
xlabel('x')
#grid(b=True, which='both', axis='both',linestyle='-')
grid(b=True, which='both', color='0.65',linestyle='-')

show()
```

4.5.6 Linerarizations of nonlinear equations on incremental form

As seen previously in equation (4.99), we may introduce the increment $\delta y_i = y_i^{m+1} - y_i^m$ in a nonlinear term as examplified by:

$$(y_i^{m+1})^2 = (y_i^m + \delta y_i)^2 = (y_i^m)^2 + 2y_i^m\ \delta y_i + (\delta y_i)^2 \tag{4.128}$$

$$\approx (y_i^m)^2 + 2y_i^m\ \delta y_i = y_i^m(2y_i^{m+1} - y_i^m) \tag{4.129}$$

i.e. we linearize by ingnoring higher order terms of the increments, and solve equation (4.129) with respect to y_i^{m+1}.

Alternatively, we may think of δy_i as the unknown quantity and solve the resulting equation system with respect to the increments. For example we may substitute $y_k^{m+1} = y_k^m + \delta y_k$ for $k = i-1,\ i,\ i+1$ and $(y_i^{m+1})^2 \approx (y_i^m)^2 + 2\, y_i^m\ \delta y_i$ into equation (4.95) (or equivalently (4.100)), the following equation system results:

$$\begin{aligned}
-(2 + 3y_1^m h^2)\ \delta y_1 + \delta y_2 &= -(y_2^m - 2y_1^m + 4) + \frac{3}{2}\,(y_1^m h)^2 \\
\delta y_{i-1} - (2 + 3y_i^m h^2)\ \delta y_i + \delta y_{i+1} &= -(y_{i+1}^m - 2y_i^m + y_{i-1}^m) + \frac{3}{2}\,(y_i^m h)^2 \\
\delta y_{N-1} - (2 + 3y_N^m h^2)\ \delta y_N &= -(1 - 2y_N^m + y_{N-1}^m) + \frac{3}{2}\,(y_N^m h)^2
\end{aligned} \tag{4.130}$$

where $i = 2, 3, \ldots, N-1$ adn $m = 0, 1, 2, \ldots$. In the equation system above we have incorporated the boundary conditions $y_0 = 4$, $\delta y_0 = 0$, $y_{N+1} = 1$, and $\delta y_{N+1} = 0$. For each iteration we update the y-values by:

$$y_i^{m+1} = y_i^m + \delta y_i, \qquad i = 1, 2, \ldots, N \tag{4.131}$$

After having gained some experience with how the iteration proceed, we may use stop criteria such ad $\max|\delta y_i| < \varepsilon_1$ or $\max|\delta y_i / y_i^{m+1}| < \varepsilon_2$, $i = 1, 2, \ldots, N$.

We have implemented the incremental formulation in the code below.

```
# src-ch3/delta34.py;ODEschemes.py @ git@lrhgit/tkt4140/src/src-ch3/ODEschemes.py;


import numpy as np
from matplotlib.pyplot import *
from TRIdiagonalSolvers import tdma
from TRIdiagonalSolvers import tripiv
# change some default values to make plots more readable
LNWDT=3; FNT=11
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT
font = {'size' : 16}; rc('font', **font)

h = 0.05 # steplength
n = int(round(1./h)-1) # number of equations
fac = (3.)*h**2
nitr = 6 # number of iterations

#initilizing:

ym = np.zeros(n) # initial guess of y. ym = np.zeros(n) will converge to y1. ym = -20*x*(1-x) for in

it, itmax, dymax, relTol = 0, 10, 1., 10**-5 # numerical tollerance limits

while (dymax > relTol) and (it < itmax):
    """iteration process of linearized system of equations"""
    it = it + 1
    a = np.ones(n)
    c = np.ones(n)
    b = -(np.ones(n)*2. + fac*ym)
    d = (np.ones(n)*fac*0.5*ym**2)
    for j in range(1,n-1):
        d[j] = d[j] - (ym[j+1] - 2*ym[j] +ym[j-1])

    d[n-1] = d[n-1]-(1 - 2*ym[n-1] + ym[n-2])
    d[0] = d[0] -(ym[1] - 2*ym[0] + 4)

    dy = tdma(a,b,c,d) # solution
    ym = ym + dy
    dymax  = np.max(np.abs((dy)/ym))


    print 'it = {},  dymax = {} '.format(it, dymax)


xv = np.linspace(h,1-h,n)
ya = 4./(1+xv)**2

print "\n"

for l in range(len(xv)):
    print 'x = {},  y = {},  ya = {}'.format(xv[l], ym[l], ya[l])


legends=[] # empty list to append legends as plots are generated

plot(xv,ya,"r") # plot analytical solution
legends.append('y1 analytic')
plot(xv,ym,"b--") # plot numerical solution
legends.append('delta linearization')
# Add the labels
legend(legends,loc='best',frameon=False) # Add the legends
```

```
ylabel('y')
xlabel('x')
#grid(b=True, which='both', axis='both',linestyle='-')
grid(b=True, which='both', color='0.65',linestyle='-')

show()
```

4.5.7 Quasi-linearization

In previous examples, we have always linearized the discretization of the original problem. However, it is possible to first linearize the differential equation and only succesively discretize it. This method is normally called quasi-linearization. Let us now look at an example given by a second order non linear equation:

$$y''(x) = f(x, y, y') \tag{4.132}$$

We now write (4.132) as:

$$g(x, y, y', y'') \equiv y''(x) - f(x, y, y') = 0 \tag{4.133}$$

and setting

$$\delta y = y_{m+1} - y_m \ , \ \delta y' = y'_{m+1} - y'_m \ , \ \delta y'' = y''_{m+1} - y''_m \tag{4.134}$$

where m stands for the iteration number (in this section we will be using sub-scripts for iteration number)

By expanding (4.133) around iteration m we get:

$$g(x, y_{m+1}, y'_{m+1}, y''_{m+1}) \approx g(x, y_m, y'_m, y''_m) + \left(\frac{\partial g}{\partial y}\right)_m \delta y + \left(\frac{\partial g}{\partial y'}\right)_m \delta y' + \left(\frac{\partial g}{\partial y''}\right)_m \delta y'' \tag{4.135}$$

Suppose that we have iterated so many times that:

$$g(x, y_{m+1}, y'_{m+1}, y''_{m+1}) \approx g(x, y_m, y'_m, y''_m) \approx 0$$

which inserted in (4.135) gives:

$$\left(\frac{\partial g}{\partial y}\right)_m \delta y + \left(\frac{\partial g}{\partial y'}\right)_m \delta y' + \left(\frac{\partial g}{\partial y''}\right)_m \delta y'' = 0 \tag{4.136}$$

After derivation of (4.133):

$$\frac{\partial g}{\partial y} = -\frac{\partial f}{\partial y} \ , \ \frac{\partial g}{\partial y'} = -\frac{\partial f}{\partial y'} \ , \ \frac{\partial g}{\partial y''} = 1 \tag{4.137}$$

which inserted in (4.136) gives:

$$\delta y'' = \left(\frac{\partial f}{\partial y}\right)_m \delta y + \left(\frac{\partial f}{\partial y'}\right)_m \delta y' \tag{4.138}$$

By inserting (4.134) in (4.138) and making use of (4.132), we get:

$$\begin{aligned} &y''_{m+1} - \left(\frac{\partial f}{\partial y'}\right)_m y'_{m+1} - \left(\frac{\partial f}{\partial y}\right)_m y_{m+1} \\ &= f(x, y_m, y'_m) - \left(\frac{\partial f}{\partial y}\right)_m y_m - \left(\frac{\partial f}{\partial y'}\right)_m y'_m \end{aligned} \tag{4.139}$$

Finally we re-write (4.139) with our regular notation, i.e. using super-scripts for iteration number:

$$\begin{aligned} &(y'')^{m+1} - \left(\frac{\partial f}{\partial y'}\right)_m (y')^{m+1} - \left(\frac{\partial f}{\partial y}\right)_m y^{m+1} \\ &= f(x, y^m, (y')^m) - \left(\frac{\partial f}{\partial y'}\right)_m (y')^m - \left(\frac{\partial f}{\partial y}\right)_m y^m \end{aligned} \tag{4.140}$$

We have used a second order equation as an example, but (4.138) - (4.140) allows for generalization to n-th order equations. Take for example a third order equation:

$$\begin{aligned} &(y''')^{m+1} - \left(\frac{\partial f}{\partial y''}\right) (y'')^{m+1} - \left(\frac{\partial f}{\partial y'}\right)_m (y')^{m+1} - \left(\frac{\partial f}{\partial y}\right)_m y^{m+1} \\ &= f(x, y^m, (y')^m, (y'')^m) - \left(\frac{\partial f}{\partial y''}\right)_m (y'')^m \\ &- \left(\frac{\partial f}{\partial y'}\right)_m (y')^m - \left(\frac{\partial f}{\partial y}\right)_m (y)^m \end{aligned} \tag{4.141}$$

4.5.8 Example:

1)

Consider the following equation

$$y''(x) = \frac{3}{2}y^2$$

One can easily check that $\frac{\partial f}{\partial y'} = 0$ and $\frac{\partial f}{\partial y} = 3y$. Inserting these terms in (4.140) gives:

$$(y'')^{m+1} - 3y^m \; y^{m+1} = -\frac{3}{2}(y^m)^2$$

By discretizing with central differences $y''_i \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}$ we get the following algebraic relation:

$$y_{i-1}^{m+1} - (2 + 3h^2 y_i^m) y_i^{m+1} + y_{i+1}^{m+1} = -\frac{3}{2}(h y_i^m)^2$$

which is in agreement with (4.100).

2)

Falkner-Skan equation reads:

$$y'' + y\, y'' + \beta\, [1 - (y')^2] = 0$$

We re-write the equation as:

$$y'' = -\left(y\, y'' + \beta\, [1 - (y')^2]\right) = f(x, y, y', y'')$$

Noting that in this case $\frac{\partial f}{\partial y''} = -y$, $\frac{\partial f}{\partial y'} = 2\beta y'$ and $\frac{\partial f}{\partial y} = -y''$, equation (4.141) gives:

$$\begin{aligned}(y''')^{m+1} + y^m\, (y'')^{m+1} - 2\beta (y')^m\, (y')^{m+1} + (y'')^m\, y^{m+1} \\ = -\beta \left(1 + [(y')^m]^2\right) + y^m\, (y'')^m\end{aligned}$$

Using central differences one can verify that the result is in agreement with equation (F.O.16) in Appendix F of the Numeriske Beregninger.

4.6 Exercises

Exercise 7: Circular clamped plate with concentrated single load

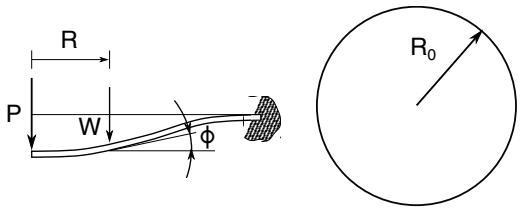

Figure 4.10: Schematic representation of a circular clamped plate with concentrated single load. The plate is clamped at $R = R_0$ and the load P is applied at $R = 0$.

Figure 4.10 show a rigidly clamped plate with radius R_0. The plate is loaded with a single load P in the plate centre. The differential equation for the deflection $W(R)$ is given by:

$$\frac{d^3W}{dR^3} + \frac{1}{R}\frac{d^2W}{dR^2} - \frac{1}{R^2}\frac{dW}{dR} = \frac{P}{2\pi D \cdot R} \tag{4.142}$$

The plate stifness D is given by: $D = \frac{Et^3}{12(1-\nu)}$, where E is the modulus of elasticity, ν is the Poissons ratio and t is the plate thickness. The boundary conditions are given by:

$$W(R_0) = \frac{dW}{dR}(R_0) = 0, \quad \frac{dW}{dR}(0) = 0 \tag{4.143}$$

In which the two first are because it is rigidly clamped, and the last due to the symmetry. We introduce dimmensionless variables: $r = \frac{R}{R_0}$, $\omega(R) = \frac{16\pi DW}{PR_0^2}$, so that Eq. (4.142) can be written:

$$\frac{d^3\omega}{dr^3} + \frac{1}{r}\frac{d^2\omega}{dr^2} - \frac{1}{r^2}\frac{d\omega}{dr} = \frac{8}{r}, \quad 0 < r < 1 \tag{4.144}$$

and Eq. (4.143):

$$\omega(1) = \frac{d\omega}{dr}(1) = 0, \quad \frac{d\omega}{dr}(0) = 0 \tag{4.145}$$

The analytical solution is given by:

$$\omega(r) = r^2\left[2\,ln(r) - 1\right] + 1, \quad \frac{d\omega}{dr}(r) = 4\,r\,ln(r) \tag{4.146}$$

Pen and paper:
The following problems should be done using pen and paper:
a) We introduce the inclination $\phi(r) = -\frac{d\omega}{dr}$ and insert into Eq. (4.144):

$$\frac{d^2\phi}{dr^2} + \frac{1}{r}\frac{d\phi}{dr} - \frac{\phi}{r^2} = -\frac{8}{r}, \tag{4.147}$$

with boundary conditions:

$$\phi(0) = \phi(1) = 0 \tag{4.148}$$

discretize Eq. (4.147) with central differences. Partion the interval $[0, 1]$ into N segments, so that $h = \Delta r = \frac{1}{N}$, which gives $r = h \cdot i$, $i = 0, 1, \ldots, N$. The coefficient matrix should be tridiagonal. Write out the discretized equation for $i = 1$, i and $i = N - 1$. Write the expressions for the diagonals, and the right hand side.

b) Choose $N = 4$ ($h = \frac{1}{4}$) in **a**) and solve the corresponding system of equations.

c) We will now find $\omega(r)$ by integrating the equation $\frac{d\omega}{dr} = -\phi(r)$ using Heuns method. Since the right hand side is independenpt of ω , Heuns method reduces to the trapes method. (The predictor is not necessary).

Find the $\omega-$ values in the points as in **b**).

d) Eq. (4.147) can also be written:

$$\frac{d}{dr}\left[\frac{1}{r}\frac{d}{dr}\left[r\cdot\phi(r)\right]\right] \tag{4.149}$$

Discretize Eq. (4.149) using Eq. (2.45). The coefficient matrix should be tridiagonal. Write out the discretized equation for $i = 1$, i and $i = N - 1$. Write the expressions for the diagonals, and the right hand side.

e) Solve Eq. (4.144) and (4.145) directly by introducing a new independant variable $z = ln(r)$ and show that Eq. (4.144) can be written: $\omega'''(z) - 2\omega''(z) = 8r^2 \equiv 8e^{2z}$. Next guess a particular solution on the form $\omega_p(z) = k \cdot z \cdot e^{2z}$, where k is a constant. Lastly decide the constants using (4.145).

Programing:

a) Write a python program that solves **b**) and **c**) from the **pen and paper** exercise, numerically. Experiment with finer segmentation. If you want you can download the python scelleton **clampedPlate.py** and fill in where applicable.

Hint 1. pen and paper:

b) Solutions: $\phi_1 = \frac{142}{105} = 1.3524$, $\phi_2 = \frac{48}{35} = 1.3714$, $\phi_3 = \frac{6}{7} = 0.8571$

c) Since ω_0 is unknown, but $\omega_4 = \phi_4 = 0$ is known, first find ω_3, then ω_2, ω_1 and ω_0. Use the ϕ-values obtained in **a**).

Solutions: $\omega_0 = \frac{94}{105} = 0.8953$, $\omega_1 = \frac{61}{84} = 0.7262$, $\omega_2 = \frac{27}{70} = 0.3857$, $\omega_2 = \frac{3}{28} = 0.1071$

e) Solution given in Eq. (4.146)

Hint 2. programming:

```
# src-ch2/clampedPlate.py

import scipy
import scipy.linalg
import scipy.sparse
import scipy.sparse.linalg

#import matplotlib; matplotlib.use('Qt4Agg')
import matplotlib.pylab as plt
#plt.get_current_fig_manager().window.raise_()
import numpy as np

#### set default plot values: ####
LNWDT=2; FNT=15
plt.rcParams['lines.linewidth'] = LNWDT; plt.rcParams['font.size'] = FNT


def solve_phi(N, h):

    i_vector = np.linspace(0, N, N + 1) # vector containing the indice number of all nodes
    i_mid = i_vector[1:-1] # vector containing the indice number of all interior nodes nodes

    # mainDiag = ?
```

```
    # subDiag = ?
    # superDiag = ?

    # RHS = ?

    # A = scipy.sparse.diags([?, ?, ?], [?, ?, ?], format='csc')

    # Phi = scipy.sparse.linalg.spsolve(A, RHS)
    # Phi = np.append(0, Phi)
    # Phi = np.append(Phi, 0)

    return Phi

def solve_omega(Phi, N, h):

    Omega = np.zeros_like(Phi)
    Omega[-1] = 0
    #for i in range(N - 1, -1, -1):
        # i takes the values N-2, N-3, .... , 1, 0
        #Omega[i] = ?

    return Omega


N = 3
r = np.linspace(0, 1, N + 1)
h = 1./N

Phi = solve_phi(N, h)

Omega = solve_omega(Phi, N, h)

Omega_analytic = r[1:]**2*(2*np.log(r[1:]) - 1) + 1
Omega_analytic = np.append(1, Omega_analytic) # log(0) not possible

Phi_analytic = - 4*r[1:]*np.log(r[1:])
Phi_analytic = np.append(0, Phi_analytic) # log(0) not possible


fig, ax = plt.subplots(2, 1, sharex=True, squeeze=False)

ax[0][0].plot(r, Phi, 'r')
ax[0][0].plot(r, Phi_analytic, 'k--')

ax[0][0].set_ylabel(r'$\phi$')

ax[1][0].plot(r, Omega, 'r')
ax[1][0].plot(r, Omega_analytic, 'k--')
ax[1][0].legend(['numerical', 'analytical'], frameon=False)
ax[1][0].set_ylabel(r'$\omega$')
ax[1][0].set_xlabel('r')

plt.show()
```

Exercise 8: Heat conduction in a triangle of two different materials

In this exercise we will look at a slightly more involved version of the trapezoidal cooling rib in 4.4.2 which is illustrated in Figure 4.11. This cooling rib is

composed av a triangle (part a) and a trapezoidal part (part b) which are made of materials of different heat properties. The purpose of this exercise is to illustrated how discontinuties in heat properties may be treated. We will find that that the temperature varies continuously along the rib, whereas the temperature gradient will have a discontinuity at the junction of the two materials.

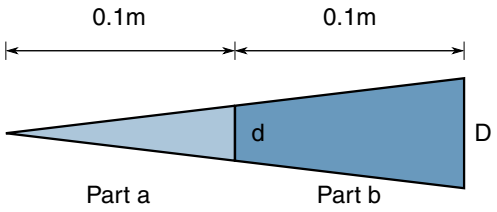

Figure 4.11: A triangle of two different materials.

In appendix B of Numeriske metoder the quasi one-dimensional differential equation for heat conduction is derived:

$$-dQ_x = P\ h\ [T(X) - T_\infty]\ dX \qquad \text{where} \qquad Q_x = -kA\frac{dT}{dX}$$

As $dX \to 0$ so will $dQ \to 0$ and consequently $\Rightarrow Q = constant$ which means that at the cross-section between the two different materials we must satisfy the following relation:

$$Q_a = Q_b \Rightarrow k_a A_a \left(\frac{dT}{dX}\right)_a = k_b A_b \left(\frac{dT}{dX}\right)_b \tag{4.150}$$

The the junction the cross-sectional areas of the two materials are the same $A_a = A_b$ and there fore $\left(\frac{dT}{dX}\right)_b = \frac{k_a}{k_b}\left(\frac{dT}{dX}\right)_a$, which may be rendered on dimensionless form by using (4.76):

$$\left(\frac{d\theta}{dx}\right)_b = \frac{k_a}{k_b}\left(\frac{d\theta}{dX}\right)_a \tag{4.151}$$

Assume the following cooling rib geometry

$$L = 0.2m, \qquad D = 0.02m, \qquad d = 0.01m \tag{4.152}$$

and let the thermodynamical properties be given by:
Leif Rune 1: ??????? the thermodynamical properties were lost
With these geometrical and thermodynamical properties equation junction condition in equation (4.151) is transformed to:

$$\left(\frac{d\theta}{dx}\right)_b = 4\left(\frac{d\theta}{dx}\right)_a \tag{4.153}$$

whereas the differential equation for quasi onedimensional heat conduction remains:

$$\frac{d}{dx}\left(\frac{x}{\beta^2}\frac{d\theta}{dx}\right) - \theta(x) = 0 \tag{4.154}$$

the expressions for β are:

$$\beta^2 = \frac{2\bar{h}\ L^2}{D\ k} \tag{4.155}$$

which in our example corresponds to

$$\beta_a^2 = 2.0 \qquad \text{and} \qquad \beta_b^2 = 8.0 \tag{4.156}$$

By rendering equation (4.154) in this way we satisfy the continuity condition in equation (4.150).

The exercise is to solve the heat equation (4.154) numerically for the triangle in Figure 4.11 with the geometry and thermodynamical parameters given above.

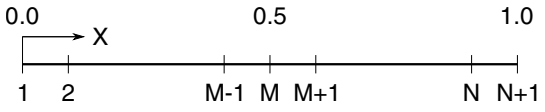

Figure 4.12: Spatial discretization for the example of heat conduction in a triangle composed by two different materials with material interface at point M.

Hint. Referring to Figure 4.12, we use the following numbering:

$$\begin{array}{l} x_i = (i-1)\ h\ ,\ i = 1, 2, ..., N+1 \text{ with } h = \frac{1}{N} \\ M = \frac{N}{2} + 1 \text{ with N being an even number and } x_M = 0.5 \end{array} \tag{4.157}$$

With $k = k_a$ for $i = M$ we get that $\beta = \beta_a$ for $i = 1, 2, ..., M$ and $\beta = \beta_b$ for $i > M$. For $i \neq M$ we can write (4.154)

$$\frac{d}{dx}\left(x\frac{d\theta}{dx}\right) - \beta^2\ \theta(x) = 0 \tag{4.158}$$

Discretizing (4.158) using (2.45) in Chapter (2):

$$-x_{i-\frac{1}{2}}\ \theta_{i-1} + (x_{i+\frac{1}{2}} + x_{i-\frac{1}{2}} + \beta^2 h^2)\ \theta_i - x_{i+\frac{1}{2}}\ \theta_{i+1} = 0$$

Noting that $x_{i-\frac{1}{2}} = (i - \frac{3}{2})\ h$ and $x_{i+\frac{1}{2}} = (i - \frac{1}{2})\ h$ we can write:

$$\left(i-\frac{3}{2}\right)\,\theta_{i-1}+\left[2(i-1)+\beta^2 h\right]\,\theta_i-\left(i-\frac{1}{2}\right)\,\theta_{i+1}=0$$

or divided by i:

$$-\left(1-\frac{3}{2i}\right)\,\theta_{i-1}+\left[2\,\left(1-\frac{1}{i}\right)+\frac{\beta^2 h}{i}\right]\,\theta_i-\left(1-\frac{1}{2i}\right)\,\theta_{i+1}=0 \tag{4.159}$$

(4.159) is used for $i=2,3,...,N$, however, except for $i=M=N/2+1$.
For i = 1
From (4.80) we get the boundary condition for $x=0$:

$$\frac{d\theta}{dx}(0)=\beta_a^2\,\theta(0)\rightarrow\frac{-3\theta_1+4\theta_2-\theta_3}{2h}=\beta_a^2\,\theta_1, \tag{4.160}$$

where we have used forward differences (See 2.5).
This results in:

$$\theta_3=4\theta_2-(3+2h\beta_a^2)\,\theta_1 \tag{4.161}$$

Writing (4.159) for $i=2$:

$$-\left(1-\frac{3}{4}\right)\,\theta_1+\left[2\,\left(1-\frac{1}{2}\right)+\frac{\beta_a^2 h}{2}\right]\,\theta_2-\left(1-\frac{1}{4}\right)\,\theta_3$$

and considering (4.155) and (4.161) results in:

$$\left(1+\frac{3h}{2}\right)\,\theta_1-\left(1-\frac{h}{2}\right)\,\theta_2=0 \tag{4.162}$$

This is the first equation of our system.
For i = M.

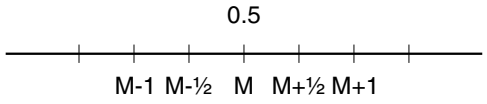

Figure 4.13: Spatial discretization for the example of heat conduction in a triangle composed by two different materials with material interface at point M. Detail of the discretization for the use of centered finite differences for point M,

According to the discretization illustrated in Figure 4.13, here we use an equation of the form given in (4.154), which discretized using (2.45) from Chapter (2), results in:

$$\beta^2_{M-\frac{1}{2}} = \beta^2_a = 2.0 \ , \ \beta^2_{M+\frac{1}{2}} = \beta^2_b = 8.0$$
$$x_{M-\frac{1}{2}} = \frac{h}{2}(N-1) = \frac{(1-h)}{2} \ , \ x_{M+\frac{1}{2}} = \frac{h}{2}(N+1) = \frac{(1+h)}{2}$$

which provides:

$$-4\,(1-h)\,\theta_{M-1} + [5 - 3h + 16h^2]\,\theta_M - (1+h)\,\theta_{M+1} = 0 \tag{4.163}$$

For i = N we can use (4.159) with $\beta^2 = \beta^2_b = 8.0$:

$$-\left(1 - \frac{3h}{2}\right)\,\theta_{N-1} + 2\,[(1-h) + 4h^2]\,\theta_N = \left(1 - \frac{h}{2}\right) \tag{4.164}$$

Equations (4.162)–(4.164) constitute a linear system of equations with a tridiagonal matrix and can be solved as such using the Thomas algorithm. In addition to the temperature, we also want to calculate the temperature gradient $\theta'(x)$. For $x = 0$ we use (4.80) and interface condition $\frac{d\theta}{dx}(0.5+) \equiv \left(\frac{d\theta}{dx}\right)_b$ found from (4.153). The other values are computed using standard central differences, besides for x = 0.5 og x = 1.0, where second order backward differences are used. Let us look at the use of the differential equation as an alternative.

We integrate (4.158):

$$\frac{d\theta_i}{dx} = \frac{\beta^2}{x_i}\int_{x_1}^{x_i} \theta(x)dx \ , \ i = 2, 3, ..., N+1 \tag{4.165}$$

Integral $I = \int_{x_1}^{x_i} \theta(x)dx$ can be calculated using the trapezium method, for example . The computation of (4.165) is shown below in pseudo-code form:

$$\begin{aligned} \theta'_1 &:= \beta^2_a\,\theta_1 \\ s &:= 0 \\ \text{Utfør for } i &:= 2, ..., N+1 \\ x &:= h\,(i-1) \\ s &:= s + 0.5h\,(\theta_i + \theta_{i-1}) \end{aligned}$$

$$\text{Dersom } (i \le M) \text{ sett } \theta'_i := \frac{\beta^2_a\,s}{x} \text{ ellers } \theta'_i := \frac{\beta^2_b\,s}{x}$$

In the following table we have used (4.165) and the trapezium method. In this case, the accuracy of the two methods for calculating $\theta'(x)$ is quite similar as both θ and θ' are smooth functions (except at x = 0.5), but generally the integration method will be more accurate if there are discontinuities..

Solution of equation (4.154) with $h = 0.01$

x	$\theta(x)$	rel. feil	$\theta'(x)$	rel. feil
0.00	0.08007	1.371E-4	0.16014	1.312E-4
0.10	0.09690	1.341E-4	0.17670	1.302E-4
0.20	0.11545	1.386E-4	0.19438	1.286E-4
0.30	0.13582	1.252E-4	0.21324	1.360E-4
0.40	0.15814	1.265E-4	0.23333	1.329E-4
0.45	0.17007	1.294E-4	0.24387	1.312E-4
0.5-	0.18253	1.260E-4	0.25472	1.021E-4
0.5+	0.18253	1.260E-4	1.01889	1.021E-4
0.55	0.23482	5.962E-5	1.07785	1.067E-4
0.60	0.29073	5.848E-5	1.16297	9.202E-5
0.70	0.41815	3.348E-5	1.39964	7.288E-5
0.80	0.57334	1.744E-5	1.71778	5.938E-5
0.90	0.76442	7.849E-6	2.11851	4.862E-5
1.00	1.00000	0.0	2.60867	1.526E-4

Chapter 5

Mathematical properties of partial differential equations

5.1 Model equations

In Appendix 2 in the Numeriske Beregninger Navier-Stokes equations for an incompressible fluid:

$$\frac{Du}{Dt} \equiv \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{5.1}$$

$$\frac{Dv}{Dt} \equiv \frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial y} + \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) \tag{5.2}$$

$$\frac{DT}{Dt} \equiv \frac{\partial T}{\partial t} + u\frac{\partial T}{\partial x} + v\frac{\partial T}{\partial y} = \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) \tag{5.3}$$

The left-hand side terms represent transport (advection/convection), while the right-hand side stands for diffusive processes. It follows that transport is expressed by first order terms and are, in this case, non-linear, while diffusive terms are given by second order derivatives, which in this case are linear. Many important engineering problems can be described by special cases of these equations, such as boundary layer problems. Since such special cases are, of course, easier to solve and analyze, these will be often used when we are going to derive a numerical scheme. These are usually one-dimensional, non-stationary cases.

5.1.1 List of some model equations

Poisson equation.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2}{\partial y^2} = f(x, y) \tag{5.4}$$

Laplace for $f(x, y) = 0$ Potential flow theory, stationary heat conduction, etc.

One-dimensional diffusion equation.

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} \tag{5.5}$$

Wave equation. Fundamental equation for acoustics and other applications.

$$\frac{\partial^2 u}{\partial t^2} = \alpha_0^2 \frac{\partial^2 u}{\partial x^2} \tag{5.6}$$

1. order linear advection equation.

$$\frac{\partial u}{\partial t} + \alpha_0 \frac{\partial u}{\partial x} = 0 \tag{5.7}$$

Inviscid Burger's equation. Model for Euler's equation of gas dynamics.

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = 0 \tag{5.8}$$

Burger's equation. Model for incompressible Navier-Stokes equations.

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2} \tag{5.9}$$

Tricomi equation. Model for transonic flow.

$$y \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{5.10}$$

Convection-diffusion equation. Linear advection and diffusion.

$$\frac{\partial u}{\partial t} + u_0 \frac{\partial u}{\partial x} = \alpha \frac{\partial^2 u}{\partial x^2} \tag{5.11}$$

The expressions transport, convection and advection are equivalent. (5.7) above is known as the advection equation.

5.2 First order partial differential equations

The left-hand side of the model equation, related to transport phenomena, consists of the term $\frac{D()}{Dt}$, which is a 1st order partial differential equation (PDE). Let us look in detail at this term by considering the following equation:

$$a \frac{\partial u}{\partial x} + b \frac{\partial u}{\partial y} + c = 0 \tag{5.12}$$

If a or b are functions of x, y and u, the equation is quasi-linear. When a and b are functions of x and/or y, or constant, the function is linear.

The equation is non-linear if a or b are functions of either u, $\frac{\partial u}{\partial x}$ and/or $\frac{\partial u}{\partial y}$. For example, the transport terms of the Navier-Stokes equations, i.e. the left-hand side of (5.1), are quasi-linear 1st order PDEs, even if we normally call them non-linear.

Writing (5.12) as:

$$a\left(\frac{\partial u}{\partial x}+\frac{b}{a}\frac{\partial u}{\partial y}\right)+c=0 \tag{5.13}$$

Assuming that u is continuously differentiable:

$$du=\frac{\partial u}{\partial x}dx+\frac{\partial u}{\partial y}dy\rightarrow\frac{du}{dx}=\frac{\partial u}{\partial x}+\frac{dy}{dx}\frac{\partial u}{\partial y}$$

Defines the characteristic of (5.12) by:

$$\frac{dy}{dx}=\frac{b}{a} \tag{5.14}$$

(5.14) inserted above provides:

$$a\left(\frac{\partial u}{\partial x}+\frac{b}{a}\frac{\partial u}{\partial y}\right)=a\frac{du}{dx}$$

which further provides:

$$a\cdot du+c\cdot dx=0 \tag{5.15}$$

Along the characteristic curve defined by $\frac{dy}{dx}=\frac{b}{a}$, (5.12) reduces to an ODE given by (5.15). (5.15) is called the compatibility equation for (5.12). If (5.12) is linear, we can in principle first find the characteristic and then solve (5.15). This is normally not possible when (5.12) is quasi-linear or non-linear because in that case $\frac{dy}{dx}$ is a function of u, ie: of the solution itself.

Example 5.2.1. .

The advection equation (5.7) given by $\frac{\partial u}{\partial t}+a_0\frac{\partial u}{\partial x}$ has the solution $u(x,t)=f(x-a_0t)$, which is a wave propagating with constant profile and velocity along x. The wave moves to the right for $a_0>0$. The characteristic is given here by: $\frac{dx}{dt}=a_0$.

This equation can be integrated directly, yielding: $x=a_0t+C_1$. Constant C_1 can be determined, for example, by noting that: $x=x_0$ for $t=0$, which implies that $C_1=x_0$. The characteristic curve equation becomes $x=a_0t+x_0$. (5.15) with $c=0$ and $a=1$ results in $du=0$. This means that u is here the particle velocity, which is constant along the characteristic.

Let us see an example with initial conditions, choose:

$$u(x,0)=\begin{cases}1-x^2\text{ for }|x|\leq 1\\ 0\text{for }|x|>1\end{cases}$$

This is a parabola. From $u(x,t) = f(x - a_0 t)$ one has that $u(x,0) = f(x)$. With $\zeta = x - a_0 t$, we can write the solution:

$$u(x,t) = \begin{cases} 1 - \zeta^2 \text{ for } |\zeta| \leq 1 \\ 0 \text{ for } |\zeta| > 1 \end{cases}$$

When we use ζ as variable, the solution is stationary: we follow the wave. Figure 5.1 summarizes the example.

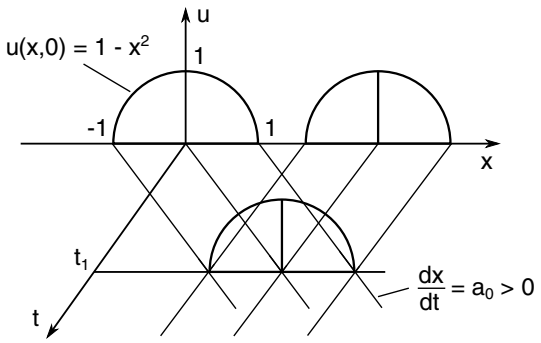

Figure 5.1: Propagation of initial condition by the linear advection equation.

Example 5.2.2. .

The inviscid Burger's equation (5.8) is given by:

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = 0$$

This has the solution $u(x,t) = f(x - u \cdot t)$ and characteristic curve given by: $\frac{dx}{dt} = u$. This is the same solution as in the previous example, with the important difference that now a_0 is replaced with u. The slope of characteristics varies now with x, as indicated in Figure 5.2.

The same initial condition as in the previous example provide the following solution:

$$u(x,t) = \begin{cases} 1 - \zeta^2 \text{ for } |\zeta| \leq 1 \\ 0 \text{ for } |\zeta| > 1 \end{cases} \quad \text{der } \zeta = x - u \cdot t$$

Again, we see that the only difference from the previous example is that a_0 is now replaced by u. Explicit solution, noting that $u(x,t) = 0$ for $|\zeta| > 1$:

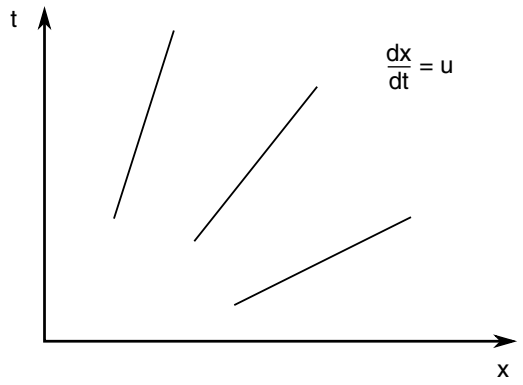

Figure 5.2: Characteristic curves for the Burger's equation (5.2.2) at different locations along x.

$$u(x,t) = \frac{1}{2t^2}[(2xt-1) \pm g(x,t)],$$

$$g(x,t) = \sqrt{1-4xt+4t^2}, \ |\zeta| \leq 1 \tag{5.16}$$

$$\frac{\partial u}{\partial x} = \frac{g(x,t) \pm 1}{t \cdot g(x,t)} \tag{5.17}$$

By this method of solution uniqueness is lost when characteristic intersect (Figure 5.3). In this case one has that $u(x,t)$ takes two possible values for $x \geq 1$, when $\frac{\partial u}{\partial x}(1,t) > 0$. The critical value t_{crit} when this occur takes place when $\frac{\partial u}{\partial x}(1,t) \to \infty$, as one can conclude from (5.17), which implies that $g(1,t) = 0 = 1 - 2t \to t_{crit} = \frac{1}{2}$.

For $t \leq \frac{1}{2}$ one uses (5.16) and (5.17) with positive sign for $g(x,t)$ for all values of x. For $t > \frac{1}{2}$ the critical value arises at $x^* = \frac{1+4t^2}{4t}$ with $u^* = 1 - \frac{1}{4t^2}$. For $-1 \leq x \leq x^*$ one can still use (5.16) and (5.17) with positive sign for $g(x,t)$, but for $1 \leq x \leq x^*$ the lower part of the solution is given by using (5.16) and (5.17) with negative sign in front of $g(x,t)$. The maximum value $u_{maks} = 1$ is always given at $x = t$.

In order to obtain a unique solution, one must introduce the concept of a moving discontinuity: in gas dynamics this is called a shock, in hydraulics a hydraulic jump.

The characteristic curves shown in Figure 5.4 are given by:

$$x = (1 - x_0^2) \cdot t + x_0, \text{ der } x = x_0 \text{ for } t = 0$$

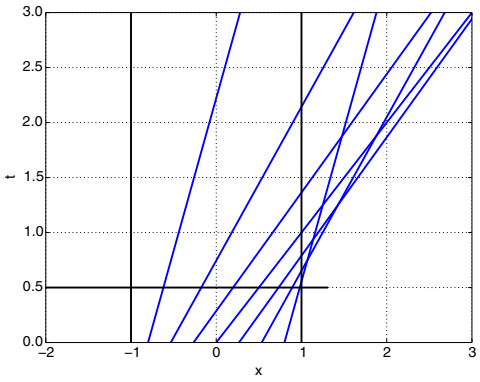

Figure 5.3: Illustration of intersecting characteristic curves.

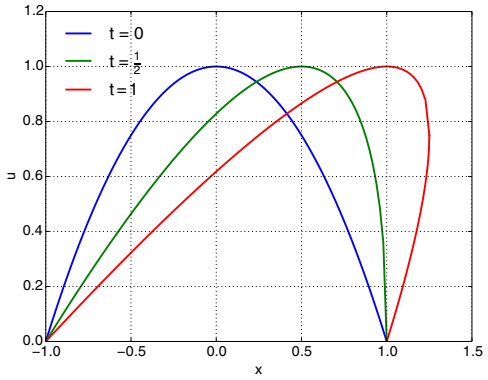

Figure 5.4: Inviscid Burger's equation solution. Note that solution for time $t = 1$ is multiply. After t_{crit} additional mathematical notions have to be applied to identify a unique solution.

5.3 Second order partial differenatial equations

A seond order PDE for two independent variables x and y can be written as:

$$A\frac{\partial^2\phi}{\partial x^2} + B\frac{\partial^2\phi}{\partial x \partial y} + C\frac{\partial^2\phi}{\partial y^2} + f = 0 \tag{5.18}$$

If A, B, C and f are functions of $x, y, \phi, \frac{\partial\phi}{\partial x}$ and $\frac{\partial\phi}{\partial y}$, (5.18) is said to be quasi-linear. If A, B and C are only functions of x and y, (5.18) is semi-linear. If f is only function of x and y, (5.18) is linear.

Example 5.3.1. .

$$\left(\frac{\partial\phi}{\partial x}\right)\frac{\partial^2\phi}{\partial x^2} - \frac{\partial\phi}{\partial x} - e^{xy}\sin\phi = 0 \quad \text{quasi-linear} \tag{5.19}$$

$$x\frac{\partial^2\phi}{\partial x^2} - \frac{\partial\phi}{\partial x} - e^{xy}\sin\phi = 0 \quad \text{semi-linear} \tag{5.20}$$

$$\left(\frac{\partial^2\phi}{\partial x^2}\right)^2\frac{\partial^2\phi}{\partial y^2} = 0 \quad \text{non-linear} \tag{5.21}$$

A PDE that can be written in the following form:

$$A\frac{\partial^2\phi}{\partial x^2} + B\frac{\partial^2\phi}{\partial x\partial y} + C\frac{\partial^2\phi}{\partial x^2} + D\frac{\partial\phi}{\partial x} + E\frac{\partial\phi}{\partial y} + F\phi + G = 0 \tag{5.22}$$

where A, B, C, D, E, F, and G are only functions of x and y, is a 2nd order linear PDE. (5.22) is consequently a special case of (5.18). Notice that usually we will use the term non-linear as opposed to linear. Let us now investigate whether (5.18) has characteristics or not.

Consider a function ϕ such that $u = \frac{\partial\phi}{\partial x}$ and $v = \frac{\partial\phi}{\partial y}$, which in turn implies that $\frac{\partial u}{\partial y} = \frac{\partial v}{\partial x}$.

(5.18) can then be written as a system of two 1st order PDEs:

$$A\frac{\partial u}{\partial x} + B\frac{\partial u}{\partial y} + C\frac{\partial v}{\partial y} + f = 0 \tag{5.23}$$

$$\frac{\partial u}{\partial y} - \frac{\partial v}{\partial x} = 0 \tag{5.24}$$

It can be shown that a higher order (quasi-linear) PDE can always be written as a system of 1st order PDEs. Moreover, the resulting first order system can have many forms. If, for example, (5.18) is the potential equation in gas dynamics, we can denote ϕ as the potential speed. (5.24) is the condition of irrotational flow.

We will now try to write (5.23) and (5.24) as a total differential (see (5.13) – (5.15)). Multiply (5.24) with any scalar σ and add to (5.23):

$$A\left[\frac{\partial u}{\partial x} + \left(\frac{B+\sigma}{A}\right)\frac{\partial u}{\partial y}\right] - \sigma\left[\frac{\partial v}{\partial x} - \frac{C}{\sigma}\frac{\partial v}{\partial y}\right] + f = 0 \tag{5.25}$$

Now we have:

$$\frac{du}{dx} = \frac{\partial u}{\partial x} + \frac{dy}{dx}\frac{\partial u}{\partial y},\ \frac{dv}{dx} = \frac{\partial v}{\partial x} + \frac{dy}{dx}\frac{\partial v}{\partial y}$$

Hence:

$$\frac{du}{dx} = \frac{\partial u}{\partial x} + \lambda\frac{\partial u}{\partial y},\ \frac{dv}{dx} = \frac{\partial v}{\partial x} + \lambda\frac{\partial v}{\partial y} \tag{5.26}$$

where we have defined λ as:

$$\lambda = \frac{dy}{dx} \tag{5.27}$$

By comparing (5.25) and (5.26):

$$\frac{dy}{dx} = \lambda = \frac{B+\sigma}{A} = -\frac{C}{\sigma} \tag{5.28}$$

(5.28) inserted in (5.25) gives the compatibility equation

$$A\frac{du}{dx} - \sigma\frac{dv}{dx} + f = 0 \tag{5.29}$$

If the characteristics are real, i.e. λ is real, we have transformed the original PDE into an ODE given by (5.29) along the directions defined by (5.27).

From (5.28):

$$\sigma = -\frac{C}{\lambda} \tag{5.30}$$

which also gives:

$$\lambda = \frac{B - \frac{C}{\lambda}}{A} \tag{5.31}$$

The following 2nd order equation can be obtained from (5.31) to determine λ:

$$A\lambda^2 - B\lambda + C = 0 \tag{5.32}$$

or by using (5.27):

$$A \cdot (dy)^2 - B \cdot dy \cdot dx + C \cdot (dx)^2 = 0 \tag{5.33}$$

After λ is found from (5.32) and (5.33), σ can be found from (5.30) and (5.31) so that the compatibility equation in (5.29) can be determined. Instead of using (5.29), we can insert σ from (5.28) in (5.29) so that we get the following compatibility equation:

$$A\frac{du}{dx} + \frac{C}{\lambda}\frac{dv}{dx} + f = 0 \tag{5.34}$$

2nd degree equations (5.32) and (5.33) have the roots λ_1 and λ_2

$$\lambda_{1,2} = \frac{B \pm \sqrt{B^2 - 4AC}}{2A} \tag{5.35}$$

We have three possibilities for the roots in (5.35):

- $B^2 - 4AC > 0 \Rightarrow \lambda_1$ and λ_2 are real
- $B^2 - 4AC < 0 \Rightarrow \lambda_1$ and λ_2 are complex

- $B^2 - 4AC = 0 \Rightarrow \lambda_1 = \lambda_2$ and real

The quasi-linear PDE (5.18) is called

- Hyperbolic if $B^2 - 4AC > 0$: λ_1 and λ are real
- Elliptic if $B^2 - 4AC < 0$: λ_1 and λ are complex
- Parabolic if $B^2 - 4AC = 0$: $\lambda_1 = \lambda_2$ are real

The above terms come from the analogy with the cone cross-section equation of the expression $Ax^2 + Bxy + Cy^2 + Dx + F = 0$. For example, $x^2 - y^2 = 1$ represents a hyperbola if $B^2 - 4AC > 0$.

The roots λ_1 and λ_2 are characteristics. Hence:

- Hyperbolic: Two real characteristics.
- Elliptic: No real characteristics.
- Parabolic: One real characteristic.

Example 5.3.2. *Examples of classification of various PDEs.*

The wave equation $\dfrac{\partial^2 u}{\partial x^2} - \dfrac{\partial^2 u}{\partial y^2} = 0$ is hyperbolic since $B^2 - 4AC = 4 > 0$

Characteristics are given by $\lambda^2 = 1 \rightarrow \dfrac{dy}{dx} = \pm 1$

Laplace equation $\dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} = 0$ is elliptic since $B^2 - 4AC = -4 < 0$

The diffusion equation $\dfrac{\partial u}{\partial y} = \dfrac{\partial^2 u}{\partial x^2}$ is parabolic since $B^2 - 4AC = 0$

Linearisert potensial-ligning for kompressibel strømning:

$$(1 - M^2)\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0, \ M = \text{Mach-number}. \tag{5.36}$$

- $M = 1$: Parabolic (degenerate).
- $M < 1$: Elliptic. Subsonic flow.
- $M > 1$: Hyperbolic. Supersonic flow.

5.4 Boundary conditions for 2nd order PDEs

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 : \text{Elliptic . No real characteristics.} \tag{5.37}$$

$$\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = 0 : \text{Hyperbolic. Two rel characteristics.} \tag{5.38}$$

$$\frac{\partial u}{\partial y} = \frac{\partial^2 u}{\partial x^2} : \text{Parabolic. One real characteristic.} \tag{5.39}$$

The model equations shown above differ by the number of real characteristics. This directly affects the boundary conditions that are possible in the three cases, as physical information propagates along characteristics.

5.4.1 Hyberbolic equations

As example we use the wave equation:

$$\frac{\partial^2 u}{\partial t^2} = \alpha_0^2 \frac{\partial^2 u}{\partial x^2} \tag{5.40}$$

(5.40) has the characteristic $\frac{dx}{dt} = \pm a_0$, where a_0 is the wave propagation speed. We denote $\frac{dx}{dt} = +a_0$ with C^+ and $\frac{dx}{dt} = -a_0$ with C^-. The area of influence for (5.40) is shown in Figure 5.5.

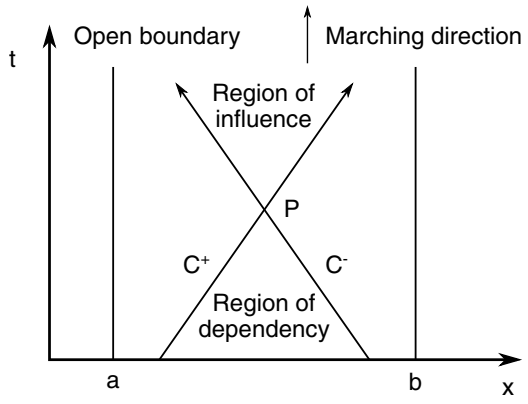

Figure 5.5: Regions of dependence and influence with respect to point P for the wave equation(5.40).

The solution at point P depends only on the solution in the region of dependence, while the value at P only affects the solution within the region of influence. Boundary values for domain borders can not be prescribed independently of

initial conditions at $t = 0$. In order to solve this equation as an initial value problem, (5.40) must also have an initial condition for $\dfrac{\partial u}{\partial t}$ at $t = 0$.

5.4.2 Elliptic equations

Model equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{5.41}$$

The solution domain is shown in Figure 5.6.

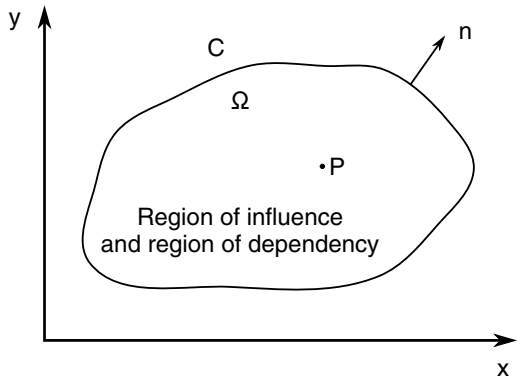

Figure 5.6: Regions of dependence and influence for the model elliptic equation (5.41).

We have no real characteristics. The entire domain Ω, including its boundary C, coincide with the regions of dependence and influence for P: Any change of a value in Ω or C will affect the solution in P. (5.41) is a purely boundary value problem and the following boundary conditions are admissible:

- u is prescribed at C: Dirichlet boundary condition.
- $\dfrac{\partial u}{\partial n}$ is prescribed at C: Neumann boundary condition.
- a weighted combination of u and $\dfrac{\partial u}{\partial n}$ are prescribed at C: Robin boundary condition.
- a combination of the above conditions in different portions of C.

5.4.3 Parabolic equations

Model equation:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \tag{5.42}$$

The solution domain for (5.42) is shown in Figure 5.7.

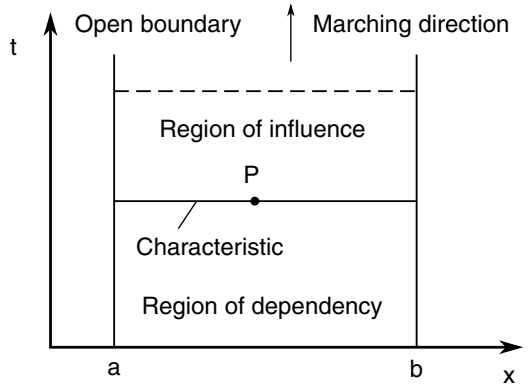

Figure 5.7: Regions of dependence and influence for the model parabolic equation (5.42).

From the classification equations (5.33) we find that:

$dt = 0 \Rightarrow t =$ constant is the characteristic curve in this case. Hence: $a_0 = \frac{dx}{dt} = \infty$, which means that the propagation velocity along characteristic $t =$ constant is infinitely large. The solution at P depends on the value in all points in the physical space for past time t, including the present. (5.42) behaves like an elliptical equation for each value of $t =$ constant.

15

Chapter 6

Elliptic partial differential equations

6.1 Introduction

Important and frequently occuring practical problems are governed by elliptic PDEs, including steady-state temperature distribution in solids.

Famous elliptic PDEs.

Some famous elliptic PDEs are better know by their given names:

- Laplace:

$$\nabla^2 u = 0 \tag{6.1}$$

- Poisson:

$$\nabla^2 u = q \tag{6.2}$$

- Helmholtz:

$$\nabla^2 u + c \cdot u = q \tag{6.3}$$

$$\text{where } \nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \text{ in 3D}$$

and

$$\text{where } \nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \text{ in 2D}$$

The 2D versions of the famous equations above are special cases of the a generic elliptic PDE may be represented by:

$$\frac{\partial}{\partial x}\left(a\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(b\frac{\partial u}{\partial y}\right) + c \cdot u = q \tag{6.4}$$

where a, b, c og q may be functions of x and y and a and b have the same sign.

- Transient heat conduction is governed by:

$$\frac{\partial T}{\partial t} = \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right) \tag{6.5}$$

which is a parabolic PDE, but at steady state when $\left(\frac{\partial T}{\partial t} = 0\right)$, we get:

$$\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right) = 0 \tag{6.6}$$

which is an elliptic PDE.

From the classification approach in Chapter 5.3 (see also Classification of linear 2nd order PDEs) we see that (6.1) to (6.3) are elliptic, meaning no real characteristics. Therefore, elliptic PDEs must be treated as boundary value problems (see 5.4) for a discussion about the conditions for a well posed problem).

For the solution of a generic elliptic PDE like (6.4), in a domain in two-dimensions bounded by the boundary curve C (6.1), we reiterate the most common boundary value conditions from 5.4:

- Dirichlet-condition: $u(x, y) = G_1(x, y)$ on the boundary cure C
- Neumann-condition: $\frac{\partial u}{\partial n} = G_2(x, y)$ on C
- Robin-condition: $a \cdot u(x, y) + b \cdot \frac{\partial u(x, y)}{\partial n} = G_3(x, y)$ on C

For the Neuman-problem, at least one value of $u(x, y)$ must be specified on C for the solution to be unique. Additionally, the contour integral of $G_2(x, y)$ on C must vanish.

Several physical problems may be modeled by the Poisson equation (6.2)

Possion problems.

- The stress equation for torsion of an elastic rod
- The displacement of an membrane under constant pressure.

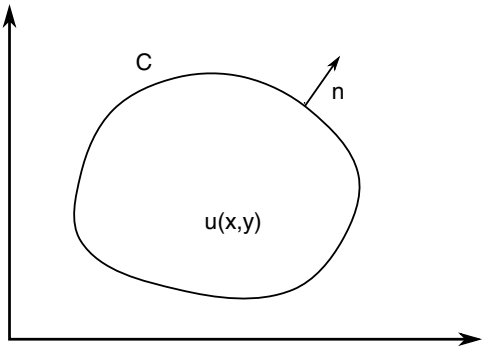

Figure 6.1: A solution domain in two-dimensions bounded by a boundary curve C.

- Stokes flow (low Reynolds number flow)
- Numerous analytical solutions (also in polar coordinantes) may be found in [15] section 3-3.

6.2 Finite differences. Notation

In Section 2.5 we used Taylor expansions to discretize differential equations for functions of one independent variable. A similar procedure can be applied to functions of more than one independent variable In particular, if the expressions to be discretized have no cross-derivatives, we can apply the formulas derived in section 2.5, keeping the index for the second variable constant.

Since we are dealing with elliptic equations, we will, for convenience, consider x and y as the independent variables. Moreover, its discretization is given by:

$$x_i = x_0 + i \cdot \Delta x, \;\; i = 0, 1, 2, \ldots$$
$$y_j = y_0 + j \cdot \Delta y, \;\; j = 0, 1, 2, \ldots$$

As before, we assume that Δx and Δy are constant, unless otherwise specified. Figure 6.2 shows how we discretize a 2D domain using a Cartesian grid.

We now recall several finite difference formulas from Section 2.5.

Forward difference:

$$\left.\frac{\partial u}{\partial x}\right|_{i,j} = \frac{u_{i+1,j} - u_{i,j}}{\Delta x} + O(\Delta x) \tag{6.7}$$

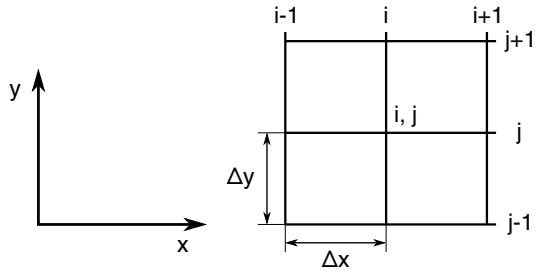

Figure 6.2: Cartesian grid discretization of a 2D domain.

Backward difference:

$$\left.\frac{\partial u}{\partial x}\right|_{i,j} = \frac{u_{i,j} - u_{i-1,j}}{\Delta x} + O(\Delta x) \tag{6.8}$$

Central differences:

$$\left.\frac{\partial u}{\partial x}\right|_{i,j} = \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} + O\left[(\Delta x)^2\right] \tag{6.9}$$

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{i,j} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2} + O\left[(\Delta x)^2\right] \tag{6.10}$$

Similar formulas for $\frac{\partial u}{\partial y}$ and $\frac{\partial^2 u}{\partial y^2}$ follow directly:
Forward difference:

$$\left.\frac{\partial u}{\partial y}\right|_{i,j} = \frac{u_{i,j+1} - u_{i,j}}{\Delta y} \tag{6.11}$$

Backward difference:

$$\left.\frac{\partial u}{\partial y}\right|_{i,j} = \frac{u_{i,j} - u_{i,j-1}}{\Delta y} \tag{6.12}$$

Central differences:

$$\left.\frac{\partial u}{\partial y}\right|_{i,j} = \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta y} \tag{6.13}$$

$$\left.\frac{\partial^2 u}{\partial y^2}\right|_{i,j} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{(\Delta y)^2} \tag{6.14}$$

6.2.1 Example: Discretization of the Laplace equation

We discretize the Laplace equation $\frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial y^2} = 0$ by using (6.10) and (6.14) and setting $\Delta x = \Delta y$.

The resulting discrete version of the original equations is:

$$u_{i+1,j} + u_{i-1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j} = 0$$

The resulting numerical stencil is shown in Figure 6.3.

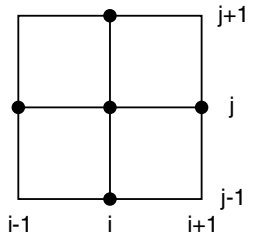

Figure 6.3: 5-point numerical stencil for the discretization of Laplace equations using central differences.

It is easy to note that in (6.2.1), the value for the central point is the mean of the values of surrounding points. Equation (6.2.1) is very well-known and is usually called the 5-point formula (used in Chapter (6)).

A problem arises at boundaries when using central differences. At those locations the stencil might fall outside of the computational domain and special strategies must be adopted. For example, one can replace central differences with backward or forward differences, depending on the case. Here we report some of these formulas with 2nd order accuracy.

Forward difference:

$$\left.\frac{\partial u}{\partial x}\right|_{i,j} = \frac{-3u_{i,j} + 4u_{i+1,j} - u_{i+2,j}}{2\Delta x} \tag{6.15}$$

Backward difference:

$$\left.\frac{\partial u}{\partial x}\right|_{i,j} = \frac{3u_{i,j} - 4u_{i-1,j} + u_{i-2,j}}{2\Delta x} \tag{6.16}$$

The formulas for 2.5 , 2.5 and 2.5 differences given in Chapter 2 can be used now if equipped with two indexes. A rich collection of difference formulas can be found in Anderson [14]. Detailed derivations are given in Hirsch [7].

6.3 Direct numerical solution

In situations where the elliptic PDE correspons to the stationary solution of a parabolic problem (6.5), one may naturally solve the parabolic equation until

stationary conditions occurs. Normally, this will be time consuming task and one may encounter limitations to ensure a stable solution. By disregarding such a timestepping approach one does not have to worry about stability. Apart from seeking a fast solution, we are also looking for schemes with efficient storage management a reasonable programming effort.

Let us start by discretizing the stationary heat equation in a rectangular plated with dimension as given in Figure 6.4:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{6.17}$$

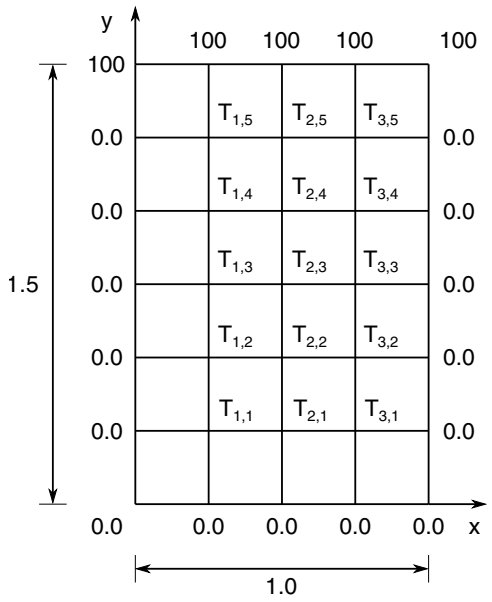

Figure 6.4: Rectangular domain with prescribed values at the boundaries (Dirichlet).

We adopt the following notation:

$$x_i = x_0 + i \cdot h, \;\; i = 0, 1, 2, \ldots$$
$$y_j = y_0 + j \cdot h, \;\; j = 0, 1, 2, \ldots$$

For convenience we assume $\Delta x = \Delta y = h$. The ordering of the unkown temperatures is illsustrated in (6.5).

By approximation the second order differentials in (6.17) by central differences we get the following numerical stencil:

$$T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1} - 4T_{i,j} = 0 \tag{6.18}$$

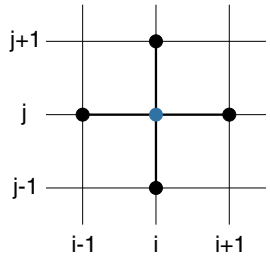

Figure 6.5: Illustration of the numerical stencil.

which states that the temperature $T_{i,j}$ in at the location (i, j) depends on the values of its neighbors to the left, right, up and down. Frequently, the neighbors are denoted in compass notation, i.e. west $= i - 1$, east $= i + 1$, south $= j - 1$, and north $= j + 1$. By referring to the compass directions with their first letters, and equivalent representation of the stencil in (6.18) reads:

$$T_e + T_w + T_n + T_s - 4T_m = 0 \tag{6.19}$$

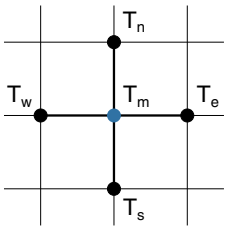

Figure 6.6: Illustration of the numerical stencil with compass notation.

The smoothing nature of elliptic problems may be seen even more clearly by isolating the $T_{i,j}$ in (6.19) on the left hand side:

$$T_m = \frac{T_e + T_w + T_n + T_s}{4} \tag{6.20}$$

showing that the temperature T_m in each point is the average temperature of the neighbors (to the east, west, north, and south).

The temperature is prescribed at the bondaries (i.e. Dirichlet boundary conditions) and are given by:

$$\begin{aligned} &T = 0.0 \quad \text{at} \quad y = 0 \\ &T = 0.0 \quad \text{at} \quad x = 0 \quad \text{and} \quad x = 1 \quad \text{for} \quad 0 \leq y < 1.5 \\ &T = 100.0 \quad \text{at} \quad y = 1.5 \end{aligned} \tag{6.21}$$

Our mission is now to find the temperature distribution over the plate by using (6.18) and (6.21) with $\Delta x = \Delta y = 0.25$. In each discretized point in (6.5) the temperatures need to satisfy (6.18), meaning that we have to satisfy as many equations as we have unknown temperatures. As the temperatures in each point depends on their neighbors, we end up with a system of algebraic equations. To set up the system of equations we traverse our our unknows one by one in a systematic manner and make use of use of (6.18) and (6.21) in each. All unknown temperatures close to any of the boundaries (left, right, top, bottom) in Figure 6.4 will be influenced by the prescribed and known temperatures the wall via the 5-point stencil (6.18). Prescribed values do not have to be calculated an can therefore be moved to the right hand side of the equation, and by doing so we modify the numerical stencil in that specific discretized point. In fact, inspection of Figure 6.4, reveals that only three unknown temperatures are not explicitly influenced by the presences of the wall ($T_{2,2}$, $T_{2,3}$, and $T_{2,3}$). The four temperatures in the corners ($T_{1,1}$,$T_{1,5}$, $T_{3,1}$, and $T_{3,5}$) have two precribed values to be accounted for on the right hand side of their specific version of the generic numerical stencil (6.18). All other unknown temperatures close to the wall have only one prescribed value to be accounted for in their specific numerical stenscil.

By starting at the lower left corner and traversing in the y-direction first, and subsequently in the x-direction we get the following system of equations:

$$
\begin{aligned}
-4 \cdot T_{11} + T_{12} + T_{21} &= 0 \\
T_{11} - 4 \cdot T_{12} + T_{13} + T_{22} &= 0 \\
T_{12} - 4 \cdot T_{13} + T_{14} + T_{23} &= 0 \\
T_{13} - 4 \cdot T_{14} + T_{15} + T_{24} &= 0 \\
T_{14} - 4 \cdot T_{15} + T_{25} &= -100 \\
T_{11} - 4 \cdot T_{21} + T_{22} + T_{31} &= 0 \\
T_{12} + T_{21} - 4 \cdot T_{22} + T_{23} + T_{32} &= 0 \\
T_{13} + T_{22} - 4 \cdot T_{23} + T_{24} + T_{33} &= 0 \\
T_{14} + T_{23} - 4 \cdot T_{24} + T_{25} + T_{34} &= 0 \\
T_{15} + T_{24} - 4 \cdot T_{25} + T_{35} &= 100 \\
T_{21} - 4 \cdot T_{31} + T_{32} &= 0, \\
T_{22} + T_{31} - 4 \cdot T_{32} + T_{33} &= 0 \\
T_{23} + T_{32} - 4 \cdot T_{33} + T_{34} &= 0 \\
T_{24} + T_{33} - 4 \cdot T_{34} + T_{35} &= 0 \\
T_{25} + T_{34} - 4 \cdot T_{35} &= -100
\end{aligned}
\tag{6.22}
$$

The equations in (6.22) represent a linear, algebraic system of equations with $5 \times 3 = 15$ unknowns, which has the more convenient and condensed symbolic representation:

$$\mathbf{A} \cdot \mathbf{T} = \mathbf{b} \tag{6.23}$$

where $\mathbf{A}$ denotes the coefficient matrix, $\mathbf{T}$ holds the unknown tempeartures, and $\mathbf{b}$ the prescribed boundary temperatures. Notice that the structure of the coefficient matrix $\mathbf{A}$ is completely dictated by the way the unknown temperatures are ordered. The non-zero elements in the coefficient matrix are markers for which unknow temperatures are coupled with each other. Below we will show an example where we order the temperatures in y-direction first and then x-direction. In this case, the components of the coefficient matrix $\mathbf{A}$ and the temperature vector $\mathbf{T}$ are given by:

$$\begin{pmatrix} -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 \end{pmatrix} \cdot \begin{pmatrix} T_{11} \\ T_{12} \\ T_{13} \\ T_{14} \\ T_{15} \\ T_{21} \\ T_{22} \\ T_{23} \\ T_{24} \\ T_{25} \\ T_{31} \\ T_{32} \\ T_{33} \\ T_{34} \\ T_{35} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -100 \\ 0 \\ 0 \\ 0 \\ 0 \\ -100 \\ 0 \\ 0 \\ 0 \\ 0 \\ -100 \end{pmatrix} \tag{6.24}$$

The analytical solution of (6.17) and (6.21) may be found to be:

$$T(x,y) = 100 \cdot \sum_{n=1}^{\infty} A_n \sinh(\lambda_n y) \cdot \sin(\lambda_n x) \tag{6.25}$$

$$\text{where } \lambda_n = \pi \cdot n \text{ and } A_n = \frac{4}{\lambda_n \sinh(\frac{3}{2}\lambda_n)}$$

The analytical solution of the temperature field $T(x,y)$ in (6.25) may be proven to be symmetric around $x = 0.5$ (see 9).

We immediately realize that it may be inefficient to solve (6.24) as it is, due to the presence of all the zero-elements. The coefficient matrix $\mathbf{A}$ has $15 \times 15 = 225$ elementes, out of which on 59 are non-zero. Further, a symmetric, band structure of $\mathbf{A}$ is evident from (6.24). Clearly, these properties may be exploited to construct an efficent scheme which does not need to store all non-zero elements of $\mathbf{A}$.

SciPy offers a sparse matrix package scipy.sparse, which has been used previously (see section 4.4.1).

```
# src-ch7/laplace_Diriclhet1.py
import numpy as np
import scipy
```

```
import scipy.linalg
import scipy.sparse
import scipy.sparse.linalg
import matplotlib; matplotlib.use('Qt4Agg')
import matplotlib.pylab as plt
import time
from math import sinh

#import matplotlib.pyplot as plt
# Change some default values to make plots more readable on the screen
LNWDT=2; FNT=15
plt.rcParams['lines.linewidth'] = LNWDT; plt.rcParams['font.size'] = FNT

# Set simulation parameters
n = 15
d = np.ones(n) # diagonals
b = np.zeros(n) #RHS
d0 = d*-4
d1 = d[0:-1]
d5 = d[0:10]

A = scipy.sparse.diags([d0, d1, d1, d5, d5], [0, 1, -1, 5, -5], format='csc')

#alternatively (scalar broadcasting version:)
#A = scipy.sparse.diags([1, 1, -4, 1, 1], [-5, -1, 0, 1, 5], shape=(15, 15)).toarray()

# update A matrix
A[4, 5], A[5, 4], A[10, 9], A[9, 10] = 0, 0, 0, 0
# update RHS:
b[4], b[9], b[14] = -100, -100, -100
#print A.toarray()


tic=time.clock()
theta = scipy.sparse.linalg.spsolve(A,b) #theta=sc.linalg.solve_triangular(A,d)
toc=time.clock()
print 'sparse solver time:',toc-tic


tic=time.clock()
theta2=scipy.linalg.solve(A.toarray(),b)
toc=time.clock()
print 'linalg solver time:',toc-tic

# surfaceplot:
x = np.linspace(0, 1, 5)
y = np.linspace(0, 1.5, 7)

X, Y = np.meshgrid(x, y)

T = np.zeros_like(X)

T[-1,:] = 100

for n in range(1,6):
    T[n,1] = theta[n-1]
    T[n,2] = theta[n+5-1]
    T[n,3] = theta[n+10-1]

from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
```

```
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import numpy as np

fig = plt.figure()
ax = fig.gca(projection='3d')
surf = ax.plot_surface(X, Y, T, rstride=1, cstride=1, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)
ax.set_zlim(0, 110)

ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('T [$^o$C]')
ax.set_xticks(x)
ax.set_yticks(y)

fig.colorbar(surf, shrink=0.5, aspect=5)
plt.show()
```

A somewhat more complicated solution of (6.17) may be found by specifying the differente temperatures on all four boundaries. However, the code structure follows the same way of reasoning as for the previous example:

```
# src-ch7/laplace_Diriclhet2.py
import numpy as np
import scipy
import scipy.linalg
import scipy.sparse
import scipy.sparse.linalg
import matplotlib; matplotlib.use('Qt4Agg')
import matplotlib.pylab as plt
import time
from math import sinh

#import matplotlib.pyplot as plt

# Change some default values to make plots more readable on the screen
LNWDT=2; FNT=15
plt.rcParams['lines.linewidth'] = LNWDT; plt.rcParams['font.size'] = FNT

# Set temperature at the top
Ttop=100
Tbottom=10
Tleft=10.0
Tright=10.0

xmax=1.0
ymax=1.5

# Set simulation parameters
#need hx=(1/nx)=hy=(1.5/ny)
Nx = 20
h=xmax/Nx
Ny = int(ymax/h)

nx = Nx-1
ny = Ny-1
n = (nx)*(ny) #number of unknowns
print n, nx, ny
```

```
d = np.ones(n) # diagonals
b = np.zeros(n) #RHS
d0 = d*-4
d1 = d[0:-1]
d5 = d[0:-ny]

A = scipy.sparse.diags([d0, d1, d1, d5, d5], [0, 1, -1, ny, -ny], format='csc')

#alternatively (scalar broadcasting version:)
#A = scipy.sparse.diags([1, 1, -4, 1, 1], [-5, -1, 0, 1, 5], shape=(15, 15)).toarray()

# set elements to zero in A matrix where BC are imposed
for k in range(1,nx):
    j = k*(ny)
    i = j - 1
    A[i, j], A[j, i] = 0, 0
    b[i] = -Ttop

b[-ny:]+=-Tright  #set the last ny elements to -Tright
b[-1]+=-Ttop      #set the last element to -Ttop
b[0:ny-1]+=-Tleft #set the first ny elements to -Tleft
b[0::ny]+=-Tbottom #set every ny-th element to -Tbottom

tic=time.clock()
theta = scipy.sparse.linalg.spsolve(A,b) #theta=sc.linalg.solve_triangular(A,d)
toc=time.clock()
print 'sparse solver time:',toc-tic

tic=time.clock()
theta2=scipy.linalg.solve(A.toarray(),b)
toc=time.clock()
print 'linalg solver time:',toc-tic

# surfaceplot:
x = np.linspace(0, xmax, Nx + 1)
y = np.linspace(0, ymax, Ny + 1)

X, Y = np.meshgrid(x, y)

T = np.zeros_like(X)


# set the imposed boudary values
T[-1,:] = Ttop
T[0,:] = Tbottom
T[:,0] = Tleft
T[:,-1] = Tright


for j in range(1,ny+1):
    for i in range(1, nx + 1):
        T[j, i] = theta[j + (i-1)*ny - 1]


from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter

fig = plt.figure()
ax = fig.gca(projection='3d')
```

```
surf = ax.plot_surface(X, Y, T, rstride=1, cstride=1, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)
ax.set_zlim(0, Ttop+10)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('T [$^o$C]')


nx=4
xticks=np.linspace(0.0,xmax,nx+1)
ax.set_xticks(xticks)

ny=8
yticks=np.linspace(0.0,ymax,ny+1)
ax.set_yticks(yticks)

nTicks=5
dT=int(Ttop/nTicks)
Tticklist=range(0,Ttop+1,dT)
ax.set_zticks(Tticklist)

#fig.colorbar(surf, shrink=0.5, aspect=5)
plt.show()
```

Numerical values are listed below with analytical values from (6.25) enclosed in paranthesis.

$$T_{11} = T_{31} = 1.578\ (1.406),\ T_{12} = T_{32} = 4.092\ (3.725)$$
$$T_{13} = T_{33} = 9.057\ (8.483),\ T_{14} = T_{34} = 19.620\ (18.945)$$
$$T_{15} = T_{35} = 43.193\ (43.483),\ T_{21} = 2.222\ (1.987),\ T_{22} = 5.731\ (5.261)$$
$$T_{23} = 12.518\ (11.924),\ T_{24} = 26.228\ (26.049),\ T_{25} = 53.154\ (54.449)$$

The structure of the coefficient matrix will not necessarily be so regular in all cases, e.g. more complicated operators than the Laplace-operator or even more so for non-linear problems. Even though the matrix will predominantly be sparse also for these problems, the requirements for fast solutions and efficent storage will be harder to obtain for the problems. For such problems iterative methods are appealing, as they often are relatively simple to program and hoffer effective memory management. However, they are often hapered by convergence challanges. We will look into iterative methods in a later section.

6.3.1 Neumann boundary conditions

Here we consider a heat conduction problem where we prescribe homogeneous Neuman boundary conditions, i.e. zero derivatives, at $x = 0$ and $y = 0$, as illustrated in figure 6.7.

Mathematically the problem in Figure 6.7, is specified with the governing equation (6.17) which we reiterate for convenience:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{6.26}$$

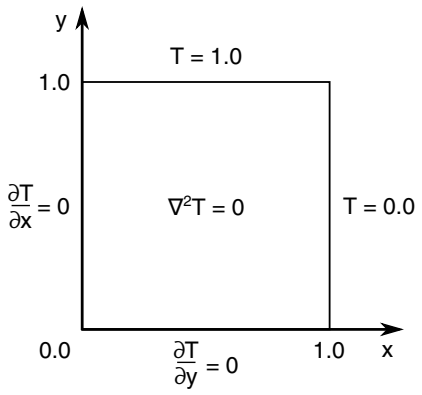

Figure 6.7: Laplace-equation for a rectangular domain with homogeneous Neumann boundary conditions for $x = 0$ and $y = 0$.

with homogeneous Neumann boundary conditions at:

$$\frac{\partial T}{\partial x} = 0 \quad \text{for } x = 0, \text{ and } 0 < y < 1 \tag{6.27}$$

$$\frac{\partial T}{\partial y} = 0 \quad \text{for } y = 0, \text{ and } 0 < x < 1 \tag{6.28}$$

and prescribed values (Dirichlet boundary conditions) for:

$$T = 1 \quad \text{for } y = 1, \text{ and } 0 \leq x \leq 1$$
$$T = 0 \quad \text{for } x = 1, \text{ and } 0 \leq y < 1$$

By assuming $\Delta x = \Delta y$, as for the Dirichlet problem, an identical, generic difference equation is obtained as in (6.18). However, at the boundaries we need to take into account the Neumann boundary conditions. We will take a closer look at $\dfrac{\partial T}{\partial x} = 0$ for $x = 0$. The other Neumann boundary condition is treated in the same manner. Notice that we have discontinuities in the corners $(1, 0)$ og $(1, 1)$, additionally the corner $(0, 0)$ may cause problems too.

A central difference approximation (see Figure 6.8) of $\dfrac{\partial T}{\partial x} = 0$ at $i = 0$ yields:

$$\frac{T_{1,j} - T_{-1,j}}{2\Delta x} = 0 \rightarrow T_{-1,j} = T_{1,j} \tag{6.29}$$

where we have introduced ghostcells with negative indices outside the physical domain to express the derivative at the boundary. Note that the requirement of a zero derivative relate the value of the ghostcells to the values inside the physical domain.

One might also approximate $\dfrac{\partial T}{\partial x} = 0$ by forward differences (see (6.15)) for a generic discrete point (i, j):

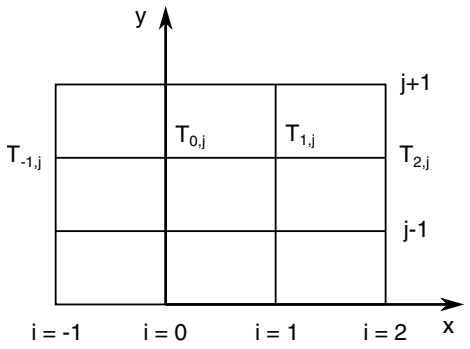

Figure 6.8: Illustration of how ghostcells with negative indices may be used to implement Neumann boundary conditions.

$$\left.\frac{\partial T}{\partial x}\right|_{i,j} = \frac{-3T_{i,j} + 4T_{i+1,j} - T_{i+2,j}}{2\Delta x}$$

which for $\dfrac{\partial T}{\partial x} = 0$ for $x = 0$ reduce to:

$$T_{0,j} = \frac{4T_{1,j} - T_{2,j}}{3} \tag{6.30}$$

For the problem at hand, illustrated in Figure 6.7, central difference approximation (6.29) and forward difference approximation (6.30) yields fairly similar results, but (6.30) results in a shorter code when the methods in 6.4 are employed.

To solve the problem in Figure 6.7 we employ the numerical stencil (6.19) for the unknows in the field (not influenced by the boundaries)

$$T_w + T_e + T_s + T_n - 4T_m = 0 \tag{6.31}$$

where we used the same ordering as given in Figure 6.9. For the boundary conditions we have chosen to implement the by means of (6.29) which is illustrated in Figure 6.9 by the dashed lines.

Before setting up the complete equation system it normally pays of to look at the boundaries, like the lowermost boundary, which by systematic usage of (6.31) along with the boundary conditions in (6.28) yield:

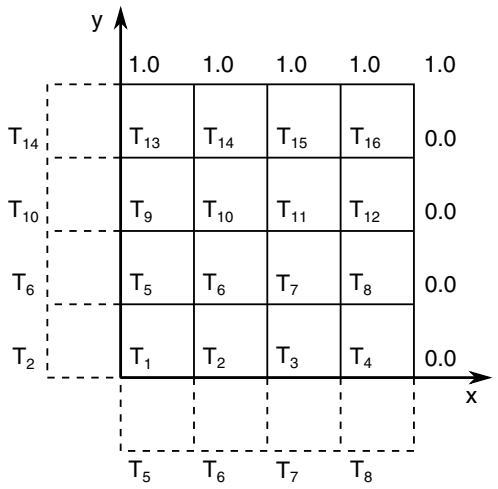

Figure 6.9: Von Neumann boundary conditions with ghost cells.

$$
\begin{aligned}
2T_2 + 2T_5 - 4T_1 &= 0 \\
T_1 + 2T_6 + T_3 - 4T_2 &= 0 \\
T_2 + 2T_7 + T_4 - 4T_3 &= 0 \\
T_3 + 2T_8 - 4T_4 &= 0
\end{aligned}
$$

The equations for the upper boundary become:

$$
\begin{aligned}
T_9 + 2T_{14} - 4T_{13} &= -1 \\
T_{10} + T_{13} + T_{15} - 4T_{14} &= -1 \\
T_{11} + T_{14} + T_{16} - 4T_{15} &= -1 \\
T_{12} + T_{15} - 4T_{16} &= -1
\end{aligned}
$$

Notice that for the coarse mesh with only 16 unknown temperatures (see Figure 6.7) only 4 (T_6, T_7, T_{10}, and T_{11}) are not explicitly influenced by the

boundaries.Finally, the complete discretized equation system for the problem may be represented:

$$\begin{pmatrix} -4 & 2 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -4 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -4 & 2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 2 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 \end{pmatrix} \cdot \begin{pmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \\ T_6 \\ T_7 \\ T_8 \\ T_9 \\ T_{10} \\ T_{11} \\ T_{12} \\ T_{13} \\ T_{14} \\ T_{15} \\ T_{16} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -1 \\ -1 \\ -1 \\ -1 \end{pmatrix} \tag{6.32}$$

```
# src-ch7/laplace_Neumann.py; Visualization.py @ git@lrhgit/tkt4140/src/src-ch7/Visualization.py;

import numpy as np
import scipy
import scipy.linalg
import scipy.sparse
import scipy.sparse.linalg
import matplotlib.pylab as plt
import time
from math import sinh

#import matplotlib.pyplot as plt

# Change some default values to make plots more readable on the screen
LNWDT=2; FNT=15
plt.rcParams['lines.linewidth'] = LNWDT; plt.rcParams['font.size'] = FNT

def setup_LaplaceNeumann_xy(Ttop, Tright, nx, ny):
    """ Function that returns A matrix and b vector  of the laplace Neumann heat problem A*T=b using
        and assuming dx=dy, based on numbering with respect to x-dir, e.g:

            1   1   1              -4  2  2  0  0  0          0
        T6  T5  T6  0               1 -4  0  2  0  0          0
        T4  T3  T4  0               1  0 -4  2  1  0          0
        T2  T1  T2  0     --> A =   0  1  1 -4  0  1    ,b =  0
            T3  T4                  0  0  1  0 -4  2         -1
                                    0  0  0  1  1 -4         -1

        T = [T1, T2, T3, T4, T5, T6]^T

    Args:
        nx(int): number of elements in each row in the grid, nx=2 in the example above
        ny(int): number of elements in each column in the grid, ny=3 in the example above
```

```
        Returns:
            A(matrix): Sparse matrix A, in the equation A*T = b
            b(array): RHS, of the equation A*t = b
    """

    n = (nx)*(ny) #number of unknowns
    d = np.ones(n) # diagonals
    b = np.zeros(n) #RHS

    d0 = d.copy()*-4
    d1_lower = d.copy()[0:-1]
    d1_upper = d1_lower.copy()

    dnx_lower = d.copy()[0:-nx]
    dnx_upper = dnx_lower.copy()

    d1_lower[nx-1::nx] = 0 # every nx element on first diagonal is zero; starting from the nx-th ele
    d1_upper[nx-1::nx] = 0
    d1_upper[::nx] = 2 # every nx element on first upper diagonal is two; stating from the first ele
                # this correspond to all equations on border (x=0, y)

    dnx_upper[0:nx] = 2 # the first nx elements in the nx-th upper diagonal is two;
                        # This correspond to all equations on border (x, y=0)

    b[-nx:] = -Ttop
    b[nx-1::nx] += -Tright

    A = scipy.sparse.diags([d0, d1_upper, d1_lower, dnx_upper, dnx_lower], [0, 1, -1, nx, -nx], forma

    return A, b

if __name__ == '__main__':
    from Visualization import plot_SurfaceNeumann_xy
    # Main program
    # Set temperature at the top
    Ttop=1
    Tright = 0.0
    xmax=1.0
    ymax=1.

    # Set simulation parameters
    #need hx=(1/nx)=hy=(1.5/ny)

    Nx = 10
    h=xmax/Nx
    Ny = int(ymax/h)

    A, b = setup_LaplaceNeumann_xy(Ttop, Tright, Nx, Ny)

    Temp = scipy.sparse.linalg.spsolve(A, b)

    plot_SurfaceNeumann_xy(Temp, Ttop, Tright, xmax, ymax, Nx, Ny)

#    figfile='LaPlace_vNeumann.png'
#    plt.savefig(figfile, format='png',transparent=True)
    plt.show()
```

The analytical solution is given by:

$$T(x,y) = \sum_{n=1}^{\infty} A_n \cosh(\lambda_n y) \cdot \cos(\lambda_n x), \tag{6.33}$$

$$\text{der } \lambda_n = (2n-1) \cdot \frac{\pi}{2}, \; A_n = 2\frac{(-1)^{n-1}}{\lambda_n \cosh(\lambda_n)}, \; n = 1, 2, \ldots \tag{6.34}$$

The solution for the problem illustrated Figure 6.7 is computed and visualized the the python code above. The solution is illustrated in Figure 6.10.

Marit 2: Har ikke endret figuren

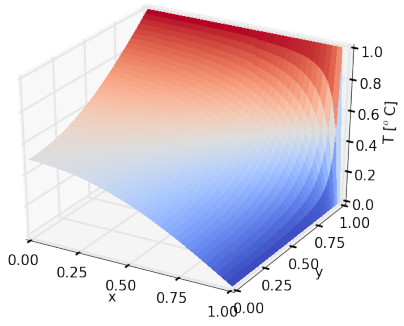

Figure 6.10: Solution of the Laplace equation with Neuman boundary conditions.

6.4 Iterative methods for linear algebraic equation systems

We will in this section seek to illustrate how classical iterative methods for linear algebraic systems of equations, such as Jacobi, Gauss-Seidel or SOR, may be applied for the numerical solution of linear, ellipitical PDEs, whereas criteria for convergence of such iterative schemes can be seen in Section 7.3.2 of the Numeriske Beregninger.

Discretizations of PDEs, in particular linear elliptic PDEs, will result in a system of linear, algebraic equations on the form $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$, which may be presented on component for a system of three equations as:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1 \tag{6.35}$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2 \tag{6.36}$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \tag{6.37}$$

We rewrite (6.37) as an iterative scheme by normally referred to as:

Jacobi's method

$$x_1^{m+1} = x_1^m + \frac{1}{a_{11}}[b_1 - (a_{11}x_1^m + a_{12}x_2^m + a_{13}x_3^m)] \tag{6.38}$$

$$x_2^{m+1} = x_2^m + \frac{1}{a_{22}}[b_2 - (a_{21}x_1^m + a_{22}x_2^m + a_{23}x_3^m)] \tag{6.39}$$

$$x_3^{m+1} = x_3^m + \frac{1}{a_{33}}[b_3 - (a_{31}x_1^m + a_{32}x_2^m + a_{33}x_3^m)] \tag{6.40}$$

where we have introduced m as an iteration counter. In the case when x_i^m is a solution of (6.37), the expression in the brackets of (6.40), will be zero and thus $x_i^{m+1} = x_i^m$, i.e. convergence is obtained. Whenever, x_i^m is *not* a solution of (6.37), the expression in the brackets of (6.40), will represent a correction to the previous guess at iteration m.

A more compact representation of (6.40) may be used for a system of n equations:

$$x_i^{m+1} = x_i^m + \delta x_i \tag{6.41}$$

$$\delta x_i = \frac{1}{a_{ii}}\left[b_i - \sum_{j=1}^{n} a_{ij}x_j^m\right], \; i = 1, 2, \ldots, n, \; m = 0, 1, \ldots \tag{6.42}$$

To start the iteration process one must chose a value $x = x_0$ at iteration $m = 0$.

From (6.40) we see that Jacobi's method may be improved by substitution of x_1^{m+1} in the second equation and for x_1^{m+1} and x_2^{m+1} in the third equation, to yield:

Gauss-Seidel's method

$$x_1^{m+1} = x_1^m + \frac{1}{a_{11}}[b_1 - (a_{11}x_1^m + a_{12}x_2^m + a_{13}x_3^m)] \tag{6.43}$$

$$x_2^{m+1} = x_2^m + \frac{1}{a_{22}}[b_2 - (a_{21}x_1^{m+1} + a_{22}x_2^m + a_{23}x_3^m)] \tag{6.44}$$

$$x_3^{m+1} = x_3^m + \frac{1}{a_{33}}[b_3 - (a_{31}x_1^{m+1} + a_{32}x_2^{m+1} + a_{33}x_3^m)] \tag{6.45}$$

The successively improved Jacobi's method, is normally referred to as Gauss-Seidel's method.

The incremental change may be multiplied with a factor ω, to yield another variant of the iterative scheme:

SOR method

$$
\begin{aligned}
x_1^{m+1} &= x_1^m + \frac{\omega}{a_{11}}[b_1 - (a_{11}x_1^m + a_{12}x_2^m + a_{13}x_3^m)] \\
x_2^{m+1} &= x_2^m + \frac{\omega}{a_{22}}[b_2 - (a_{21}x_1^{m+1} + a_{22}x_2^m + a_{23}x_3^m)] \\
x_3^{m+1} &= x_3^m + \frac{\omega}{a_{33}}[b_3 - (a_{31}x_1^{m+1} + a_{32}x_2^{m+1} + a_{33}x_3^m)]
\end{aligned} \tag{6.46}
$$

A general version of (6.46) may be presented:

$$x_i^{m+1} = x_i^m + \delta x_i \tag{6.47}$$

$$\delta x_i = \frac{\omega}{a_{ii}}\left[b_i - \left(\sum_{k=1}^{i-1} a_{ik}x_k^{m+1} + \sum_{k=1}^{n} a_{ik}x_k^m\right)\right],\ i = 1, 2, \ldots, n, \tag{6.48}$$

The factor ω is denoted the *relaxation parameter* or the *relaxation factor*. The method in (6.48) is commonly referred to as the *successive over relaxation method* when $\omega > 1$ or simply abbreviated to the SOR method. With $\omega = 1$ Gauss-Seidel's method is retrieved.

The relaxation factor ω may be shown to be in the range $(0, 2)$ for Laplace/Poisson equations, but naturally $\omega > 1$ is most efficient. We will not use the SOR method is presented in (6.37), but rather use the difference equations directly.

Let us first consider Poisson's equation in two physical dimensions:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y) \tag{6.49}$$

which after discretization with central differences, with $\Delta x = \Delta y = h$ results in the following difference equation:

$$u_{i-1,j} + u_{i+1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j} - h^2 \cdot f_{i,j} = 0 \tag{6.50}$$

We discretize our two-dimensional domain in the normal manner as:

$$
\begin{aligned}
x_i &= x_0 + i \cdot h, \qquad i = 0, 1, 2, \ldots, n_x \\
y_j &= y_0 + j \cdot h, \qquad j = 0, 1, 2, \ldots, n_y
\end{aligned} \tag{6.51}
$$

where n_x and n_y denote the number of grid cells in the $x-$ and y-direction, respectively (see 6.3).

By using the general SOR method (6.48) on (6.50) and (6.51) we get the following iterative scheme:

$$u_{i,j}^{m+1} = u_{i,j}^m + \delta u_{i,j} \tag{6.52}$$

$$\delta u_{i,j} = \frac{\omega}{4}\left[u_{i-1,j}^{m+1} + u_{i,j-1}^{m+1} + u_{i+1,j}^m + u_{i,j+1}^m - 4u_{i,j}^m - h^2 \cdot f_{i,j}\right] \tag{6.53}$$

The scheme in (6.53) may be reformulated by introducing the residual $R_{i,j}$

$$R_{i,j} = \left[u^{m+1}_{i-1,j} + u^{m+1}_{i,j-1} + u^{m}_{i+1,j} + u^{m}_{i,j+1} - 4u^{m}_{i,j} - h^2 \cdot f_{i,j}\right] \tag{6.54}$$

Note that the residual $R_{i,j}$ is what is left when a non correct solution is plugged into to the difference equation (6.50) in a dicrete point (i, j). By introducing the residual (6.54) into (6.53), the following iterative scheme may be obtained:

$$u^{m+1}_{i,j} = u^{m}_{i,j} + \frac{\omega}{4} R_{i,j} \tag{6.55}$$

We will now solve the example in Figure 6.4, 6.3, with the iterative SOR-scheme in (6.53).

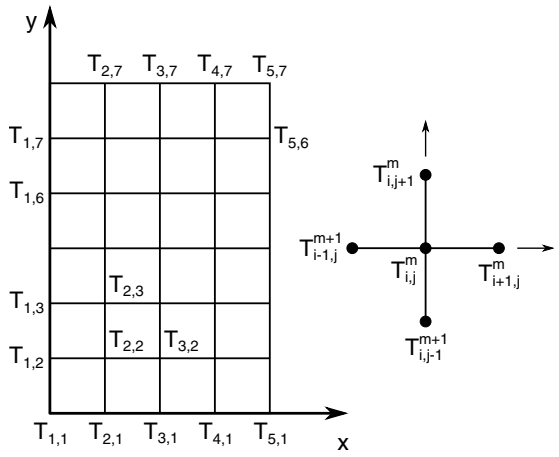

Figure 6.11: Rectangular domain as in Figure 6.4 but boundary nodes are unknown due to Neumann boundary conditions.

In Figure 6.11 we have introduced a new indexation as compared to the problem in Figure 6.4, and we do not explicitly account for symmetry.

The boundary values are imposed at the boundaries, whereas the rest of the unknown values are initialized to zero:

```
# Initialize T and impose boundary values
T = np.zeros_like(X)

T[-1,:] = Ttop
T[0,:] = Tbottom
T[:,0] = Tleft
T[:,-1] = Tright
```

In this first simple program we use $\omega = 1.5$ and perform a fixed number of 20 iterations as shown in the program `laplace_sor.py` below with $h = 0.25$:

```
# src-ch7/laplace_Diriclhet2.py
import numpy as np
import scipy
import scipy.linalg
import scipy.sparse
import scipy.sparse.linalg
import matplotlib.pylab as plt
import time
from math import sinh
from astropy.units import dT

#import matplotlib.pyplot as plt

# Change some default values to make plots more readable on the screen
LNWDT=2; FNT=15
plt.rcParams['lines.linewidth'] = LNWDT; plt.rcParams['font.size'] = FNT

# Set temperature at the top
Ttop=10
Tbottom=0.0
Tleft=0.0
Tright=0.0

xmax=1.0
ymax=1.5

# Set simulation parameters
#need hx=(1/nx)=hy=(1.5/ny)
Nx = 40
h=xmax/Nx
Ny = int(ymax/h)

nx = Nx-1
ny = Ny-1
n = (nx)*(ny) #number of unknowns

# surfaceplot:
x = np.linspace(0, xmax, Nx + 1)
y = np.linspace(0, ymax, Ny + 1)

X, Y = np.meshgrid(x, y)



# Initialize T and impose boundary values
T = np.zeros_like(X)

T[-1,:] = Ttop
T[0,:] = Tbottom
T[:,0] = Tleft
T[:,-1] = Tright

tic=time.clock()
omega = 1.5
for iteration in range(20):
    for j in range(1,ny+1):
        for i in range(1, nx + 1):
            R = (T[j,i-1]+T[j-1,i]+T[j,i+1]+T[j+1,i]-4.0*T[j,i])
            dT = 0.25*omega*R
            T[j,i]+=dT
```

```
toc=time.clock()
print 'GS solver time:',toc-tic
```

As opposed to the above scheme with a fixed number of iterations, we seek an iteration scheme with a proper stop criteria, reflecting that our solution approximates the solution with a certain accuracy. Additionally, we would like to find and relaxation parameter ω which reduces the number of required iterations for a given accuracy. These topics will be addressed in the following section.

6.4.1 Stop criteria

Examples of equivalent stop criteria are:

- $\max(\delta T_{i,j}) < \varepsilon_a$
- $\max\left(\dfrac{\delta T_{i,j}}{T_{i,j}}\right) < \varepsilon_r$
- The residual $R_{i,j}$ (6.54) may also be used
- Other alternatives:

$$\frac{1}{N}\sum_i\sum_j |\delta T_{i,j}| < tol_a,\ \ \frac{1}{N}\sum_i\sum_j \left|\frac{\delta T_{i,j}}{T_{i,j}}\right| < tol_r,\ \ |T_{i,j}| \neq 0 \tag{6.56}$$

where $N = n_x\, n_y$ is the total number of unknowns. In (6.56) the residual may be used rather than $\delta T_{i,j}$.

In the first expression we use an abolsute tolerance, whereas a relative tolerance is suggested in the latter. We choose to use the following alternative for (6.56):

$$\frac{\sum_i \sum_j |\delta T_{i,j}|}{\sum_i \sum_j |T_{i,j}|} < tol_r, \qquad \frac{\max\left(|\delta T_{i,j}|\right)}{\max\left(|T_{i,j}|\right)} < tol_r \tag{6.57}$$

The expression (6.57) represents some kind of an average relative stop criteria, as we sum over all computational grid points in these formulas.

From Figure 6.12, we observe that the number of iterations is a function of both the relaxation parameter ω and the grid size h.

Marit 2: Har ikke endret figuren

A more generic program for the Poisson problem (6.49) with a stop criteria $\frac{\max(|\delta T_{i,j}|)}{\max(|T_{i,j}|)} < tol_r$ and variable grid size h is included below. As for the previous code with a fixed number of iterations, both a Jacobi-scheme with array-slicing and a Gauss-Seidel scheme is included for comparison.

```
# src-ch7/laplace_Diriclhet2.py
import numpy as np
import scipy
import scipy.linalg
import scipy.sparse
```

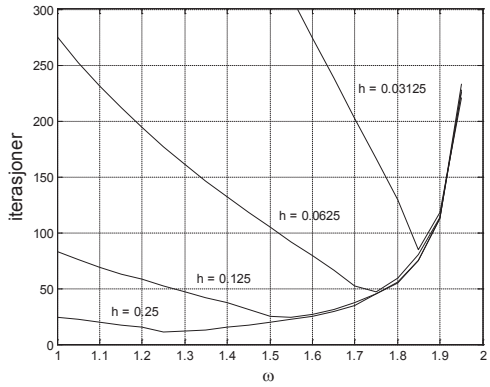

Figure 6.12: Number of iterations as a function of ω and h with $tol_r = 10^{-5}$.

```
import scipy.sparse.linalg
import matplotlib.pylab as plt
import time
from math import sinh
from astropy.units import dT

#import matplotlib.pyplot as plt

# Change some default values to make plots more readable on the screen
LNWDT=2; FNT=15
plt.rcParams['lines.linewidth'] = LNWDT; plt.rcParams['font.size'] = FNT

# Set temperature at the top
Ttop=10
Tbottom=0.0
Tleft=0.0
Tright=0.0

xmax=1.0
ymax=1.5

# Set simulation parameters
#need hx=(1/nx)=hy=(1.5/ny)
Nx = 20
h=xmax/Nx
Ny = int(ymax/h)

nx = Nx-1
ny = Ny-1
n = (nx)*(ny) #number of unknowns

# surfaceplot:
x = np.linspace(0, xmax, Nx + 1)
y = np.linspace(0, ymax, Ny + 1)

X, Y = np.meshgrid(x, y)

T = np.zeros_like(X)
```

```
# set the imposed boudary values
T[-1,:] = Ttop
T[0,:] = Tbottom
T[:,0] = Tleft
T[:,-1] = Tright

T2 = T.copy()

reltol=1.0e-3

omega = 1.5
iteration = 0
rel_res=1.0

# Gauss-Seidel iterative solution
tic=time.clock()
while (rel_res > reltol):
    dTmax=0.0
    for j in range(1,ny+1):
        for i in range(1, nx + 1):
            R = (T[j,i-1]+T[j-1,i]+T[j,i+1]+T[j+1,i]-4.0*T[j,i])
            dT = 0.25*omega*R
            T[j,i]+=dT
            dTmax=np.max([np.abs(dT),dTmax])

    rel_res=dTmax/np.max(np.abs(T))
    iteration+=1

toc=time.clock()
print "Gauss-Seidel solver time:\t{0:0.2f}. \t Nr. iterations {1}".format(toc-tic,iteration)


iteration = 0
rel_res=1.0
# Jacobi iterative solution
tic=time.clock()
while (rel_res > reltol):
    R2 = (T2[1:-1,0:-2]+T2[0:-2,1:-1]+T2[1:-1,2:]+T2[2:,1:-1]-4.0*T2[1:-1,1:-1])
    dT2 = 0.25*R2
    T2[1:-1,1:-1]+=dT2
    rel_res=np.max(dT2)/np.max(T2)
    iteration+=1

toc=time.clock()
print "Jacobi solver time:\t\t{0:0.2f}. \t Nr. iterations {1}".format(toc-tic,iteration)
```

6.4.2 Optimal relaxation parameter

By optimal, we mean the relaxation parameter value that results in the lowest possible iteration number for ω hold constant throughout the computation. For Laplace and Poisson equations in a rectangular domain it is possible to calculate an optimal ω, which we will call as theoretically optimal.

Let L_x and L_y be the extent of the rectangular domain in the x- and y-directions. Taking the same space discretization h in both directions, we set: $n_x = \frac{L_x}{h}$, $n_y = \frac{L_y}{h}$, with n_x and n_y being the number of intervals in the x- and

y-directions, respectively. n_x and n_y must be integers. The theoretical optimal ω is then given by:

$$\rho = \frac{1}{2}[\cos(\pi/n_x) + \cos(\pi/n_y)] \tag{6.58}$$

$$\omega = \frac{2}{1+\sqrt{1-\rho^2}} \tag{6.59}$$

If the interval length h is different in x- og y-directions with $h = h_x$ in x-direction and $h = h_y$ in y-direction, instead of (6.59), the following holds:

$$\rho = \frac{\cos(\pi/n_x) + (h_x/h_y)^2 \cdot \cos(\pi/n_y)}{1+(h_x/h_y)^2} \tag{6.60}$$

Moreover, if the domain is not rectangular, one can use Garabedian's estimate:

$$\omega = \frac{2}{1+3.014 \cdot h/\sqrt{A}}, \tag{6.61}$$

where A is the area of the domain.

Let us compute some numerical values for these formulas for the case in which $L_x = 1,\ L_y = 1.5$ and $A = L_x \cdot L_y = 1.5$.

h	(6.59)	(6.61)
0.25	1.24	1.24
0.125	1.51	1.53
0.0625	1.72	1.74
0.03125	1.85	1.86

We see that Garabedian's estimate is well in line with the theoretically exact values in this case. Results reported in Figure 6.12 are also in good agreement with the values in this table.

6.4.3 Example using SOR

We now solve the temperature problem shown in Figure 6.7 (se 6.3.1). The numbering convention used here is depicted in Figure 6.13.

Here we use ghost points as indicated by the dashed lines. For $T_{1,1}$ we get:

$$T_{1,1} = \frac{1}{4}(T_{1,2} + T_{1,2} + T_{2,1} + T_{2,1}) = \frac{1}{2}(T_{1,2} + T_{2,1}) \tag{6.62}$$

The calculation is started by iterating along $y = 0$, starting with $T_{2,1}$. Then iteration along $x = 0$ follows, starting with $T_{1,2}$. Afterwards we iterate in a double loop over inner points. Finally, $T_{1,1}$ is computed from (6.62). The algorithm is shown in code **lapsor2** in the next page. We have used the stopping criteria $\frac{\sum_i \sum_j |\delta T_{i,j}|}{\sum_i \sum_j |T_{i,j}|} < tol_r$ and optimal ω from (6.59) and denote $T = 0.5$ as initial guess for the entire domain, expect where boundary conditions are prescribed. Accuracy is as in the print out for **lap2v3** in 6.3.1.

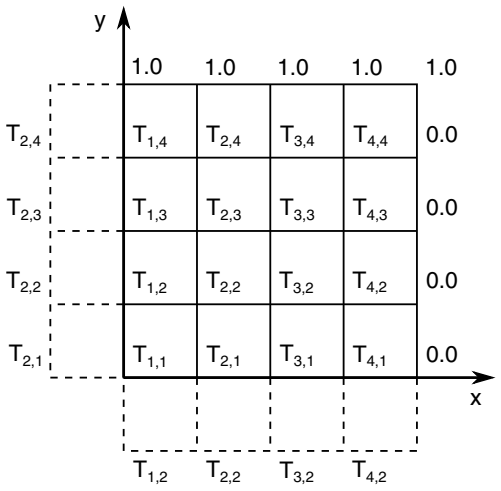

Figure 6.13: Ghost-cells are used to implement Neumann boundary conditions.

```
% program lapsor2
clear
net = 1;
h = 0.25;
hn = h/2^(net -1);
nx = 1/hn; ny = nx;
imax = nx + 1; % points in x-direction
jmax = ny + 1; % points in y-direction
T = 0.5*ones(imax,jmax); % temperatures
% --- Compute optimal omega ---
ro = cos(pi/nx);
omega = 2/(1 + sqrt(1 - ro^2));
T(1:imax,jmax) = 1;  % boundary values along y = 1
T(imax,1:jmax-1) = 0;% boundary values along x = 1
reltol = 1.0e-5; % relative iteration error
relres = 1.0; it = 0;
% --- Start iteration ---
while relres > reltol
it = it + 1;
Tsum = 0.0; dTsum = 0.0;
% --- boundary values along y = 0 ---
for i = 2: imax - 1
resid = 2*T(i,2) + T(i-1,1) + T(i+1,1) - 4*T(i,1);
dT = 0.25*omega*resid;
dTsum = dTsum + abs(dT);
T(i,1) = T(i,1) + dT;
Tsum = Tsum + abs(T(i,1));
end
% --- boundary values along x = 0 ---
for j = 2: jmax - 1
resid = 2*T(2,j) + T(1,j-1) + T(1,j+1) - 4*T(1,j);
dT = 0.25*omega*resid;
```

```
dTsum = dTsum + abs(dT);
T(1,j) = T(1,j) + dT;
Tsum = Tsum + abs(T(1,j));
end
for i = 2: imax-1
for j = 2: jmax-1
resid = T(i-1,j) + T(i,j-1) + T(i+1,j) + T(i,j+1)-4*T(i,j);
dT = 0.25*omega*resid;
dTsum = dTsum + abs(dT);
T(i,j) = T(i,j) + dT;
Tsum = Tsum + abs(T(i,j));
end
end
T(1,1) = 0.5*(T(2,1) + T(1,2));
relres = dTsum/Tsum;
end
```

6.4.4 Initial guess and boundary conditions

We expect faster convergence if we use initial guesses that are close to the problem's solution. This is typical of nonlinear equations, while we are more free to choose initial guesses when we solve linear equations without exceeding the convergence rate. For example, for the temperature problem in Figure 6.4, there is little difference in number of iterations if we start the iteration by choosing $T = 0$ in the entire domain or if we start with $T = 1004$. The optimal ω for this case is also independent of the starting values. The situation is completely different for the case in Figure 6.7. Here we also solve a linear equation but we have more complicated boundary conditions, where we prescribe the temperature along two sides of the domain (Dirichlet conditions) and the derivative of the temperature along the two other sides (Neumann conditions) . In addition, the temperature is discontinuous in the corner x = 1, y = 1. In the corner x = 0, y = 0 the correct solution is $T = 0.5$, as shown by the analytical solution. If we exclude $T = 0.5$ as initial guess throughout the domain, we get fast convergence with optimal ω equal to the theoretical value obtained using (6.59). If we deviate slightly from $T = 0.5$ as the initial guess, then the optimal ω is no longer the theoretical optimal. The situation is as shown in Figure 6.14 below.

Marit 2: Har ikke endret figuren

The table below shows the theoretical optimal ω obtained with (6.59) and (6.61)

h	(6.59)	(6.61)
0.25	1.17	1.14
0.125	1.45	1.45
0.0625	1.67	1.68
0.03125	1.82	1.83

We see that the values reported in the table match values shown in Figure 6.14 when the initial guess for the iterative process is 0.5.

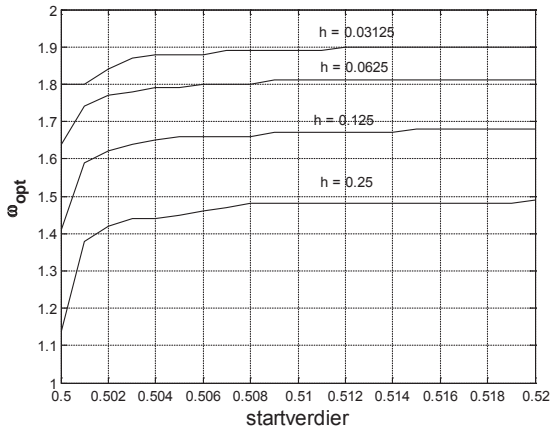

Figure 6.14: Effect of initial guesses on the number of iterations for a problem with sub-optimal ω.

6.4.5 Example: A non-linear elliptic PDE

In this example we will solve a non-linear elliptic Poisson equation for a square 2D-domain (see Figure 6.15) given by:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + u^2 = -1 \tag{6.63}$$

with Dirichlet boundary conditions, i.e precribed values $u = 0$ on all boundaries (see Figure 6.15).

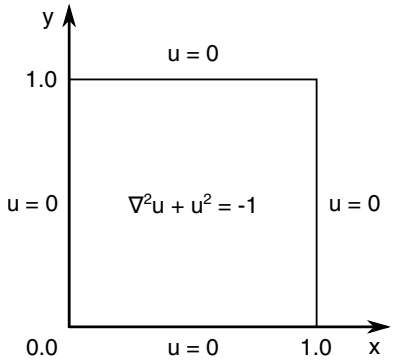

Figure 6.15: Solution domain and boundary conditions for a non-linear Poisson equation.

The PDE in (6.63) is only weakly non-linear, so-called semi-linear, but the approach for how to solve a non-linear elliptic PDE is still illustrated.

We discretize (6.63) with central differences over the domain in Figure 6.15 by a constant stepsize h in both physical directions and get the following system of equations when the source term on the right hands side has been moved to the left hand side:

$$\frac{1}{h^2}[u_{i+1,j} + u_{i-1,j} + u_{i,j-1} + u_{i,j`1} - 4u_{i,j}] + u_{i,j}^2 + 1 = 0$$

or equivalently with by introducing the function $f_{i,j}$:

$$f_{i,j} = u_{i+1,j} + u_{i-1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j} + h^2(u_{i,j}^2 + 1) = 0 \qquad (6.64)$$

with $x_i = h \cdot (i-1),\ i = 1, 2, \ldots$ and $y_j = h \cdot (j-1),\ j = 1, 2, \ldots$. A computational mesh for $h = 0.25$ is illustrated in Figure 6.16.

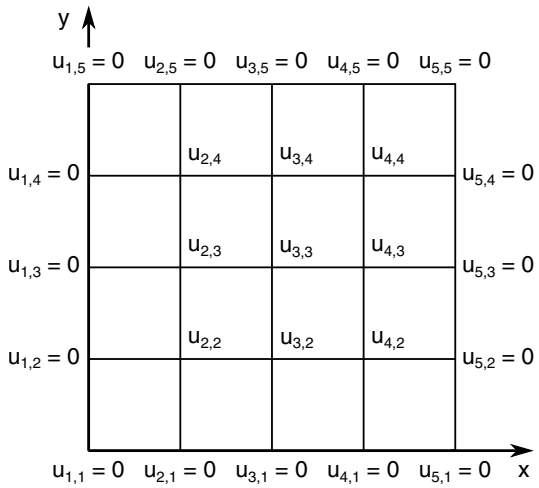

Figure 6.16: Computational mesh for the problem in Figure 6.15.

We will solve the non-linear function in (6.64) with Newton's method by changing each variable one at a time. In this case the iteration process becomes:

$$u_{i,j}^{m+1} = u_{i,j}^m + \delta u_{i,j} \qquad (6.65)$$

$$\delta u_{i,j} = -\omega \frac{f(u_{k,l})}{\frac{\partial f(u_{k,l})}{\partial u_{i,j}}} \qquad (6.66)$$

where:

$$u_{k,l} = u_{k,l}^{m+1} \text{ for } k < i,\ l < j \tag{6.67}$$

$$u_{k,l} = u_{k,l}^{m} \text{ ellers} \tag{6.68}$$

and

$$\frac{\partial f}{\partial u_{i,j}} = -4 + 2h^2 \cdot u_{i,j} \qquad \text{and} \qquad \delta u_{i,j} = \omega \frac{f}{4 - 2h^2 u_{i,j}} \tag{6.69}$$

We have implemented (6.64) and (6.66) to (6.69) in the python code `nonlin_poisson` below:

```
# Python Gauss-Seidel method
tic=time.clock()
while (rel_res > reltol):
    du_max=0.0
    for j in range(1,ny+1):
        for i in range(1, nx + 1):
            R = (U[j,i-1]+U[j-1,i]+U[j,i+1]+U[j+1,i]-4.0*U[j,i]) + h**2*(U[j,i]**2+1.0)
            df=4-2*h**2*U[j,i]
            dU = omega*R/df
            U[j,i]+=dU
            du_max=np.max([np.abs(dU),du_max])


    rel_res=du_max/np.max(np.abs(U))

    iteration+=1

toc=time.clock()
print "Python Gauss-Seidel CPU-time:\t{0:0.2f}. \t Nr. iterations {1}".format(toc-tic,iteration)
```

In this implementation we have used the following stop criterium:

$$\frac{\max \delta u_{i,j}}{\max u_{i,j}} \leq \text{tol}_r \tag{6.70}$$

and initialize the solution in the field (excluding boundaries) with $u = 0$.

We have also used (6.59) to estimat an optimal ω_{est}. In the table below we compare the estimated optimal ω_{est} with the real optimal ω_{opt} in case of an initial field of $u = 0$.

h	ω_{est}	ω_{opt}
0.25	1.17	1.19
0.125	1.45	1.46
0.0625	1.67	1.69
0.03125	1.82	1.83

We observe relatively good agreement between the two ω-values for a range of grid sizes h, even though the PDE is weakly non-linear.

As the Gauss-Seidel algorithm above involves a triple-loop (the iterative while-construct, pluss one loop in each physical direction), the naive python implementation above must be expected to be computationally expensive.

For comparison we have also implemented another solution to the problem by making use of numpy's array slicing capabilities:

```
# Jacobi iterative solution
tic=time.clock()
while (rel_res > reltol):
    R2 = (U2[1:-1,0:-2]+U2[0:-2,1:-1]+U2[1:-1,2:]+U2[2:,1:-1]-4.0*U2[1:-1,1:-1]) + h**2*(U2[1:-1,1:-1
    df=4-2*h**2*U2[1:-1,1:-1]
    dU2 = R2/df
    U2[1:-1,1:-1]+=dU2
    rel_res=np.max(dU2)/np.max(U2)
    iteration+=1

toc=time.clock()
print "Jacobi CPU-time:\t\t{0:0.2f}. \t Nr. iterations {1}".format(toc-tic,iteration)
```

Note with this implementation all values on the right hand side are at the previous iteration, and thus the method must me denoted a Jacobian algorithm.

Finally, we have implemented a third method the Gauss-Seidel method (6.66) with Cython. Cython is an optimising static compiler (based on Pyrex) for both the Python programming language and the extended Cython programming language. The ambition is to makes the writing of computationally superior C extensions for Python as easy as Python itself.

The expensive triple loop is implemented in a typed Cython function which looks very much like the Python implementation, save for the type declarations.

```
import cython
cimport cython

import numpy as np
cimport numpy as np

DTYPE = np.float64
ctypedef np.float64_t DTYPE_t

@cython.boundscheck(False)
@cython.wraparound(False)
@cython.nonecheck(False)


cpdef gauss(np.ndarray[DTYPE_t, ndim=2] U, double reltol, double h, double omega):
    cdef Py_ssize_t i, j, it
    cdef double rel_res, dU_max, df, dU, R

    cdef unsigned int rows = U.shape[0]
    cdef unsigned int cols = U.shape[1]

    it=0
    rel_res=1.0

    itmax=100

    while ((rel_res>reltol) and (it<=itmax)):
```

```
        dU_max=0.0
        for j in range(1,rows-2):
                for i in range(1,cols-2):
                    R = (U[j,i-1]+U[j-1,i]+U[j,i+1]+U[j+1,i]-4.0*U[j,i]) + h**2*(U[j,i]**2+1.0)
                    df=4.0-2*h**2*U[j,i]
                    dU =  omega*R/df
                    U[j,i]+=dU
                    dU_max=np.max([np.abs(dU),dU_max])

        rel_res=dU_max/np.max(np.abs(U[:,:]))
#         print 'rel_res', rel_res
        it+=1
    if (it>=itmax): print 'Terminated after max iterations'
    return U, rel_res, it
```

After compilation the Cython module is easily imported into our python-code which allows for comparison with the methods above as illustrated in the code below:

```
# Python Gauss-Seidel method
tic=time.clock()
while (rel_res > reltol):
    du_max=0.0
    for j in range(1,ny+1):
        for i in range(1, nx + 1):
            R = (U[j,i-1]+U[j-1,i]+U[j,i+1]+U[j+1,i]-4.0*U[j,i]) + h**2*(U[j,i]**2+1.0)
            df=4-2*h**2*U[j,i]
            dU = omega*R/df
            U[j,i]+=dU
            du_max=np.max([np.abs(dU),du_max])


    rel_res=du_max/np.max(np.abs(U))

    iteration+=1

toc=time.clock()
print "Python Gauss-Seidel CPU-time:\t{0:0.2f}. \t Nr. iterations {1}".format(toc-tic,iteration)


iteration = 0
rel_res=1.0

# Second method
# Jacobi iterative solution
tic=time.clock()
while (rel_res > reltol):
    R2 = (U2[1:-1,0:-2]+U2[0:-2,1:-1]+U2[1:-1,2:]+U2[2:,1:-1]-4.0*U2[1:-1,1:-1]) + h**2*(U2[1:-1,1:-1
    df=4-2*h**2*U2[1:-1,1:-1]
    dU2 = R2/df
    U2[1:-1,1:-1]+=dU2
    rel_res=np.max(dU2)/np.max(U2)
    iteration+=1

toc=time.clock()
print "Jacobi CPU-time:\t\t{0:0.2f}. \t Nr. iterations {1}".format(toc-tic,iteration)

# Third method
# Cython Gauss-Seidel method
rel_res=1.0
```

```
tic=time.clock()
U3, relreturn, itsused=gs.gauss(U3,reltol,h, omega)
toc=time.clock()
print "Cython Gauss-Seidel CPU-time:\t{0:0.2f}. \t Nr. iterations {1}".format(toc-tic,itsused)
```

By running the code we get the follwing results for comparison:

```
omega=1.85
Python Gauss-Seidel CPU-time:        1.62.          Nr. iterations 56
Jacobi CPU-time:                0.04.          Nr. iterations 492
Cython Gauss-Seidel CPU-time:        0.97.          Nr. iterations 56
```

which illstrates the extreme efficiency whenever a numerical scheme may be expressed by means of numpy-array-slicing. Note that the numpy-array-slicing Jacobi scheme converge slower than the Gauss-Seidel scheme in terms of iterations, and need apporximately 10 times as many iterations as the Gauss-seidel algorithm. But even in this situation the CPU-speed of the Jacobi scheme with array-slicing is approximately 40 times faster than the Gauss-Seidel scheme in python. We also observe that the Cython implementation Gauss-Seidel scheme is approximately 1.7 times faster than the python counter part, but not by far as fast as the Jacobi scheme with array-slicing, which is approximately 24 times faster.

```
emfs /src-ch7/ #python #package nonlin_poisson_sor.py  @ git@lrhgit/tkt4140/src/src-ch7/nonlin_poiss
```

Exercise 9: Symmetric solution

Prove that the analytical solution of the temperature field $T(x, y)$ in (6.25) is symmetric around $x = 0.5$.

Exercise 10: Stop criteria for the Poisson equation

Implement the various stop criteria outlined in 6.4.1 for the Possion equation in two dimensions (6.49).

Chapter 7

Diffusjonsproblemer

7.1 Introduction

A one-dimensional diffusion equation takes the canonical form:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} \tag{7.1}$$

where t is an evolutionary variable, which might be both a time-coordinaten and a spatial coordinate. Some classical diffusion problems are listed below:

- Heat conduction

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2}$$

- Unsteady boundary layers (Stokes' problem):

$$\frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial y^2}$$

- Linearized boundary layer equation with x as an evolutionary variable:

$$\frac{\partial u}{\partial x} = \frac{\nu}{U_0} \frac{\partial^2 u}{\partial y^2}$$

- Flow in porous media:

$$\frac{\partial u}{\partial t} = c \frac{\partial^2 u}{\partial x^2}$$

Our model equation (7.1) may be classified according to (5.18):

$$A \frac{\partial^2 \phi}{\partial x^2} + B \frac{\partial^2 \phi}{\partial x \partial y} + C \frac{\partial^2 \phi}{\partial y^2} + f = 0 \tag{7.2}$$

$$A \cdot (dy)^2 - B \cdot dy \cdot dx + C \cdot (dx)^2 = 0 \tag{7.3}$$

$$\lambda_{1,2} = \frac{B \pm \sqrt{B^2 - 4AC}}{2A} \tag{7.4}$$

$B = C = 0$ and $A = 1$ which by substitution in (5.33) and (5.35) yield:

$$dt = 0,\ B^2 - 4AC = 0$$

And we find that (7.1) is a parabolic PDE with the characteristics given by $t =$ constant. By dividing dt with dx we get:

$$\frac{dt}{dx} = 0 \rightarrow \frac{dx}{dt} = \infty \tag{7.5}$$

which corresponds to an infinite propagation speed along the characteristic curve $t =$ constant.

7.2 Confined, unsteady Couette flow

The classical versicon of unsteady Couette flow with $b = \infty$ has been presented as Stokes first problem and discussed in section (3.3).

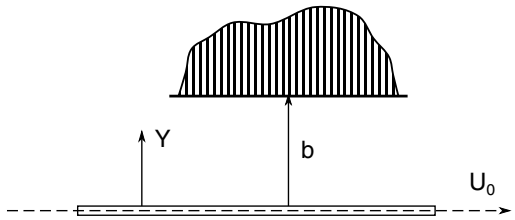

Figure 7.1: Confined, unsteady Couette flow or channel flow. The channel with is b

In this section we will look at the problem of unsteady Couette flow, confined by two walls, which has the following governing equation:

$$\frac{\partial U}{\partial \tau} = \nu \frac{\partial^2 U}{\partial Y^2},\ 0 < Y < b \tag{7.6}$$

with the following boundary conditions, representing the presence of the two walls or channel if you like:

$$\left.\begin{array}{rr} U(0,\ \tau) = & U_0 \\ U(b,\ \tau) = & 0 \end{array}\right\} = \tau \geq 0 \tag{7.7}$$

Futher, the parabolic problem also needs initial conditions to be solved and we assume:

$$U(Y,\ \tau) = 0,\ \tau < 0 \tag{7.8}$$

In section 3.3 we have presented several ways to render (7.6)) dimensionless, and for the current problem we introduce the following dimesionless variables:

$$y = \frac{Y}{b},\ u = \frac{U}{U_0},\ t = \frac{\tau\nu}{b^2} \tag{7.9}$$

which allow (7.6) to be written:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial y^2},\ 0 < y < 1 \tag{7.10}$$

with the corresponding boundary conditions:

$$\left.\begin{array}{rl} u(0,\ t) = & 1 \\ u(1,\ t) = & 0 \end{array}\right\},\ t \geq 0 \tag{7.11}$$

and initial conditions:

$$u(y,\ t) = 0,\ t < 0 \tag{7.12}$$

As for the problemn in section 3.3, this present example may also be formulated as a heat conduction problem. The problem of confined, unsteady Couette flow has the following analytical solution:

$$u(y,t) = 1 - y - \frac{2}{\pi} \cdot \sum_{n=1}^{\infty} \frac{1}{n} \exp[-(n\pi)^2 t] \sin(n\pi y) \tag{7.13}$$

A detailed derivation of (7.13) may be found in appendix G.6 of Numeriske beregninger.

We discretize (7.10) by a forward difference for the time t:

$$\left.\frac{\partial u}{\partial t}\right|_j^n \approx \frac{u_j^{n+1} - u_j^n}{\Delta t} \tag{7.14}$$

and central differences for the spatial coordinate y:

$$\frac{\partial^2 u}{\partial y^2} \approx \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta y)^2} \tag{7.15}$$

where:

$$t_n = n \cdot \Delta t,\ n = 0, 1, 2, \ldots,\ y_j = j \cdot \Delta y,\ j = 0, 1, 2, \ldots$$

Substitution of (7.14) in (7.10) results in the following difference equation:

$$u_j^{n+1} = D\,(u_{j+1}^n + u_{j-1}^n) + (1 - 2D)\,u_j^n \tag{7.16}$$

where D is a dimesionless group, commonly denoted the diffusion number or the Fourier number in heat conduction. See section 13.1 in [2])) for a discussion of (7.16).

$$D = \frac{\Delta t}{(\Delta y)^2} = \nu \frac{\Delta \tau}{(\Delta Y)^2} \tag{7.17}$$

A scheme with Forward differences for the Time (evolutionary) variable and Central differences for the Space variable, is commonly referred to as a FTCS (Forward Time Central Space). For a FTCS-scheme we normally mean a scheme which is first order in t og and second order in y. Another common name for this scheme is the Euler-scheme.

In section 7.6 we show that for $D = 1/6$, the scheme (7.16)) is second order in t og and fourth order in y. Further, in fluid mechanics it is customary to write u_j^n rather than $u_{j,n}$, such that index for the evolutionary variable has a super index. We seek to adopt this convention in general.

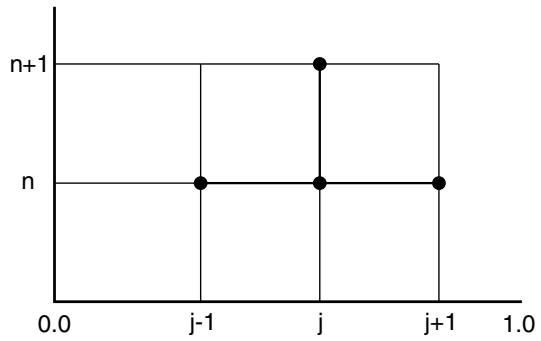

Figure 7.2: Numerical stencil of the FTCS scheme.

In Figure 7.2 we try to illustrate that the scheme is explicitt, meaning that the unknown value at time $n+1$ can be found *explicitly* from the formula without having to solve an equation system. In other words, the unknown value at time $n+1$ is *not* implicitly dependent on other values at other spatial locations at time $n+1$.

The above example is implemented in the code below. Download the code and experiment using different diffusion numbers. The FTCS-scheme is explicit and thus has a stability constraint. We will look further into stability in the next sections, but as we will see in the animations displayed below, the stability limit in this example is $D = \frac{1}{2}$.

```
# src-ch5/couette_FTCS.py;Visualization.py @ git@lrhgit/tkt4140/src/src-ch5/Visualization.py;

import matplotlib; matplotlib.use('Qt5Agg')
import matplotlib.pylab as plt
```

```
#plt.get_current_fig_manager().window.raise_()

import numpy as np
from math import exp, sin, pi


def analyticSolution(y, t, N=100):

    """ Method that calculates the analytical solution to the differential equation:
        du/dt = d^2(u)/dx^2 , u = u(y,t), 0 < y < 1
        Boundary conditions: u(0, t) = 1, u(1, t) = 0
        Initial condition: u(t, 0) = 0 t<0,  u(t, 0) = 1 t>0

        Args:
            y(np.array): radial coordinat
            t(float): time
            N(int): truncation integer. Truncate sumation after N elements


        Returns:
            w(float): velocity, us - ur
    """
    sumValue = 0
    for n in range(1,N+1):
        temp = np.exp(-t*(n*np.pi)**2)*np.sin(n*np.pi*y)/n
        sumValue += temp
    u = 1 - y - (2/pi)*sumValue
    return u

def solveNextTimestepFTCS(Uold, D, U_b=1, U_t=0):
    """ Method that solves the transient couetteflow using the FTCS-scheme..
        At time t=t0 the plate starts moving at y=0
        The method solves only for the next time-step.
        The Governing equation is:

        du/dt = d^2(u)/dx^2 , u = u(y,t), 0 < y < 1

        Boundary conditions: u(0, t) = 1, u(1, t) = 0

        Initial condition: u(t, 0) = 0 t<0,  u(t, 0) = 1 t>0

        Args:
            uold(array): solution from previous iteration
            D(float): Numerical diffusion number

        Returns:
            unew(array): solution at time t^n+1
    """
    Unew = np.zeros_like(Uold)

    Uold_plus = Uold[2:]
    Uold_minus = Uold[:-2]
    Uold_mid = Uold[1:-1]

    Unew[1:-1] = D*(Uold_plus + Uold_minus) + (1 - 2*D)*Uold_mid
    Unew[0] = U_b
    Unew[-1] = U_t

    return Unew

if __name__ == '__main__':
```

```
    import numpy as np
    from Visualization import createAnimation

    D = 0.505 # numerical diffusion number

    N = 20
    y = np.linspace(0, 1, N + 1)
    h = y[1] - y[0]
    dt = D*h**2
    T = 0.4 # simulation time
    time = np.arange(0, T + dt, dt)

    # Spatial BC
    U_bottom = 1.0 # Must be 1 for analytical solution
    U_top = 0.0 # Must be 0 for analytical solution

    # solution matrices:
    U = np.zeros((len(time), N + 1))
    U[0, 0] = U_bottom   # no slip condition at the plate boundary
    U[0,-1] = U_top
#
    Uanalytic = np.zeros((len(time), N + 1))
    Uanalytic[0, 0] = U[0,0]


    for n, t in enumerate(time[1:]):

        Uold = U[n, :]

        U[n + 1, :] = solveNextTimestepFTCS(Uold, D, U_b=U_bottom, U_t=U_top)

        Uanalytic[n + 1, :] = analyticSolution(y,t)

    U_Visualization = np.zeros((1, len(time), N + 1))
    U_Visualization[0, :, :] = U

    createAnimation(U_Visualization, Uanalytic, ["FTCS"], y, time, symmetric=False)
```

In the following two animations we show numerical solutions obtained with the explicit FTCS scheme, as well as with two implicit schemes (Crank-Nicolson and Laasonen schemes), along with the analytical solution. The first animation shows results for $D = 0.5$, for which the FTCS scheme is stable. In the second animation we use $D = 0.504$, a value for which the FTCS scheme's stability limit is exceeded. This observation is confirmed by the oscillatory character of the numerical solution delivered by this scheme. .

Movie 3: Animation of numerical results obtained using three numerical schemes as well as the analytical solution using $D = 0.5$. `mov-ch5/couette_0.5.mp4`

Movie 4: Animation of numerical results obtained using three numerical schemes as well as the analytical solution using $D = 0.504$. The FTCS displays an unstable solution. `mov-ch5/couette_0.504.mp4`

7.3 Stability: Criterion for positive coefficients. PC-criterion

Consider the following sum:

$$s = a_1\,x_1 + a_2\,x_2 + \cdots + a_k\,x_k \tag{7.18}$$

where $a_1, a_2, \ldots, a_k$ are positive coefficients. Now, by introuducing the extrema of x_i as:

$$x_{min} = \min(x_1, x_2, \ldots, x_k) \qquad \text{and} \qquad x_{max} = \max(x_1, x_2, \ldots, x_k) \tag{7.19}$$

we may deduce from (7.18):

$$x_{min} \cdot (a_1 + a_2 + \cdots + a_k) \leq s \leq x_{max} \cdot (a_1 + a_2 + \cdots + a_k) \tag{7.20}$$

Note, that the above is valid only when $a_1, a_2, \ldots, a_k$ are all positive. In the following we will consider two cases:

Case 1: $a_1 + a_2 + \cdots + a_k = 1$

In this case (7.20) simplifies to:

$$x_{min} \leq s \leq x_{max} \tag{7.21}$$

Equality in (7.21) is obtained when $x_1 = x_2 = \cdots = x_k$.

Let us now apply (7.21) on the difference equation in (7.16):

$$u_j^{n+1} = D\,(u_{j+1}^n + u_{j-1}^n) + (1 - 2\,D)\,u_j^n$$

In this case the coefficients are $a_1 = D,\ a_2 = D,\ a_3 = 1 - 2D$ such that the sum of all coefficents is: $a_1 + a_2 + a_3 = 1$.

From (7.21) we get:

$$\min(u_{j+1}^n, u_j^n, u_{j-1}^n) \leq u_j^{n+1} \leq \max(u_{j+1}^n, u_j^n, u_{j-1}^n)$$

Meaning that u_j^{n+1} is restricted by the extrema of u_j^n, i.e. all the solutions at the previous timestep, and will thus not have to ability to grow without bounds and therefore be stable.

The conditions for stability are that all the coefficients $a_1, a_2 \ldots, a_k$ are positive. As $D > 0$, this means that only $a_3 = 1 - 2D$ may become negative. The condition for a_3 to be positive becomes: $1 - 2D > 0$ which yields $D < \frac{1}{2}$. When $D = \frac{1}{2}$, the coefficient $a_3 = 0$, such that $a_1 + a_2 = 1$, which still satisfies the condition. Thus the condition for stability becomes:

$$D \leq \frac{1}{2} \tag{7.22}$$

which is commonly referred to as the Bender-Schmidt formula.

For explicit, homogenous schemes which has a constant solution $u = u_0$, the sum of the coefficients will often be one. (Substitute e.g. $u = u_0$ in (7.16)). This property is due to the form of the difference equations from the Taylor-expansions. (See (2.5) , (2.5) and (2.5) differences in (2)).

Case 2: $a_1 + a_2 + \cdots + a_k < 1$

As a remedy to that the sum of the coefficient do not sum to unity we define $b = 1 - (a_1 + a_2 + \cdots + a_k) > 0$ such that $a_1 + a_2 + \cdots + a_k + b = 1$.

We may the construct the following sum: $a = a_1x_1 + a_2x_2 + \cdots + a_k + b \cdot 0$ which satisfies the conditions for Case 1 and we get:

$$\min(0, x_1, x_2, \ldots, x_k) \leq s \leq \max(0, x_1, x_2, \ldots, x_k) \tag{7.23}$$

The only difference from (7.21) being that we have introduced 0 for the x-es, which naturally has concequences for the extremal values.

Let us look at an example:

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2} + bT, \ b = \text{konstant}, \ t < t_{maks}$$

which may be discretized with the FTCS-scheme to yield:

$$T_j^{n+1} = D\left(T_{j+1}^n + T_{j-1}^n\right) + (1 - 2D + \Delta t \cdot b)T_j^n, \qquad D = \alpha \frac{\Delta t}{(\Delta x)^2}$$

and with the following coefficients $a_1 = a_2 = D$ og $a_3 = 1 - 2D + \Delta t \cdot b$ we get:

$$a_1 + a_2 + a_3 = 1 + \Delta t \cdot b \leq 1$$

only for negative b-values.

The condition of positive coefficients then becomes: $1 - 2D + \Delta t \cdot b > 0$ which corresponds to

$$0 < D < \frac{1}{2} + \frac{\Delta t \cdot b}{2}$$

where $b < 0$.

In this situation the criterion implies that the $T - values$ from the difference equation will not increase or be unstable for a negative b. This result agrees well the physics, as a a negative b corresponds to a heat sink.

Note that (7.21) and (7.23) provide limits within which the solution is bounded, and provides a *sufficient* criteria to prevent the occurrence of unstable ocillations in the solution. This criteria may be far more restrictive that what is necessary for a stable solution. However, in many situations we may be satisfied with such a criteria. The PC-criterion is used frequently on difference equations for which a more exact analysis is difficult to pursue. Note that the PC-criterion may only be applied for explicit schemes if no extra information is provided. For parabolic equations we often have such extra information by means of max/min principles (see (7.5.3)). Further, the criterion must be modified in case of increasing amplitudes.

One would of course hope for the existence of a *necessary and sufficient* condition for numerical stability. However, for general difference equations we have no such condition, which is hardly surprising. But a method which often leads to sufficient, and in some cases necessary, conditions for stabily, is von Neumann's method. This method involves Fourier-analysis of the linearized difference equation and may be applied for both explicit and implicit numerical schemes. We will present this method in 7.4.

7.4 Stability analysis with von Neumann's method

The von Neumann analysis is commonly used to determine stability criteria as it is generally easy to apply in a straightforward manner. Unfortunately, it can only be used to find necessary and sufficient conditions for the numerical stability of linear initial value problems with constant coefficients. Practical problems may typically involve variable coefficients, nonlinearities and complicated boundary conditions. For such cases the von Neumann analysis may only be applied locally on linearized equations. In such situations the von Neumann analysis provides sufficient, but not alwasy necessary conditions for stability [5]. Further, due to the basis on Fourier analysis, the method is strictly valid only for interior points, i.e. excluding boundary conditions.

In this section we will show how von Neumann stability analysis may be applied to the parabolic PDE:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \tag{7.24}$$

To motivate the rationale for the basis of the von Neumann analysis, we will start by a revisit on the analytical solution of (7.24) by the method of separation of variables. The aim of the method is to simplify the PDE to two ODEs which has analytical solutions. With this approach we assume that the solution may constructed by means of separation of variables as $u(x,t)$, i.e. as a product of to $f(t)$ og $g(x)$, which each are only functions of time and space, repsectively:

$$u(x,t) = f(t)\, g(x) \tag{7.25}$$

We may now differentiate (7.25) with respect to time and space:

$$\frac{\partial u}{\partial t} = \frac{df(t)}{dt}\, g(x), \;\; \frac{\partial^2 u}{\partial x^2} = f(t)\, \frac{d^2 g(x)}{dx^2}\, \frac{1}{g(x)} \tag{7.26}$$

which upon substitution in (7.25) results in the following equation:

$$\frac{df(t)}{dt}\, g(x) = f(t)\, \frac{d^2 g(x)}{dx^2} \tag{7.27}$$

or more conveniently:

$$\frac{df(t)}{dt}\, \frac{1}{f(t)} = \frac{d^2 g(x)}{dx^2}\, \frac{1}{g(x)} \tag{7.28}$$

Observe that the left hand side of (7.28) is a function of t only, whereas the right hand side is a function of x only. As the both sides of (7.28) must be satisfied for arbitrary values of t and x, the only possible solution is that both sides of the equation equals a constant, which we for convenience denote $-\beta^2$, thus our original PDE in (7.24) has been transformed to two ODEs:

$$\frac{df(t)}{dt}\frac{1}{f(t)} = -\beta^2 \rightarrow \frac{df(t)}{dt} + \beta^2 f(t) = 0 \tag{7.29}$$

$$\frac{d^2g(x)}{dx^2}\frac{1}{g(x)} = -\beta^2 \rightarrow \frac{d^2g(x)}{dx^2} + \beta^2 g(x) = 0 \tag{7.30}$$

The first ODE (7.29) is of first order and has the solution (verify by susbtitution):

$$f(t) = e^{-\beta^2 t} \tag{7.31}$$

whereas the second ODE is of second order with solution (7.30):

$$g(x) = A\sin(\beta x) + B\cos(\beta x) \tag{7.32}$$

such that a particluar solution to (7.25) is found by the product of (7.31) and (7.32):

$$u(x,t) = e^{-\beta^2 t}\,[A\sin(\beta x) + B\cos(\beta x)] \tag{7.33}$$

and since (7.25) is a linear PDE, the sum or superposition of solutions like (7.33) will also represent a solution:

$$u(x,t) = \sum_{m=0}^{m=\infty} e^{-\beta_m^2 t}\,[A_m\sin(\beta_m x) + B_m\cos(\beta_m x)]$$

The coefficients A_m, B_m og β_m may be determined from the initial conditions and the boundary conditions as demonstrated in Appendix G.6 of Numeriske Beregninger.

However, for the current purpose of demonstration of von Neumann analysis, two particular solutions suffice:

$$u(x,t) = \begin{cases} e^{-\beta^2 t}\sin(\beta x) \\ e^{-\beta^2 t}\cos(\beta x) \end{cases} \tag{7.34}$$

which may be presented in a more compact for by making use of Euler's formula:

$$e^{i\,x} = \cos(x) + i\,\sin(x), \qquad i = \sqrt{-1} \tag{7.35}$$

The more compact form of (7.34) is then:

$$u(x,t) = e^{-\beta^2 t}\, e^{i\,\beta x} = e^{-\beta^2 t + i\,\beta x} \tag{7.36}$$

Note. Both the real and the imaginary part of the complex (7.36) satisfy (7.24) and is therefore included in the somewhat general solution. For particular problem (7.36) will be multiplied with complex cofficients such that the solution is real.

By adpoting the common notation: $x_j = j\,\Delta x,\ j = 0, 1, 2, \ldots$ og $t_n = n\,\Delta t,\ n = 0, 1, 2, \ldots$ for (7.36), the solution at location x_j and time t_n is:

$$u(x_j, t_n) = e^{-\beta^2 t_n}\, e^{i\,\beta x_j} = e^{-\beta^2\, n\,\Delta t}\, e^{i\beta x_j} = (e^{-\beta^2\,\Delta t})^n\, e^{i\beta x_j} \tag{7.37}$$

and the solution at location x_j at the next timestep $n+1$ is:

$$u(x_j, t_{n+1}) = e^{-\beta^2 t_{n+1}}\, e^{i\,\beta x_j} = e^{-\beta^2\,(n+1)\,\Delta t}\, e^{i\beta x_j} = (e^{-\beta^2\,\Delta t})^{n+1}\, e^{i\beta x_j} \tag{7.38}$$

If we divide (7.37) with (7.38), the spatial dependency vanishes and a we get a spatially independent expression for how the solution is amplified (or damped):

$$G_a = \frac{u(x_j, t_{n+1})}{u(x_j, t_n)} = e^{-\beta^2 \Delta t} \tag{7.39}$$

For this reason, G_a is commonly denoted the *analytical* amplificaiton factor. (See section 1.6.1 i Numeriske Beregninger). In our particular case we find that $G_a < 1$. Having introduced the the amplification factor G_a we may rewrite (7.37) as:

$$u(x_j, t_n) = (G_a)^n\, e^{i\,\beta x_j} \equiv G_a^n\, e^{i\,\beta x_j} \tag{7.40}$$

For the current problem $G_a < 1$, and consequently $G_a^n \to 0$ for $n \to \infty$.

As shown previously in (7.16), the following difference equation for the solution of (7.24) may be obtained:

$$u_j^{n+1} = D\,(u_{j+1}^n + u_{j-1}^n) + (1 - 2D)u_j^n, \qquad D = \frac{\Delta t}{(\Delta x)^2} \tag{7.41}$$

Clearly, the solution of the difference equation (7.41) will deviate from the analytical solution to the differential equation for finite values of Δx and Δt. The deviation/error will increase for increasing values of Δx and Δt and we have seen that when $D > \frac{1}{2}$, the difference equation becomes unstable, with constantly increasing amplitudes and alternating signs. As (7.40) is a solution of the differential equuation, we introuce a simpliar expression for the difference equation (7.41):

$$u_j^n \to E_j^n = G^n\, e^{i\,\beta x_j} \tag{7.42}$$

where we have introduced the *numerical ampflicication factor* G:

$$G = \frac{E_j^{n+1}}{E_j^n} \tag{7.43}$$

which may be complex and is a function og Δt and β. From (7.43) we may relate the error E_j^n at the n-th timestep with the initial error E_j^0:

$$E_j^n = G^n \, E_j^0, \;\; E_j^0 = e^{i \, \beta \, x_j} \tag{7.44}$$

Given that G_a is derived from the analytical solution of the differetial equation, and that G is derived from the difference equation approximating the same differential equation, we can not expect perfect agreement between the two factors. We will discuss this further in 7.4.1 and illustrated in Figure 7.3.

From (7.44) we find a that if E_j^n is to be limited and not grow exponentially the following condition must be satisfied, which is denoted:

The strict von Neumann condition.

$$|G| \leq 1 \tag{7.45}$$

The condition (7.45) is denoted *strict* as no increase in amplitude is allowed for. This condition will be relaxed in section (7.5.4).

Even though we have demostrated the von Neumann method for a relatively simple diffusion equation, the method is applicable for more generic equations. The method is simple:

The strict von Neumann method.

- Substitute (7.42) in the difference equation in question.
- Test if the condition (7.45) is satisfied.

Some properties of the condition:

1. The linear difference equation must have constant coefficients. In case of variable coefficients, the condition may be applied on

linearized difference equations with *frozen* coefficients locally. Based on experience this results in a necessary condition for stability.

1. The criterion do not consider the effect of boundary conditions as it is based on periodic initital data. If the impact of boundary

conditions is to be investigated on may use matrix methods for the coefficient matrix of the difference scheme [3].

7.4.1 Example: Practical usage of the von Neumann condition

In this example we will demonstrate how the von Neumann method may be used on the simple scheme (7.16) on the following form:

$$E_j^{n+1} = D\,(E_{j+1}^n + E_{j-1}^n) + (1-2D)\,E_j^n \tag{7.46}$$

We have previously (see (7.42)) established how the error E_j^n is related with the numerical amplification factor: $E_j^n = G^n\, e^{i\,\beta\, y_j}$ which upon substitution in (7.46) yields:

$$G^{n+1} e^{i\,\beta y_j} = D\,(G^n\, e^{i\,\beta y_{j+1}} + G^n\, e^{i\,\beta y_{j-1}}) + (1-2D) G^n e^{i\,\beta y_j}$$

Note that G^n means G in the n-th power, whereas E_j^n denotes the error at timelevel n and the spatial location y_j.

The expression in (7.4.1) is a nonlinear, $n+1$-order polynomial in G which may be simplified by division of $G^n\, e^{i\,\beta y_j}$:

$$G = D\,(e^{i\,\beta h} + e^{-i\,\beta h}) + (1-2D) = D\,(e^{i\delta} + e^{-i\delta}) + (1-2D) \tag{7.47}$$

where we introduce δ as:

$$\delta = \beta h \tag{7.48}$$

By using terminology for periodical functions, β may be thought of as a wavenumber (angular frequency) and δ as a phase angle. (See appendix A.3 in Numeriske Beregninger)

For further simplification we introduce some standard trigonometric formulas:

$$\begin{aligned} 2\cos(x) &= e^{ix} + e^{-ix} \\ i\,2\sin(x) &= e^{ix} - e^{-ix} \\ \cos(x) &= 1 - 2\sin^2(\tfrac{x}{2}) \end{aligned} \tag{7.49}$$

By substitution (7.49) in (7.47) we get the simpler expression:

$$G = 1 - 2D\,(1-\cos(\delta)) = 1 - 4D\,\sin^2\left(\frac{\delta}{2}\right) \tag{7.50}$$

As G is real, the condition $|G| \leq 1$ has the following mathematical interpretation:

$$-1 \leq G \leq 1 \qquad \text{or} \qquad -1 \leq 1 - 4D\sin^2\left(\frac{\delta}{2}\right) \leq 1 \tag{7.51}$$

The right hand side of (7.4.1) is always true as $D \geq 0$. For the left hand side of (7.4.1) we have

$$D \leq \frac{1}{2\sin^2(\frac{\delta}{2})}$$

which is true for all δ ($-\pi \leq \delta \leq \pi$) when $D \leq \frac{1}{2}$. A von Neumann condition for stability of (7.16) may then be presented as:

$$0 < D < \frac{1}{2} \tag{7.52}$$

where $D = \frac{\Delta t}{(\Delta y)^2}$ from (7.17).

As (7.16) is a two-level scheme with constant coefficients, the condition in (7.52) is both sufficient and necessary for numerical stability. This condition agrees well in the previous condition in (7.21), which is sufficient only.

The stability condition impose a severe limitation on the size of the timestep Δt, which of course influence the CPU-time.

A rough estimate of the CPU-time for the FTCS-schemes with constant D-value yields:

$$\frac{T_2}{T_1} \approx \left(\frac{h_1}{h_2}\right)^3$$

where T_1 and T_2 are the CPU-times for $\Delta y = h_1$ and $\Delta y = h_2$, respectively. For example for a reduction in spatial resolution from $h_1 = 0.1$ by a factor 10 to $h_2 = 0.01$ we get an increase in CPU-time by a factor $\frac{T_2}{T_1} = 1000$.

Differentiation to find stability conditions. An alternative method to find extremal values (i.e. max/min) for G as a function of δ, is to compute $\frac{dG}{d\delta}$ and then set it to $\frac{dG}{d\delta} = 0$. A stability condition may then be found from the criterion $|G| < 1$. For the current example we have from (7.50):

$$G = 1 - 2D\,(1 - cos(\delta)) \qquad \Rightarrow \qquad \frac{dG}{d\delta} = -2D\,\sin(\delta)$$

which has extremal values for $\delta = 0$,
$\delta = \pm\pi$, $\delta = 0$ gir $G = 1$, whereas $\delta = \pm\pi$ yields the condition $G = 1 - 4D$. Finally, the condition $|G| \leq 1$ yields:

$$-1 \leq 1 - 4D \leq 1$$

The right hand side of the inequality will always be satisfied for positive D, whereas the left hand side yields $D \leq \frac{1}{2}$ as before.

In many situations we will find that $\delta = \pm\pi$ are critical values, and it may therefore be wise to assess these values, but remember it might not be sufficent in order to prove stability. On the other hand these values might sufficient to prove instabilities for those values, as the condition $|G| \leq 1$ must be satisfied for all δ-values in the range $[-\pi,\ \pi]$.

Comparison of the amplification factors. In this section we will compare the numerical amplification factor G in (7.50) with the analytical amplification factor G_a in (7.39)

By making use of (7.41) and (7.48) we may present the analytical amplification factor (7.39) as:

$$G_a = \exp(-\delta^2 D) \tag{7.53}$$

And from (7.50) we have an expression for the numerical amplification factor:

$$G = 1 - 2D\,(1 - \cos(\delta)) = 1 - 4D\,\sin^2\left(\frac{\delta}{2}\right) \tag{7.54}$$

In figure 7.3 we plot G and G_a as functions of $\delta \in [0,\ \pi]$ for selected values of D. For small values of δ we observe small differences between G and G_a, with slightly larger values of G than for G_a, with progressively increasing differences as a function of δ.

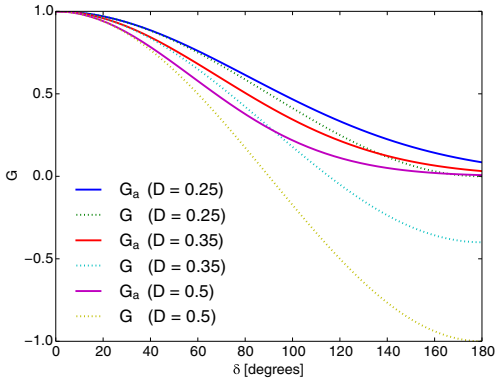

Figure 7.3: The amplification factors G (dotted) and G_a (solid) as a function of δ for specific values of D.

Further, we observe large errors for $\delta \in [90°,\ 180°]$ and that the amplification factor G even has the wrong sign, which will lead to unpysical oscillations in the numerical solution.

The reason why the solution may still be usable is due to that the analytical solution has an amplitude G_a which diminishes strongly with increasing frequency (see the analytical solution in (7.13)). Such a smoothing effect is typical for parbolic PDEs. Yet the effect is noticable as we in the current example have a discontinuity at $y = 0$, such that the solution contains many high frequent components.

Errors in the amplitude is commonly quantified with $\varepsilon_D = \left|\frac{G}{G_a}\right|$ and denoted diffusion error or dissipation error. No diffusion error corresponds to $\varepsilon_D = 1$. The term *diffusive scheme* will normally refer to a numerical scheme with decreasing

amplitude with increasing t. For our FTCS-scheme applied for the diffusion equation we have:

$$\varepsilon_D = \left|1 - 4D\sin^2(\delta/2)\right| \exp(\delta^2 D) \tag{7.55}$$

The expression in (7.55) may be simplified by a Taylor-expansion:

$$\varepsilon_D = 1 - D^2\delta^4/2 + D\delta^4/12 + O(\delta^6)$$

which confirms that the dissipation error is smal for low frequencies when $D \leq 1/2$.

7.5 Some numerical schemes for parabolic equations

7.5.1 Richardson scheme (1910)

The FTCS scheme is 1st order accurate in time and 2nd order accurate in space. We would like to have a scheme that is also 2nd order accurate in time. This can be achieved by using central finite differences for $\frac{\partial u}{\partial t}$:

$$\left.\frac{\partial u}{\partial t}\right|_j^n \approx \frac{u_j^{n+1} - u_j^{n-1}}{2\Delta t}$$

which results in:

$$u_j^{n+1} = u_j^{n-1} + 2D\left(u_{j-1}^n - 2u_j^n + u_{j+1}^n\right) \tag{7.56}$$

This is an explicit, 3-level scheme called *Richardson scheme*; see Figure 7.4.

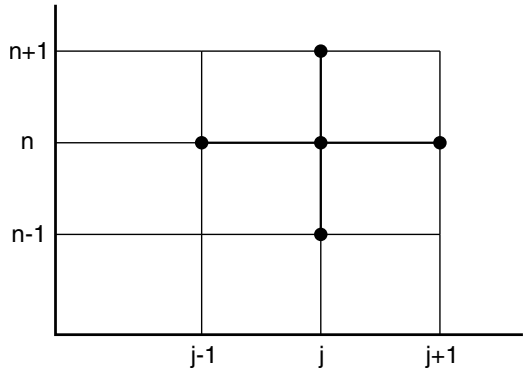

Figure 7.4: 3-level stencil of the Richardson scheme (7.56).

Stability analysis

Let us first try with the criterion of positive coefficients introduced in (7.21). This condition can not be fulfilled for the coefficient in front of u_j^n for $D > 0$. Next, we try to gain insight on the stability of the scheme by using von Neumann's method. (7.42) inserted in (7.56) gives:

$$G^{n+1}e^{i\cdot\beta y_j} = G^{n-1}e^{i\cdot\beta y_j} + 2D[G^n e^{i\cdot\beta y_{j-1}} - 2G^n e^{i\cdot\beta y_j} + G^n e^{i\cdot\beta y_{j+1}}]$$

Dividing with $G^{n-1}e^{i\beta y_j}$ where $y_j = j \cdot h$:

$$\begin{aligned} G^2 &= 1 + 2DG \cdot \left(e^{-i\delta} + e^{i\delta} - 2\right) = 1 + 4DG \cdot (\cos(\delta) - 1) \\ &= 1 - 8GD \cdot \sin^2\left(\tfrac{\delta}{2}\right) \end{aligned}$$

where we have used (7.48) and (7.49). We have obtained a 2nd order equation because we are dealing with a 3-level scheme:

$$\begin{aligned} &G^2 + 2bG - 1 = 0 \text{ med løsning} \\ &G_{1,2} = -b \pm \sqrt{b^2+1},\ b = 4D\sin^2\left(\tfrac{\delta}{2}\right) \geq 0 \end{aligned}$$

$|G| = 1$ for $b = 0$. For all other values of b we have that $|G_2| > 1$. The scheme is therefore unstable for all actual values of D. Such schemes are said to be *unconditionally unstable*. This example also shows that stability and accuracy can be rather independent concepts depending on how they are defined.

Use of derivation

We can also use derivation here:

$$\begin{aligned} &G^2 = 1 + 4DG \cdot (\cos(\delta) - 1), \\ &2G\tfrac{dG}{d\delta} = 4D\left[(\cos(\delta) - 1)\tfrac{dG}{d\delta} - G\sin(\delta)\right], \end{aligned}$$

that with $\frac{dG}{d\delta} = 0$ gives max-min for $\delta = 0$, $\delta = \pm\pi$, as for the FTCS scheme. For $\delta = 0$ we have $G_{1,2} = \pm 1$, while $\delta = \pm\pi$, resulting in

$$G_{1,2} = -4D \pm \sqrt{1 + (4D)^2}$$

with instability for $|G_2| > 1$ as before.

7.5.2 Dufort-Frankel scheme (1953)

Richardson scheme in (7.56) can be made stable by the following modification:

$$u_j^n = \frac{1}{2}(u_j^{n+1} + u_j^{n-1}) \tag{7.57}$$

which inserted in (7.56) gives:

$$u_j^{n+1} = \frac{1}{1+2D}\left[(1-2D)u_j^{n-1} + 2D(u_{j+1}^n + u_{j-1}^n)\right] \tag{7.58}$$

This is an explicit 3-level scheme called DuFort-Frankel scheme, see Figure 7.5.

3-level schemes where u_j^n is missing are called *Leap-frog*-type schemes for obvious reasons. The sufficient criterion of positive coefficients in (7.21) requires

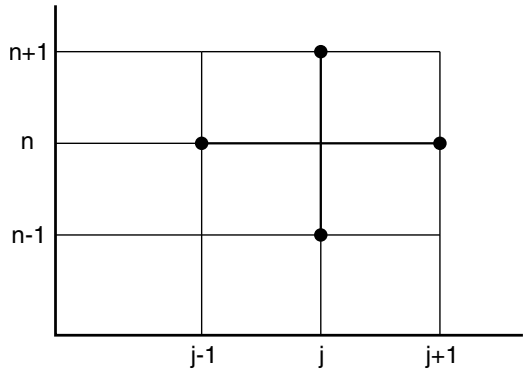

Figure 7.5: 3-level stencil of the DuFort-Frankel scheme (7.58).

that $D \leq \frac{1}{2}$ for a stable scheme. The stability analysis here is slightly more complicated as with previous cases because we have to discuss what happens when G takes complex values.

Inserting (7.42) in (7.58) and division by $G^{n-1}e^{e^{i\delta j}}$ yields:

$$G^2 = \frac{1}{1+2r}\left[(1-2r) + 2r \cdot G \cdot (e^{i\delta} + e^{-i\delta})\right] = \frac{1}{1+2r}\left[(1-2r) + 4r \cdot G \cdot \cos(\delta)\right]$$

which gives the following 2nd order equation:

$$(1+2D) \cdot G^2 - 4D \cdot G\cos(\delta) - (1-2D) = 0\,,$$

with roots:

$$\begin{aligned} G_{1,2} &= \frac{4D\cos(\delta) \pm \sqrt{(4D\cos(\delta))^2 + 4(1+2D)\cdot(1-2D)}}{2(1+2D)} \\ &= \frac{2D\cos(\delta) \pm \sqrt{1 - D^2\sin^2(\delta)}}{1+2D}. \end{aligned}$$

For stability, both roots must meet the condition $|G| \leq 1$. In general, we must distinguish between real and complex roots to take care of the case for which $G \leq 0$ when G is real.

1. Real roots: $1 - 4D^2\sin(\delta) \geq 0$
   $$|G_{1,2}| \leq \frac{2D\cdot|\cos(\delta)| + \sqrt{1-4D^2\sin^2(\delta)}}{1+2r} \leq \frac{1+2D}{1+2D} \leq 1$$

2. Complex roots: $1-4D^2\sin^2(\delta) < 0 \rightarrow \sqrt{1-4D^2\sin^2(\delta)} = i\cdot\sqrt{4D^2\sin^2(\delta)-1}$
   $$|G_{1,2}|^2 = \left|\frac{\left(2D\cos(\delta)\right)^2 + 4D^2\sin^2(\delta) - 1}{(1+2D)^2}\right| = \left|\frac{4D^2-1}{4D^2+4D+1}\right| = \left|\frac{2D-1}{2D+1}\right| < 1$$

Analysis shows that (7.58) is actually *unconditionally stable*. The DuFort-Frankel scheme is the only simple know explicit scheme with 2nd order accuracy in space and time that has this property. Therefore it has been in part used to solve the Navier-Stokes equations. In Section (7.6) we shall see that there is however a severe problem with this scheme. For the first time level one lacks the lowest time level required by the scheme. One option is to evolve the solution for the first time step using the FTCS scheme.

7.5.3 Crank-Nicolson scheme. θ-scheme

One of the bad characteristics of the DuFort-Frankel scheme is that one needs a special procedure at the starting time, since the scheme is a 3-level scheme. Therefore, we try now to find a second order approximation for $\frac{\partial u}{\partial t}$ where only two time levels are required..

Using central differences centered at the half time interval we obtain:

$$\left.\frac{\partial u}{\partial t}\right|_j^{n+\frac{1}{2}} = \frac{u_j^{n+1} - u_j^n}{\Delta t} + O(\Delta t)^2 \tag{7.59}$$

The problem now is to approximate $\left.\frac{\partial u}{\partial t}\right|_j^{n+\frac{1}{2}}$ without knowing without including any term evaluate at $n+\frac{1}{2}$ explicitly in the scheme. This was achieved by the Crank-Nicolson approximation (1947):

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_j^{n+\frac{1}{2}} = \frac{1}{2}\left[\frac{u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}}{(\Delta x)^2} + \frac{u_{j+1}^{n} - 2u_j^{n} + u_{j-1}^{n}}{(\Delta x)^2}\right] + O(\Delta x)^2 \tag{7.60}$$

The difference equation is now:

$$u_{j-1}^{n+1} - 2(1 + \frac{1}{D})u_j^{n+1} + u_{j+1}^{n+1} = -u_{j-1}^n + 2(1 - \frac{1}{D})u_j^n - u_{j+1}^n \tag{7.61}$$

with $D = \dfrac{\Delta t}{(\delta x)^2}$.

The stencil of the resulting numerical scheme is shown in Figure 7.6.

From (7.61) and Figure 7.6 we see that the scheme is implicit. This in turn implies that the algebraic equations resulting from the discretization of the original differential problem are coupled and, differently from an explicit scheme, we have to solve them as a system. In this case the system is tridiagonal, so we can use the Thomas' algorithm, for which the code 4.3 was introduce in Section 4.3.

Before proceeding with a stability analysis for (7.61), we will re-write it a more general form, by introducing the parameter θ:

$$u_j^{n+1} = u_j^n + D\left[\theta(u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}) + (1-\theta)(u_{j+1}^n - 2u_j^n + u_{j-1}^n)\right] \tag{7.62}$$

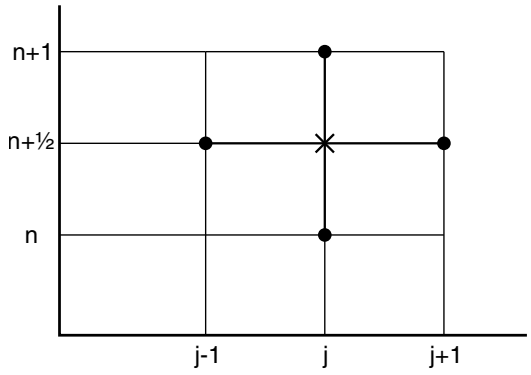

Figure 7.6: 2-level stencil of the Crank-Nicolson scheme. Note that the scheme does not require any evaluation of the solution at time level $n + \frac{1}{2}$.

with $0 \leq \theta \leq 1$. This scheme is a generalization of Crank-Nicolson original scheme and is called the θ-scheme.

For $\theta = 0$ one recovers the explicit FTCS-scheme. On the other hand, for$\theta = \frac{1}{2}$ one has the original Crank-Nicolson scheme. Finally, ror $\theta = 1$ one gets an implicit scheme some times called the Laasonen-scheme (1949). In fluid mechanics this last scheme is often called BTCS-scheme. (Backward in Time and Central in Space).

Inserting (7.42) in (7.62) and dividing by $G^n \cdot e^{i\cdot\beta x_j}$ gives:

$$\begin{aligned} G &= 1 + D[G\theta(e^{i\delta} + e^{-i\cdot\delta} - 2) + (1-\theta)(e^{i\delta} + e^{-i\delta} - 2)] \\ &= 1 + D(e^{i\delta} + e^{-i\delta} - 2) \cdot (G\theta + 1 - \theta) \end{aligned}$$

with $\delta = \beta \cdot h$. Moreover, using (7.49) results in:

$$G = \frac{1 - 4D(1-\theta)\sin^2(\frac{\delta}{2})}{1 + 4D\theta\sin^2(\frac{\delta}{2})} \tag{7.63}$$

The stability condition is $|G| \leq 1$ or, since G is real: $-1 \leq G \leq 1$. Since we have that $0 \leq \theta \leq 1$, the condition of the right side is satisfied for $D \geq 0$. For the left side of the inequality we have that:

$$2D\sin^2\left(\frac{\delta}{2}\right)(1 - 2\theta) \leq 1 \tag{7.64}$$

or

$$D(1-2\theta) \leq \frac{1}{2} \text{ da } \sin^2\left(\frac{\delta}{2}\right) \leq 1 \tag{7.65}$$

For $\frac{1}{2} \leq \theta \leq 1$, stability is given for all $D \geq 0$, ie.: the scheme is unconditionally stable.

For $0 \leq \theta \leq \frac{1}{2}$, the stability condition requires that:

$$D(1-2\theta) \leq \frac{1}{2},\ 0 \leq \theta \leq 1 \tag{7.66}$$

Use of derivation
Now we write:

$$G = 1 + D(e^{i\delta} + e^{-i\delta} - 2) \cdot (G\theta + 1 - \theta) = 1 + 2D\big(\cos(\delta) - 1\big) \cdot (G\theta + 1 - \theta)$$
$$\frac{dG}{d\delta} = 2D\left[(G\theta + 1 - \theta)\frac{d}{d\delta}\big(\cos(\delta) - 1\big) + \big(\cos(\delta) - 1\big)\theta \cdot \frac{dG}{d\delta}\right]$$

With $\frac{dG}{d\delta} = 0$ we get the max-min for $\delta = 0,\ \delta = \pm\pi$ ($\delta = 0$ gives $G = 1$ as expected) $\delta = \pm\delta$ gives $G = \frac{1 - 4D(1-\theta)}{1 + 4D\theta}$ which is identical to (7.63) for $\frac{\delta}{2} = 90°$. The condition $|G| \leq 1$ becomes $-1 \leq \frac{1 - 4D(1-\theta)}{1 + 4D\theta} \leq 1$ with the same results as before.

Accuracy
Let us now look at the accuracy of the scheme presented in (7.62).
To do so, we write a simple Maple-program:

```
> eq1:= u(x+h,t+k) - 2*u(x,t+k) + u(x-h,t+k):
> eq2:= u(x+h,t) - 2*u(x,t) + u(x-h,t):
> eq:=  (u(x,t+k) - u(x,t))/k -(theta*eq1 + (1-theta)*eq2)/h^2:
> Tnj:= mtaylor(eq,[h,k]):
> Tnj:= simplify(Tnj):
> Tnj:= convert(Tnj,diff);
```

Note that we have used $h = \Delta x$ and $k = \Delta t$. Tnj is the truncation error, which is discussed in detail in Section 7.6. Suppose now that $u(x,t)$ is the analytical solution for the partial differential equation $\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$ so that we can say that $\frac{\partial}{\partial t}() = \frac{\partial^2}{\partial x^2}$, etc. If we use these relationships in the Maple-program for Tnj, we will obtain the following result:

$$T_j^n = \left[\left(\frac{1}{2} - \theta\right) \cdot \Delta t - \frac{1}{12}(\Delta x)^2\right] \cdot \frac{\partial^4 u}{\partial x^4} + \frac{1}{6}(\Delta t)^2(1 - 3\theta) \cdot \frac{\partial^6 u}{\partial x^6} + \ldots \tag{7.67}$$

We see that the truncation error is $T_j^n = O(\Delta t) + O(\Delta x)^2$ for $\theta = 0$ and 1, i.e. for Euler-scheme and Laasonen-scheme, while it becomes $T_j^n = O(\Delta t)^2 + O(\Delta x)^2$ for $\theta = \frac{1}{2}$, which corresponds to the Crank-Nicolson scheme.
If we add the following line

```
Tnj:= simplify(subs(theta =(1-h^2/(6*k))/2,Tnj));
```

to the Maple-program, we find that $T_j^n = O(\Delta t)^2 + O(\Delta x)^4$ if we choose $D = \dfrac{1}{6(1-2\theta)}$ when $D \leq \dfrac{1}{2(1-2\theta)}$.

More about stability

We have found that the Crank-Nicolson scheme is unconditionally stable for $\frac{1}{2} \leq \theta \leq 1$. In practice, the scheme can cause oscillations around discontinuities for $\theta = \frac{1}{2}$. Values of θ above 0.5 will dampen such oscillations with strongest attenuation for $\theta = 1$. Let's now look at an example where we use the heat equation in dimensional form.

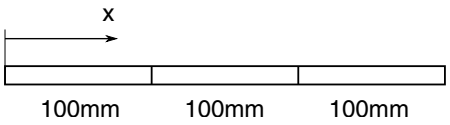

Figure 7.7: Thin aluminum rod for one-dimensional heat equation example.

Figure 7.7 shows a thin aluminum rod of length equal to 300 mm.
The one-dimensional heat equation is given by:

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2},\ T = T(x,t)$$

Boundary conditions are:

$$T(0,t) = T(300,t) = 20^\circ C,$$

while initial conditions are:

$$T(x,0) = 270^\circ C \text{ for } x \in (100,200)$$
$$T(x,0) = 20^\circ C \text{ for } x \in (0,100) \text{ and } x \in (200,300)$$

Thermal diffusivity is:

$$\alpha = 100 mm^2/s$$

The numerical Fourier number is:

$$D = \alpha \frac{\Delta t}{(\Delta x)^2}$$

Furthermore, we set $\Delta t = 0.25$ s so that $\Delta x = 5/\sqrt{D}$. By selecting $D = 4$, we get that $\Delta x = 2.5$ mm. Figures 7.8 and 7.9 show computational results with $\theta = \frac{1}{2}$ and with $\theta = 1$; the Crank-Nicolson and Laasonen schemes, respectively.

We see that the solution obtained using the C-N-scheme contains strong oscillations at discontinuities present in the initial conditions at $x = 100$ of $x = 200$. These oscillation will be dampen by the numerical scheme, in this

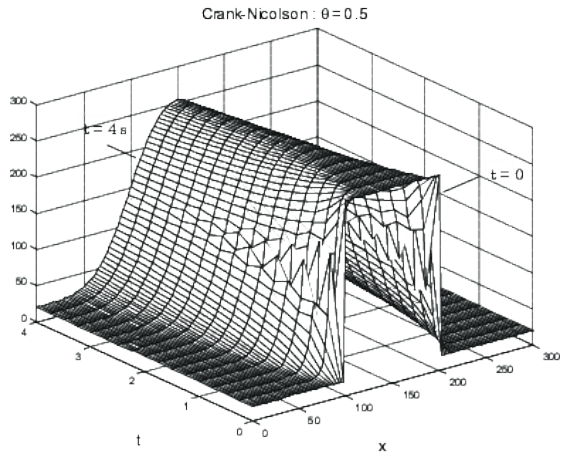

Figure 7.8: Solution for the one-dimensional heat equation problem using Crank-Nicolson scheme.

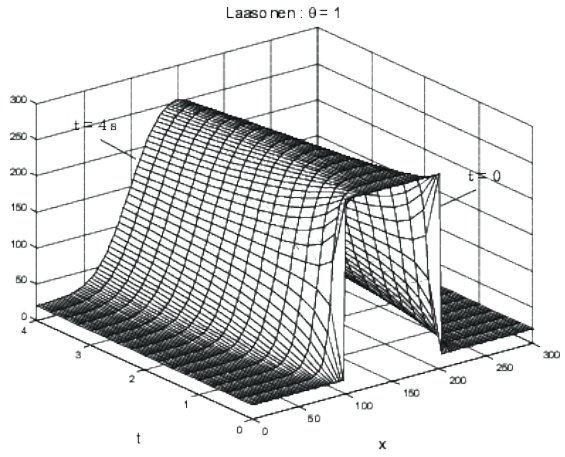

Figure 7.9: Solution for the one-dimensional heat equation problem using Laasonen scheme.

case they tend to disappear for $t = 4$ s. The solution delivered by the Laasonen-scheme presents no oscillations, even around $t = 0$. To explain the behavior of these solutions we have to go back to equation (7.63):

$$G = \frac{1 - 4D(1-\theta)\sin^2(\frac{\delta}{2})}{1 + 4D\sin^2(\frac{\delta}{2})}$$

For $\theta = \frac{1}{2}$ we get:

$$G = \frac{1 - 2D\sin^2(\frac{\delta}{2})}{1 + 2D\sin^2(\frac{\delta}{2})}$$

For δ near π, G will have a value close to -1 for large values of D. This can be clearly seen if one sets $\delta = \pi = \beta \cdot h$, which results in $G = \dfrac{1-2D}{1+2D}$.

From (7.44) we have:

$$E_j^n = G^n \cdot E_j^0,\ E_j^0 = e^{i\cdot\beta x_j}$$

Therefore, we see that we will get oscillations for δ close to π. These high wave numbers will be dampened as the simulation proceeds because the initial condition discontinuities are smoothed by dissipation. If we set $D = 4$ as used in this example, after 16 time increments with $\Delta t = 0.25$s we get that $G^{16} = \left(-\dfrac{7}{9}\right)^{16} \approx 0.018$, whereas with $D = 1$ we would obtain $G^{16} = 2.3 \cdot 10^{-8}$. This means that high wave-numbers oscillations dampening slows down for large values of D. This is very similar to what happens in the case of the Gibbs-phenomenon for Fourier series of discontinuous functions (See Appendix A.3, in the Numeriske Beregninger).

On the other hand, for the Laasonen-scheme, i.e. the θ-scheme, we have that:

$$G = \frac{1}{1 + 4D\sin^2\left(\dfrac{\delta}{2}\right)}$$

We do not get oscillations for any wave number.

If we use terms from the stability analysis of ordinary differential equations we can say that the θ-scheme is absolute stable for (A-stable) for $\theta = \frac{1}{2}$ and L-stable for $\theta = 1$. (See Section 1.6.1 in the Numeriske Beregninger)

If we use the θ-scheme for the heat equation in a problem setting as the one described above with prescribed temperature at both ends (Dirichlet boundary conditions), we know that the maximum temperature at any given time must be between the largest value of the initial condition and the smallest given on the edge of the domain, or $T_{min} \leq T_j^n \leq T_{max}$. For the example presented above one has $T_{min} = 20°C$ and $T_{max} = 270°C$.

Writing (7.62) for u_j^{n+1} yields:

$$u_j^{n+1} = \frac{1}{1 + 2\theta D}\left[\theta D(u_{j-1}^{n+1} + u_{j+1}^{n+1}) + (1-\theta)D(u_{j-1}^n + u_{j+1}^n) + \big(1 - (1-\theta)2D\big)\right] \tag{7.68}$$

By summing the coefficients on the right-hand side we find that the sum is equal to 1. Next, we check for which conditions the coefficients are positive, which is equivalent to require that $0 < \theta < 1$ and $1 - (1-\theta)\cdot 2D > 0$. The last inequality is met for $D \cdot (1-\theta) < \frac{1}{2}$. By setting $\theta = 0$ and $\theta = 1$, we find that

the sum is equal to 1 for these values, so that we get the following condition for fulfilling the PC-criterion:

$$D \cdot (1 - \theta) \leq \frac{1}{2} \tag{7.69}$$

By the von Neumann-analysis we found that the following relation should hold (see (7.66)):

$$D \cdot (1 - 2\theta) \leq \frac{1}{2} \tag{7.70}$$

We see that (7.69) is significantly stricter than (7.70). While the θ-scheme is unconditionally stable for $\theta = \frac{1}{2}$ according to the von Neumann stability analysis, we must have $D \leq 1$ according to the PC-criterion. The PC-criterion provides a sufficient condition which ensures that the physical max-min criterion is also fulfilled by the solution delivered by the numerical scheme. The example shown above confirms the prediction provided by the PC-criterion. Now, is there a criterion that is both necessary and sufficient for this simple model equation? Kraaijevanger found the following necessary and sufficient criterion in 1992:

$$D \cdot (1 - \theta) \leq \frac{2 - \theta}{4(1 - \theta)} \tag{7.71}$$

We see that for $\theta = \frac{1}{2}$ this criterion implies the condition $D \leq \frac{3}{2}$. For $\theta = \frac{3}{4}$ it gives (7.71) $D \leq 5$, while the PC-criterion yields $D \leq 2$.

The purpose of allowing for large D values is related to the fact that one might be interested in simulations where a large time step is set to let reach a steady state. But on must not forget about accuracy. Moreover, we should remember that this is a simple one-dimensional partial differential equation with constant coefficients, and as such it can be solver very quickly with a modern PC with for any reasonable Δt and Δx, as long as we stay within the stability area. However, three-dimensional non-linear problems will in general result in very computational demanding simulations.

7.5.4 Generalized von Neumann stability analysis

We have referred to the stability condition $|G| \leq 1$ as the von Neumann's strong stability condition. The reason for this designation is that if $|G| \leq 1$ is fulfilled, then the amplitude of the solution can not increase. However, there are many physical problems for which amplitude grows in time (in a bounded manner). A simple example is that of heat conduction with a source such as:

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2} + bT, \; b = const., \; t < t_{max} \tag{7.72}$$

$b < 0$ represents a heat sink, while $b > 0$ is said to be a heat source. In the first case we can apply the von Neumann strong criterion, whereas in the later case it is necessary to allow for $|G| > 1$.

A particular solution of (7.72) is given by:

$$T(x,t) = e^{bt} \cdot e^{-\alpha\beta^2 \cdot t} \cos(\beta x) = e^{(b-\alpha\beta^2)\cdot t} \cos(\beta x) \tag{7.73}$$

Now, let us use (7.73) to determine an analytical growth factor, see (7.39):

$$G_a = \frac{T(x_j,\ t_{n+1})}{T(x_j,\ t_n)} = \exp\left[(b-\alpha\beta^2)\cdot t_{n+1} - (b-\alpha\beta^2)\cdot t_n\right] = e^{b\Delta t} \cdot e^{-\alpha\beta^2\Delta t} \tag{7.74}$$

We see that the term $e^{b\cdot\Delta t}$ causes increases of amplitude for positive b.
A series expansion for small Δt:

$$e^{b\Delta t} = 1 + b\cdot\Delta t + \frac{b^2}{2}(\Delta t)^2 + \ldots \tag{7.75}$$

If we use the FTCS-scheme we obtain:

$$T_j^{n+1} = D(T_{j+1}^n + T_{j-1}^n) + (1-2D)T_j^n + \Delta t b T_j^n \tag{7.76}$$

with

$$D = \alpha\frac{\Delta t}{(\Delta x)^2} \tag{7.77}$$

If we use the PC-criterion we find that the sum of coefficients is equal to $1 + b\cdot\Delta t$, which implies that this criterion can only be used for $b < 0$.
On the other hand, using the von Neumann's method yields (7.76):

$$G = 1 - 4D\sin^2\left(\frac{\delta}{2}\right) + \Delta t\cdot b \tag{7.78}$$

By setting $D = \frac{1}{2}$, we have:

$$|G| \le \left|1 - 2\sin^2\left(\frac{\delta}{2}\right)\right| + |\Delta t\cdot b| \le 1 + \Delta t\cdot b,\ b > 0$$

If we compare this expression with (7.75), we see that the agreement between the analytical and the numerical growth factor for this case.
We can now introduce the generalized von Neumann's stability condition:

$$|G| \le 1 + K\cdot\Delta t \tag{7.79}$$

with K being a positive constant.

This means that we allow the amplitude to grow exponentially for $t < t_{max}$. In this case we can use the strong stability condition if we reason this way: since the source term in (7.72) doe not contain any derivative term, we can ignore this term in the stability analysis. We have the same kind of problem in Section 1.6 in the Numeriske Beregninger, where stiff ordinary differential equations are discussed (See for example equation (1.6.8) in Section 1.6.1 of the Numeriske Beregninger).

For a growing amplitude , we must decrease the step lenth of the independent variable if we are to achieve a prescribed accuracy. For a decreasing amplitude, however, we must keep it under a maximum step lenth to get a stable calculation.

Let us look at another example using the FTCS-scheme.

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} + a_0 \frac{\partial u}{\partial x},\ \alpha > 0 \tag{7.80}$$

This equation is called the advection-diffusion equation and is a parabolic equation according to the classification scheme discussed in Section 4.3 of the Numeriske Beregninger.

Using the FTCS scheme for (7.80) with central differences for $\frac{\partial u}{\partial x}$:

$$u_j^{n+1} = u_j^n + D \cdot (u_{j+1}^n - 2u_j^n + u_{j-1}^n) + a_0 \frac{\Delta t}{2\Delta x}(u_{j+1}^n - u_{j-1}^n),\ D = \alpha \frac{\Delta t}{(\Delta x)^2}$$

Using von Neumann's method we get that:

$$G = 1 - 4D \sin^2\left(\frac{\delta}{2}\right) + i \cdot a_0 \frac{\Delta t}{\Delta x} \sin(\delta)$$

which further provides:

$$|G|^2 = \left(1 - 4D \sin^2\left(\frac{\delta}{2}\right)\right)^2 + \left(a_0 \frac{\Delta t}{\Delta x} \sin(\delta)\right)^2 = \left(1 - 4D \sin^2\left(\frac{\delta}{2}\right)\right)^2 + \frac{a_0^2 \cdot D}{\alpha} \cdot \Delta t \sin^2(\delta)$$

Setting now $D = \frac{1}{2}$:

$$|G| = \sqrt{\left(1 - 2\sin^2\left(\frac{\delta}{2}\right)\right)^2 + \frac{a_0^2}{2\alpha} \cdot \Delta t \cdot \sin^2(\delta)} \leq 1 + \frac{a_0^2}{2\alpha} \cdot \Delta t$$

Here we have used the inequality $\sqrt{x^2 + y^2} \leq |x| + |y|$

With $K = \frac{a_0^2}{2\alpha}$, we see that the generalized condition (7.79) is met.

The advection-diffusion equation is discussed in detail in Section6.10 of the Numeriske Beregninger.

7.6 Truncation error, consistency and convergence

7.6.1 Truncation error

Let $U(x,t)$ be the exact solution of a PDE, written as $L(U) = 0$, where L(U) is the differential operator, and u the numerical solution given by a generic numerical scheme, also written in operator form as $F(u) = 0$. In this last case $F(u)$ is the so-called numerical operator. The exact solution at $(x_i,\ t_n)$ is given by:

$$U_i^n \equiv U(x_i, t_n) \qquad \text{der} \quad x_i = i \cdot \Delta x = i \cdot h, \qquad i = 0, 1, 2, \ldots \tag{7.81}$$

$$t_n = n \cdot \Delta t = n \cdot k, \;\; n = 0, 1, 2, \ldots \tag{7.82}$$

We define the local truncation error T_i^n as:

$$T_i^n = F(U_i^n) - L(U_i^n) = F(U_i^n) \tag{7.83}$$

Since in general the exact solution $U(x,t)$ is not available, one finds T_i^n by expressing the exact solution at required space-time locations using Taylor expansions.

Some expansions are given to illustrate the concept:

$$U_{i\pm1}^n \equiv U(x_{i\pm h}, t_n) = U_i^n \pm h \cdot \left.\frac{\partial U}{\partial x}\right|_i^n + \frac{h^2}{2} \cdot \left.\frac{\partial^2 U}{\partial x^2}\right|_i^n \pm \frac{h^3}{6} \cdot \left.\frac{\partial^3 U}{\partial x^3}\right|_i^n + \ldots \tag{7.84}$$

$$U_i^{n\pm1} \equiv U(x_i, t_{n\pm k}) = U_i^n \pm k \cdot \left.\frac{\partial U}{\partial t}\right|_i^n + \frac{k^2}{2} \cdot \left.\frac{\partial^2 U}{\partial t^2}\right|_i^n \pm \frac{k^3}{6} \cdot \left.\frac{\partial^3 U}{\partial t^3}\right|_i^n + \ldots \tag{7.85}$$

For example, let us find the local truncation error T_i^n for the FTCS scheme applied to the diffusion equation. In this case the analytical operator is

$$L(U) = \frac{\partial U}{\partial t} - \frac{\partial^2 U}{\partial x^2} = 0$$

$$T_i^n = F(U_i^n) = \frac{U_i^{n+1} - U_i^n}{k} - \frac{U_{i-1}^n - 2U_i^n + U_{i+1}^n}{h^2} \tag{7.86}$$

Replacing (7.84) in (7.86):

$$T_i^n = \left(\frac{\partial U}{\partial t} - \frac{\partial^2 U}{\partial x^2}\right)_i^n + \left(\frac{k}{2}\frac{\partial^2 U}{\partial t^2} - \frac{h^2}{12}\frac{\partial^4 U}{\partial x^4}\right)_i^n + \frac{k^2}{6}\left.\frac{\partial^3 U}{\partial t^3}\right|_i^n + O(k^3, h^4)$$

$$\text{Da} \;\; \frac{\partial U}{\partial t} - \frac{\partial^2 U}{\partial x^2} = 0$$

$$T_i^n = \left(\frac{k}{2}\frac{\partial^2 U}{\partial t^2} - \frac{h^2}{12}\frac{\partial^4 U}{\partial x^4}\right)_i^n + \text{ higher order terms} \tag{7.87}$$

(7.87) shows that $T_i^n = O(k) + O(h^2)$, as expected.

(7.87) can also be written as:

$$T_i^n = \frac{h^2}{12} \cdot \left(6\frac{k}{h^2}\frac{\partial^2 U}{\partial t^2} - \frac{\partial^4 U}{\partial x^4}\right)_i^n + O(k^2) + O(h^4)$$

By choosing $D = \dfrac{k}{h^2} = \dfrac{1}{6}$, we get:

$$T_i^n = O(k^2) + O(h^4) \tag{7.88}$$

Δt becomes very small for $D = 1/6$, but with today's computers this should not be a problem, except for the possible accumulation of rounding errors.

7.6.2 Consistency

We say that a the discretization of a differential equation, i.e. the numerical scheme, is consistent with the original differential if the local truncation error $T_i^n \to 0$ when Δx and $\Delta t \to 0$ independent of each other.

7.6.3 Example: Consistency of the FTCS-scheme

From (7.87)

$$T_i^n = \left(\frac{k}{2} \frac{\partial^2 U}{\partial t^2} - \frac{h^2}{12} \frac{\partial^4 U}{\partial x^4} \right)_i^n \to 0 \text{ for } h \text{ and } k \to 0$$

This means that the FTCS scheme is consistent with the diffusion equation.

7.6.4 Example: Consistency of the DuFort-Frankel scheme

Let us look at the DuFort-Frankel scheme introduced in Section efch5:sec42:

$$T_i^n = \frac{U_i^{n+1} - U_i^{n-1}}{2k} - \frac{\left[U_{i-1}^n + U_{i+1}^n - (U_i^{n+1} + U_i^{n-1}) \right]}{h^2}$$

Using the series expansions (7.84) and (7.85):

$$T_i^n = \left[\frac{\partial U}{\partial t} - \frac{\partial^2 U}{\partial x^2} + \left(\frac{k}{h} \right)^2 \frac{\partial^2 U}{\partial t^2} \right]_i^n + \left[\frac{k^2}{6} \frac{\partial^3 U}{\partial t^3} - \frac{h^2}{12} \frac{\partial^4 U}{\partial x^4} \right]_i^n + O\left(\frac{k^4}{h^2}, k^4, h^4 \right) \tag{7.89}$$

Due to the factor $\left(\frac{k}{h} \right)^2$ it is important to specify how k and $h \to 0$. The scheme is not necessarily consistent with the underlying differential equation. Such schemes are normally called conditionally consistent.

Case 1

Set $r_0 = \frac{k}{h} \to k = r_0 \cdot h$, and let $r_0 =$ be a positive constant.

Inserting r_0 in (7.89):

$$T_i^n = \left(\frac{\partial U}{\partial t} - \frac{\partial^2 U}{\partial x^2} + r_0^2 \cdot \frac{\partial^2 U}{\partial t^2} \right)_i^n + O(h^2)$$

For $h \to 0$, we see that the DuFort-Frankel scheme is consistent with the hyperbolic equation $\frac{\partial U}{\partial t} + r_0^2 \frac{\partial^2 U}{\partial t^2} = \frac{\partial^2 U}{\partial x^2}$ and not with the original diffusuion equation.

Case 2

Setting $r_0 = \frac{k}{h^2} \to k = r_0 \cdot h^2$. Inserting r_0 in (7.89):

$$
\begin{aligned}
T_i^n &= \left[\frac{\partial U}{\partial t} - \frac{\partial^2 U}{\partial x^2}\right]_i^n + \left[r_0^2 h^2 \frac{\partial^2 U}{\partial t^2} + \frac{k^2}{6}\frac{\partial^3 U}{\partial t^3} - \frac{h^2}{12}\frac{\partial^4 U}{\partial x^4}\right]_i^n + O\left(\frac{k^4}{h^2}, k^4, h^4\right) \\
&= \left[r_0^2 h^2 \frac{\partial^2 U}{\partial t^2} + \frac{k^2}{6}\frac{\partial^3 U}{\partial t^3} - \frac{h^2}{12}\frac{\partial^4 U}{\partial x^4}\right]_i^n + O\left(r_0^4 h^6,\ k^4,\ h^4\right)
\end{aligned}
$$

$$
\text{da } \left[\frac{\partial U}{\partial T} - \frac{\partial^2 U}{\partial x^2}\right]_i^n = 0
$$

We see that in this case $T_i^n \to 0$ for h and $t \to 0$ with $T_i^n = O(k^2) + O(h^2)$.

The scheme is now consistent with the diffusion equation. Therefore, the DuFort-Frankel scheme can be used with $k = r_0 \cdot h^2$. However, this poses a restriction in Δt, arising from consistency constrains and not from stability considerations. Non-consistent schemes usually arise when we change the scheme after we have made the Taylor expansions in the usual way.

7.6.5 Convergence

Lucas 5: this must be considerably improved. No concepts given to understand convergence properly

It is generally difficult to prove the convergence of a difference scheme. Therefore, many attempts have been made to replace the above definition with conditions that are easier to prove individually but which together are sufficient for convergence.

A very important result in this direction is given by the following theorem

Lax's equivalence theorem: Given a well-posed linear initial value problem and a consistent numerical scheme, stability of the same scheme is a necessary and sufficient condition for convergence.

See Section 4.3 in the Numeriske Beregninger for more details. When we see all the conditions that must be fulfilled in order for Lax's theorem to be used, we understand the difficulties of proving convergence in more general issues.

7.7 Example with radial symmetry

If we write the diffusion equation $\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}$ for cylindrical and spherical coordinates and require u to be only a function of time t and radius r, we have that:

Cylinder:

$$
\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial r^2} + \frac{1}{r}\frac{\partial u}{\partial r}
$$

Sphere:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial r^2} + \frac{2}{r}\frac{\partial u}{\partial r}$$

Both equations can be written as:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial r^2} + \frac{\lambda}{r}\frac{\partial u}{\partial r},\ \lambda = 0, 1, 2 \tag{7.90}$$

$\lambda = 0$ with $r \to x$ gives the well-known Cartesian case.

(7.90) is a partial differential equation with variable coefficients. We will now perform a von Neumann stability analysis for this equation for the θ-scheme from Section (7.5.3).

Stability analysis using θ -scheme for radius $r > 0$

Setting $r_j = \Delta r \cdot j,\ j = 0, 1, \ldots$ and introducing $D = \frac{\Delta t}{(\Delta r)^2}$

For $r > 0$:

$$\begin{aligned} u_j^{n+1} = \quad & u_j^n + D\left[\theta(u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}) + (1-\theta)(u_{j+1}^n - 2u_j^n + u_{j-1}^n)\right] \\ & + \tfrac{\lambda D}{2j}\left[\theta(u_{j+1}^{n+1} - u_{j-1}^{n+1}) + (1-\theta)(u_{j+1}^n - u_{j-1}^n)\right] \end{aligned} \tag{7.91}$$

We proceed with the von Neumann analysis by inserting $E_j^n = G^n \cdot e^{i\cdot\beta r_j} = G^n e^{i\cdot\delta\cdot j}$ with $\delta = \beta \cdot \Delta r$ and using the usual formulas (7.49) and (7.50).

We get:

$$G\cdot\left(1 + 4\theta D\sin^2\left(\frac{\delta}{2}\right) - i\cdot\frac{\theta\lambda D}{j}\sin(\delta)\right) = 1 - 4(1-\theta)D\cdot\sin^2\left(\frac{\delta}{2}\right) + i\frac{(1-\theta)\lambda D}{j}\sin(\delta)$$

which, using the formula $\sin(\delta) = 2\sin\left(\frac{\delta}{2}\right)\cos\left(\frac{\delta}{2}\right)$ and the condition $|G| \leq 1$, becomes:

$$\left(1 - 4(1-\theta)D\sin^2\left(\frac{\delta}{2}\right)\right)^2 + \frac{(1-\theta)^2\lambda^2 D^2}{j^2}\sin^2(\delta) \leq \left(1 + 4\theta D\sin^2\left(\frac{\delta}{2}\right)\right)^2 + \frac{\theta^2\lambda^2 D^2}{j^2}\sin^2(\delta)$$

and which further provides:

$$D\cdot(1-2\theta)\cdot\left(\sin^2\left(\frac{\delta}{2}\right)\cdot\left(4 - \frac{\lambda^2}{j^2}\right) + \frac{\lambda^2}{j^2}\right) \leq 2,\ j \geq 1 \tag{7.92}$$

It is not difficult to see that the term in parenthesis has its largest value for $\sin^2\left(\frac{\delta}{2}\right) = 1$; i.e. for $\delta = \pi$. (This can also be found by deriving the term with respect to δ, which has its maximum for $\delta = \pi$). Factors $\frac{\lambda^2}{j^2}$ fall then out.

We get:

$$D\cdot(1-2\theta)\cdot 2 \leq 1 \tag{7.93}$$

As in Section 7.5.3, we must distinguish between two cases:

1.

$$0 \leq \theta \leq \frac{1}{2}$$

$$D = \frac{\Delta t}{(\Delta r)^2} \leq \frac{1}{2(1-2\theta)} \tag{7.94}$$

2.

$$\frac{1}{2} \leq \theta \leq 1$$

Next we multiply (7.93) by -1:

$$D \cdot (2\theta - 1) \cdot 2 \geq -1$$

This condition is always satisfied for the specified range of θ, so that in such cases the scheme is unconditionally stable.

In other words, we have obtain the same stability condition as for the equation with constant coefficients: $\dfrac{\partial u}{\partial t} = \dfrac{\partial^2 u}{\partial r^2}$, where we have the FTCS scheme for $\theta = 0$, the Crank-Nicolson scheme for $\theta = 1/2$, and the Laasonen scheme for $\theta = 1$.

Note that this analysis is only valid for $r > 0$.

We now have a look at the equation for $r = 0$.

The term $\dfrac{\lambda}{r}\dfrac{\partial u}{\partial r}$ must be treated with special care for $r = 0$.

L'Hopital's rule:

$$\lim_{r \to 0} \frac{\lambda}{r}\frac{\partial u}{\partial r} = \lambda \frac{\partial^2 u}{\partial r^2} \to \frac{\partial u}{\partial t} = (1+\lambda)\frac{\partial^2 u}{\partial r^2} \text{ for } r = 0 \tag{7.95}$$

We have found that using the FTCS scheme requires the usual constrain $D \leq 1/2$. Now we will check if boundary conditions pose some constrains when using the FTCS scheme.

Version 1

Discretize (7.95) for $r = 0$ and use the symmetry condition $\frac{\partial u}{\partial r}(0) = 0$:

$$u_0^{n+1} = [1 - 2(1+\lambda)D] \cdot u_0^n + 2(1+\lambda)D \cdot u_1^n \tag{7.96}$$

If we use the PC-criterion on (7.96), we get:

$$D \leq \frac{1}{2(1+\lambda)} \tag{7.97}$$

For $\lambda = 0$ we get the well-known condition $D \leq 1/2$, whereas for the cylinder ($\lambda = 1$) we get $D \leq 1/4$ and for the sphere, ($\lambda = 2$) we have $D \leq 1/6$. The question to be answered now is whether this conditions are for cylinder and sphere are necessary. We know that $D \leq 1/2$ is necessary and sufficient for $\lambda = 0$.

It is difficult to find a necessary and sufficient condition in this case. Therefore, we concentrate in a special case: flow start-up in a tube, which is described in Example 7.7.1.

Version 2

To avoid using a separate equation for $r = 0$, we discretize $\frac{\partial u}{\partial r}(0) = 0$ with a second order forward difference:

$$\frac{\partial u}{\partial r}(0) = 0 \rightarrow \frac{-3u_0^n + 4u_1^n - u_2^n}{2\Delta r} \rightarrow u_0^n = \frac{1}{3}(4u_1^n - u_2^n) \tag{7.98}$$

For a sphere, there is a detailed analysis by Dennis Eisen in the Journal *Numerische Mathematik* vol. 10, 1967, pages 397-409. The author shows that a necessary and sufficient condition for the solution of (7.91), along with (7.96) (for $\lambda = 2$ and $\theta = 0$) is that $D < 1/3$. In addition, he shows that by avoiding the use of (7.96), the stability constrain for the FTCS scheme results to be $D < 1/2$.

Next we will go through two cases to see which stability requirements are obtained when one uses the two types of boundary conditions.

7.7.1 Example: Start-up flow in a tube

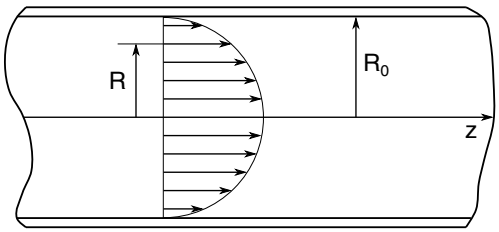

Figure 7.10: Velocity profile in a pipe at a given time.

Figure 7.10 shows the velocity profile of an incompressible fluid in a tube at a given time. The profile has reached the current configuration after evolving from a static configuration (velocity equal to 0 over the entire domain) by the application of a constant pressure gradient $\frac{dp}{dz} < 0$, so that we also know the velocity profile for the steady state configuration, which is the well-known parabolic profile for Poiseuille flow.

Under the above mentioned conditions, the momentum equation reads:

$$\frac{\partial U}{\partial \tau} = -\frac{1}{\rho}\frac{dp}{dz} + \nu\left(\frac{\partial^2 U}{\partial R^2} + \frac{1}{R}\frac{\partial U}{\partial R}\right) \tag{7.99}$$

with $U = U(R,\ \tau)$. $0 \leq R \leq R_0$, being the velocity profile and τ the physical time.

Non-dimensional variables are:

$$t = \nu \frac{\tau}{R_0^2},\ r = \frac{R}{R_0},\ u = \frac{U}{k},\ u_s = \frac{U_s}{k} \text{ der } k = -\frac{R_0^2}{4\mu}\frac{dp}{dz} \tag{7.100}$$

which introduced in (7.99) yield:

$$\frac{\partial u}{\partial t} = 4 + \frac{\partial^2 u}{\partial r^2} + \frac{1}{r}\frac{\partial u}{\partial r} \tag{7.101}$$

Boundary conditions are:

$$u(\pm 1, t) = 0,\ \frac{\partial u}{\partial r}(0, t) = 0 \tag{7.102}$$

The later is a symmetry condition. Finding a stationary solution requires that $\frac{\partial u}{\partial t} = 0$:

$$\frac{d^2 u_s}{dr^2} + \frac{1}{r}\frac{du_s}{dr} = -4 \rightarrow \frac{1}{r}\frac{d}{dr}\left(r\frac{du_s}{dr}\right) = -4 \text{ som gir:}$$

$$\frac{du_s}{dr} = -2r + \frac{C_1}{r} \text{ med } C_1 = 0 \text{ da } \frac{du_s(0)}{dr} = 0$$

After a new integration and application of boundary conditions we obtain the familiar parabolic velocity profile:

$$u_s = 1 - r^2 \tag{7.103}$$

We assume now that we have a fully developed profile as given in (7.103) and that suddenly we remove the pressure gradient. From (7.99) we see that this results in a simpler equation. The velocity $\omega(r, t)$ for this case is given by:

$$\omega(r, t) = u_s - u(r, t) \text{ med } \omega = \frac{W}{k} \tag{7.104}$$

We will now solve the following problem:

$$\frac{\partial \omega}{\partial t} = \frac{\partial^2 \omega}{\partial r^2} + \frac{1}{r}\frac{\partial \omega}{\partial r} \tag{7.105}$$

with boundary conditions:

$$\omega(\pm 1, t) = 0,\ \frac{\partial \omega}{\partial r}(0, t) = 0 \tag{7.106}$$

and initial conditions:

$$\omega(r, 0) = u_s = 1 - r^2 \tag{7.107}$$

The original problem now reads:

$$u(r, t) = 1 - r^2 - \omega(r, t) \tag{7.108}$$

The analytical solution of (7.105), (7.108), first reported by Szymanski in 1932, can be found in Appendix G.8 of the Numeriske Beregninger.

Let us now look at the FTCS scheme.

From (7.91), with $\lambda = 1$, $\theta = 0$ and $j \geq 0$:

$$\omega_j^{n+1} = \omega_j^n + D \cdot (\omega_{j+1}^n - 2\omega_j^n + \omega_{j-1}^n) + \frac{D}{2j} \cdot (\omega_{j+1}^n - \omega_{j-1}^n) \tag{7.109}$$

For $j = 0$ (7.96) yields:

$$\omega_0^{n+1} = (1 - 4D) \cdot \omega_0^n + 4D \cdot \omega_1^n \tag{7.110}$$

From (7.94) we obtain that the stability range is $0 < D \leq \frac{1}{2}$ for $r > 0$ and $0 < D \leq \frac{1}{4}$
from (7.97) for $r = 0$, which is an adequate condition.

By solving (7.109), we get the following table for stability limits:

Δr	D
0.02	0.413
0.05	0.414
0.1	0.413
0.2	0.402
0.25	0.394
0.5	0.341

For satisfactory accuracy, we should have that $\Delta r \leq 0.1$. From the table above we see that in terms of stabilty, $D < 0.4$ is sufficient. In order words, a sort of mean value between $D = \frac{1}{2}$ og $D = \frac{1}{4}$.

The trouble causing equation is (7.110). We can avoid this problem by using the following equation instead of (7.98):

$$\omega_0^n = \frac{1}{3}(4\omega_1^n - \omega_2^n), \ n = 0, 1, \ldots \tag{7.111}$$

A new stability analysis shows then that the new stability condition for the entire system is now $0 < D \leq \frac{1}{2}$ for the FTCS scheme. Figure 7.11 shows the velocity profile u for $D = 0.45$ with $\Delta r = 0.1$ after 60 time increments using (7.110). The presence of instabilities for $r = 0$ can be clearly observed.

Marit 6: Har ikke endret noe her

Programming of the θ-scheme:

Boundary conditions given in (7.110)

We set $r_j = \Delta r \cdot (j)$, $\Delta r = \frac{1}{N}$, $j = 0, 1, 2, \ldots, N$ as shown in Fig. 7.12. From Eq. (7.91) and (7.95) we get:

For $j = 0$:

$$(1 + 4D \cdot \theta) \cdot \omega_0^{n+1} - 4D \cdot \theta \cdot \omega_1^{n+1} = \omega_0^n + 4D(1 - \theta) \cdot (\omega_1^n - \omega_0^n) \tag{7.112}$$

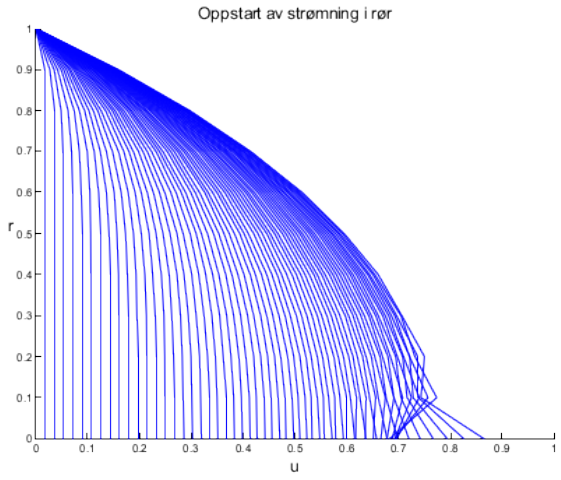

Figure 7.11: Velocity profile for start-up flow in a tube for different time step obtained using the FTCS scheme. Stability development can be observed.

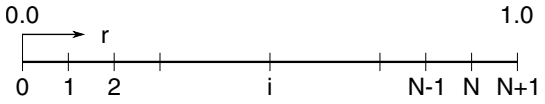

Figure 7.12: Spatial discretization for the programing of the $\theta scheme$.

For $j = 1, \ldots, N-1$:

$$\begin{aligned} &-D\theta \cdot \left(1 - \tfrac{1}{2j}\right) \cdot \omega_{j-1}^{n+1} + (1 + 2D\theta) \cdot \omega_j^{n+1} - D\theta \cdot \left(1 + \tfrac{1}{2j}\right) \cdot \omega_{j+1}^{n+1} \\ &= D(1-\theta) \cdot \left(1 - \tfrac{1}{2j}\right) \cdot \omega_{j-1}^{n} + [1 - 2D(1-\theta)] \cdot \omega_j^{n} \\ &+D(1-\theta) \cdot \left(1 + \tfrac{1}{2j}\right) \cdot \omega_{j+1}^{n} \end{aligned} \tag{7.113}$$

Initial values:

$$\omega_j^0 = 1 - r_j^2 = 1 - [\Delta r \cdot j]^2, \; j = 0, \ldots, N \tag{7.114}$$

In addition we have that $\omega_N = 0$ for every n-values.

The system written in matrix form now becomes:

$$\begin{pmatrix} 1+4D\theta & -4D\theta & 0 & 0 & 0 & 0 & 0 \\ -D\theta(1-\frac{0.5}{1}) & 1+2D\theta & -D\theta(1+\frac{0.5}{1}) & 0 & 0 & 0 & 0 \\ 0 & -D\theta(1-\frac{0.5}{2}) & 1+2D\theta & -D\theta(1+\frac{0.5}{2}) & 0 & 0 & 0 \\ 0 & 0 & \cdot & \cdot & \cdot & 0 & 0 \\ 0 & 0 & 0 & \cdot & \cdot & \cdot & 0 \\ 0 & 0 & 0 & 0 & -D\theta(1-\frac{0.5}{N-2}) & 1+2D\theta & -D\theta(1+ \\ 0 & 0 & 0 & 0 & 0 & -D\theta(1-\frac{0.5}{N-1}) & 1+2 \end{pmatrix} \tag{7.115}$$

Note that we can also use a **linalg-solver** for $\theta = 0$ (FTCS-scheme). In this case the elements in the first upper- and lower diagonal are all 0. The determinant of the matrix is now the product of the elements on the main diagonal. The criterion for non singualar matrix is then that all the elements on the main diagonal are $\neq 0$, which is satisfied.

The python function **thetaSchemeNumpyV1** show how one could implement the algorithm for solving w^{n+1}. The function is part of the script/module **startup.py** which may be downloaded in your LiClipse workspace.

```
# Theta-scheme and using L'hopital for r=0
def thetaSchemeNumpyV1(theta, D, N, wOld):
    """ Algorithm for solving w^(n+1) for the startup of pipeflow
        using the theta-schemes. L'hopitals method is used on the
        governing differential equation for r=0.

        Args:
            theta(float): number between 0 and 1. 0->FTCS, 1/2->Crank, 1->Laasonen
            D(float): Numerical diffusion number [dt/(dr**2)]
            N(int): number of parts, or dr-spaces. In this case equal to the number of unknowns
            wOld(array): The entire solution vector for the previous timestep, n.

        Returns:
            wNew(array): solution at timestep n+1
    """

    superDiag = np.zeros(N - 1)
    subDiag = np.zeros(N - 1)
    mainDiag = np.zeros(N)

    RHS = np.zeros(N)

    j_array = np.linspace(0, N, N + 1)
    tmp = D*(1. - theta)

    superDiag[1:] = -D*theta*(1 + 0.5/j_array[1:-2])
    mainDiag[1:] = np.ones(N - 1)*(1 + 2*D*theta)
    subDiag[:] = -D*theta*(1 - 0.5/j_array[1:-1])

    a = tmp*(1 - 1./(2*j_array[1:-1]))*wOld[0:-2]
    b = (1 - 2*tmp)*wOld[1:-1]
    c = tmp*(1 + 1/(2*j_array[1:-1]))*wOld[2:]

    RHS[1:] = a + b + c
```

```
superDiag[0] = -4*D*theta
mainDiag[0] = 1 + 4*D*theta
RHS[0] = (1 - 4*tmp)*wOld[0] + 4*tmp*wOld[1]

A = scipy.sparse.diags([subDiag, mainDiag, superDiag], [-1, 0, 1], format='csc')

wNew = scipy.sparse.linalg.spsolve(A, RHS)
wNew = np.append(wNew, 0)


return wNew
```

Boundary condition given in (7.98)

Now let us look at the numerical solution when using the second order forward difference for the bc at $r = 0$ repeated here for convinience:

$$\omega_0^n = \frac{1}{3}(4\omega_1^n - \omega_2^n),\ n = 0, 1, 2 \ldots \tag{7.116}$$

The difference equation for the θ-scheme for $j = 1, \ldots, N-1$ are the same as in the previous case:

$$\begin{aligned} &-D\theta \cdot \left(1 - \tfrac{1}{2j}\right) \cdot \omega_{j-1}^{n+1} + (1 + 2D\theta) \cdot \omega_j^{n+1} - D\theta \cdot \left(1 + \tfrac{1}{2j}\right) \cdot \omega_{j+1}^{n+1} \\ &= D(1-\theta) \cdot \left(1 - \tfrac{1}{2j}\right) \cdot \omega_{j-1}^{n} + [1 - 2D(1-\theta)] \cdot \omega_j^n \\ &+D(1-\theta) \cdot \left(1 + \tfrac{1}{2j}\right) \cdot \omega_{j+1}^{n} \end{aligned} \tag{7.117}$$

Now inserting Eq. (7.116) into Eq. (7.117) and collecting terms we get the following difference equation for $j = 1$:

$$(1 + \frac{4}{3}D\theta) \cdot \omega_1^{n+1} - \frac{4}{3}D\theta\omega_2^{n+1} = [1 - \frac{4}{3}D(1-\theta)] \cdot \omega_1^n + \frac{4}{3}D(1-\theta)\omega_2^n \tag{7.118}$$

When $\omega_1, \omega_2, \ldots$ are found, we can calculate ω_0 from (7.116).

The initial values are as in Eq. (7.114), repeated here for convonience:

$$\omega_j^0 = 1 - r_j^2 = 1 - (\Delta r \cdot j)^2,\ j = 0, \ldots, N+1 \tag{7.119}$$

The system written in matrix form now becomes:

$$\begin{pmatrix} 1+\frac{4}{3}D\theta & -\frac{4}{3}D\theta & 0 & 0 & 0 & 0 & 0 \\ -D\theta(1-\frac{0.5}{2}) & 1+2D\theta & -D\theta(1+\frac{0.5}{2}) & 0 & 0 & 0 & 0 \\ 0 & -D\theta(1-\frac{0.5}{3}) & 1+2D\theta & -D\theta(1+\frac{0.5}{3}) & 0 & 0 & 0 \\ 0 & 0 & \cdot & \cdot & \cdot & 0 & 0 \\ 0 & 0 & 0 & \cdot & \cdot & \cdot & 0 \\ 0 & 0 & 0 & 0 & -D\theta(1-\frac{0.5}{N-2}) & 1+2D\theta & -D\theta(1+ \\ 0 & 0 & 0 & 0 & 0 & -D\theta(1-\frac{0.5}{N-1}) & 1+2 \end{pmatrix} \tag{7.120}$$

The python function **thetaSchemeNumpyV2** show how one could implement the algorithm for solving w^{n+1}. The function is part of the script/module **startup.py** which may be downloaded in your LiClipse workspace.

In Fig. 7.13 we see that the stability-limit for the two version is different for the to versions of treating the BC, for **FTCS**. Using L'hopital's rule for r=0 give a smaller stability-limit, and we see that unstability arises at r=0, befor the general stability-limit for the FTSC scheme ($D = 1/2$).

```
# Theta-scheme and using 2nd order forward difference for r=0
def thetaScheme_numpy_V2(theta, D, N, wOld):
    """ Algorithm for solving w^(n+1) for the startup of pipeflow
        using the theta-schemes. 2nd order forward difference is used
        on the von-Neumann bc at r=0.

        Args:
            theta(float): number between 0 and 1. 0->FTCS, 1/2->Crank, 1->Laasonen
            D(float): Numerical diffusion number [dt/(dr**2)]
            N(int): number of parts, or dr-spaces.
            wOld(array): The entire solution vector for the previous timestep, n.

        Returns:
            wNew(array): solution at timestep n+1
    """
    superDiag = np.zeros(N - 2)
    subDiag = np.zeros(N - 2)
    mainDiag = np.zeros(N-1)

    RHS = np.zeros(N - 1)

    j_array = np.linspace(0, N, N + 1)
    tmp = D*(1. - theta)

    superDiag[1:] = -D*theta*(1 + 0.5/j_array[2:-2])
    mainDiag[1:] = np.ones(N - 2)*(1 + 2*D*theta)
    subDiag[:] = -D*theta*(1 - 0.5/j_array[2:-1])

    a = tmp*(1 - 1./(2*j_array[2:-1]))*wOld[1:-2]
    b = (1 - 2*tmp)*wOld[2:-1]
    c = tmp*(1 + 1/(2*j_array[2:-1]))*wOld[3:]

    RHS[1:] = a + b + c

    superDiag[0] = -(4./3)*D*theta
    mainDiag[0] = 1 + (4./3)*D*theta
    RHS[0] = (1 - (4./3)*tmp)*wOld[1] + (4./3)*tmp*wOld[2]

    A = scipy.sparse.diags([subDiag, mainDiag, superDiag], [-1, 0, 1], format='csc')

    wNew = scipy.sparse.linalg.spsolve(A, RHS)
    w_0 = (1./3)*(4*wNew[0] - wNew[1])

    wNew = np.append(w_0, wNew)
    wNew = np.append(wNew, 0)


    return wNew
```

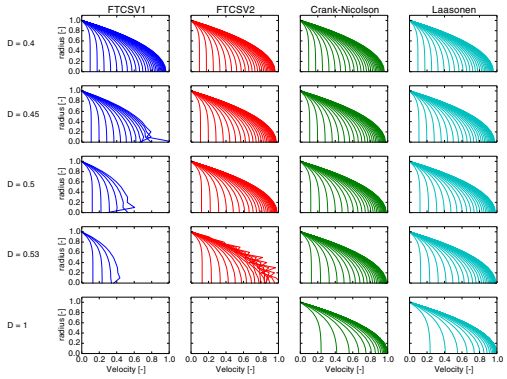

Figure 7.13: Results for stability tests performed for the two versions of the FTCS scheme, the Crank-Nicolson and the Laasonen schemes. Note that version 1 of FTCS scheme develops instabilities before $D = 0.5$, and that instabilities start at $r = 0$. For version 2 of FTCS scheme instability occurs for $D = 0.5$ and for all values of r, as expected.

Lucas 7: I do not see the point of this video, one should use a spatial resolution that somehow illustrates differences in accuracy, if all results are mesh independent you can not make any interesting observation ...

Movie 5: Animation of numerical results obtained using three numerical schemes as well as the analytical solution. `mov-ch5/startup.mp4`

```
emfs /src-ch5/ #python #package startup.py  @ git@lrhgit/tkt4140/src/src-ch5/startup_compendium.py;
```

7.7.2 Example: Cooling of a sphere

Figure 7.14 shows our problem setup, which consists of a sphere immersed in water and exchanging heat with it. The sphere radius is $b = 5$ and its temperature before it is immersed in water is T_k. Moreover, we assume that the water keeps a constant temperature T_v throughout the entire process and we neglect heat exchange with the environment.

Additional data are

- Heat conductivity: $k = 0.1W/(cm \cdot^\circ C)$
- Heat transfer coefficient: $\bar{h} = 0.2W/(cm \cdot^\circ C)$
- Thermal diffusivity: $\alpha = 0.04cm^2/s$

We have chosen the above reported coefficients so that $\dfrac{\bar{h} \cdot b}{k} = 1$, since this choice leads to a simple analytical solution. Moreover, such values correspond to characteristic values found for nickel alloys.

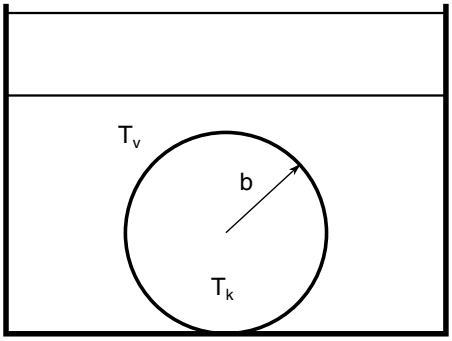

Figure 7.14: Problem setup for example on the cooling of a sphere.

We must now solve the following problem with $T = T(r,t)$:

$$\frac{\partial T}{\partial t} = \alpha\left(\frac{\partial^2 T}{\partial r^2} + \frac{2}{r}\frac{\partial T}{\partial r}\right), \tag{7.121}$$

with boundary conditions:

$$\frac{\partial T}{\partial r}(0,t) = 0, \text{ (symmetribetingelse)} \tag{7.122}$$

For $r = b$:

$$k\frac{\partial T}{\partial r} = \bar{h}\cdot(T_v - T_b) \tag{7.123}$$

Initial conditions are:

$$T(r,0) = T_k \tag{7.124}$$

In this example we use the FTCS scheme, but the derivation can be easily extended to the θ-scheme as shown in Example 7.7.1. With $r_j = \Delta r \cdot j,\ \Delta r = \frac{1}{N+1},\ j = 0,1,\ldots,N+1$ and $D = \alpha\frac{\Delta t}{(\Delta r)^2}$ from (7.91) we get that when $u(r,t) \to T(r,t),\ \theta = 0$ and $\lambda = 2$:

$$T_j^{n+1} = (1-2D)\cdot T_j^n + D[(1-1/j)\cdot T_{j-1}^n + (1+1/j)\cdot T_{j+1}^n],\ j = 1,2,\ldots \tag{7.125}$$

As in Example 7.7.1, we employ two versions of the symmetry condition for $r = 0$. **1)**

From (7.96) with $\lambda = 2$:

$$T_0^{n+1} = (1-6D)\cdot T_0^n + 6D\cdot T_1^n \tag{7.126}$$

2) From (7.98):

$$T_0^n = \frac{1}{3}(4T_1^n - T_2^n), \text{ alle } n \tag{7.127}$$

Boundary conditions for $r = b$.
Discretizing (7.122) using 2nd order backward differences:

$$k \cdot \left(\frac{3T_{N+1}^n - 4T_N^n + T_{N-1}^n}{2 \cdot \Delta r} \right) = \bar{h} \cdot (T_v - T_b)$$

solving for T_{N+1}^n yields:

$$T_{N+1}^n = \frac{4T_N^n - T_{N-1}^n + 2\delta \cdot T_v}{3 + 2\delta} \tag{7.128}$$

with

$$\delta = \frac{\Delta r \cdot \bar{h}}{k} \tag{7.129}$$

We have previously shown that we must have $D < 1/3$ when we use boundary condition (7.126). In Example 7.7.1 for a cylindrical domain, we found that the stability limit of D increased when we reduced Δr using the FTCS scheme. On the other hand, for the spherical case it appears that the condition $D < 1/3$ is independent of Δr. The reason for this can be understood by writing (7.125) for $j = 1$:

$$T_1^{n+1} = (1 - 2D) \cdot T_1^n + 2D \cdot T_2^n$$

The term $(1 - 1/j) \cdot T_{j-1}^n = (1 - 1) \cdot T_0^n$ vanishes for $j = 1$, so that the temperature in the center of the ball does not affect any other point. This explains why the stability limit is independent of Δr. We can actually solve for $j = 1, 2, \ldots, N$ without bothering about boundary conditions (7.126) and (7.127) (see (**??**in [**?**]) for further details). For boundary condition (7.127) we only need to find the temperature T_0^n at the center of the sphere.

von Neumann stability analysis without considering the boundary conditions showed that (7.125) is stable for $D \leq 1/2$. In addition, the analysis showed that the influence of the variable coefficient disappeared for both, the cylindrical and the spherical cases (see discussion in connection with (7.92)). We have not shown that $D \leq 1/2$ is an adequate condition for the entire system, since that requires to include the boundary condition (7.128).

We can summarize the usage of the FTCS scheme for the spherical case as follows:

The scheme in (7.128) is stable for $D < 1/2$ for $j = 1, 2, \ldots$

If the temperature at the center of the sphere is also required we must ensure that $D < 1/3$ when (7.126) is used.

Using (7.127), the temperature at the center of the sphere can be computed for $D < 1/2$.

This is in agreement with Eisen's analysis, mentioned in connection to (7.98).

Below we show the printout resulting from running program **kule** for a computation where we used $D = 0.4$ and $\Delta r = 0.1$cm. $T_k = 300°C$, $T_v = 20°C$ and a simulation time of 10 minutes, with time steps of 1 second. The results for the analytical solution are computed using the program function **kanalyt**. We see that there is a good agreement between analytical and numerical values (the analytical solution is derived in Appendix G.9 of the Numeriske Beregninger.)

r(cm)	T ($°C$)	T_a	r(cm)	T ($°C$)	T_a
0.00	53.38	53.37	2.60	49.79	49.78
0.10	53.37	53.36	2.70	49.52	49.51
0.20	53.36	53.35	2.80	49.24	49.23
0.30	53.33	53.32	2.90	48.95	48.94
0.40	53.29	53.28	3.00	48.65	48.64
0.50	53.24	53.23	3.10	48.35	48.34
0.60	53.18	53.17	3.20	48.03	48.03
0.70	53.11	53.10	3.30	47.71	47.71
0.80	53.03	53.02	3.40	47.38	47.38
0.90	52.93	52.93	3.50	47.05	47.04
1.00	52.83	52.82	3.60	46.70	46.70
1.10	52.72	52.71	3.70	46.35	46.35
1.20	52.59	52.58	3.80	46.00	45.99
1.30	52.46	52.45	3.90	45.63	45.63
1.40	52.31	52.30	4.00	45.26	45.26
1.50	52.16	52.15	4.10	44.89	44.88
1.60	51.99	51.98	4.20	44.50	44.50
1.70	51.81	51.81	4.30	44.11	44.11
1.80	51.63	51.62	4.40	43.72	43.71
1.90	51.43	51.42	4.50	43.32	43.31
2.00	51.22	51.22	4.60	42.92	42.91
2.10	51.01	51.00	4.70	42.51	42.50
2.20	50.78	50.78	4.80	42.09	42.09
2.30	50.55	50.54	4.90	41.67	41.67
2.40	50.30	50.30	5.00	41.25	41.24
2.50	50.05	50.04			

Movie 6: Animation for example on the cooling of a sphere using the FTCS scheme, as well as analytical solution. `mov-ch5/sphere.mp4`

Chapter 8

Convection problems and hyperbolic PDEs

8.1 The advection equation

The classical advection equation is very often used as an example of a hyperbolic partial differential equation which illustrates many features of convection problems, while still being linear:

$$\frac{\partial u}{\partial t} + a_0 \frac{\partial u}{\partial x} = 0 \tag{8.1}$$

Another convenient feature of the model equation (8.1) is that is has an analytical solution:

$$u = u_0\, f(x - a_0\, t) \tag{8.2}$$

and represents a wave propagating with a constant velocity a_0 with unchanged shape. When $a_0 > 0$, the wave propagates in the positive x-direction, whereas for $a_0 < 0$, the wave propagates in the negative x-direction.

Equation (8.1) may serve as a model-equation for a compressible fluid, e.g if u denote pressure it represents a pressure wave propagating with the velocity a_0. The advection equation may also be used to model the propgation of pressure or flow in a compliant pipe, such as a blood vessel.

To allow for generalization we will also when appropriate write (8.1) on the following form:

$$\frac{\partial u}{\partial t} + \frac{\partial F}{\partial x} = 0 \tag{8.3}$$

where for the linear advection equation $F(u) = a_0\, u$.

8.2 Forward in time central in space discretization

We may discretize (8.1) with a forward difference in time and a central difference in space, normally abberviated as the FTCS-scheme:

$$\frac{\partial u}{\partial t} \approx \frac{u_j^{n+1} - u_j^n}{\Delta t}, \qquad \frac{\partial u}{\partial x} \approx \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}$$

and we may substitute the approximations (8.2) into the advection equation (8.1) to yield:

$$u_j^{n+1} = u_j^n - \frac{C}{2}(u_{j+1}^n - u_{j-1}^n) \tag{8.4}$$

For convenience we have introduced the non-dimensional Courant-Friedrich-Lewy number (or CFL-number or Courant-number for short):

$$C = a_0 \frac{\Delta t}{\Delta x} \tag{8.5}$$

The scheme in (8.4) is first order in time and second order in space (i.e. ($O(\Delta t)$+ $O(\Delta x^2)$)), and explicit in time as can bee seen both from Figure 8.1 and (8.4).

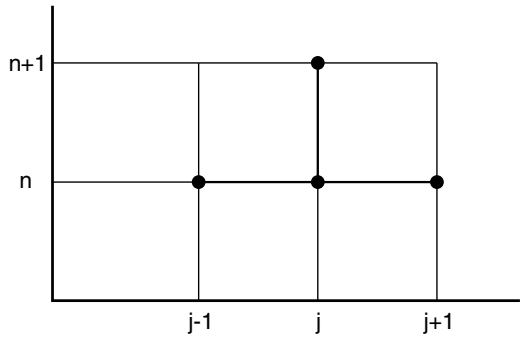

Figure 8.1: Illustration of the first order in time central in space scheme.

We will try to solve model equation (8.1) with the scheme (8.4) and initial conditions illustrated in Fig (8.2) with the mathematical representation:

$$u(x,0) = 1 \text{ for } x < 0.5$$
$$u(x,0) = 0 \text{ for } x > 0.5$$

Solutions for three CFL-numbers: C=0.25, 0.5 and 1.0 are illustrated in Figure 8.3. Large oscillations are observed for all values of the CFL-number, even though they seem to be sligtly reduced for smaller C-values,; thus we

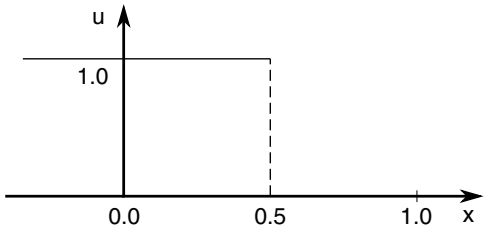

Figure 8.2: Initial values for the advection equaution (8.1).

have indications of an unstable scheme. As a first approach observe that the coefficient for u_{j+1}^n in (8.4) always will be negative, and thus the criterion of positive coefficients (PC-criterion) may not be satisfied for any value of C.

Marit 2: Har ikke endret figuren

However, as we know that the PC-criterion may be too strict in some cases, we proceed with a von Neumann analysis by introducing the numerical amplification factor G^n for the error E_j^n in the numerical scheme to be analyzed

$$u_j^n \rightarrow E_j^n = G^n \cdot e^{i\cdot\beta x_j} \tag{8.6}$$

Substitution of (8.6) into (8.4) yields:

$$G^{n+1}e^{i\cdot\beta\cdot x_j} = G^n e^{i\cdot\beta\cdot x_j} - \frac{C}{2}\left(G^n e^{i\cdot\beta x_{j+1}} - G^n e^{i\cdot\beta x_{j-1}}\right)$$

which after division with $G^n e^{i\cdot\beta\cdot x_j}$ and introduction of the simplified notation $\delta = \beta \cdot h$ yields:

$$G = 1 - \frac{C}{2}\left(e^{i\cdot\beta h} - e^{-i\cdot\beta h}\right) = 1 - i\cdot C\sin(\delta)$$

where the trigonometric relations:

$$2\cos(x) = e^{ix} + e^{-ix} \tag{8.7}$$

$$i\cdot 2\sin(x) = e^{ix} - e^{-ix} \tag{8.8}$$

$$\cos(x) = 1 - 2\sin^2(\frac{x}{2}) \tag{8.9}$$

have been introduced for convenience. Finally, we get the following expression for the numerical ampliciation factor:

$$|G| = \sqrt{1 + C^2\sin^2(\delta)} \geq 1 \text{ for all } C \text{ and } \delta$$

and concequently the FTCS-scheme is unconditionally unstable for the advection equation and is thus not a viable scheme. Even a very small value of C will not suffice to dampe the oscillations.

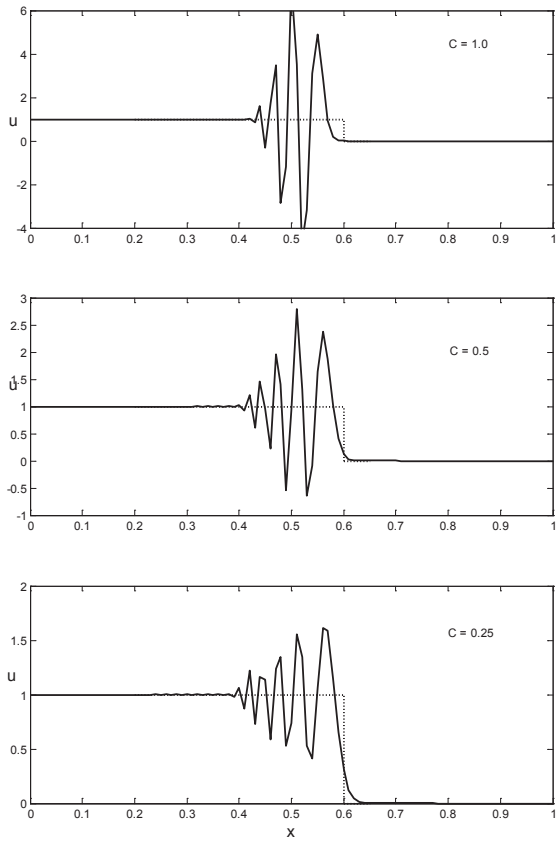

Figure 8.3: Computed solutions with the (8.4). Dotted line: analytical solution, solid line: computed soultion.

8.3 Upwind schemes

Upwind differences are also called upstream differences. Numerically, this term indicates that we use backward differences with respect to the direction from which information is coming. Upwind differences are used only for convection terms, never for diffusive terms. Figure 8.4 shows a graphical interpretation of upwind differences when applied to the advection equation . The resulting scheme is called the upwind scheme.

Upwind finite difference for the convective term:

$$\frac{\partial u}{\partial x} = \frac{u_j^n - u_{j-1}^n}{\Delta x} + O(\Delta x) \tag{8.10}$$

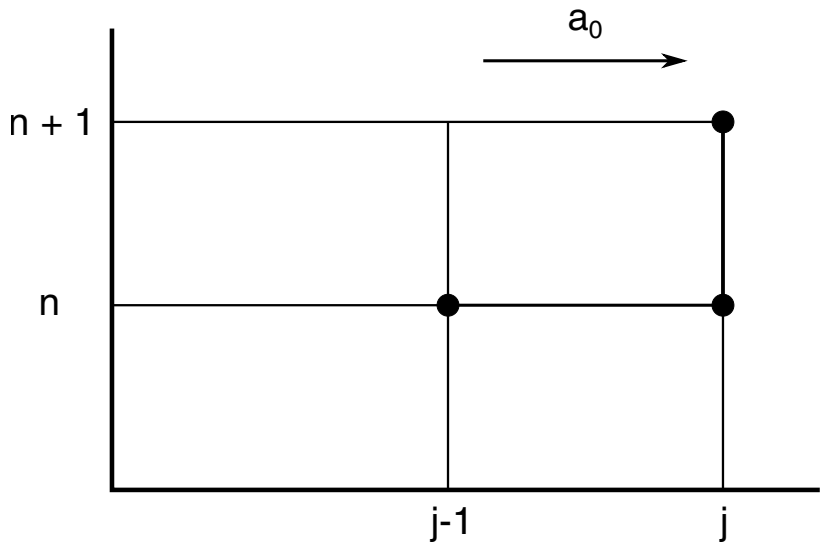

Figure 8.4: Stencil of the upwind scheme for the linear advection equation with characteristic speed a_0.

Replacing (8.10) in (8.1) and using a forward difference for $\frac{\partial u}{\partial t}$ and $C = \frac{a_0 \Delta t}{\Delta x}$ yields:

$$u_j^{n+1} = u_j^n - C \cdot (u_j^n - u_{j-1}^n) = (1 - C)u_j^n + C \cdot u_{j-1}^n \tag{8.11}$$

(8.11) has an a truncation error of order $O(\Delta t) + O(\Delta x)$

If we set $C = 1$ into (8.11), we get that $u_j^{n+1} = u_{j-1}^n$, which in turn is the exact solution of the partial differential equation $\frac{\partial u}{\partial t} + a_0 \frac{\partial u}{\partial x} = 0$ (see Section 5.2). For $C < 1$ the wavefront of the numerical solution is smoothed, indicating that the scheme introduces a so called *numerical viscosity* for these values of C. We will come back to the term *numerical viscosity* later on. Computational results shown in Figure 8.5 indicate that the scheme should be stable for $C \leq 1$. In fact, the PC criterion shows that the scheme should be stable for that range of C. Let us use the von Neumann stability analysis to see if the stability range extends beyond that range. Insert (8.11) from (7.42) in Chapter 7:

$$G^{n+1} e^{i \cdot \beta \cdot x_j} = (1 - C) \cdot G^n e^{i \cdot \beta \cdot x_j} + C \cdot G^n e^{i \cdot \beta \cdot x_{j-1}}$$

dividing by $G^n e^{i \cdot \beta \cdot x_j}$ gives:

$$G = (1-C)+C\cdot e^{-i\cdot\beta\cdot h} = 1-C+C\cdot(cos(\delta)-i\cdot\sin(\delta)) = 1+C\cdot(\cos(\delta)-1)-i\cdot C\cdot\sin(\delta)$$

$$|G| = \sqrt{[1 + C\cos(\delta - 1)]^2 + C^2\sin^2(\delta)} = \sqrt{1 - 2C(1 - \cos(\delta)) \cdot (1 - C)} \tag{8.12}$$

Enforcing the stability criterion $|G| \leq 1$ results in the following:

$$1 - 2C \cdot (1 - \cos(\delta)) \cdot (1 - C) \leq 1 \Rightarrow C \cdot (1 - \cos(\delta)) \cdot (1 - C) \geq 0$$

eller

$$C \cdot \sin^2\left(\frac{\delta}{2}\right)(1 - C) \geq 0$$

Stability interval:

$$0 < C \leq 1 \tag{8.13}$$

(8.13) also applies when we set $C = |a_0| \cdot \Delta t/\Delta x$. All stable explicit 2-level schemes for the advection equation have a restriction on the admissible Courant-number .

Marit 9: Har ikke endret figur

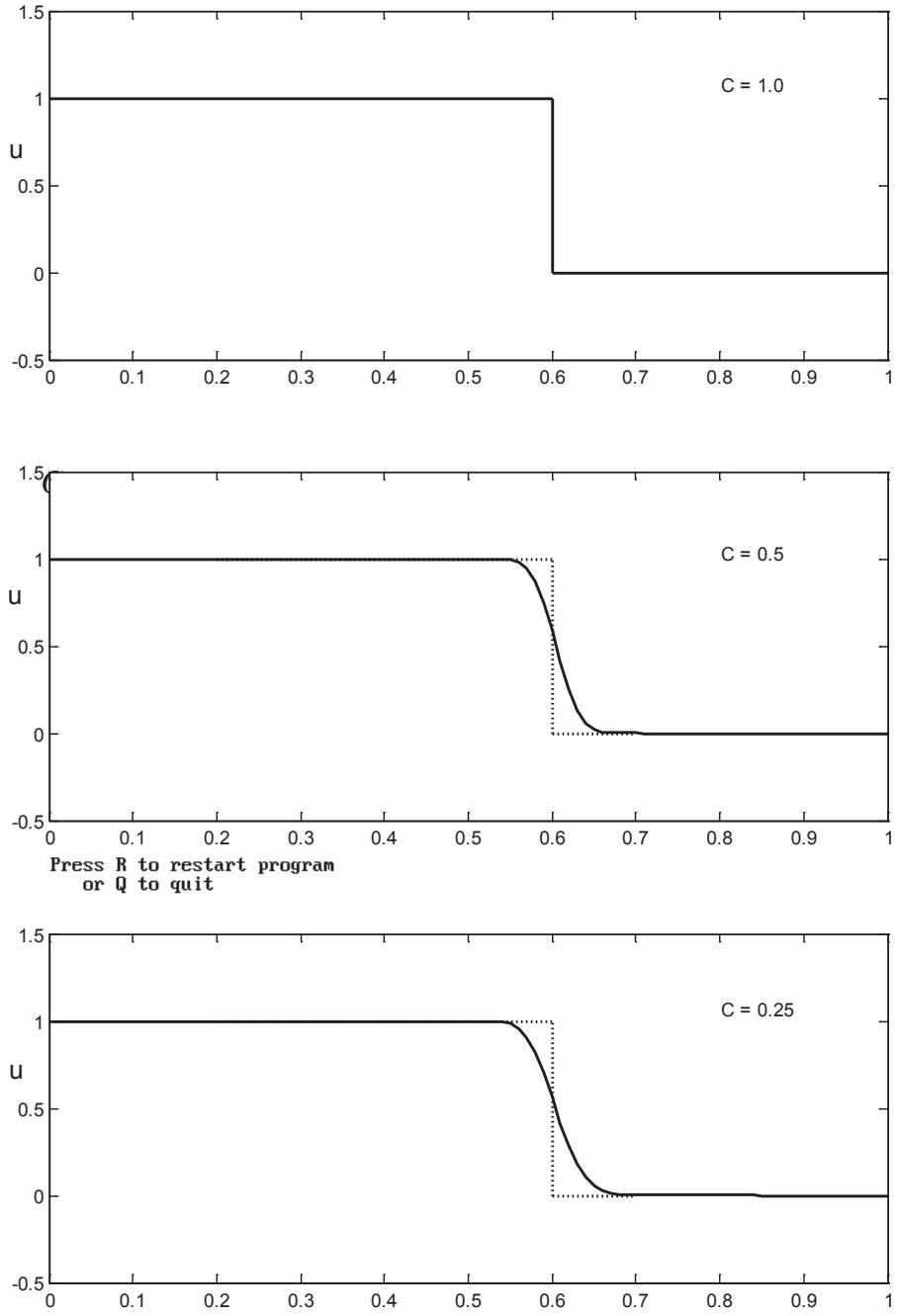

Figure 8.5: Numerical solution for the linear advection equation using the upwind scheme and different values of Courant-number C.

8.4 The modified differential equation

In Section 7.6 we have seen how to determine the local truncation error and rthe consistency of numerical schemes. We will now see how to use a similar method to gain understanding on the stability of the resulting scheme. We will restrcit the discussion to the linear advection equation

We recall the FTCS scheme from Section 8.2:

$$\frac{u_j^{n+1} - u_j^n}{k} + a_0 \frac{u_{j+1}^n - u_{j-1}^n}{2h} = 0 \tag{8.14}$$

Inserting in (8.14) the Taylor expansions introduced in Section 7.6, such as (7.84), we obtain:

$$\frac{1}{k}\left\{\left[u + ku_t + \tfrac{k^2}{2}u_{tt} + \cdots\right]\Big|_j^n - u_j^n\right\} + \frac{a_0}{2h}\left[u + hu_x + \tfrac{h^2}{2}u_{xx} + \tfrac{h^3}{6}u_{xxx} + \cdots\right]\Big|_j^n$$

$$-\frac{a_0}{2h}\left[u - hu_x + \tfrac{h^2}{2}u_{xx} - \tfrac{h^3}{6}u_{xxx} + \cdots\right]\Big|_j^n = 0$$

Reordering:

$$\frac{\partial u}{\partial t} + a_0\frac{\partial u}{\partial x} = -\frac{k}{2}\frac{\partial^2 u}{\partial t^2} - \frac{a_0 h^2}{6}\frac{\partial^3 u}{\partial x^3} + \cdots \tag{8.15}$$

We now use the differential equation $\frac{\partial u}{\partial t} + a_0\frac{\partial u}{\partial x} = 0$ to transform the term $-\frac{k}{2}\frac{\partial^2 u}{\partial t^2}$ in a spatial derivative.

The resulting differential equation is:

$$\frac{\partial u}{\partial t} = -a_0\frac{\partial u}{\partial x} \tag{8.16}$$

Deriving with respect to t:

$$\frac{\partial^2 u}{\partial t^2} = -a_0\frac{\partial^2 u}{\partial t \partial x}$$

and then we derive (8.16) with respect to x:

$$\frac{\partial^2 u}{\partial t \partial x} = -a_0\frac{\partial^2 u}{\partial x^2} \rightarrow -a_0\frac{\partial^2 u}{\partial t \partial x} = a_0^2\frac{\partial^2 u}{\partial x^2}$$

so that we obtain:

$$\frac{\partial^2 u}{\partial t^2} = a_0^2\frac{\partial^2 u}{\partial x^2} \tag{8.17}$$

Inserting (8.17) in (8.15):

$$\frac{\partial u}{\partial t} + a_0 \frac{\partial u}{\partial x} = -\frac{k a_0^2}{2} \frac{\partial^2 u}{\partial x^2} - \frac{a_0 h^2}{6} \frac{\partial^3 u}{\partial x^3} + \cdots$$

which, when introducing the Courant-number $C = \frac{a_0 k}{h}$ gives:

$$\frac{\partial u}{\partial t} + a_0 \frac{\partial u}{\partial x} = -\frac{C h a_0^2}{2} \frac{\partial^2 u}{\partial x^2} - \frac{a_0 h^2}{6} \frac{\partial^3 u}{\partial x^3} + \cdots \tag{8.18}$$

(8.18) clearly shows why the scheme in (8.14) is unstable. In computational fluid mechanics the term $\nu_N = -\frac{C h a_0}{2}$ is often called *numerical viscosity* in analogy to the physical viscosity that accompanies the diffusive term of advection-diffusion equations like $\frac{\partial u}{\partial t} + a_0 \partial u \partial x = \nu \frac{\partial^2 u}{\partial x^2}$. When using the scheme given in (8.14), this results in solving the advection-diffusion equation with a negative viscosity coefficient , which is certainly unstable. (8.18) provides information on why oscillations in Figure 8.3 decrease for decreasing C. Equation (8.18) is called the *modified equation* for a given numerical scheme, since it shows which is the differential problem that the scheme is actually solving and it therefore provides a deep insight in the behavior that the numerical solution will exhibit.

Now we will perform the same procedure as above but for the upwind scheme presented in Section 8.3. The scheme can be written as:

$$\frac{u_j^{n+1} - u_j^n}{k} + a_0 \frac{(u_j^n - u_{j-1}^n)}{h} = 0 \tag{8.19}$$

After introducing Taylor expansions as before and reordering we get:

$$\frac{\partial u}{\partial t} = a_0 \frac{\partial u}{\partial x} = -\frac{k}{2} \frac{\partial^2 u}{\partial t^2} + \frac{a_0 h}{2} \frac{\partial^2 u}{\partial x^2} + \cdots \tag{8.20}$$

Since the differential equation is the same as for the FTCS scheme, we use the same procedure to replace time derivative terms with spatial derivative ones, yielding:

$$\frac{\partial u}{\partial t} + a_0 \frac{\partial u}{\partial x} = \frac{a_0 h}{2}(1 - C) \frac{\partial^2 u}{\partial x^2} + \cdots \tag{8.21}$$

The coefficient $\nu_N = \frac{a_0 h}{2}(1 - C)$ is the so called numerical viscosity.

For $C > 1$ this term becomes negative and this in turn means that the scheme turns unstable. The necessary condition for stability is then $C \leq 1$, in agreement with what seen before in terms of stability analysis. Note that the numerical viscosity increases with decreasing C, meaning that sharp fronts will be smoothed more for lower Courant-numbers, as shown in Figure 8.5.

Remark: We have used the original differential equation for the derivation of both modified equations (8.18) og (8.21), i.e. $\frac{\partial^2 u}{\partial t^2} = a_0^2 \frac{\partial^2 u}{\partial x^2}$. This procedure is called the Cauchy-Kowalewsky procedure. The problem is that the modified equation does not generally corresponds to the original equation, as expected. This means that instead of deriving the original equation, we have to derive (8.15) and (8.20). This procedure is called the Warming-Hyett procedure and must be generally adopted. For the above made computations, where we stop

after one iteration of this procedure, both procedures are equivalent. For more complex differential equations the use of symbolic calculus programs such as Maple or Maxima becomes mandatory. The Maple-program below illustrate how to implement such procedure.

```
> restart:
> EQ:= (u(x,t+k)-u(x,t))/k + a*(u(x+h,t)-u(x-h,t))/(2*h):
> EQT:= mtaylor(EQ,[h,k]):
> MDE:=EQT:
> ELIM:=proc(i::integer,j::integer)
     local DE,uxt,UXT:
     global EQT,MDE:
     DE:=convert(diff(EQT,x$i,t$j-1,D):
     uxt:=convert(diff(u(x,t),x$i,t$j),D):
     UXT:=solve(DE = 0,uxt):
     subs(uxt = UXT,MDE):
 end:
> MDE:=ELIM(0,2):
> MDE:=ELIM(1,1):
> MDE:=ELIM(0,3):
> MDE:=ELIM(1,2):
> MDE:=ELIM(2,1):
> # Substitute the Courant number C = a*k/h
> MDE:=expand(subs(k=C*h/a,MDE)):
> u2x:=convert(diff(u(x,t),x$2),D):
> u3x:=convert(diff(u(x,t),x$3),D):
> collect(MDE,[u2x,u3x]):
> RHSMDE:=-coeff(MDE,u2x)*convert(u2x,diff)
          -coeff(MDE,u3x)*convert(u3x,diff);
```

The resulting right-hand side of the modified equation delivered by the program is

$$\text{RHSMDE:=} \; -\tfrac{1}{2}Cha\left(\tfrac{\partial^2}{\partial x^2}u(x,t)\right) - \left(\tfrac{1}{6}ah^2 + \tfrac{1}{3}C^2h^2a\right)\left(\tfrac{\partial^2}{\partial x^2}u(x,t)\right)$$

Note that in this case we have obtained the term $\frac{1}{3}C^2h^2a$, which was missing in (8.18). The technique of the modified equation can be also applied to non-linear equations. Moreover, it can be shown that the method relates to von Neumann stability method. Hirsch [7] and Anderson [14] use the modified equation to study fundamental properties of finite difference schemes.

8.5 Errors due to diffusion and dispersion

We have previously seen that the numerical amplification factor G is complex, i.e. it has a real part G_r and a imaginary part G_i. Being a complex number G may be represented in the two traditional ways:

$$G = G_r + i \cdot G_i = |G| \cdot e^{-i\phi} \tag{8.22}$$

where the latter is normally referred to as the polar form which provides a means to express G by the amplitude (or absolute value) $|G|$ and the phase ϕ.

$$|G| = \sqrt{G_r^2 + G_i^2}, \qquad \phi = \arctan\left(\frac{-G_i}{G_r}\right) \tag{8.23}$$

I section 7.4 we introduced the diffusion/dissipation error ε_D (dissipasjonsfeilen) as:

$$\varepsilon_D = \frac{|G|}{|G_a|} \tag{8.24}$$

where $|G_a|$ is the amplituden of analytical amplification factor G_a, i.e. we have no dissipative error when $\varepsilon_D = 1$.

For problems and models related with convection we also need to consider the error related with timing or *phase*, and introduce a measure for this kind error as the *dispersion error* ε_ϕ:

$$\varepsilon_\phi = \frac{\phi}{\phi_a} \tag{8.25}$$

where ϕ_a is the phase for the analytical amplification factor. And again $\varepsilon_\phi = 1$ corresponds to no dispersion error. Note for parabolic problems with $\phi_a = 0$, it is common to use $\varepsilon_\phi = \phi$.

8.5.1 Example: Advection equation

Consider the linear advection equation, given by:

$$\frac{\partial u}{\partial t} + a_0 \frac{\partial u}{\partial x} = 0$$

Similarly to what introduced in Section 7.4, we use

$$u(x,t) = e^{i(\beta x - \omega t)}, \tag{8.26}$$

with $\omega = a_0\beta$, where β is the wave number. With the notation introduced above we get:

$$u(x,t) = e^{i(\beta x - \omega t)} = e^{-i\cdot\phi_a} \ , \ \phi_a = \omega t - \beta x \tag{8.27}$$

From equation (7.39) in Section 7.4:

$$G_a = \frac{u(x_j, t_{n+1})}{u(x_j, t_n)} = \frac{e^{i(\beta x_j - \omega t_{n+1})}}{e^{i(\beta x_j - \omega t_n)}} = \exp[i(\beta x_j - \omega t_{n+1}) - i(\beta x_j - \omega t_n)] = \exp(-i\omega\cdot\Delta t)$$

where we have inserted (8.27). Thus, $|G_a| = 1 \rightarrow \varepsilon_D = |G|$.

Hence:

$$\phi_a = \omega \cdot \Delta t = a_0\beta \cdot \Delta t = a_0 \frac{\Delta t}{\Delta x}\beta \cdot \Delta x = C \cdot \delta \tag{8.28}$$

C is the Courant-number and $\delta = \beta \cdot \Delta x$ as usual.

From (8.28) we also get:

$$a_0 = \frac{\phi_a}{\beta \cdot \Delta t} \tag{8.29}$$

Analogously to (8.29), we can define a numerical propagation velocity a_{num}:

$$a_{num} = \frac{\phi}{\beta \cdot \Delta t} \tag{8.30}$$

The dispersion error in (8.25) can then be written as:

$$\varepsilon_\phi = \frac{a_{num}}{a_0} \tag{8.31}$$

When the dispersion error is greater than one we have that the numerical propagation speed is greater than the physical one. This results in the fact that the numerical solution will move faster than the physical one. On the other hand, a $\varepsilon_\phi < 1$ results in slower numerical propagation speed compared to the physical one, with obvious consequences for the numerical solution.

Let us know have a closer look to the upwind scheme and the Lax-Wendroff scheme.

8.5.2 Example: Diffusion and dispersion errors for the upwind schemes

From equation (8.12) in Section 8.3:

$$G = 1 + C \cdot (\cos(\delta) - 1) - i \cdot C \sin(\delta) = G_r + i \cdot G_i \tag{8.32}$$

which inserted in (8.23) and (8.25) gives:

$$\begin{aligned} \varepsilon_D = |G| \quad &= \sqrt{[1 + C \cdot (\cos(\delta) - 1)]^2 + [C \sin(\delta)]^2} \\ &= \sqrt{1 - 4C(1 - C)\sin^2(\tfrac{\delta}{2})} \end{aligned} \tag{8.33}$$

$$\varepsilon_\phi = \frac{\phi}{\phi_a} = \frac{1}{C\delta} \arctan\left[\frac{C \sin(\delta)}{1 - C(1 - \cos(\delta))}\right] \tag{8.34}$$

Figure 8.6 shows (8.33) and (8.34) as a function of δ for different Courant-number values. We see that ε_D decreases strongly for larger frequencies, which means that the numerical amplitude becomes much smaller than the exact one when use a large time step. This applies even for $C = 0.8$. Remember that the amplitude factor for the n-th time iteration is $|G|^n$ (see also Figure 8.5). The scheme is performance is rather modest for general use even though it is stable.

For $C = 0.5$ we have no dispersion errors. For $C < 0.5$ we have that $\varepsilon_\phi < 1$, so that the numerical propagation speed is smaller than the physical one a_0. On the other hand, for $C > 0.5$ we have that $\varepsilon_\phi > 1$.

Lucas 10: axes labels are wrong, also legend is not consistent with text for denoting the Courant-number

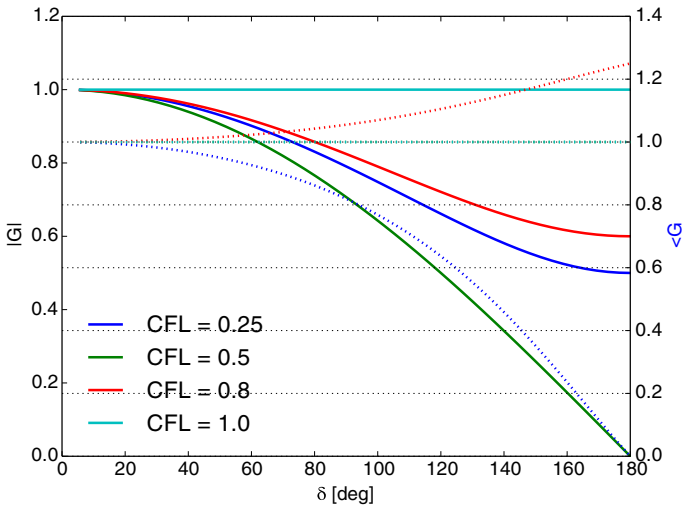

Figure 8.6: Diffusion error ε_D (left y-axis) and dispersion error ε_ϕ (right y-axis) for the upwind scheme as a function of frequency for the upwind scheme. The dotted lines of ε_ϕ correspond to same CFL-numbers as solid lines of ε_D with the same color.

8.5.3 Example: Diffusion and disperision errors for the Lax-Wendroff scheme

From equation (8.47):

$$G = 1 + C^2 \cdot (\cos(\delta) - 1) - i \cdot C \sin(\delta) = G_r + i \cdot G_i \tag{8.35}$$

which inserted in (8.23) and (8.25) gives:

$$\begin{aligned} \varepsilon_D = |G| \quad &= \sqrt{[1 + C^2 \cdot (\cos(\delta) - 1)]^2 + (C \sin(\delta))^2} \\ &= \sqrt{1 - 4C^2(1 - C^2) \sin^4\left(\tfrac{\delta}{2}\right)} \end{aligned} \tag{8.36}$$

$$\varepsilon_\phi = \frac{\phi}{\phi_a} = \frac{1}{C\delta} \arctan\left[\frac{C sin(\delta)}{1 + C^2(cos(\delta) - 1)}\right] \tag{8.37}$$

Figures 8.7 and 8.9 show (8.36) and (8.37) as a function of δ for different values of the Courant-number. The range for which ε_D is close to 1 is larger than in Figure 8.7 for the upwind scheme. This puts in evidence an important difference between first and second order schemes Figure 8.8 shows that ε_ϕ is generally less than 1, so that the numerical propagation speed is smaller than the physical one. This is the reason for the occurrence of oscillations as the ones shown in animation 7. A von Neumann criterion guarantees stability for $|G| \leq 1$,

but does not guarantees that the scheme is monotone. On the other hand, the PC criterion ensures that no oscillations can occur. In fact, this criterion can not be verified for the Lax-Wendroff scheme applied to the linear advection equation. In fact, monotone schemes for the linear advection equation can be at most first order accurate.

Lucas 11: we could say something about Godunov's theorem here! **Lucas 10**: axes labels are wrong, also legend is not consistent with text for denoting the Courant-number

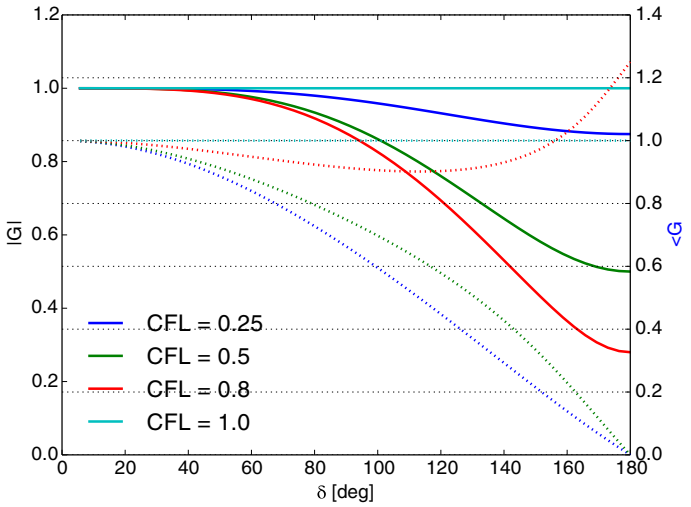

Figure 8.7: Diffusion error ε_D (left y-axis) and dispersion error ε_ϕ (right y-axis) as a function of frequency for the Lax-Wendroff scheme. The dotted lines of ε_ϕ correspond to same CFL-numbers as solid lines of ε_D with the same color.

Marit 2: Har ikke endret figuren

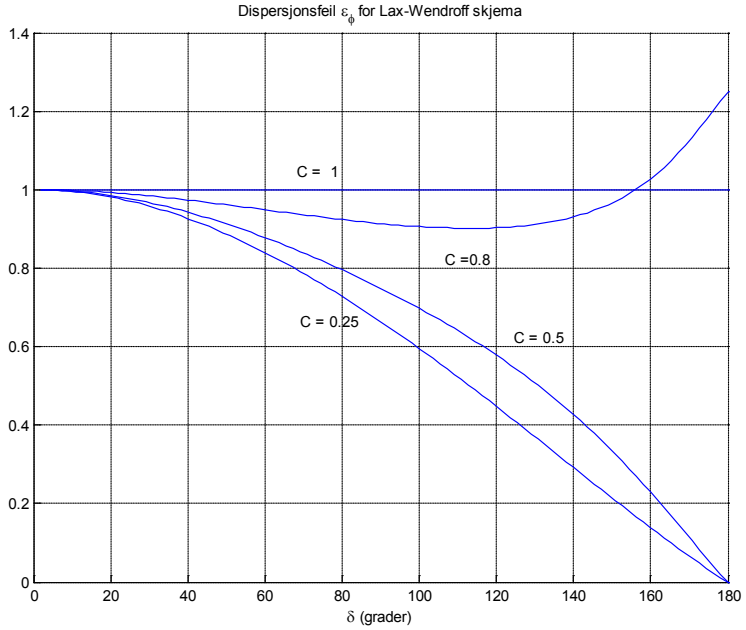

Figure 8.8: Dispersion error ε_ϕ as a function of frequency for the Lax-Wendroff scheme.

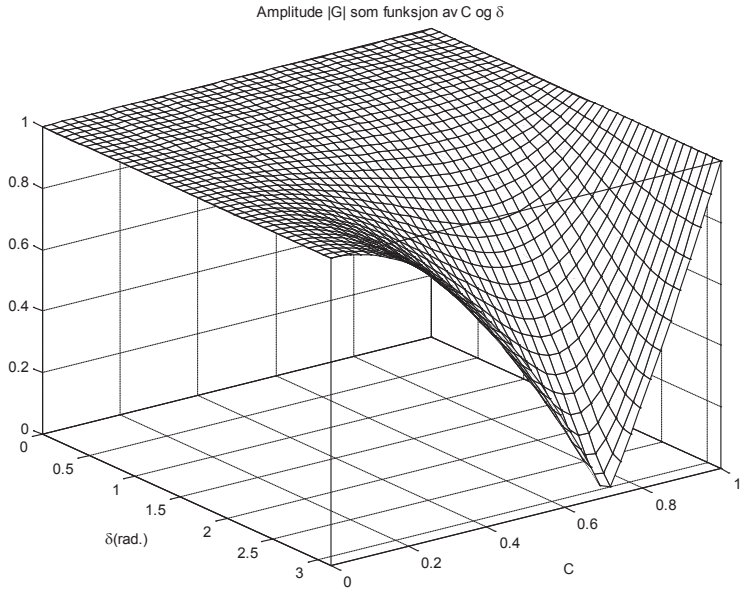

Figure 8.9: Diffusion error ε_D as a function of frequency and Courant-number for the Lax-Wendroff scheme.

8.6 The Lax-Friedrich Scheme

Lax-Friedrichs scheme is an explicit, first order scheme, using forward difference in time and central difference in space. However, the scheme is stabilized by averaging u_j^n over the neighbour cells in the in the temporal approximation:

$$\frac{u_j^{n+1} - \frac{1}{2}(u_{j+1}^n + u_{j-1}^n)}{\Delta t} = -\frac{F_{j+1}^n - F_{j-1}^n}{2\Delta x} \tag{8.38}$$

The Lax-Friedrich scheme is the obtained by isolation u_j^{n+1} at the right hand side:

$$u_j^{n+1} = \frac{1}{2}(u_{j+1}^n + u_{j-1}^n) - \frac{\Delta t}{2\Delta x}(F_{j+1}^n - F_{j-1}^n) \tag{8.39}$$

By assuming a linear flux $F = a_0\, u$ it may be shown that the Lax-Friedrich scheme takes the form:

$$u_j^{n+1} = \frac{1}{2}(u_{j+1}^n + u_{j-1}^n) - \frac{C}{2}(u_{j+1}^n - u_{j-1}^n) \tag{8.40}$$

where we have introduced the CFL-number as given by (8.5) and have the simple python-implementation:

```
def lax_friedrich(u):
    u[1:-1] = (u[:-2] +u[2:])/2.0 -  c*(u[2:] - u[:-2])/2.0
    return u[1:-1]
```

whereas a more generic flux implementation is implemented as:

```
def lax_friedrich_Flux(u):
    u[1:-1] = (u[:-2] +u[2:])/2.0 -  dt*(F(u[2:])-F(u[:-2]))/(2.0*dx)
    return u[1:-1]
```

8.7 Lax-Wendroff Schemes

These schemes were proposed in 1960 by P.D. Lax and B. Wendroff [11] for solving, approximately, systems of hyperbolic conservation laws on the generic form given in (8.3).

A large class of numerical methods for solving (8.3) are the so-called conservative methods:

$$u_j^{n+1} = u_j^n + \frac{\Delta t}{\Delta x}\left(F_{j-1/2} - F_{j+1/2}\right) \tag{8.41}$$

Linear advection. The Lax–Wendroff method belongs to the class of conservative schemes (8.3) and can be derived in various ways. For simplicity, we will derive the method by using a simple model equation for (8.3), namely the linear advection equation with $F(u) = a\, u$ as in (8.1), where a is a constant

propagation velocity. The Lax-Wendroff outset is a Taylor approximation of u_j^{n+1}:

$$u_j^{n+1} = u_j^n + \Delta t \frac{\partial u}{\partial t}\bigg|_j^n + \frac{(\Delta t)}{2}\frac{\partial^2 u}{\partial t^2}\bigg|_j^n + \cdots \tag{8.42}$$

From the differential equation (8.3) we get by differentiation

$$\frac{\partial u}{\partial t}\bigg|_j^n = -a_0 \frac{\partial u}{\partial x}\bigg|_j^n \qquad \text{and} \qquad \frac{\partial^2 u}{\partial t^2}\bigg|_j^n = a_0^2 \frac{\partial^2 u}{\partial x^2}\bigg|_j^n \tag{8.43}$$

Before substitution of (8.43) in the Taylor expansion (8.42) we approximate the spatial derivatives by central differences:

$$\frac{\partial u}{\partial x}\bigg|_j^n \approx \frac{u_{j+1}^n - u_{j-1}^n}{(2\Delta x)} \qquad \text{and} \qquad \frac{\partial^2 u}{\partial x^2}\bigg|_j^n \approx \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2} \tag{8.44}$$

and then the Lax-Wendroff scheme follows by substitution:

$$u_j^{n+1} = u_j^n - \frac{C}{2}\left(u_{j+1}^n - u_{j-1}^n\right) + \frac{C^2}{2}\left(u_{j+1}^n - 2u_j^n + u_{j-1}^n\right) \tag{8.45}$$

with the local truncation error T_j^n:

$$T_j^n = \frac{1}{6} \cdot \left[(\Delta t)^2 \frac{\partial^3 u}{\partial t^3} + a_0 (\Delta x)^2 \frac{\partial^3 u}{\partial x^3}\right]_j^n = O[(\Delta t)^2, (\Delta x)^2] \tag{8.46}$$

The resulting difference equation in (8.45) may also be formulated as:

$$u_j^{n+1} = \frac{C}{2}(1+C)u_{j-1}^n + (1-C^2)u_j^n - \frac{C}{2}(1-C)u_{j+1}^n \tag{8.47}$$

The explicit Lax-Wendroff stenticl is illustrated in Figure 8.10

An example of how to implement the Lax-Wendroff scheme is given as follows:

```
def lax_wendroff(u):
    u[1:-1] = c/2.0*(1+c)*u[:-2] + (1-c**2)*u[1:-1] - c/2.0*(1-c)*u[2:]
    return u[1:-1]
```

8.7.1 Lax-Wendroff for non-linear systems of hyperbolic PDEs

For non-linear equations (8.3) the Lax–Wendroff method is no longer unique and naturally various methods have been suggested. The challenge for a non-linear $F(u)$ is that the substitution of temporal derivatives with spatial derivatives (as we did in (8.43)) is not straightforward and unique.

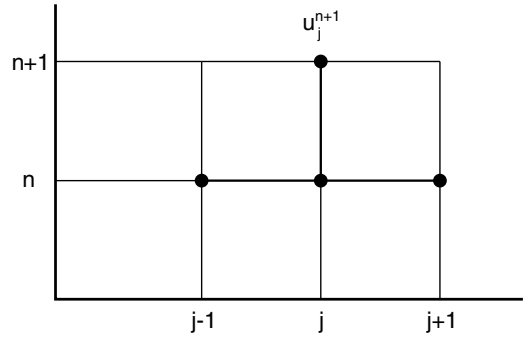

Figure 8.10: Schematic of the Lax-Wendroff scheme.

Ricthmyer Scheme. One of the earliest extensions of the scheme is the Richtmyer two-step Lax–Wendroff method, which is on the conservative form (8.41) with the numerical fluxes computed as follows:

$$u_{j+1/2}^{n+1/2} = \frac{1}{2}\left(u_j^n + u_{j+1}^n\right) + \frac{1}{2}\frac{\Delta t}{\Delta x}\left(F_j^n - F_{j+1}^n\right) \tag{8.48}$$

$$F_{j+1/2} = F(u_{j+1/2}^{n+1/2}) \tag{8.49}$$

Lax-Wendroff two step. A Lax-Wendroff two step method is outlined in the following. In the first step $u(x,t)$ is evaluated at half time steps $n+1/2$ and half grid points $j+1/2$. In the second step values at the next time step $n+1$ are calculated using the data for n and $n+1/2$.

First step:

$$u_{j+1/2}^{n+1/2} = \frac{1}{2}\left(u_{j+1}^n + u_j^n\right) - \frac{\Delta t}{2\Delta x}\left(F(u_{j+1}^n) - F(u_j^n)\right) \tag{8.50}$$

$$u_{j-1/2}^{n+1/2} = \frac{1}{2}\left(u_j^n + u_{j-1}^n\right) - \frac{\Delta t}{2\Delta x}\left(F(u_j^n) - F(u_{j-1}^n)\right) \tag{8.51}$$

Second step:

$$u_j^{n+1} = u_j^n - \frac{\Delta t}{\Delta x}\left(F(u_{j+1/2}^{n+1/2}) - F(u_{j-1/2}^{n+1/2})\right) \tag{8.52}$$

Notice that for a linear flux $F = a_0\, u$, the two-step Lax-Wendroff method ((8.51) and (8.52)) may be shown to reduce to the one-step Lax-Wendroff method outlined in (8.45) or (8.47).

MacCormack Scheme. A simpler and popular extension/variant of Lax-Wendroff schemes like in the previous section, is the MacCormack scheme [13]:

$$u_j^p = u_j^n + \frac{\Delta t}{\Delta x}\left(F_j^n - F_{j+1}^n\right)$$
$$u_j^{n+1} = \frac{1}{2}\left(u_j^n + u_j^p\right) + \frac{1}{2}\frac{\Delta t}{\Delta x}\left(F_{j-1}^p - F_j^p\right) \tag{8.53}$$

where we have introduced the convention $F_j^p = F(u_j^p)$.

Note that in the predictor step we employ the conservative formula (8.41) for a time Δt with forward differencing, i.e. . $F_{j+1/2} = F_{j+1}^n = F(u_{j+1}^n)$. The corrector step may be interpreted as using (8.41) for a time $\Delta t/2$ with initial condition $\frac{1}{2}\left(u_j^n + u_{j+1}^p\right)$ and backward differencing.

Another MacCormack scheme may be obtained by reversing the predictor and corrector steps. Note that the MacCormack scheme (8.53) is not written in conservative form (8.41). However, it easy to express the scheme in conservative form by expressing the flux in (8.41) as:

$$F_{j+1}^m = \frac{1}{2}\left(F_j^p + F_{j+1}^n\right) \tag{8.54}$$

For a linear flux $F(u) = a_0\, u$, one may show that the MacCormack scheme in (8.53) reduces to a two-step scheme:

$$u_j^p = u_j^n + C\left(u_j^n - u_{j+1}^n\right) \tag{8.55}$$
$$u_j^{n+1} = \frac{1}{2}\left(u_j^n + u_j^p\right) + \frac{C}{2}\left(u_{j-1}^p - u_j^p\right) \tag{8.56}$$

and substitution of (8.55) into (8.56) shows that the MacCormack scheme is identical to the Lax-Wendroff scheme (8.47) for the linear advection flux. A python implementation is given by:

```
def macCormack(u):
    up = u.copy()
    up[:-1] = u[:-1] - c*(u[1:]-u[:-1])
    u[1:] = .5*(u[1:]+up[1:] -  c*(up[1:]-up[:-1]))
    return u[1:-1]


# Constants and parameters
a = 1.0 # wave speed
tmin, tmax = 0.0, 1.0 # start and stop time of simulation
xmin, xmax = 0.0, 2.0 # start and end of spatial domain
Nx = 80 # number of spatial points
c = 0.9 # courant number, need c<=1 for stability
```

8.7.2 Code example for various schemes for the advection equation

A complete example showing how a range of hyperbolic schemes are implemented and applied to a particular example:

```
# src-ch6/advection_schemes.py

import numpy as np
from matplotlib import animation
from scipy import interpolate
from numpy import where
from math import sin

import matplotlib; matplotlib.use('Qt4Agg')
import matplotlib.pylab as plt
plt.get_current_fig_manager().window.raise_()


LNWDT=2; FNT=15
plt.rcParams['lines.linewidth'] = LNWDT; plt.rcParams['font.size'] = FNT


init_func=1   # Select stair case function (0) or sin^2 function (1)

# function defining the initial condition
if (init_func==0):
    def f(x):
        """Assigning a value of 1.0 for values less than 0.1"""
        f = np.zeros_like(x)
        f[np.where(x <= 0.1)] = 1.0
        return f
elif(init_func==1):
    def f(x):
        """A smooth sin^2 function between x_left and x_right"""
        f = np.zeros_like(x)
        x_left = 0.25
        x_right = 0.75
        xm = (x_right-x_left)/2.0
        f = where((x>x_left) & (x<x_right), np.sin(np.pi*(x-x_left)/(x_right-x_left))**4,f)
        return f

def ftbs(u): # forward time backward space
    u[1:-1] = (1-c)*u[1:-1] + c*u[:-2]
    return u[1:-1]

# Lax-Wendroff
def lax_wendroff(u):
    u[1:-1] = c/2.0*(1+c)*u[:-2] + (1-c**2)*u[1:-1] - c/2.0*(1-c)*u[2:]
    return u[1:-1]

# Lax-Friedrich Flux formulation
def lax_friedrich_Flux(u):
    u[1:-1] = (u[:-2] +u[2:])/2.0 -  dt*(F(u[2:])-F(u[:-2]))/(2.0*dx)
    return u[1:-1]

# Lax-Friedrich Advection
def lax_friedrich(u):
    u[1:-1] = (u[:-2] +u[2:])/2.0 -  c*(u[2:] - u[:-2])/2.0
    return u[1:-1]

# macCormack for advection quation
def macCormack(u):
    up = u.copy()
    up[:-1] = u[:-1] - c*(u[1:]-u[:-1])
    u[1:] = .5*(u[1:]+up[1:] -  c*(up[1:]-up[:-1]))
    return u[1:-1]
```

```
# Constants and parameters
a = 1.0 # wave speed
tmin, tmax = 0.0, 1.0 # start and stop time of simulation
xmin, xmax = 0.0, 2.0 # start and end of spatial domain
Nx = 80 # number of spatial points
c = 0.9 # courant number, need c<=1 for stability


# Discretize
x = np.linspace(xmin, xmax, Nx+1) # discretization of space
dx = float((xmax-xmin)/Nx) # spatial step size
dt = c/a*dx # stable time step calculated from stability requirement
Nt = int((tmax-tmin)/dt) # number of time steps
time = np.linspace(tmin, tmax, Nt) # discretization of time

# solve from tmin to tmax

solvers = [ftbs,lax_wendroff,lax_friedrich,macCormack]
#solvers = [ftbs,lax_wendroff,macCormack]
#solvers = [ftbs,lax_wendroff]
#solvers = [ftbs]

u_solutions=np.zeros((len(solvers),len(time),len(x)))
uanalytical = np.zeros((len(time), len(x))) # holds the analytical solution



for k, solver in enumerate(solvers): # Solve for all solvers in list
    u = f(x)
    un = np.zeros((len(time), len(x))) # holds the numerical solution

    for i, t in enumerate(time[1:]):

        if k==0:
            uanalytical[i,:] = f(x-a*t) # compute analytical solution for this time step

        u_bc = interpolate.interp1d(x[-2:], u[-2:]) # interplate at right bndry

        u[1:-1] = solver(u[:]) # calculate numerical solution of interior
        u[-1] = u_bc(x[-1] - a*dt) # interpolate along a characteristic to find the boundary value

        un[i,:] = u[:] # storing the solution for plotting

    u_solutions[k,:,:] = un



### Animation

# First set up the figure, the axis, and the plot element we want to animate
fig = plt.figure()
ax = plt.axes(xlim=(xmin,xmax), ylim=(np.min(un), np.max(un)*1.1))

lines=[]     # list for plot lines for solvers and analytical solutions
legends=[]   # list for legends for solvers and analytical solutions

for solver in solvers:
    line, = ax.plot([], [])
    lines.append(line)
```

```
    legends.append(solver.func_name)

line, = ax.plot([], []) #add extra plot line for analytical solution
lines.append(line)
legends.append('Analytical')

plt.xlabel('x-coordinate [-]')
plt.ylabel('Amplitude [-]')
plt.legend(legends, loc=3, frameon=False)

# initialization function: plot the background of each frame
def init():
    for line in lines:
        line.set_data([], [])
    return lines,

# animation function.  This is called sequentially
def animate(i):
    for k, line in enumerate(lines):
        if (k==0):
            line.set_data(x, un[i,:])
        else:
            line.set_data(x, uanalytical[i,:])
    return lines,

def animate_alt(i):
    for k, line in enumerate(lines):
        if (k==len(lines)-1):
            line.set_data(x, uanalytical[i,:])
        else:
            line.set_data(x, u_solutions[k,i,:])
    return lines,


# call the animator.  blit=True means only re-draw the parts that have changed.
anim = animation.FuncAnimation(fig, animate_alt, init_func=init, frames=Nt, interval=100, blit=False)


plt.show()
```

Movie 7: Result from code example above using a step function as the initial value. `mov-ch6/step.mp4`

Movie 8: Result from code example above using a sine squared function as the initial value `mov-ch6/sine.mp4`

8.8 Order analysis on various schemes for the advection equation

The schemes presented have different theoretical order; **ftbs**: $\mathcal{O}(\Delta x, \Delta t)$, **Lax-Friedrich**: $\mathcal{O}(\Delta x^2, \Delta t)$, **Lax-Wendroff**: $\mathcal{O}(\Delta x^2, \Delta t^2)$, **MacCormack**: $\mathcal{O}(\Delta x^2, \Delta t^2)$. For the linear advection equation the MacCormack and Lax-Wendroff schemes are identical, and we will thus only consider Lax-Wendroff in this section. Assuming the wavespeed a_0 is not very big, nor very small we will have $\Delta t = \mathcal{O}(\Delta x)$, because of the cfl constraint condition. Thus for the advection schemes the

discretization error ϵ will be of order $\min(p, q)$. Where p is the spatial order, and q the temporal order. Thus we could say that ftbs, and Lax-Friedrich are both first order, and Lax-Wendroff is of second order. In order to determine the observed (dominating) order of accuracy of our schemes we could adapt the same procedure outlined in Example 2.8.1, where we determined the observed order of accuracy of ODEschemes. For the schemes solving the Advection equation the discretization error will be a combination of Δx and Δt, however since $\Delta t = \mathcal{O}(\Delta x)$, we may still assume the following expression for the discretization error:

$$\epsilon \approx Ch^p \tag{8.57}$$

where h is either Δx or Δt, and p is the dominating order. Further we can calculate the discretization error, observed for our schemes by successively refining h with a ratio r, keeping the cfl-number constant. Thus refining Δx and Δt with the same ratio r. Now dividing ϵ_{n-1} by ϵ_n and taking the log on both sides and rearranging lead to the following expression for the observed order of accuracy:

$$p = \frac{log\left(\frac{\epsilon_{n-1}}{\epsilon_n}\right)}{log\,(r)} = log_r\left(\frac{\epsilon_{n-1}}{\epsilon_n}\right)$$

To determine the observed discretization error we will use the root mean square error of all our discrete soultions:

$$E = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\hat{f} - f\right)^2}$$

where N is the number of sampling points, $\hat{f}$ is the numerical-, and f is the analytical solution. A code example performing this test on selected advections schemes, is showed below. As can be seen in Figure 8.11, the Lax-Wendroff scheme quickly converge to it's theoretical order, whereas the ftbs and Lax Friedrich scheme converges to their theoretical order more slowly.

```
def test_convergence_MES():
    from numpy import log
    from math import sqrt
    global c, dt, dx, a

    # Change default values on plots
    LNWDT=2; FNT=13
    plt.rcParams['lines.linewidth'] = LNWDT; plt.rcParams['font.size'] = FNT

    a = 1.0 # wave speed
    tmin, tmax = 0.0, 1 # start and stop time of simulation
    xmin, xmax = 0.0, 2.0 # start and end of spatial domain
    Nx = 160 # number of spatial points
    c = 0.8 # courant number, need c<=1 for stability

    # Discretize
    x = np.linspace(xmin, xmax, Nx + 1) # discretization of space
```

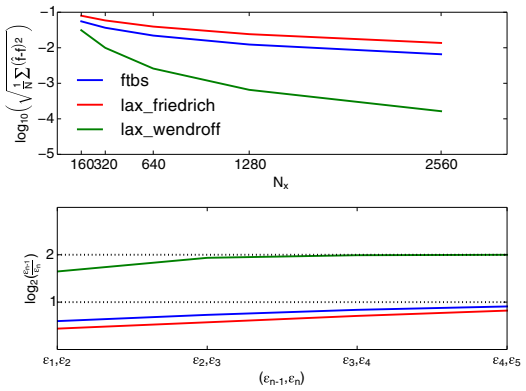

Figure 8.11: The root mean square error E for the various advection schemes as a function of the number of spatial nodes (top), and corresponding observed convergence rates (bottom).

```
dx = float((xmax-xmin)/Nx) # spatial step size
dt = c/a*dx # stable time step calculated from stability requirement
time = np.arange(tmin, tmax + dt, dt) # discretization of time

init_funcs = [init_step, init_sine4] # Select stair case function (0) or sin^4 function (1)
f = init_funcs[1]

solvers = [ftbs, lax_friedrich, lax_wendroff]

errorDict = {} # empty dictionary to be filled in with lists of errors
orderDict = {}

for solver in solvers:
    errorDict[solver.func_name] = [] # for each  solver(key) assign it with value=[], an emp
    orderDict[solver.func_name] = []

hx = [] # empty list of spatial step-length
ht = [] # empty list of temporal step-length

Ntds = 5 # number of grid/dt refinements
# iterate Ntds times:
for n in range(Ntds):
    hx.append(dx)
    ht.append(dt)

    for k, solver in enumerate(solvers): # Solve for all solvers in list
        u = f(x) # initial value of u is init_func(x)
        error = 0
        samplePoints = 0

        for i, t in enumerate(time[1:]):
            u_bc = interpolate.interp1d(x[-2:], u[-2:]) # interplate at right bndry
            u[1:-1] = solver(u[:]) # calculate numerical solution of interior
            u[-1] = u_bc(x[-1] - a*dt) # interpolate along a characteristic to find the boun

            error += np.sum((u - f(x-a*t))**2) # add error from this timestep
```

```
                samplePoints += len(u)

            error = sqrt(error/samplePoints) # calculate rms-error
            errorDict[solver.func_name].append(error)

            if n>0:
                previousError = errorDict[solver.func_name][n-1]
                orderDict[solver.func_name].append(log(previousError/error)/log(2))

        print " finished iteration {0} of {1}, dx = {2}, dt = {3}, tsample = {4}".format(n+1, Nto
        # refine grid and dt:
        Nx *= 2
        x = np.linspace(xmin, xmax, Nx+1) # new x-array, twice as big as the previous
        dx = float((xmax-xmin)/Nx) # new spatial step size, half the value of the previous
        dt = c/a*dx # new stable time step
        time = np.arange(tmin, tmax + dt, dt) # discretization of time

    # Plot error-values and corresponding order-estimates:
    fig , axarr = plt.subplots(2, 1, squeeze=False)
    lstyle = ['b', 'r', 'g', 'm']
    legendList = []

    N = Nx/2**(Ntds + 1)
    N_list = [N*2**i for i in range(1, Ntds+1)]
    N_list = np.asarray(N_list)

    epsilonN = [i for i in range(1, Ntds)]
    epsilonlist = ['$\epsilon_{0} , \epsilon_{1}$'.format(str(i), str(i+1)) for i in range(1, Nto

    for k, solver in enumerate(solvers):
        axarr[0][0].plot(N_list, np.log10(np.asarray(errorDict[solver.func_name])),lstyle[k])
        axarr[1][0].plot(epsilonN, orderDict[solver.func_name],lstyle[k])

        legendList.append(solver.func_name)



    axarr[1][0].axhline(1.0, xmin=0, xmax=Ntds-2, linestyle=':', color='k')
    axarr[1][0].axhline(2.0, xmin=0, xmax=Ntds-2, linestyle=':', color='k')
```

8.8.1 Separating spatial and temporal discretization error

The test performed in the previous example verify the dominating order of accuracy of the advection schemes. However in order to verify our implementations of the various schemes we would like to ensure that both the observed temporal- and spatial order are close to the theoretical orders. The expression for the discretization error in (8.57) is a simplification of the more general expression

$$\epsilon \approx C_x\, h_x^p + C_t\, h_t^q \tag{8.58}$$

where C_x, and C_t are constants, h_x and h_t are the spatial- and temporal step lengths and p and q are the spatial- and temporal orders respectively. We are interested in confirming that p is close to the theoretical spatial order of the scheme, and that q is close to the theoretical temporal order of the scheme. The method of doing this is not necessarily straight forward, especially since h_t and h_x are coupled through the cfl-constraint condition. In chapter one we showed that

we were able to verify C and p in the case of only one step-length dependency (time or space), by solving two nonlinear equations for two unknowns using a Newton-Rhapson solver. To expand this method would now involve solving four nonlinear algebraic equations for the four unknowns C_x, C_t, p, q. However since it is unlikely that the observed discretization error match the expression in (8.58) exactly, we now sugest a method based on optimization and curve-fitting. From the previous code example we calculated the root mean square error $E(h_x, h_t)$ of the schemes by succecively refining hx and ht by a ratio r. We now assume that E is related to C_x, C_t, p, q as in (8.58). In other words we would like to find the parameters C_x, C_t, p, q, so that the difference between our caluclated errors $E(hx, ht)$, and the function $\epsilon\left(h_x, h_t; C_x, p, C_t, q\right) = C_x\, h_x^p + C_t\, h_t^q$ is as small as possible. This may be done by minimizing the sum (S) of squared residuals of E and ϵ:

$$S = \sum_{i=1}^{N} r_i^2, \quad r_i = E_i - \epsilon\left(h_x, h_t; C_x, p, C_t, q\right)_i$$

The python module scipy.optimize has many methods for parameter optimization and curve-fitting. In the code example below we use `scipy.optimize.curve_fit` which fits a function "f(x;params)" to a set of data "y-data" using a Levenberg-Marquardt algorithmwith a least square minimization criteria. In the code example below we start by loading the calculated root mean square errors $E\left(h_x, h_t\right)$ of the schemes from "`advection_scheme_errors.txt`", which where calculated in the same manner as in the previous example. As can be seen by Figure 8.11 the ftbs, and Lax Friedriech scheme takes a while before they are in their asymptotic range (area where they converge at a constant rate). In "`advection_scheme_errors.txt`" we have computed $E\left(h_x, h_t\right)$ up to $N_x = 89120$ in which all schemes should be close to their asymptotic range. This procedure is demonstrated in the code example below in which the following expressions for the errors are obtained:

$$ftbs \rightarrow \epsilon = 1.3\, h_x^{0.98} + 6.5\, h_t^{0.98} \tag{8.59}$$

$$laxFriedrich \rightarrow \epsilon = -1484\, h_x^{1.9} + 26\, h_t^{1.0} \tag{8.60}$$

$$laxWendroff \rightarrow \epsilon = -148\, h_x^{2.0} + 364.\, h_t^{2.0} \tag{8.61}$$

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
from sympy import symbols, lambdify, latex

def optimize_error_cxct(errorList, hxList,
                        htList, p=1.0, q=1.0):
    """ Function that optimimze the values Cx and Ct in the expression
        E = epsilon = Cx hx^p + Ct ht^q, assuming p and q are known,
        using scipy's optimization tool curve_fit

        Args:
            errorList(array): array of calculated numerical discretization errors E
            hxList(array): array of spatial step lengths corresponding to errorList
```

```
            htList(array): array of temporal step lengths corresponding to errorList
            p (Optional[float]): spatial order. Assumed to be equal to theoretical value
            q (Optional[float]): temporal order. Assumed to be equal to theoretical value

        Returns:
            Cx0(float): the optimized values of Cx
            Ct0(float): the optimized values of Ct
    """

    def func(h, Cx, Ct):
        """ function to be matched with ydata:
            The function has to be on the form func(x, parameter1, parameter2,...,parametern)
            where where x is the independent variable(s), and parameter1-n are the parameters to be
        """
        return Cx*h[0]**p + Ct*h[1]**q


    x0 = np.array([1,2]) # initial guessed values for Cx and Ct
    xdata = np.array([hxList, htList])# independent variables
    ydata = errorList # data to be matched with expression in func
    Cx0, Ct0 = curve_fit(func, xdata, ydata, x0)[0] # call scipy optimization tool curvefit

    return Cx0, Ct0

def optimize_error(errorList, hxList, htList,
                   Cx0, p0, Ct0, q0):
    """ Function that optimimze the values Cx, p Ct and q in the expression
        E = epsilon = Cx hx^p + Ct ht^q, assuming p and q are known,
        using scipy's optimization tool curve_fit

        Args:
            errorList(array): array of calculated numerical discretization errors E
            hxList(array): array of spatial step lengths corresponding to errorList
            htList(array): array of temporal step lengths corresponding to errorList
            Cx0 (float): initial guessed value of Cx
            p (float): initial guessed value of p
            Ct0 (float): initial guessed value of Ct
            q (float): initial guessed value of q

        Returns:
            Cx(float): the optimized values of Cx
            p (float): the optimized values of p
            Ct(float): the optimized values of Ct
            q (float): the optimized values of q
    """


    def func(h, gx, p, gt, q):
        """ function to be matched with ydata:
            The function has to be on the form func(x, parameter1, parameter2,...,parametern)
            where where x is the independent variable(s), and parameter1-n are the parameters to be
        """
        return gx*h[0]**p + gt*h[1]**q


    x0 = np.array([Cx0, p0, Ct0, q0]) # initial guessed values for Cx, p, Ct and q
    xdata = np.array([hxList, htList]) # independent variables
    ydata = errorList # data to be matched with expression in func

    gx, p, gt, q = curve_fit(func, xdata, ydata, x0)[0] # call scipy optimization tool curvefit
    gx, p, gt, q = round(gx,2), round(p, 2), round(gt,2), round(q, 2)
```

```
    return gx, p, gt, q

# Program starts here:

# empty lists, to be filled in with values from 'advection_scheme_errors.txt'
hxList = []
htList = []

E_ftbs = []
E_lax_friedrich = []
E_lax_wendroff = []

lineNumber = 1

with open('advection_scheme_errors.txt', 'r') as FILENAME:
    """ Open advection_scheme_errors.txt for reading.
        structure of file:
        hx    ht    E_ftbs    E_lax_friedrich    E_lax_wendroff

        with the first line containing these headers, and the next lines containing
        the corresponding values.
    """
    # iterate all lines in FILENAME:
    for line in FILENAME:
        if lineNumber ==1:
            # skip first line which contain headers
            lineNumber += 1
        else:
            lineList = line.split() # sort each line in a list: lineList = [hx, ht, E_ftbs, E_lax_fr

            # add values from this line to the lists
            hxList.append(float(lineList[0]))
            htList.append(float(lineList[1]))

            E_ftbs.append(float(lineList[2]))
            E_lax_friedrich.append(float(lineList[3]))
            E_lax_wendroff.append(float(lineList[4]))

            lineNumber += 1

FILENAME.close()

# convert lists to numpy arrays:
hxList = np.asarray(hxList)
htList = np.asarray(htList)



E_ftbs = np.asarray(E_ftbs)
E_lax_friedrich = np.asarray(E_lax_friedrich)
E_lax_wendroff = np.asarray(E_lax_wendroff)


ErrorList = [E_ftbs, E_lax_friedrich, E_lax_wendroff]
schemes = ['ftbs', 'lax_friedrich', 'lax_wendroff']

p_theoretical = [1, 2, 2] # theoretical spatial orders
q_theoretical = [1, 1, 2] # theoretical temporal orders

h_x, h_t = symbols('h_x h_t')
```

```
XtickList = [i for i in range(1, len(hxList)+1)]
Xticknames = [r'$(h_x , h_t)_{0}$'.format(str(i)) for i in range(1, len(hxList)+1)]
lstyle = ['b', 'r', 'g', 'm']
legendList = []

# Optimize Cx, p, Ct, q for every scheme
for k, E in enumerate(ErrorList):

    # optimize using only last 5 values of E and h for the scheme, as the first values may be outside

    Cx0, Ct0 = optimize_error_cxct(E[-5:], hxList[-5:], htList[-5:],
                                   p=p_theoretical[k], q=q_theoretical[k]) # Find appropriate initia

    Cx, p, Ct, q = optimize_error(E[-5:], hxList[-5:], htList[-5:],
                                  Cx0, p_theoretical[k], Ct0, q_theoretical[k]) # Optimize for all p

    # create sympy expressions of e, ex and et:
    errorExpr = Cx*h_x**p + Ct*h_t**q

    print errorExpr
    errorExprHx = Cx*h_x**p
    errorExprHt = Ct*h_t**q

    # convert e, ex and et to python functions:
    errorFunc = lambdify([h_x, h_t], errorExpr)
    errorFuncHx = lambdify([h_x], errorExprHx)
    errorFuncHt = lambdify([h_t], errorExprHt)

    # plotting:
    fig , ax = plt.subplots(2, 1, squeeze=False)

    ax[0][0].plot(XtickList, np.log10(E),lstyle[k])
    ax[0][0].plot(XtickList, np.log10(errorFunc(hxList, htList)),lstyle[k] + '--')

    ax[1][0].plot(XtickList[-5:], E[-5:],lstyle[k])
    ax[1][0].plot(XtickList[-5:], errorFunc(hxList, htList)[-5:],lstyle[k] + '--')
    ax[1][0].plot(XtickList[-5:], errorFuncHx(hxList[-5:]), lstyle[k] + '-.')
    ax[1][0].plot(XtickList[-5:], errorFuncHt(htList[-5:]),lstyle[k] + ':')
```

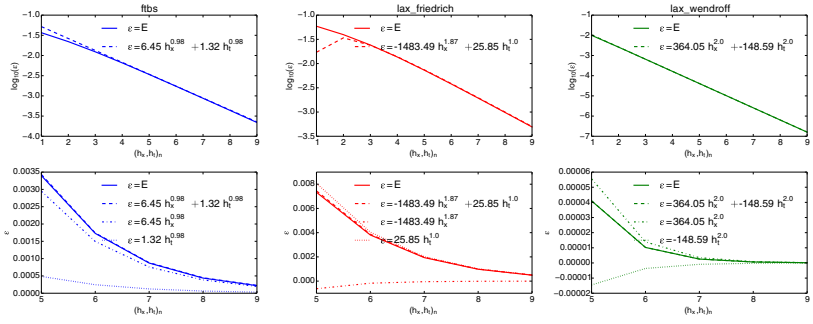

Figure 8.12: Optimized error expressions for all schemes. Test using doconce combine images

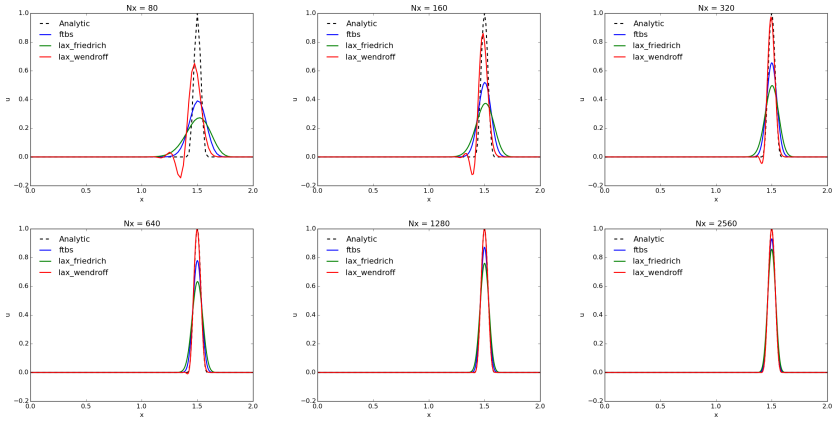

Figure 8.13: The solutions at time t=T=1 for different grids corresponding to the code-example above. A sin^4 wave with period $T = 0.4$ travelling with wavespeed $a = 1$. The solutions were calculated with a cfl-number of 0.8

Marit 2: Har ikke endret figuren

8.9 Flux limiters

Looking at the previous examples and especially Figure 8.13 we clearly see the difference between first and second order schemes. Using a first order scheme require a very high resolution (and/or a cfl number close to one) in order to get a satisfying solution. What characterizes the first order schemes is that they are highly diffusive (loss of amplitude). Unless the resolution is very high or the cfl-number is very close to one, the first order solutions will lose amplitude as time passes. This is evident in the animation (8). (To see how the different schemes behave with different combinations of CFL-number, and frequencies try the sliders app located in the git repo in chapter 6.**Fredrik 15**: add a link or something?) However if the solution contain big gradients or discontinuities, the second order schemes fail, and introduce nonphysical oscillations due to their dispersive nature. In other words the second order schemes give a much higher accuracy on smooth solutions, than the first order schemes, whereas the first order schemes behave better in regions containing sharp gradients or discontinuities. First order upwind methods are said to behave monotonically varying in regions where the solution should be monotone (cite:Second Order Positive Schemes by means of Flux Limiters for the Advection Equation.pdf). Figure 8.14 show the behavior of the different advection schemes in a solution containing discontinuities; oscillations arises near discontinuities for the Lax-Wendroff scheme, independent of the resolution. The dissipative nature of the first order schemes are evident in solutions with "low" resolution.

Marit 2: Har ikke endret figuren

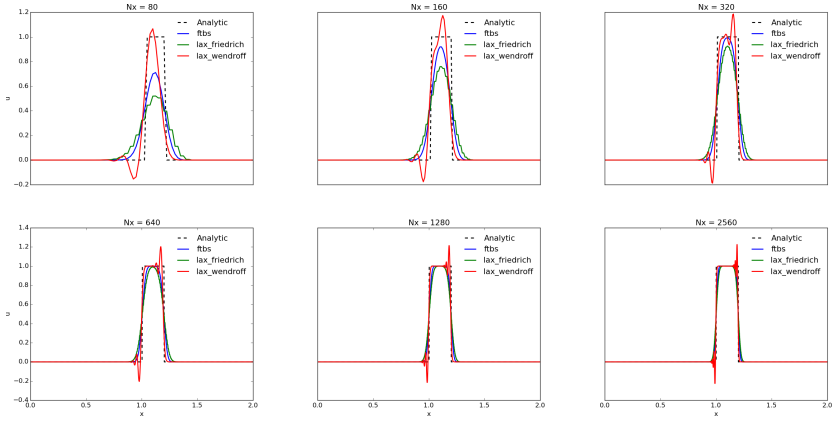

Figure 8.14: Effect of refining grid on a solution containing discontinuities; a square box moving with wave speed a=1. Solutions are showed at t=1, using a cfl-number of 0.8.

The idea of Flux limiters is to combine the best features of high and low order schemes. In general a flux limiter method can be obtained by combining any low order flux F_l, and high order flux F_h. Further it is convenient/required to

write the schemes in the so the so-called conservative form as in (8.41) repeated here for convenience

$$u_j^{n+1} = u_j^n + \frac{\Delta t}{\Delta x}\left(F_{j-1/2} - F_{j+1/2}\right) \tag{8.62}$$

Fredrik 17: do we need clarifying more about conservative methods and Fluxes calculated at midpoints and halftime?

The general flux limiter method solves (8.62) with the following definitions of the fluxes $F_{i-1/2}$ and $F_{i+1/2}$

$$F_{j-1/2} = F_l(j-1/2) + \phi(r_{j-1/2})\big[F_h(j-1/2) - F_l(j-1/2)\big] \tag{8.63}$$
$$F_{j+1/2} = F_l(j+1/2) + \phi(r_{j+1/2})\big[F_h(j+1/2) - F_l(j+1/2)\big] \tag{8.64}$$

where $\phi(r)$ is the limiter function, and r is a measure of the smoothness of the solution. The limiter function ϕ is designed to be equal to one in regions where the solution is smooth, in which F reduces to F_h, and a pure high order scheme. In regions where the solution is not smooth (i.e. in regions containing sharp gradients and discontinuities) ϕ is designed to be equal to zero, in which F reduces to F_l, and a pure low order scheme. As a measure of the smoothness, r is commonly taken to be the ratio of consecutive gradients

$$r_{j-1/2} = \frac{u_{j-1} - u_{j-2}}{u_j - u_{j-1}}, \qquad r_{j+1/2} = \frac{u_j - u_{j-1}}{u_{j+1} - u_j} \tag{8.65}$$

In regions where the solution is constant (zero gradients), some special treatment of r is needed to avoid division by zero. However the choice of this treatment is not important since in regions where the solution is not changing, using a high or low order method is irrelevant.

Lax-Wendroff limiters. The Lax-Wendroff scheme for the advection equation may be written in the form of (8.62) by defining the Lax-Wendroff Flux F_{LW} as:

$$F_{LW}\left(j-1/2\right) = F_{LW}\left(u_{j-1}, u_j\right) = a\, u_{j-1} + \frac{1}{2}a\left(1 - \frac{\Delta t}{\Delta x}a\right)\left[u_j - u_{j-1}\right] \tag{8.66}$$

$$F_{LW}\left(j+1/2\right) = F_{LW}\left(u_j, u_{j+1}\right) = a\, u_j + \frac{1}{2}a\left(1 - \frac{\Delta t}{\Delta x}a\right)\left[u_{j+1} - u_j\right] \tag{8.67}$$

which may be showed to be the Lax-Wendroff two step method condensed to a one step method as outlined in (8.7.1). Notice the term $\frac{\Delta t}{\Delta x}a = c$. Now the Lax-Wendroff flux assumes that of an upwind flux ($a\, u_{j-1}$ or $a\, u_j$) with an additional anti diffusive term ($\frac{1}{2}a\,(1-c)\left[u_j - u_{j-1}\right]$ for $F_{LW}\left(j-1/2\right)$). A flux

limited version of the Lax-Wendroff scheme could thus be obtained by adding a limiter function ϕ to the second term

$$F\left(j-1/2\right) = a\, u_{j-1} + \phi\left(r_{j-1/2}\right)\frac{1}{2}a\left(1-c\right)\left[u_j - u_{j-1}\right] \tag{8.68}$$

$$F\left(j+1/2\right) = a\, u_j + \phi\left(r_{j+1/2}\right)\frac{1}{2}a\left(1-c\right)\left[u_{j+1} - u_j\right] \tag{8.69}$$

When $\phi = 0$ the scheme is the upwind scheme, and when $\phi = 1$ the scheme is the Lax-Wendroff scheme. Many different limiter functions have been proposed, the optimal function however is dependent on the solution. Sweby (**Fredrik 18**: Cite sweby) showed that in order for the Flux limiting scheme to possess the wanted properties of a low order scheme; TVD, or monotonically varying in regions where the solution should be monotone, the following equality must hold **Fredrik 19**: definition of TVD? more theory backing these statements?

$$0 \leq \left(\frac{\phi(r)}{r}, \phi(r)\right) \leq 2 \tag{8.70}$$

Where we require

$$\phi(r) = 0, \quad r \leq 0 \tag{8.71}$$

Hence for the scheme to be TVD the limiter must lie in the shaded region of Figure 8.15, where the limiter function for the two second order schemes, Lax-Wendroff and Warming and Beam are also plotted.

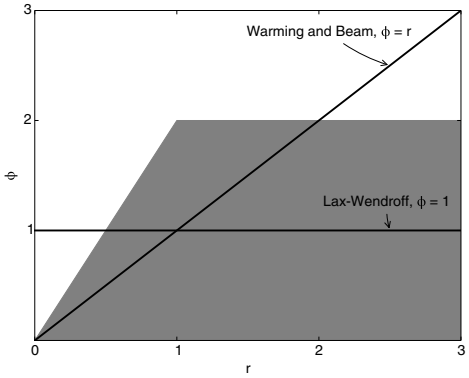

Figure 8.15: TVD region for flux limiters (shaded), and the limiters for the second order schemes; Lax-Wendroff, and Warming and Beam

For the scheme **Fredrik 20**: maybe add only using $u_{j-2}, u_{j-1}u_j u_{j+1}$ to be second order accurate whenever possible, the limiter must be an arithmetic average of the limiter of Lax-Wendroff ($\phi = 1$) and that of Warming and

Beam($\phi = r$) **Fredrik 21**: need citing(Sweby), and maybe more clearification . With this extra constraint a second order TVD limiter must lie in the shaded region of Figure 8.16

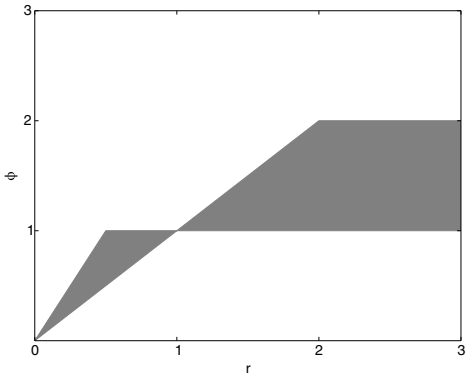

Figure 8.16: Second orderTVD region for flux limiters; sweby diagram **Fredrik 22**: cite sweby?

Note that $\phi(0) = 0$, meaning that second order accuracy must be lost at extrema. All schemes pass through the point $\phi(1) = 1$, which is a general requirement for second order schemes. Many limiters that pass the above constraints have been proposed. Here we will only consider a few:

$$Superbee: \quad \phi(r) = max\,(0, min(1, 2r), min(2, r)) \tag{8.72}$$

$$Van - Leer: \quad \phi(r) = \frac{r + |r|}{1 + |r|} \tag{8.73}$$

$$minmod: \quad \phi(r) = minmod(1, r) \tag{8.74}$$

where minmod of two arguments is defined as

$$minmod(a, b) = \begin{cases} a & if \quad a \cdot b > 0 \quad and \quad |a| > |b| \\ b & if \quad a \cdot b > 0 \quad and \quad |b| > |a| > 0 \\ 0 & if \quad a \cdot b < 0 \end{cases} \tag{8.75}$$

The limiters given in (8.72) -(8.74) are showed in Figure 8.17. Superbee traverses the upper bound of the second order TVD region, whereas minmod traverses the lower bound of the region. Keeping in mind that in our scheme, ϕ regulates the anti diffusive flux, superbee is thus the least diffusive scheme of these.

In addition we will consider the two base cases:

$$upwind: \quad \phi(r) = 0 \tag{8.76}$$

$$Lax - Wendroff: \quad \phi(r) = 1 \tag{8.77}$$

in which upwind, is TVD, but first order, and Lax-Wendroff is second order, but not TVD.

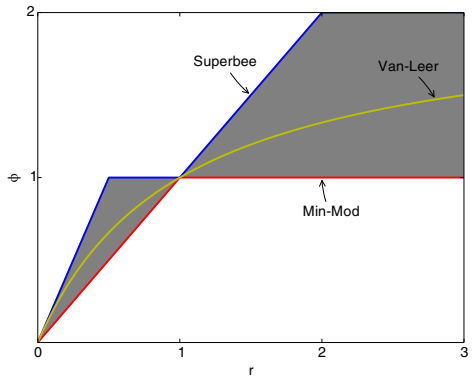

Figure 8.17: Second order TVD limiters

Example of Flux limiter schemes on a solution with continuos and discontinuous sections. All of the above schemes have been implemented in the python class **Fluxlimiters**, and a code example of their application on a solution containing combination of box, and sine moving with wavespeed a=1, may be found in the python script **advection-schemes-flux-limiters.py**. The result may be seen in the video (9)

Movie 9: The effect of Flux limiting schemes on solutions containing continuos and discontinuous sections `mov-ch6/flux_limiter.mp4`

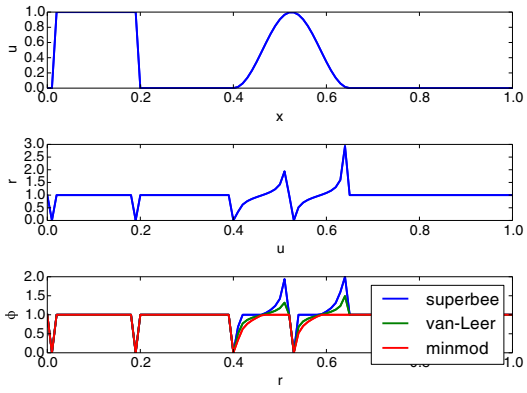

Figure 8.18: Relationship between u and smoothness measure r, and different limiters ϕ, on continuos and discontinuous solutions

8.10 Example: Burger's equation

The 1D Burger's equation is a simple (if not the simplest) non-linear hyperbolic equation commonly used as a model equation to illustrate various numerical schemes for non-linear hyperbolic differential equations. It is normally prestented as:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0 \tag{8.78}$$

To enable us to present schemes for a greater variety of hyperbolic differenctial equations and to better handle shocks (i.e discontinuities in the solution), we will present our model equation on conservative form:

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\left(\frac{u^2}{2}\right) = 0 \tag{8.79}$$

and by introducing a flux function

$$F(u) = \frac{u^2}{2} \tag{8.80}$$

the conservative formulation of the Burger's equation may be represented by a generic transport equation:

$$\frac{\partial u}{\partial t} + \frac{\partial F(u)}{\partial x} = 0 \tag{8.81}$$

8.10.1 Upwind Scheme

The upwind scheme for the general conservation equation take the form:

$$u_j^{n+1} = u_j^n - \frac{\Delta t}{\Delta x}\left(F(u_{j+1}^n) - F(u_j^n)\right) \tag{8.82}$$

where we have assumed a forward propagating wave ($a(u) = F'(u) > 0$, i.e. $u > 0$ for the burger equation). In the opposite case $\frac{\partial F(u)}{\partial x}$ will be approximated by $\frac{1}{\Delta x}\left(F(u_j^n) - F(u_{j-1}^n)\right)$

8.10.2 Lax-Friedrich

The Lax-Friedrich conservation equation take the form as given in (8.38), repeated here for convinience:

$$u_j^{n+1} = \frac{\Delta t}{2}(u_{j+1}^n + u_{j-1}^n) - \frac{F_{j+1}^n - F_{j-1}^n}{2\Delta x} \tag{8.83}$$

8.10.3 Lax-Wendroff

As outlined in ((8.7.1)), the general Lax-Wendroff two step method takes the form as given in (8.51) and (8.52) repeated here for convinience:

First step:

$$u_{j+1/2}^{n+1/2} = \frac{1}{2}\left(u_{j+1}^n + u_j^n\right) - \frac{\Delta t}{2\Delta x}\left(F(u_{j+1}^n) - F(u_j^n)\right) \tag{8.84}$$

$$u_{j-1/2}^{n+1/2} = \frac{1}{2}\left(u_j^n + u_{j-1}^n\right) - \frac{\Delta t}{2\Delta x}\left(F(u_j^n) - F(u_{j-1}^n)\right) \tag{8.85}$$

$$u_j^{n+1} = u_j^n - \frac{\Delta t}{\Delta x}\left(F(u_{j+1/2}^{n+1/2}) - F(u_{j-1/2}^{n+1/2})\right) \tag{8.86}$$

In the previous section ((8.7.1)) we showed how the two step Lax-Wendroff method could be condensed to a one step method. The same procedure may be applied to a the general transport equation given by (8.81). However for the nonlinear case (8.43) no longer holds. This may be overcome be rewriting (8.43):

$$\begin{aligned}\left.\frac{\partial u}{\partial t}\right|_j^n = -\left.\frac{\partial F}{\partial x}\right|_j^n \qquad \text{and} \qquad \left.\frac{\partial^2 u}{\partial t^2}\right|_j^n &= -\left.\frac{\partial^2 F(u)}{\partial t \partial x}\right|_j^n = -\left.\frac{\partial\left(\frac{\partial F(u)}{\partial t}\right)}{\partial x}\right|_j^n \\ = -\left.\frac{\partial\left(\frac{\partial F(u)}{\partial u}\frac{\partial u}{\partial t}\right)}{\partial x}\right|_j^n &= \left.\frac{\partial\left(a(u)\frac{\partial F}{\partial x}\right)}{\partial x}\right|_j^n\end{aligned} \tag{8.87}$$

Now inserting into the Taylor series we get

$$u_j^{n+1} = u_j^n - \Delta t\frac{\partial F(u)}{\partial x} + \frac{\Delta t^2}{2}\frac{\partial\left(a(u)\frac{\partial F}{\partial x}\right)}{\partial x} \tag{8.88}$$

and further we may obtain the general Lax-Wendroff one-step method for a generic transport equation

$$u_j^{n+1} = u_j^n - \frac{\Delta t}{2\Delta x}\left(F_{j+1} - F_{j-1}\right) + \frac{\Delta t^2}{2\Delta x^2}\left[a_{j+1/2}\left(F_{j+1} - F_j\right) - a_{j-1/2}\left(F_j - F_{j-1}\right)\right] \tag{8.89}$$

where $a(u)$ is the wavespeed, or the Jacobian of F, $F'(u)$, which is u for the burger equation. As indicated $a(u)$ has to be approximated at the indice $(j+1/2)$ and $(j-1/2)$. This may simply be done by averaging the neighboring values:

$$a_{j+1/2} = \frac{1}{2}\left(u_j^n + u_{j+1}^n\right) \tag{8.90}$$

for the burger equation. Another method that assure conservation is to use the following approximation

$$a_{j+1/2} = \begin{cases} \frac{F_{j+1}^n - F_j^n}{u_{j+1}^n - u_j^n} & if \quad u_{j+1} \neq u_j \\ u_j & otherwise \end{cases} \tag{8.91}$$

8.10.4 MacCormack

The MacCormack scheme was discussed in ((8.7.1)) and is given by (8.94) repeated her for convinience

$$u_j^p = u_j^n + \frac{\Delta t}{\Delta x}\left(F_j^n - F_{j+1}^n\right) \tag{8.92}$$

$$u_j^{n+1} = \frac{1}{2}\left(u_j^n + u_j^p\right) + \frac{1}{2}\frac{\Delta t}{\Delta x}\left(F_{j-1}^p - F_j^p\right) \tag{8.93}$$

$$\tag{8.94}$$

8.10.5 Method of Manufactured solution

For the Advection equation we were able to verify our schemes by comparing with exact solutions, using MES. For the burger equation it is not easy to find an analytical solution, so in order to verify our schemes we use the MMS approach. However this requires that our schemes can handle source terms. The new equation to solve is thus the modified burgers equation

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = Q \tag{8.95}$$

In this chapter we will consider source terms that are a function of x and t only. The basic approach to adding source terms to our schemes is to simply add a term Q_i^n to our discrete equations. The schemes mentioned above with possibility to handle source terms are summarized in Table (8.10.5).

Name of Scheme	Scheme	order
Upwind	$u_j^{n+1} = u_j^n - \frac{\Delta t}{\Delta x} + \Delta t Q_j^n$	1
Lax-Friedrichs	$u_j^{n+1} = \frac{\Delta t}{2}(u_{j+1}^n + u_{j-1}^n) - \frac{F_{j+1}^n - F_{j-1}^n}{2\Delta x} + \Delta t Q_j^n$	1
Lax-Wendroff	$u_{j+1/2}^{n+1/2} = \frac{1}{2}\left(u_{j+1}^n + u_j^n\right) - \frac{\Delta t}{2\Delta x}\left(F(u_{j+1}^n) - F(u_j^n)\right) + \frac{\Delta t}{2}Q_{j+1/2}^n$ $u_{j-1/2}^{n+1/2} = \frac{1}{2}\left(u_j^n + u_{j-1}^n\right) - \frac{\Delta t}{2\Delta x}\left(F(u_j^n) - F(u_{j-1}^n)\right) + \frac{\Delta t}{2}Q_{j-1/2}^n$ $u_j^{n+1} = u_j^n - \frac{\Delta t}{\Delta x}\left(F(u_{j+1/2}^{n+1/2}) - F(u_{j-1/2}^{n+1/2})\right) + \Delta t Q_j^n$	2
macCormack	$u_j^p = u_j^n - \frac{\Delta t}{\Delta x}\left(F_{j+1}^n - F_j^n\right) + \Delta t Q_j^{n+1/2}$ $u_j^{n+1} = \frac{1}{2}\left(u_j^n + u_j^p\right) - \frac{1}{2}\frac{\Delta t}{\Delta x}\left(F_j^p - F_{j-1}^p\right) + \frac{\Delta t}{2}Q_j^{n+1/2}$	2

Examples on how to implement these schemes are given below, where we have used $RHS(x,t)$ instead of Q for the source term:

ftbs or **upwind**:

```
def ftbs(u, t):
    """method that solves u(n+1), for the scalar conservation equation with source term:
        du/dt + dF/dx = RHS,
        where F = 0.5u^2 for the burger equation
        with use of the forward in time backward in space (upwind) scheme

        Args:
```

```
            u(array): an array containg the previous solution of u, u(n). (RHS)
            t(float): an array
        Returns:
            u[1:-1](array): the solution of the interior nodes for the next timestep, u(n+1).
    """
    u[1:-1] = u[1:-1] -  (dt/dx)*(F(u[1:-1])-F(u[:-2])) + dt*RHS(t-0.5*dt, x[1:-1])
    return u[1:-1]
```

Lax-Friedrichs:

```
def lax_friedrich_Flux(u, t):
    """method that solves u(n+1), for the scalar conservation equation with source term:
        du/dt + dF/dx = RHS,
        where F = 0.5u^2 for the burger equation
        with use of the lax-friedrich scheme

        Args:
            u(array): an array containg the previous solution of u, u(n). (RHS)
            t(float): an array
        Returns:
            u[1:-1](array): the solution of the interior nodes for the next timestep, u(n+1).
    """
    u[1:-1] = (u[:-2] +u[2:])/2.0 -  dt*(F(u[2:])-F(u[:-2]))/(2.0*dx) + dt*(RHS(t, x[:-2]) + RHS(t, x
    return u[1:-1]
```

Lax-Wendroff-Two-step:

```
def Lax_W_Two_Step(u, t):
    """method that solves u(n+1), for the scalar conservation equation with source term:
        du/dt + dF/dx = RHS,
        where F = 0.5u^2 for the burger equation
        with use of the Two-step Lax-Wendroff scheme

        Args:
            u(array): an array containg the previous solution of u, u(n).
            t(float): time at t(n+1)
        Returns:
            u[1:-1](array): the solution of the interior nodes for the next timestep, u(n+1).
    """
    ujm = u[:-2].copy() #u(j-1)
    uj = u[1:-1].copy() #u(j)
    ujp = u[2:].copy() #u(j+1)
    up_m = 0.5*(ujm + uj) - 0.5*(dt/dx)*(F(uj)-F(ujm)) + 0.5*dt*RHS(t-0.5*dt, x[1:-1] - 0.5*dx) #u(n
    up_p = 0.5*(uj + ujp) - 0.5*(dt/dx)*(F(ujp)-F(uj)) + 0.5*dt*RHS(t-0.5*dt, x[1:-1] + 0.5*dx)#u(n+

    u[1:-1] = uj -(dt/dx)*(F(up_p) - F(up_m)) + dt*RHS(t-0.5*dt, x[1:-1])
    return u[1:-1]
```

macCormack:

```
def macCormack(u, t):
    """method that solves u(n+1), for the scalar conservation equation with source term:
        du/dt + dF/dx = RHS,
        where F = 0.5u^2 for the burger equation
        with use of the MacCormack scheme

        Args:
            u(array): an array containg the previous solution of u, u(n). (RHS)
            t(float): an array
        Returns:
            u[1:-1](array): the solution of the interior nodes for the next timestep, u(n+1).
```

```
    """
    up = u.copy()
    up[:-1] = u[:-1] - (dt/dx)*(F(u[1:]) - F(u[:-1])) + dt*RHS(t-0.5*dt, x[:-1])
    u[1:] = .5*(u[1:] + up[1:] -  (dt/dx)*(F(up[1:]) - F(up[:-1])) + dt*RHS(t-0.5*dt, x[1:]))
    return u[1:-1]
```

Exercise 11: Stability analysis of the Lax-Friedrich scheme

In this exercise we want to assess the stability of the Lax-Friedrich scheme for the advection equation as formulated in (8.40).

a) Use the PC-criterion to find a sufficient condition for stability for (8.40).

b) Use the von Neumann method to find a sufficient and necessary condition for stability for (8.40) and compare with the PC-condition

Chapter 9

Python Style Guide

PEP 8, and Google have python style guides which we generally try to follow, though we have made some different choices in some respect.

- https://www.python.org/dev/peps/pep-0008/
- https://google-styleguide.googlecode.com/svn/trunk/pyguide.html

9.1 The conventions we use

First off we try to be compliant to the https://www.python.org/dev/peps/pep-0008/ the summary is as follows:

- Use 4 spaces, not tabs (you can get your editor to do this automatically, and it is recommended for coding in general).
- If your function has many inputs, list them vertically by having an extra indent or more
- We use CapitalizedWords and MixedCase mostly in our code. We capitalize the first letter to signal that this is a class, and we keep it in lowercase if it is an instance of a class, or a function. This allows us to easily see what we´re working with.
- We disobey the convention on function names, where PEP 8 wants them to be lowercase and separated by underscores, we use mixedCase for functions as well. This is due to underscore being a hassle to use.
- If you have a list of things, add list to the name. Do not use plurals. good: vesselList, bad: vessels
- Further, as we have a project with a lot of functional files, we often use "import .... as" method, where we have a list of abbreviations that are convenient to follow

9.2 Summary

- Have clear names and use proper namespacing in your code
- Document your code with docstrings adhering to the Goodle docstring standard. Clearly indicate what is input and what is outout. Especially side-effects!
- Structure your code so that it is as self-explanatory as possible, and use comments where additional clarification is useful.
- Remember that all code you write will end up being treated as a "black box" sooner or later. So make sure that it actually works like one, with clean input-output dichotomy.
- Exceptions should crash your program, unless you have very specific reasons why it should not.
- If you absolutely want to stop exceptions, you should not use "except", as this will catch the system exit and keyboard interrupts. If you want catch-all use "except Exception as e"
- You should catch, append and re-raise the exception at each level of the code, so that an informative traceback will appear when the program dies.
- Unit Testing is desirable, and you should test the crucial behavior of each feature you wish to push to the main project. Also, test that the code fails the way it should. Few things are as hard to track down as code that passes wrong data onto other parts in the program
- Keep your Unit tests. Write them well. Systematize them. Make sure they´re thorough, independent of each other, and fast.

9.3 The code is the documentation... i.e. the docs always lie!

Write the code as clearly as possible to what it is doing. Clear, succinct variable names and conventions, and ideally code review.

9.4 Docstring Guide

The google docstring guide is located at Google Style Python Docstrings

Chapter 10

Sympolic computation with SymPy

In this chapter we provide a very short introduction to SymPy customized for the applications and examples in the current setting. For a more thorough presentation see e.g. [9].

10.1 Introduction

SymPy is a Python library for symbolic mathematics, with the ambition to offer a full-featured computer algebra system (CAS). The library design makes SymPy ideally suited to make symbolic mathematical computations integrated in numerical Python applications.

10.2 Basic features

The SymPy library is implemented in the Python module sympy. To avoid namespace conflicts (e.g. with other modules like NumPy/SciPy) we propose to import the SymPy as:

```
import sympy
```

Symbols. SymPy introduces the class symbols (or Symbol) to represent mathematical symbols as Python objects. An instance of type symbols has a set of attributes to hold the its properties and methods to operate on those properties. Such symbols may be used to represent and manipulate algebraic expressions. Unlike many symbolic manipulation systems, variables in SymPy must be defined before they are used (for justification see sympy.org)

As an example, let us define a symbolic expression, representing the mathematical expression $x^2 + xy - y$

```
import sympy
x, y = sympy.symbols('x y')
expr = x**2 + x*y -y
expr
```

Note that we wrote the expression as if "x" and "y" were ordinary Python variables, but instead of being evaluated the expression remains unaltered.

To make the output look nicer we may invoke the pretty print feature of SymPy by:

```
sympy.init_printing(use_latex=True)
expr
```

The expression is now ready for algebraic manipulation:

```
expr + 2
x**2 + x*y -y + 2
```

and

```
expr + y
x**2 + x*y
```

Note that the result of the above is not $x^2 + xy - y + y$ but rather $x^2 + xy$, i.e. the $-y$ and the $+y$ are added and found to cancel automatically by SymPy and a simplified expression is outputted accordingly. Appart from rather obvious simplifications like discarding subexpression that add up to zero (e.g. $y - y$ or $\sqrt{9} = 3$), most simplifications are not performed automatically by SymPy.

```
expr2 = x**2 + 2*x + 1
expr3 = sympy.factor(expr2)
expr3
```

Matrices. Matrices in SymPy are implemented with the Matrix class and are constructed by providing a list of row the vectors in the following manner:

```
sympy.init_printing(use_latex='mathjax')
M = sympy.Matrix([[1, -1], [2, 1], [4, 4]])
M
```

A matrix with symbolic elements may be constructed by:

```
a, b, c, d = sympy.symbols('a b c d')
M = sympy.Matrix([[a, b],[c, d]])
M
```

The matrices may naturally be manipulated like any other object in SymPy or Python. To illustrate this we introduce another $2x2$-matrix

```
n1, n2, n3, n4 =sympy.symbols('n1 n2 n3 n4')
N=sympy.Matrix([[n1, n2],[n3, n4]])
N
```

The two matrices may then be added, subtracted, multiplied, and inverted by the following simple statements

```
M+N, M-N, M*N, M.inv()

M=sympy.Matrix([[0, a], [a, 0]])
```

Diagonaliztion:

```
L, D = M.diagonalize()
L, D
```

Differentiating and integrating. Consider also the parabolic function which may describe the velocity profile for fully developed flow in a cylinder.

```
from sympy import integrate, diff, symbols, pi
v0, r = symbols('v0 r')
v = v0*(1 - r**2)
Q = integrate(2*pi*v*r, r)
Q

Q = sympy.factor(Q)
Q

newV = diff(Q, r)/(r*2*pi)
newV
sympy.simplify(newV)
```

compute $\int cos(x)$

```
from sympy import cos
integrate(cos(x), x)
```

compute $\int_{-\infty}^{\infty} sin(x^2)$

```
from sympy import sin, oo
integrate(sin(x**2), (x, -oo, oo))
```

limits. perform $\lim_{x\to 0} \frac{sin(x)}{x}$

```
from sympy import limit
limit(sin(x)/x, x, 0)
```

solving equations. solve the algebraic equation $x^2 - 4 = 0$

```
from sympy import solve
solve(x**2 - 4*x, x)
```

solve the differential equation $y'' - y' = e^t$

```
from sympy import dsolve, Function, Eq, exp
y = Function('y')
t = symbols('t')
diffeq = Eq(y(t).diff(t, t) - y(t), exp(t))
dsolve(diffeq, y(t))
```

Bibliography

[1] P.W. Bearman and J.K. Harvey. Golf ball aerodynamics. *Aeronaut Q*, 27(pt 2):112–122, 1976. cited By 119.

[2] E. Cheney and David Kincaid. *Numerical Mathematics and Computing*. Cengage Learning, 4th edition, 1999.

[3] E. Cheney and David Kincaid. *Numerical Mathematics and Computing 7th*. Cengage Learning, 7th edition, 2012.

[4] J. Evett and C. Liu. *2,500 Solved Problems in Fluid Mechanics and Hydraulics*. Schaum's Solved Problems Series. McGraw-Hill Education, 1989.

[5] C. A. J. Fletcher. *Computational Techniques for Fluid Dynamics. 1. , Fundamental and General Techniques*. Springer series in computational physics. Springer-Verlag, Berlin, Paris, 1991.

[6] Ernst Hairer, Syvert Paul Norsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*, volume 1. Springer Science & Business, 2008.

[7] C. Hirsch. *Numerical Computation of Internal and External Flows*. Elsevier, 2007.

[8] George A. Hool and Nathan Clarke Johnson. *Elements of Structural Theory-Definitions. Handbook of Building Construction*. New York. McGraw-Hill. Google Books, 1920. p.2.

[9] Robert Johansson. *Numerical Python. a Practical Techniques Approach for Industry*. Springer, 2015.

[10] Hans Petter Langtangen. *A Primer on Scientific Programming With Python*. Springer, Berlin; Heidelberg; New York, fourth edition, 2011.

[11] P. D. Lax. *Hyperbolic Systems of Conservation Laws and the Mathematical Theory of Shock Waves*. Society for Industrial and Applied Mathematics, 1973. Regional conference series in applied mathematics.

[12] Randall J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2007. OCLC: ocm86110147.

[13] R. W. MacCormack. The effect of viscosity in hypervelocity impact cratering. *Astronautics, AIAA*, 69:354, 1969.

[14] John. C. Tannehil, Dale A. Anderson, and Richard H. Pletcher. *Computational Fluid Mechanics and Heat Transfer*. Taylor & Francis, 1997.

[15] F.M. White. *Viscous Fluid Flow*. WCB/McGraw-Hill, 1991.

[16] F.M. White. *Fluid Mechanics*. McGraw-Hill series in mechanical engineering. WCB/McGraw-Hill, 1999.