Numerical Methods for Engineers

A digital compendium

Leif Rune Hellevik

Department of Structural Engineering, NTNU

Jan 4, 2016

Contents

Chapter 1

Introduction

This digital compendium is based on the first chapter of the compendium *Numeriske Beregninger* by J.B. Aarseth. It's intended to be used in the course TKT4140 Numerical Methods with Computer Laboratory at NTNU.

The development of the technical solutions for this digital compendium results from collaborations with professor Hans Petter Langtangen at UiO (hpl@simula.no), who has developed Doconce for flexible typesetting, and associate professor Hallvard Trætteberg at IDI, NTNU (hal@idi.ntnu.no), who has developed the webpage-parser which identifies Python-code for integration in Eclipse IDEs (such as LiClipse). The latter part of the development has been funded by the project IKTiSU.

1.1 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.

$$\begin{aligned} y'(x) &= f(x,y) \\ y(x_0) &= a \end{aligned} \tag{2.1}$$

while for a second order ODE we get

$$\begin{aligned} y''(x) &= f(x,y,y') \\ y(x_0) &= a,\ y'(x_0) = b \end{aligned} \tag{2.2}$$

A first order ODE, as shown in Equation (2.1), will always be an *initial value problem*. For Equation (2.2), 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 (2.2) 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.

If we are to solve an initial value problem of the type in Equation (2.1), we must first be sure that it has a solution. In addition we will demand that this solution is unique, together the two criteria above lead to the following criteria:

The criteria for existence and uniqueness

A sufficient criteria for existence and uniqueness of a solution of the problem in Equation (2.1) is that both $f(x,y)$ and $\frac{\partial f}{\partial y}$ are continuous in and around x_0.

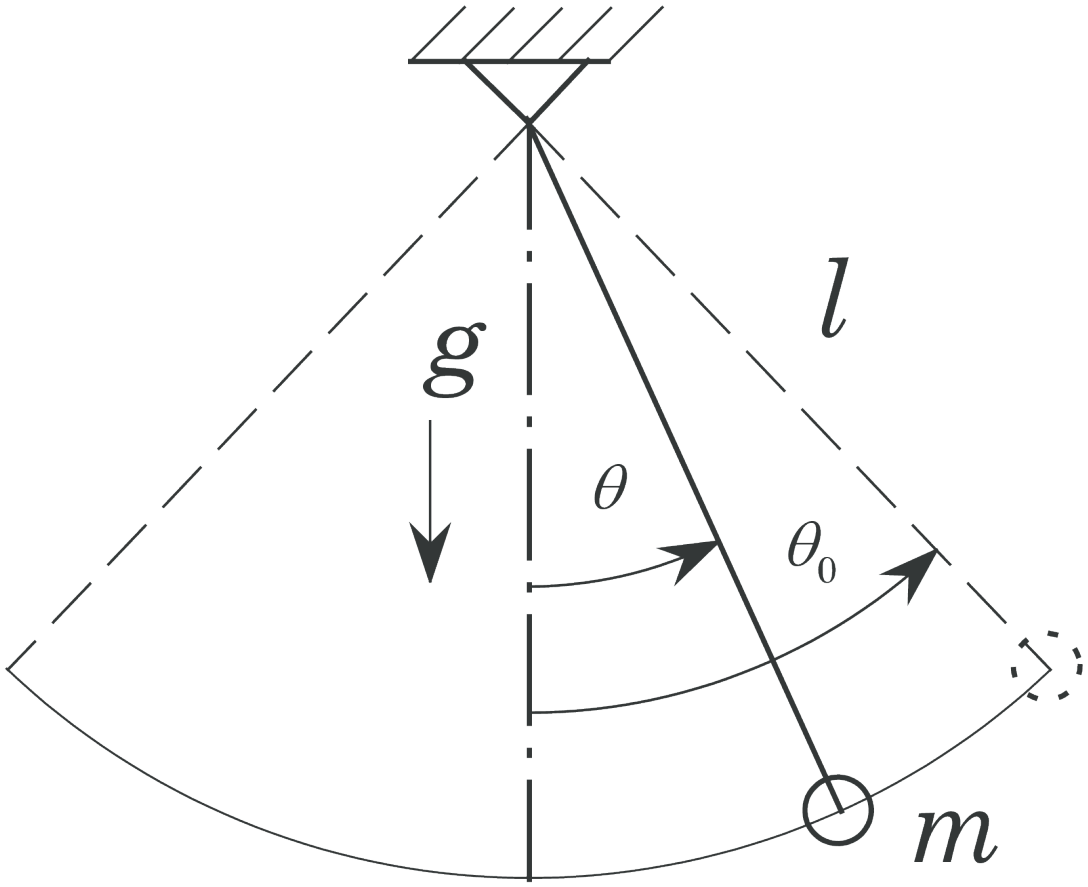
For (2.2) this conditions becomes that $f(x,y)$, $\frac{\partial f}{\partial y}$ and $\frac{\partial f}{\partial y'}$ are continuous in and around x_0. Similarly for higher order equations.

Violation of the criteria for existence and uniqueness

$$y' = y^{\frac{1}{3}},\ y(0) = 0$$

Here $f = y^{\frac{1}{3}}$ and $\frac{\partial f}{\partial y} = \frac{1}{3y^{\frac{2}{3}}}$. f is continuous in $x = 0$, but that's not the case for $\frac{\partial f}{\partial y}$. It may be shown that this ODE has two solutions: $y = 0$ and $y = (\frac{2}{3}x)^{\frac{2}{3}}$. Hopefully this equation doesn't present a physical problem.

A mathematical pendulum A problem of more interest is shown below.



The figure shows a mathematical pendulum where the motion is described by the following equation:

$$\frac{\partial^2 \theta}{\partial \tau^2} + \frac{g}{l}\sin(\theta) = 0 \tag{2.3}$$

$$\theta(0) = \theta_0,\ \frac{d\theta}{d\tau}(0) = 0 \tag{2.4}$$

We introduce a dimensionless time t given by $t = \sqrt{\frac{g}{l}} \cdot \tau$ such that (2.3) and (2.4) may be written as

$$\ddot{\theta}(t) + \sin(\theta(t)) = 0 \tag{2.5}$$

$$\theta(0) = \theta_0, \ \dot{\theta}(0) = 0 \tag{2.6}$$

The dot denotes derivation with respect to the dimensionless time t. For small displacements we can set $\sin(\theta) \approx \theta$, such that (2.5) and (2.6) becomes

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

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

The difference between (2.5) and (2.7) is that the latter is linear, while the first is non-linear. The analytical solution of Equations (2.5) and (2.6) is given in Appendix G.2. in the compendium. An n'th order linear ODE may be written on the form

$$a_n(x)y^{(n)}(x) + a_{n-1}(x)y^{(n-1)}(x) + \cdots + a_1(x)y'(x) + a_0(x)y(x) = b(x) \tag{2.9}$$

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 (2.9) it follows that (2.5) is non-linear and (2.7) 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 change 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 Taylor's method

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

$$f(x) = f(x_0) + (x - x_0) \cdot f'(x_0) + \frac{(x - x_0)^2}{2} f''(x_0) + \cdots + \frac{(x - x_0)^n}{n!} f^{(n)}(x_0) + \text{remainder}$$

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 (2.7):

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

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

We set $\theta(t) \approx \theta(0) + t \cdot \dot{\theta}(0) + \frac{t^2}{2}\ddot{\theta}(0) + \frac{t^3}{6}\dddot{\theta}(0) + \frac{t^4}{24}\theta^{(4)}(0)$. By use of the initial conditions $\theta(0) = \theta_0, \ \dot{\theta}(0) = 0$ we get

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

From the differential equation we have $\ddot{\theta}(t) = -\theta(t) \to \ddot{\theta}(0) = -\theta(0) = -\theta_0$

By differentiation we get $\dddot{\theta}(t) = -\dot{\theta}(t) \to \ddot{\theta}(0) = -\theta(0) = -\theta_0$

We now get

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

Setting this into the expression for $\theta(t)$ gives $\theta(t) \approx \theta_0 \left(1 - \frac{t^2}{2} + \frac{t^4}{24}\right) = \theta_0 \left(1 - \frac{t^2}{2!} + \frac{t^4}{4!}\right)$

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 $\cos(t)$.

Let's try the same procedure on the non-linear version (2.5)

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

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.

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

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

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. [10] section 1.2 for more details on classical differential equations.

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

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

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

Another example:

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

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, men 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_1. We then get the following procedure:

$$\begin{aligned} y' &= y_1' = y_2 \\ y'' &= y_1'' = y_2' = y_3 \end{aligned}$$

The system may then be written as

$$\begin{aligned} y_1'(x) =& y_2(x) \\ y_2'(x) =& y_3(x) \\ y_3'(x) =& -2y_3(x) + (y_2(x))^2 - 2y_1(x) + x^2 \end{aligned}$$

with initial conditions $y_1(0) = a_0,\ y_2(0) = b_0,\ y_3(0) = c_0$.

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

Given the equation

$$y^{(m)} = f(x, y, y', y'', \ldots, y^{(m-1)}) \tag{2.10}$$

$$y(x_0) = a_1,\ y'(x_0) = a_2,\ \ldots, y^{(m-1)}(x_0) = a_m \tag{2.11}$$

where

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

with $y = y_1$, we set

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= y_3 \\ &. \\ &. \\ y_{m-1}' &= y_m \end{aligned} \tag{2.12}$$

$$y_1(x_0) = a_1, y_2(x_0) = a_2, \ldots, y_m(x_0) = a_m$$

2.3.1 Example: Reduction of higher order systems

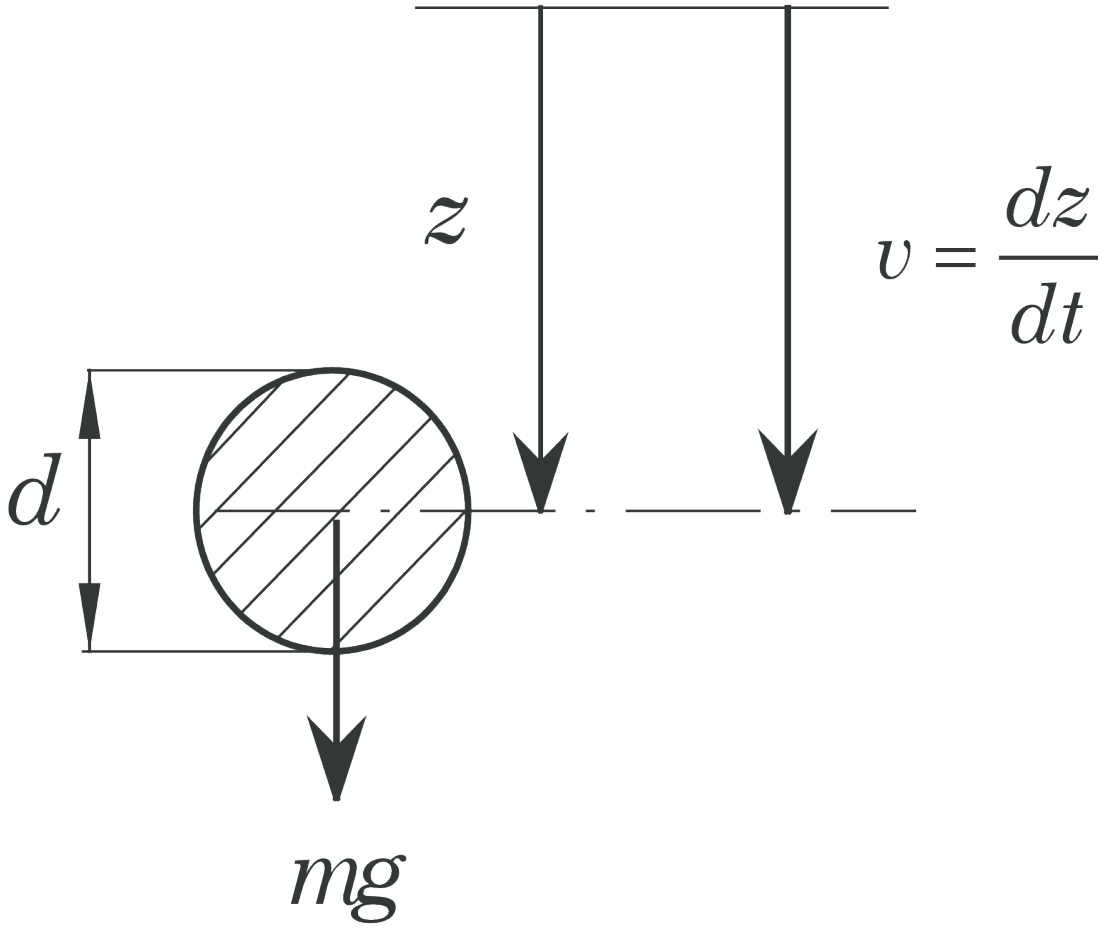
Write the following ODE as a system of first order ODEs:

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

First we write $y''' = y'y'' + (y')^2 - 2y + x^3$.
By use of (2.12) we get

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= y_3 \\ y_3' &= y_2y_3 + (y_2)^2 - 2y_1 + x^3 \\ y_1(0) = a,\ y_2(0) &= b,\ y_3 = c \end{aligned}$$

2.3.2 Example: Sphere in free fall



The figure shows a falling sphere with a diameter d and mass m that falls vertically in a fluid. Use of Newton's 2nd law in the z-direction gives

$$m\frac{dv}{dt} = mg - m_f g - \frac{1}{2}m_f\frac{dv}{dt} - \frac{1}{2}\rho_f v\,|v|\,A_k C_D, \tag{2.13}$$

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 [22], page 539-540. To write Equation (2.13) on a more convenient form we introduce the following abbreviations:

$$\rho = \frac{\rho_f}{\rho_k},\ A = 1 + \frac{\rho}{2},\ B = (1-\rho)g,\ C = \frac{3\rho}{4d}. \tag{2.14}$$

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 (2.13) may then be written as

$$\frac{dv}{dt} = \frac{1}{A}(B - C\cdot v\,|v|\,C_d). \tag{2.15}$$

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 (2.15), we get a 2nd order ODE as follows

$$\frac{d^2z}{dt^2} = \frac{1}{A}\left(B - C \cdot \frac{dz}{dt}\left|\frac{dz}{dt}\right| C_d\right) \tag{2.16}$$

For Equation (2.16) two initial conditions must be specified, e.g. $v = v_0$ and $z = z_0$ for $t = 0$.

Figure 2.1 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 (2.15). We will use the function $C_D(Re)$ as an example of how functions may be implemented in Python.

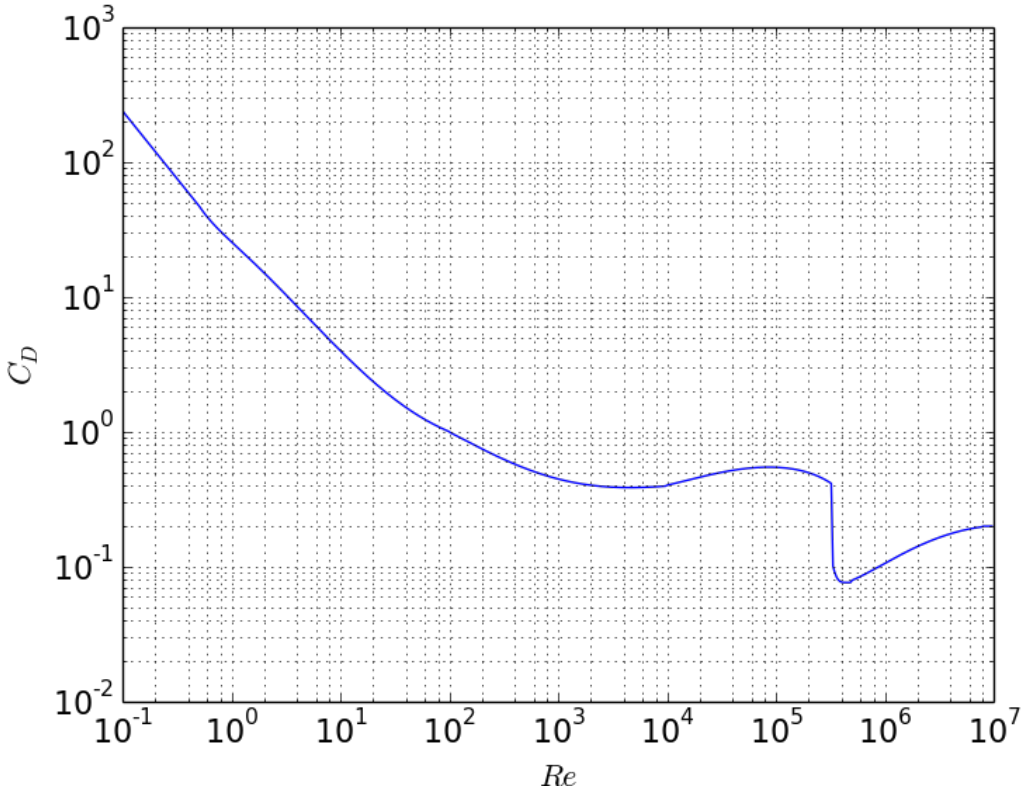


Figure 2.1: Drag coefficient C_D as function of the Reynold's number R_e.

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 [7]. 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 (2.15) and (2.16).

```
# chapter1/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"

    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
pyplot.plot(Re, CD, '-b')
font = {'size' : 16}
pyplot.rc('font', **font)
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
pyplot.plot(Re, CD, '-b')
font = {'size' : 16}
pyplot.rc('font', **font)
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.

2.4 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 2.5.

2.5 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 (2.3.2). 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 [14], 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` [14], 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.

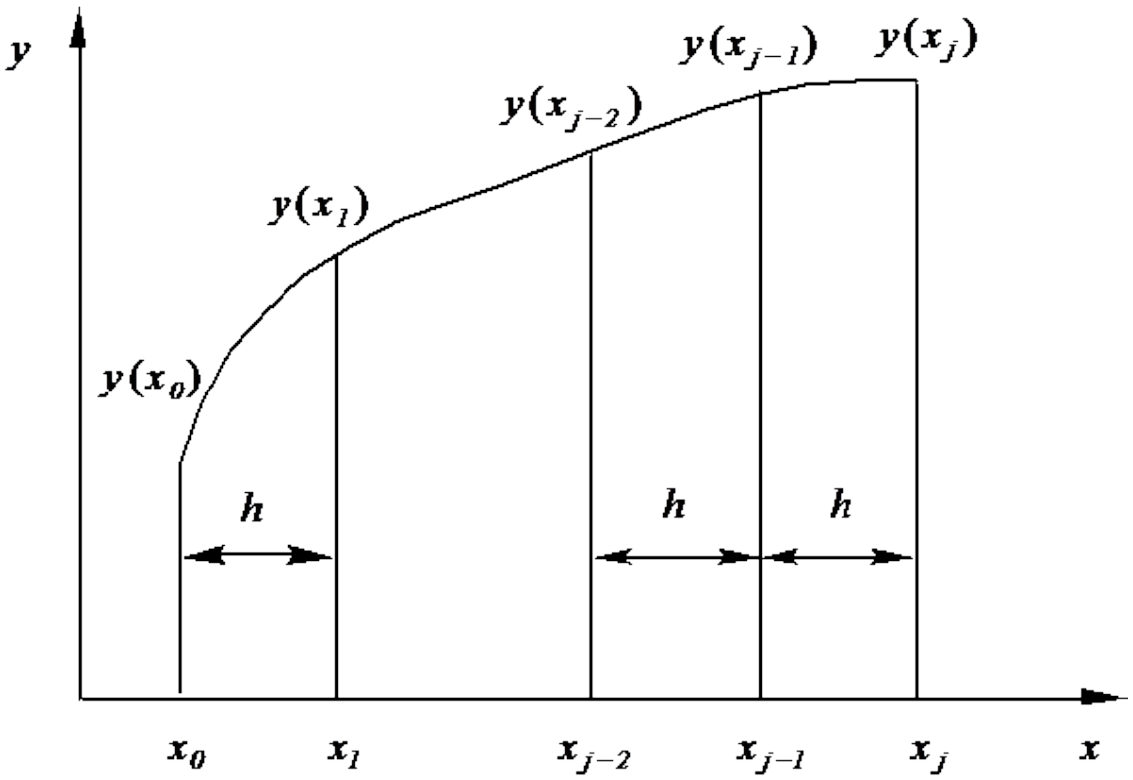
```
# chapter1/src-ch1/DragCoefficientGeneric.py

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()
```

2.6 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.



$$x_j = x_0 + jh$$

where $h = \Delta x$ is assumed constant unless otherwise stated.

Forward differences:

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

Backward differences:

$$\nabla y_j = y_j - y_{j-1} \tag{2.17}$$

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,

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

We will mainly write out the formulas entirely instead of using operators.

We shall find difference formulas and need again **Taylor's theorem**:

$$\begin{aligned} y(x) =& y(x_0) + y'(x_0) \cdot (x - x_0) + \frac{1}{2} y''(x_0) \cdot (x - x_0)^2 + \\ & \cdots + \frac{1}{n!} y^{(n)}(x_0) \cdot (x - x_0)^n + R_n \end{aligned} \tag{2.18}$$

The remainder R_n is given by

$$R_n = \frac{1}{(n+1)!} y^{(n+1)}(\xi) \cdot (x - x_0)^{n+1} \tag{2.19}$$

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

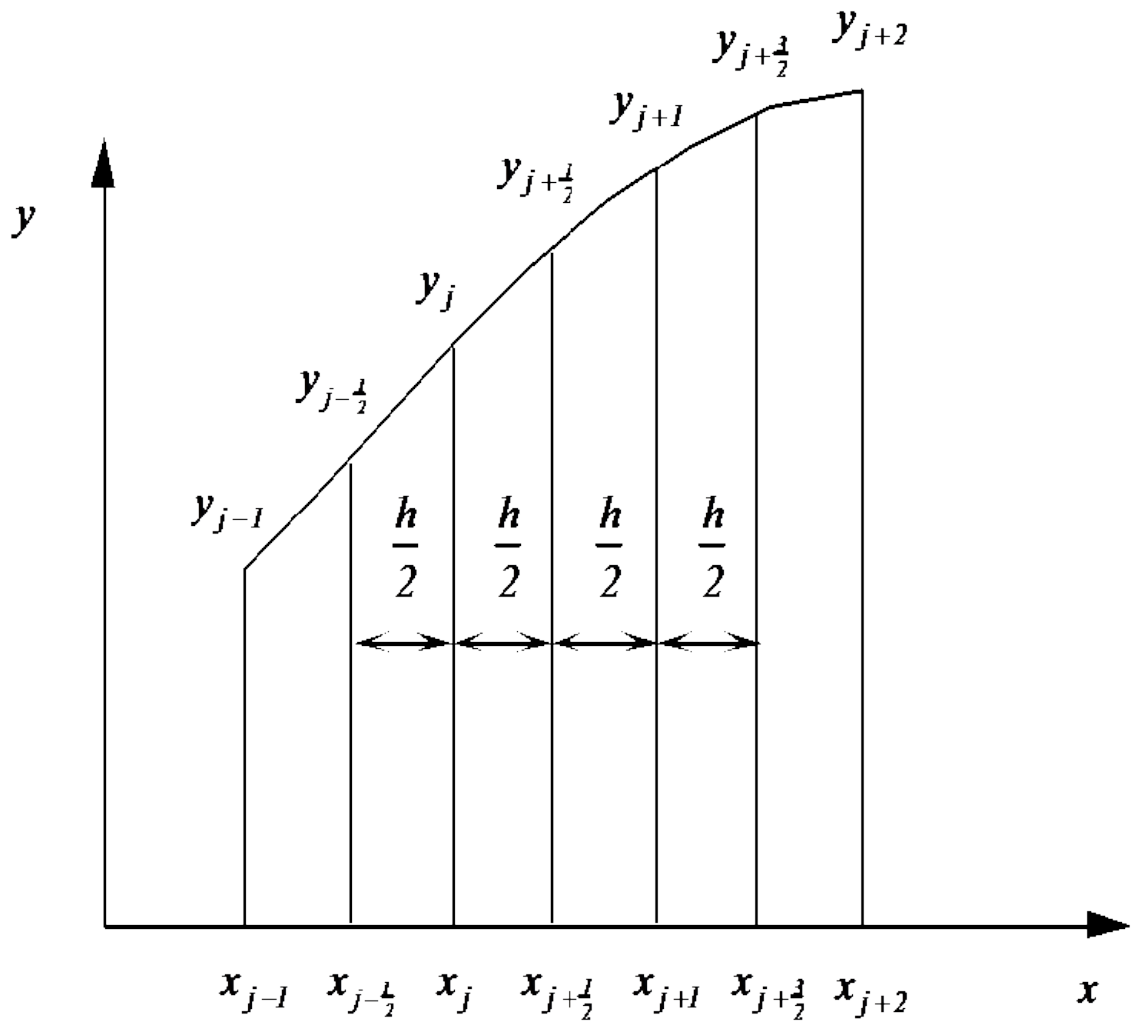


Figure 2.2: Illustration of how to obtain difference equations.

By use of (2.18) we get

$$y(x_{j+1}) \equiv y(x_j + h) = y(x_j) + hy'(x_j) + \frac{h^2}{2}y''(x_j) + \cdots + \frac{h^n y^{(n)}(x_j)}{n!} + R_n \tag{2.20}$$

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

From (2.20) we also get

$$y(x_{j-1}) \equiv y(x_j - h) = y(x_j) - hy'(x_j) + \frac{h^2}{2}y''(x_j) + \cdots + \frac{h^k(-1)^k y^{(k)}(x_j)}{k!} + \ldots \tag{2.21}$$

We will here and subsequently assume that h is positive.

We solve (2.20) with respect to y':

$$y'(x_j) = \frac{y(x_{j+1}) - y(x_j)}{h} + O(h) \tag{2.22}$$

We solve (2.21) with respect to y':

$$y'(x_j) = \frac{y(x_j) - y(x_{j-1})}{h} + O(h) \tag{2.23}$$

By addition of (2.21) and (2.20) we get

$$y''(x_j) = \frac{y(x_{j+1}) - 2y(x_j) + y(x_{j-1})}{h^2} + O(h^2) \tag{2.24}$$

By subtraction of (2.21) from (2.20) we get

$$y'(x_j) = \frac{y(x_{j+1}) - y(x_{j-1})}{2h} + O(h^2) \tag{2.25}$$

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. Which is which will be implied.

Equations (2.22), (2.23), (2.24) and (2.25) then gives the following difference expressions:

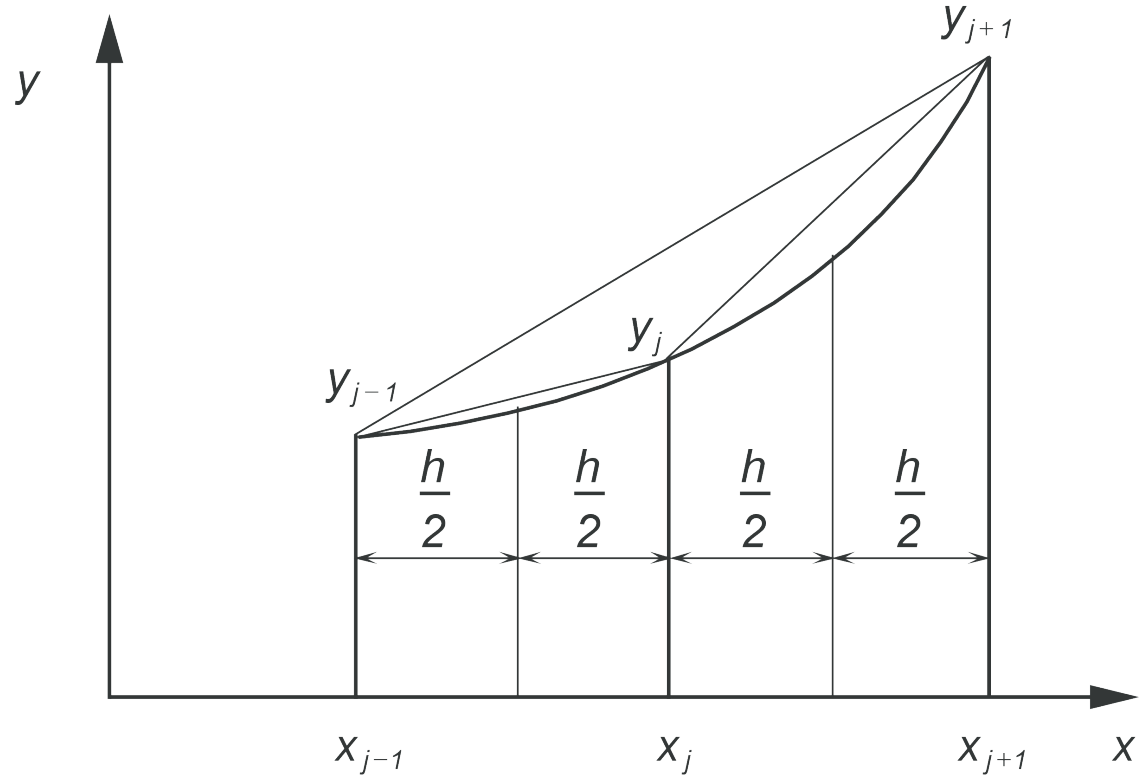
$$y'_j = \frac{y_{j+1} - y_j}{h} \ ; \text{ truncation error } O(h) \tag{2.26}$$

$$y'_j = \frac{y_j - y_{j-1}}{h} \ ; \text{ truncation error } O(h) \tag{2.27}$$

$$y''_j = \frac{y_{j+1} - 2y_j + y_{j-1}}{h^2} \ ; \text{ truncation error } O(h^2) \tag{2.28}$$

$$y'_j = \frac{y_{j+1} - y_{j-1}}{2h} \ ; \text{ truncation error } O(h^2) \tag{2.29}$$

(2.26) is a forward difference, (2.27) is a backward difference while (2.28) and (2.29) are central differences.



The expressions in (2.26), (2.27), (2.28) and (2.29) are easily established from the figure.

(2.26) follows directly.
(2.28):

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

(2.29):

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

To find the truncation error we must use the Taylor series expansion.
The derivation above may be done more systematically. We set

$$y'(x_j) = a \cdot y(x_{j-1}) + b \cdot y(x_j) + c \cdot y(x_{j+1}) + O(h^m) \tag{2.30}$$

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 (2.20) and (2.21) we get

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

Collecting terms:

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

We determine a, b and c such that y_j' gets as high accuracy as possible:

$$\begin{aligned}
a + b + c &= 0 \\
(c-a) \cdot h &= 1 \\
a + c &= 1
\end{aligned} \tag{2.31}$$

The solution to (2.31) is

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

which when inserted in (2.30) gives

$$y_j' = \frac{y_{j+1} - y_{j-1}}{2h} - \frac{h^2}{6}y'''(\xi) \tag{2.32}$$

Comparing (2.32) with (2.30) we see that the error term is $O(h^m) = -\frac{h^2}{6}y'''(\xi)$, which means that $m = 2$. As expected, (2.32) is identical to (2.25).

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,

$$y'(x_j) = a \cdot y_j + b \cdot y_{j+1} + c \cdot y_{j+2} + O(h^m) \tag{2.33}$$

The procedure goes as in the previous example as follows,

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

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

$$\begin{aligned}
a + b + c &= 0 \\
(b + 2c) \cdot h &= 1 \\
\frac{b}{2} + 2c &= 0
\end{aligned} \tag{2.34}$$

The solution of (2.34) is

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

which inserted in (2.33) gives

$$y_j' = \frac{-3y_j + 4y_{j+1} - y_{j+2}}{2h} + \frac{h^2}{3} y'''(\xi) \tag{2.35}$$

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:

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

Backward differences:

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

Central differences:

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

Treatment of the term $\frac{d}{dx}\left[p(x)\frac{d}{dx}u(x)\right]$. This term often appears in difference equations, and it may be clever to treat the term as it is instead of first execute the differentiation.

Central differences: We use central differences (recall Figure 2.2) 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

$$\frac{d}{dx}\left[p(x)\cdot\frac{d}{dx}u(x)\right]\Bigg|_i \approx \frac{p_{i-\frac{1}{2}}\cdot u_{i-1} - (p_{i+\frac{1}{2}} + p_{i-\frac{1}{2}})\cdot u_i + p_{i+\frac{1}{2}}\cdot u_{i+1}}{h^2} + \text{error term} \tag{2.36}$$

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

$$p(x_{1+\frac{1}{2}}) \approx \frac{1}{2}(p_{i+1} + p_i),\ p(x_{1-\frac{1}{2}}) \approx \frac{1}{2}(p_i + p_{i-1}) \tag{2.37}$$

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

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

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

$$\frac{d}{dx}\left[p(x)\cdot\frac{du}{dx}\right]\Big|_i = \frac{2\cdot[p(x_{i+\frac{1}{2}})\cdot(u_{i+1}-u_i)-h\cdot p(x_i)\cdot u'(x_i)]}{h^2} + \text{error term} \tag{2.38}$$

where

$$\text{error term} = -\frac{h}{12}[3p''(x_i)\cdot u'(x_i)+6p'(x_i)\cdot u''(x_i)+4p(x_i)\cdot u'''(x_i)]+O(h^2) \tag{2.39}$$

We have kept the term $u'(x_i)$ since (2.38) usually is used at the boundary, and $u'(x_i)$ may be prescribed there. For $p(x)=1=$ constant we get the expression

$$u_i'' = \frac{2\cdot[u_{i+1}-u_i-h\cdot u'(x_i)]}{h^2} - \frac{h}{3}u'''(x_i)+O(h^2) \tag{2.40}$$

Backward Differences: We start with

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

which gives

$$\frac{d}{dx}\left[p(x)\frac{du}{dx}\right]\Big|_i = \frac{2\cdot[h\cdot p(x_i)u'(x_i)-p(x_{i-\frac{1}{2}})\cdot(u_i-u_{i-1})]}{h^2} + \text{error term} \tag{2.41}$$

where

$$\text{error term} = \frac{h}{12}[3p''(x_i)\cdot u'(x_i)+6p'(x_i)\cdot u''(x_i)+4p(x_i)\cdot u'''(x_i)]+O(h^2) \tag{2.42}$$

This is the same error term as in (2.39) except from the sign. Also here we have kept the term $u'(x_i)$ since (2.42) usually is used at the boundary where $u'(x_i)$ may be prescribed. For $p(x)=1=$ constant we get the expression

$$u_i'' = \frac{2\cdot[h\cdot u'(x_i)-(u_i-u_{i-1})]}{h^2} + \frac{h}{3}u'''(x_i)+O(h^2) \tag{2.43}$$

2.7 Euler's method

The ODE is given as

$$\frac{dy}{dx} = y'(x) = f(x,y) \tag{2.44}$$

$$y(x_0) = y_0 \tag{2.45}$$

By using a first order forward approximation (2.22) of the derivative in (2.44) we obtain:

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

or

$$y_{n+1} = y_n + h\cdot f(x_n, y_n) \tag{2.46}$$

(2.46) is a difference equation and the scheme is called **Euler's method** (1768). The scheme is illustrated graphically in Figure 2.3. 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.

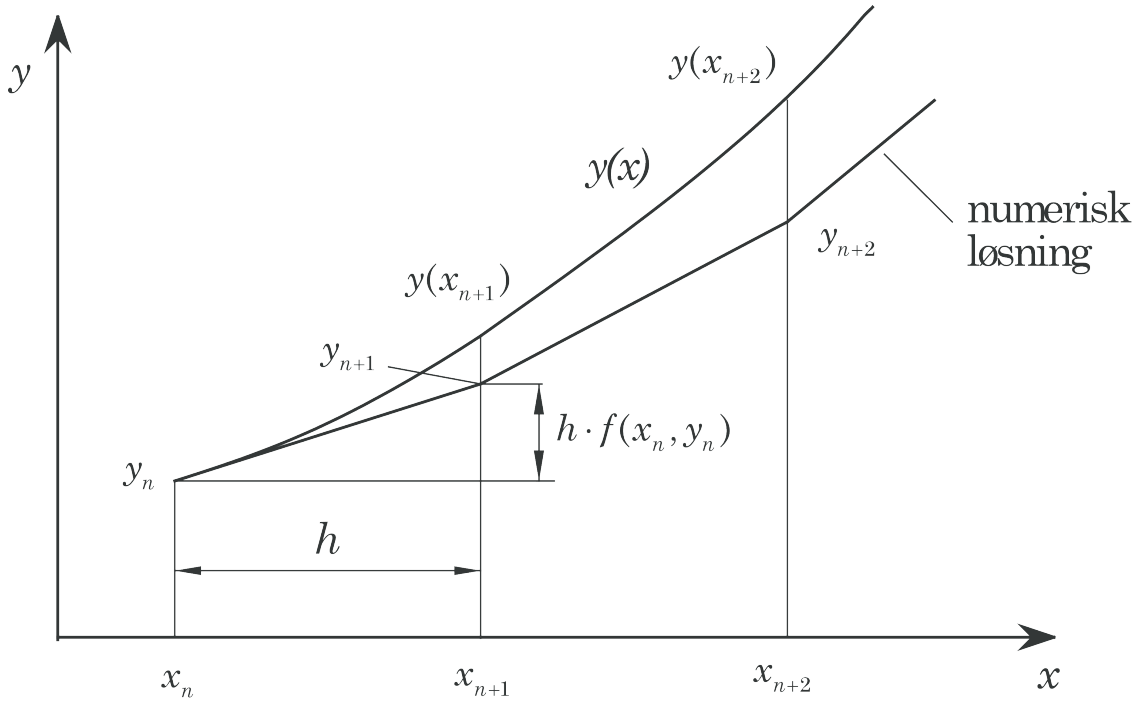


Figure 2.3: Graphical illustration of Euler's method.

2.7.1 Example: Falling sphere with constant and varying drag

We write (2.15) and (2.16) as a system as follows,

$$\frac{dz}{dt} = v \tag{2.47}$$

$$\frac{dv}{dt} = g - \alpha v^2 \tag{2.48}$$

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

$$z(t) = \frac{\ln(\cosh(\sqrt{\alpha g} \cdot t)}{\alpha} \tag{2.49}$$

$$v(t) = \sqrt{\frac{g}{\alpha}} \cdot \tanh(\sqrt{\alpha g} \cdot t) \tag{2.50}$$

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

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 2.2 we get the following expression by using four terms in the series expansion:

$$z(t) = \frac{1}{2}gt^2 \cdot (1 - \frac{1}{6}\alpha g t^2) \tag{2.51}$$

$$v(t) = gt \cdot (1 - \frac{1}{3}\alpha g t^2) \tag{2.52}$$

The Euler scheme (2.46) used on (2.48) gives

$$v_{n+1} = v_n + \Delta t \cdot (g - \alpha \cdot v_n^2),\ n = 0, 1, \ldots \tag{2.53}$$

with $v(0) = 0$.

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 2.4. 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 (2.3.2). The complete program is as follows,

```
# chapter1/src-ch1/FallingSphereEuler.py;DragCoefficientGeneric.py @ git@lrhgit/tkt4140/allfiles/digital_compendiu
from DragCoefficientGeneric import cd_sphere
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

# 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()
```

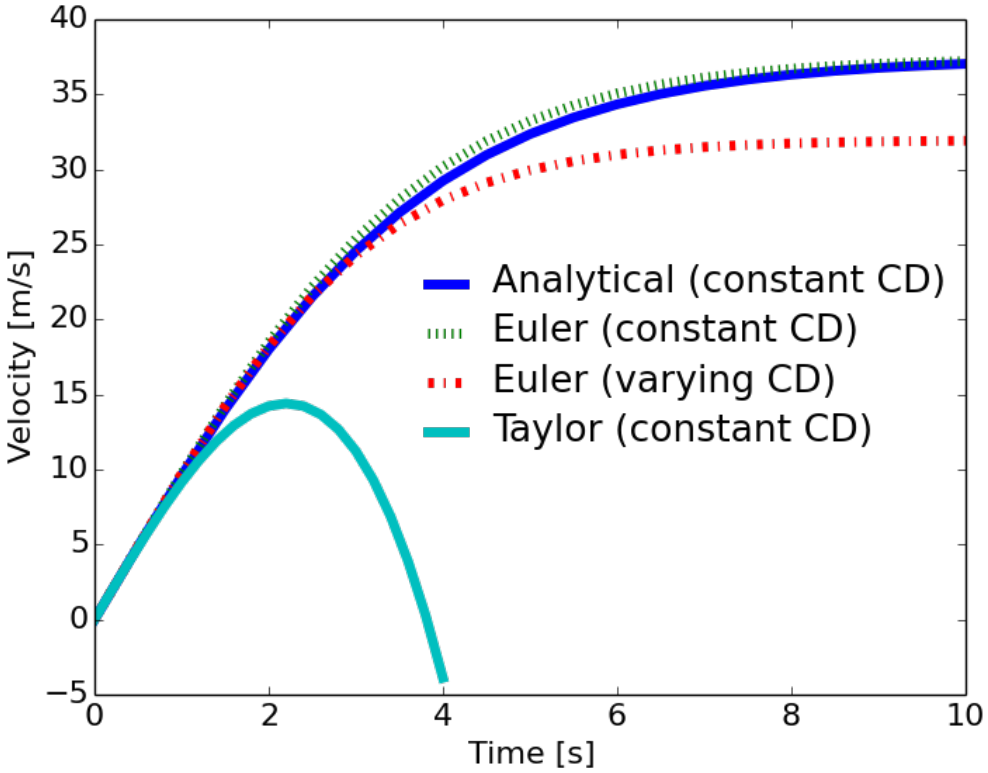


Figure 2.4: Euler's method with $\Delta t = 0.5$ s.

2.7.2 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 timestep 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.log10(abs_error[1:])`.

```
# chapter1/src-ch1/Euler_timestep_ctrl.py;DragCoefficientGeneric.py @ git@lrhgit/tkt4140/allfiles/digital_compend
from DragCoefficientGeneric import cd_sphere
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
font = {'size' : 16}; rc('font', **font)



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.log10(abs_error[1:])
    max_log_error = np.max(log_error)
    #plot(time, ze[:,1])
    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 10**(np.mean(error_diff)), 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('log10-error')
savefig('example_euler_timestep_study.png', transparent=True)
show()
```

The plot resulting from the code above is shown in Figure (2.5). 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 $d_{i+1} = \log 10\epsilon_i - \log 10\epsilon_{i+1} = \log 10\dfrac{\epsilon_i}{\epsilon_{i+1}}$. A rough visual inspection of Figure (2.5) yields $d_{i+1} \approx 0.3$, from which we may deduce:

$$\log 10\frac{\epsilon_i}{\epsilon_{i+1}} \approx 0.3 \Rightarrow \epsilon_{i+1} \approx 10^{-0.3}\,\epsilon_i \approx 0.501\,\epsilon_i \tag{2.54}$$

The print statement `print 10**(np.mean(error_diff)), np.mean(error_diff)` returns `2.04715154702 0.311149993907`, 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 $\epsilon_{i+1} \approx \epsilon_i/2$.

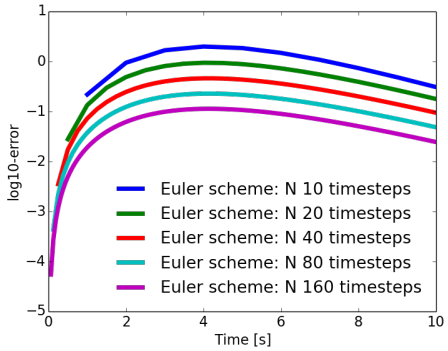


Figure 2.5: Plots for the logarithmic errors for a falling sphere with constant drag. The timestep Δt is reduced by a factor two from one curve to the one immediately below.

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

$$\begin{aligned} y_1' &= f_1(x, y_1, y_2, \ldots y_p) \\ y_2' &= f_2(x, y_1, y_2, \ldots y_p) \\ &. \\ &. \\ y_p' &= f_p(x, y_1, y_2, \ldots y_p) \end{aligned} \tag{2.55}$$

with initial values

$$y_1(x_0) = a_1,\ y_2(x_0) = a_2, \ldots,\ y_p(x_0) = a_p \tag{2.56}$$

Or, in vectorial format as follows,

$$\begin{aligned} \mathbf{y}' &= \mathbf{f}(x, \mathbf{y}) \\ \mathbf{y}(x_0) &= \mathbf{a} \end{aligned} \tag{2.57}$$

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

The Euler scheme (2.46) used on (2.57) gives

$$\mathbf{y_{n+1}} = \mathbf{y_n} + h \cdot \mathbf{f}(x_n, \mathbf{y_n}) \tag{2.58}$$

For a system of three equations we get

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= y_3 \\ y_3' &= -\, y_1 y_3 \end{aligned} \tag{2.59}$$

In this case (2.58) gives

$$\begin{aligned} (y_1)_{n+1} &= (y_1)_n + h \cdot (y_2)_n \\ (y_2)_{n+1} &= (y_2)_n + h \cdot (y_3)_n \\ (y_3)_{n+1} &= (y_3)_n - h \cdot (y_1)_n \cdot (y_3)_n \end{aligned} \tag{2.60}$$

(2.61)

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

In section 2.3 we have seen how we can reduce a higher order ODE to a set of first order ODEs. In (2.47) and (2.48) 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,

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

2.8 Heun's method

From (2.22) or (2.26) we have

$$y''(x_n, y_n) = f'\left(x_n, y(x_n, y_n)\right) \approx \frac{f(x_n + h) - f(x_n)}{h} \tag{2.62}$$

The Taylor series expansion (2.20) 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 (2.62), gives

$$y_{n+1} = y_n + \frac{h}{2} \cdot [f(x_n, y_n) + f(x_{n+1}, y(x_{n+1}))] \tag{2.63}$$

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 (2.46) to calculate $y(x_{n+1})$ on the right side of (2.63). The resulting scheme is often denoted **Heun's method**.

The scheme for Heun's method becomes

$$y^p_{n+1} = y_n + h \cdot f(x_n, y_n) \tag{2.64}$$

$$y_{n+1} = y_n + \frac{h}{2} \cdot [f(x_n, y_n) + f(x_{n+1}, y^p_{n+1})] \tag{2.65}$$

Index p stands for "predicted". (2.64) is then the predictor and (2.65) is the corrector. This is a second order method. For more details, see [5]. Figure 2.6 is a graphical illustration of the 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

$$\mathbf{y^p_{n+1}} = \mathbf{y_n} + h \cdot \mathbf{f}(x_n, \mathbf{y_n}) \tag{2.66}$$

$$\mathbf{y_{n+1}} = \mathbf{y_n} + \frac{h}{2} \cdot [\mathbf{f}(x_n, \mathbf{y_n}) + \mathbf{f}(x_{n+1}, \mathbf{y^p_{n+1}})] \tag{2.67}$$

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

If we use (2.66) and (2.67) on the example in (2.59) 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:

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

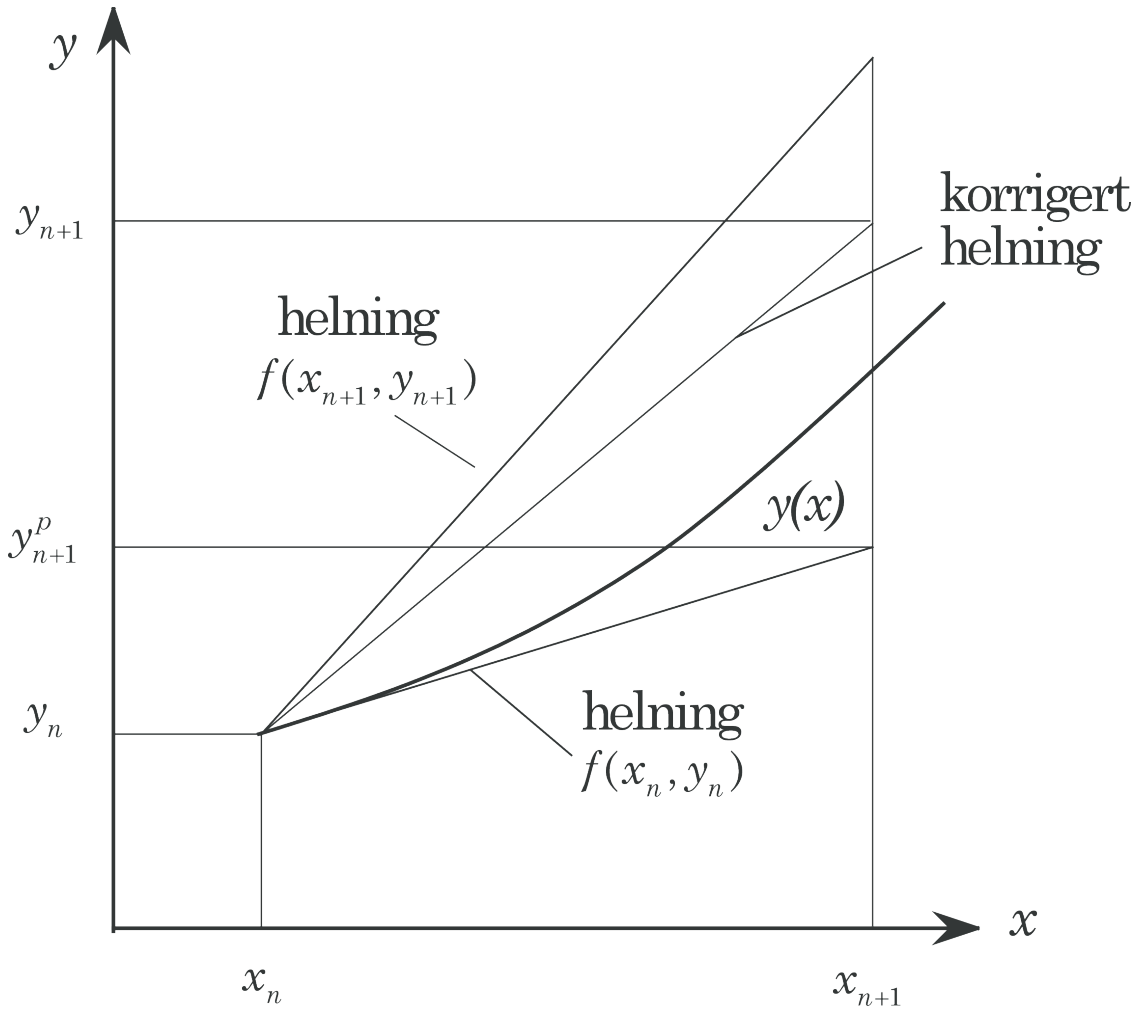


Figure 2.6: Illustration of Heun's method.

2.8.1 Example: Newton's equation

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

$$y'(x) = 1 - 3x + y + x^2 + xy, \; y(0) = 0 \tag{2.68}$$

with analytical solution

$$\begin{aligned} y(x) =& 3\sqrt{2\pi e} \cdot \exp\left(x\left(1 + \frac{x}{2}\right)\right) \cdot \left[\operatorname{erf}\left(\frac{\sqrt{2}}{2}(1+x)\right) - \operatorname{erf}\left(\frac{\sqrt{2}}{2}\right)\right] \\ &+ 4 \cdot \left[1 - \exp\left(x\left(1 + \frac{x}{2}\right)\right)\right] - x \end{aligned} \tag{2.69}$$

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 2.7.

```
# chapter1/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 py
py.plot(x, y, '-b.', x, ya, '-g.')
py.xlabel('x')
py.ylabel('y')
font = {'size' : 16}
py.rc('font', **font)
py.title('Solution to Newton\'s equation')
py.legend(['Heun', 'Analytical'], loc='best', frameon=False)
py.grid()
py.savefig('newton_heun.png', transparent=True)
py.show()
```

2.8.2 Example: Falling sphere with Heun's method

Let's go back to (2.7.1), and implement a new function `heun()` in the program FallingSphereEuler.py.

We recall the system of equations as

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

which by use of Heun's method in (2.66) and (2.67) becomes

Predictor:

$$z^p_{n+1} = z_n + \Delta t v_n \tag{2.70}$$
$$v^p_{n+1} = v_n + \Delta t \cdot (g - \alpha v_n^2)$$

Corrector:

$$z_{n+1} = z_n + 0.5\Delta t \cdot (v_n + v^p_{n+1}) \tag{2.71}$$
$$v_{n+1} = v_n + 0.5\Delta t \cdot \left[2g - \alpha[v_n^2 + (v^p_{n+1})^2\right]$$

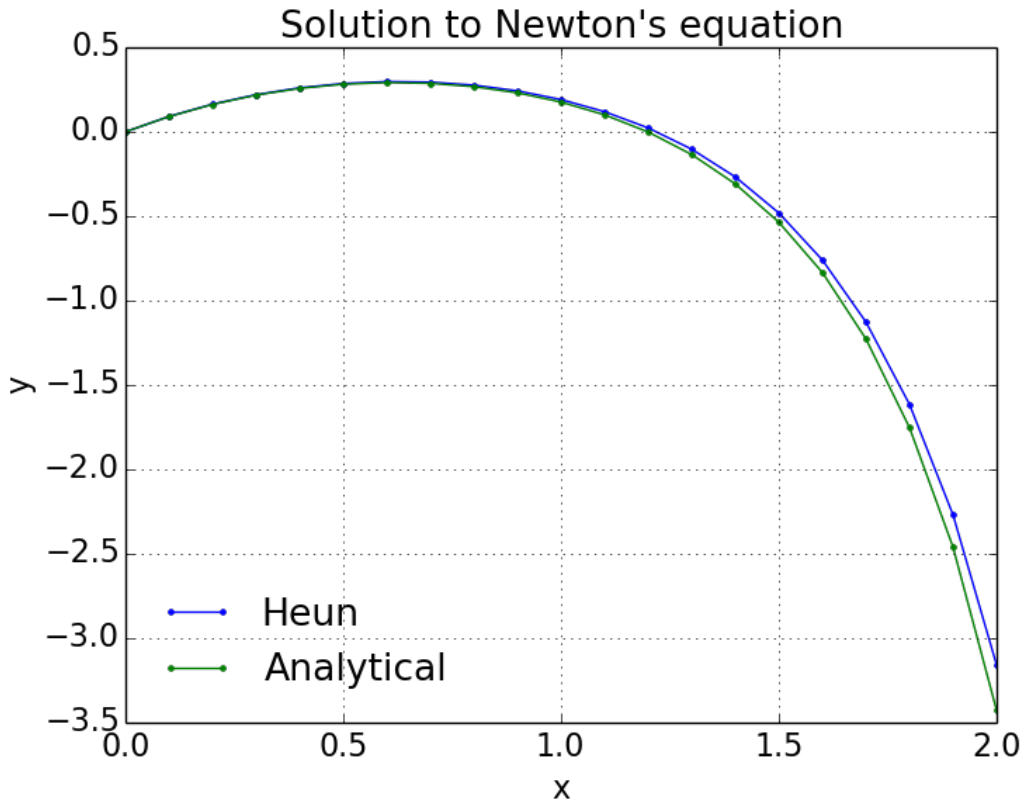


Figure 2.7: Velocity of falling sphere using Euler's and Heun's methods.

with initial values $z_0 = z(0) = 0,\ v_0 = v(0) = 0$. Note that we don't use the predictor z^p_{n+1} 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 ."""

    def f_np(z,t):
        """A local function to ensure that the return of func is an np array
        and to avoid lengthy code for implementation of the Heun algorithm"""
        return np.asarray(func(z,t))

    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,:] + f_np(z[i,:],t)*dt    # Predictor step
        z[i+1,:] = z[i,:] + (f_np(z[i,:],t) + f_np(zp,t+dt))*dt/2.0 # Corrector step
```

Using the same time steps as in (2.7.1), we get the response plotted in Figure 2.8.

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

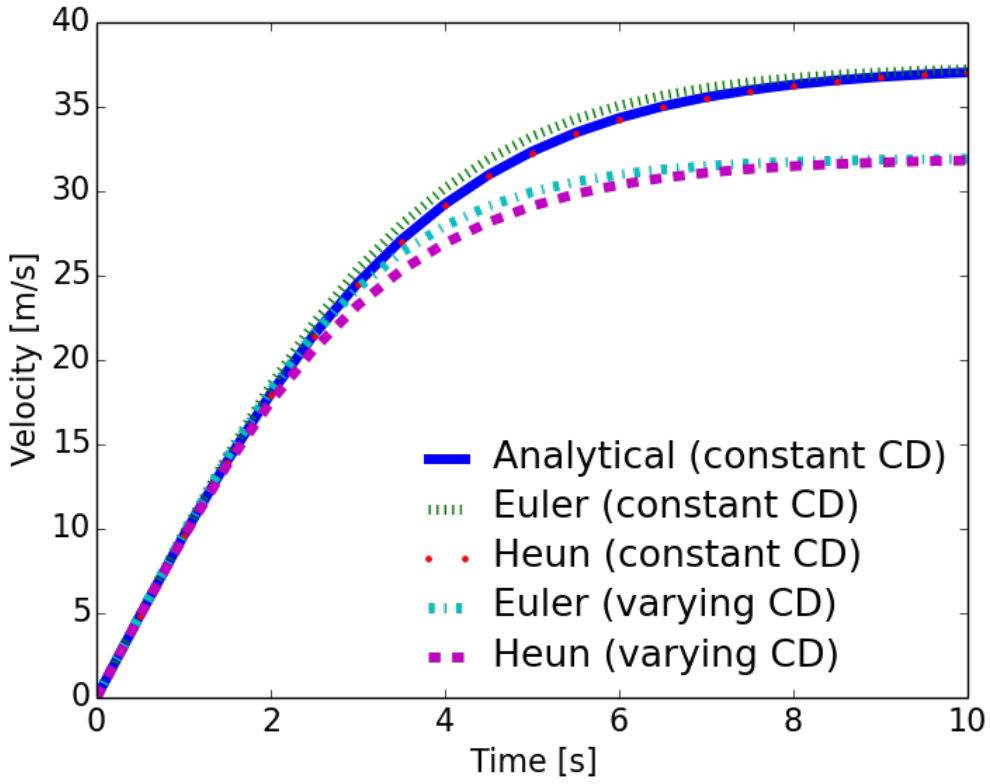


Figure 2.8: Velocity of falling sphere using Euler's and Heun's methods.

```
# chapter1/src-ch1/FallingSphereEulerHeun.py;ODEschemes.py @ git@lrhgit/tkt4140/allfiles/digital_compendium/chapte
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)
font = {'size' : 16}
rc('font', **font)
xlabel('Time [s]')
ylabel('Velocity [m/s]')
savefig('example_sphere_falling_euler_heun.png', transparent=True)
show()
```

2.9 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

$$\begin{aligned} k_1 &= f(x_n, y_n) \\ k_2 &= f(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \\ k_3 &= f(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_2) \\ k_4 &= f(x_n + h, y_n + hk_3) \\ y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \tag{2.72}$$

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 [5].

Figure 2.9 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.

Using (2.72) on the equation system in (2.59) we get

$$(y_1)_{n+1} = (y_1)_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

$$(y_2)_{n+1} = (y_2)_n + \frac{h}{6}(l_1 + 2l_2 + 2l_3 + l_4) \tag{2.73}$$

$$(y_3)_{n+1} = (y_3)_n + \frac{h}{6}(m_1 + 2m_2 + 2m_3 + m_4) \tag{2.74}$$

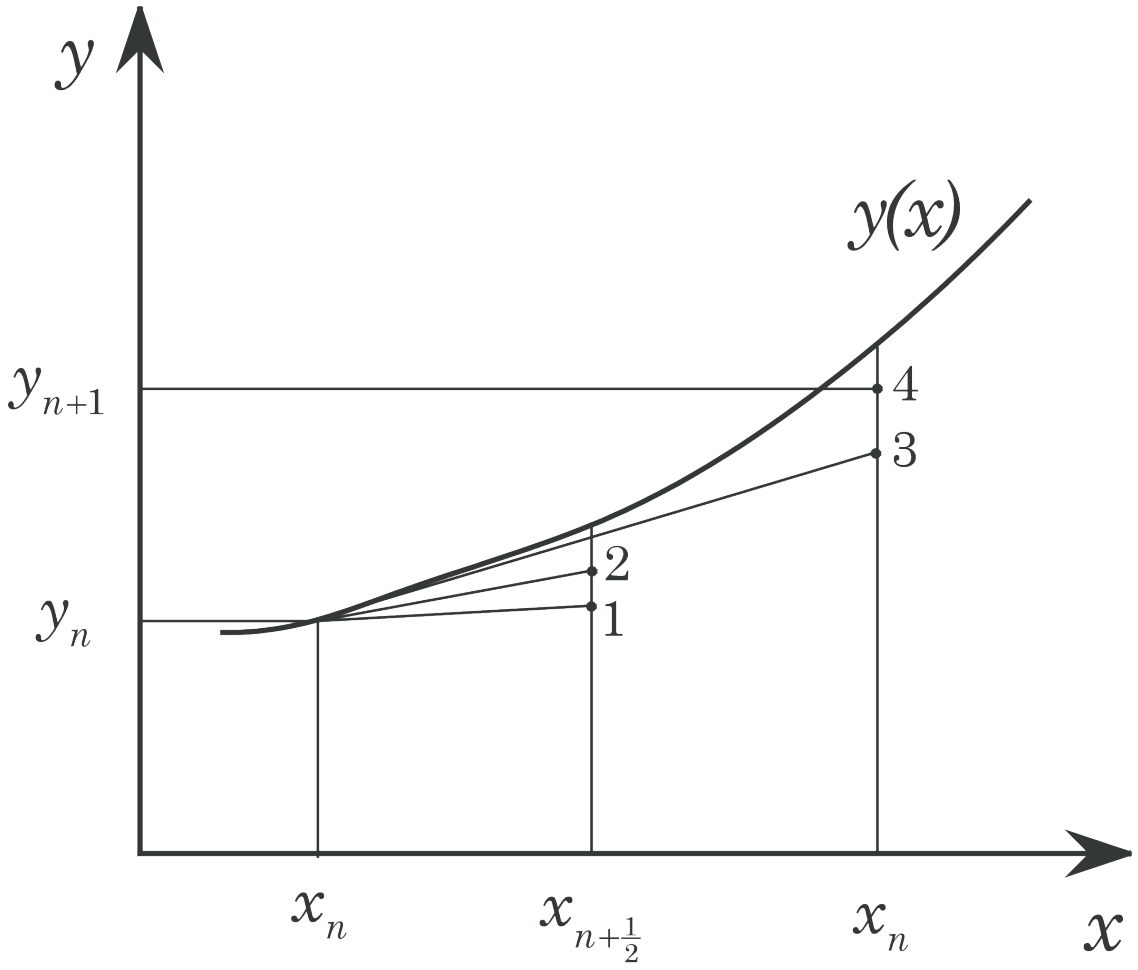


Figure 2.9: Illustration of the RK4 scheme.

where

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

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

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

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

2.9.1 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 2.10 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**.

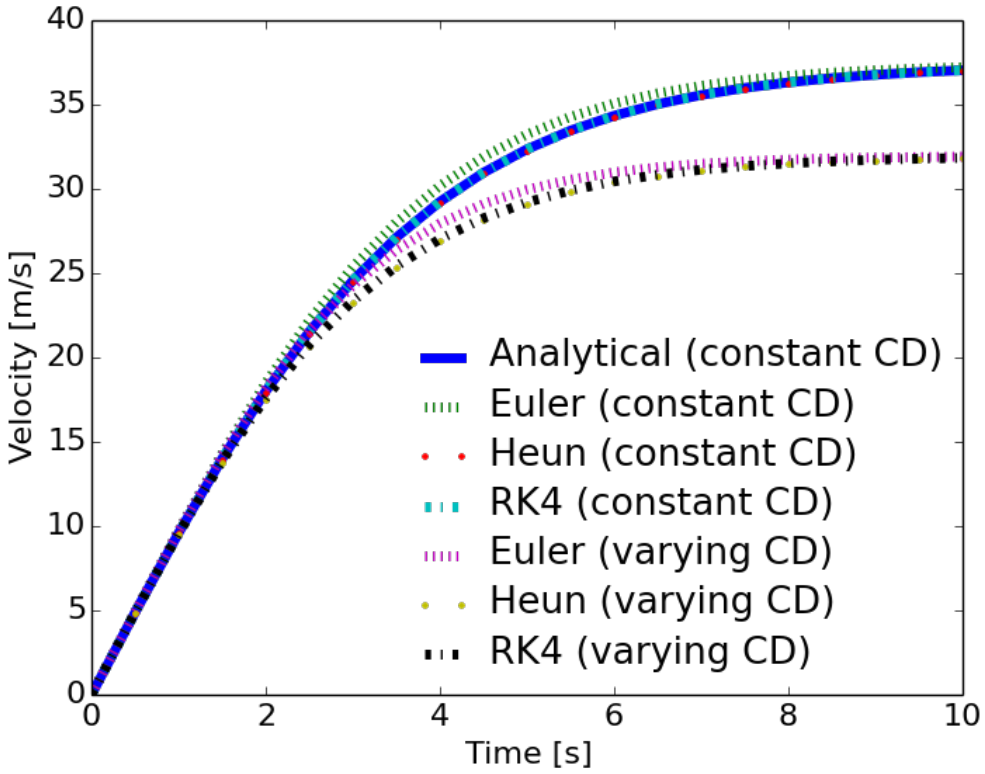


Figure 2.10: Velocity of falling sphere using Euler, Heun and RK4.

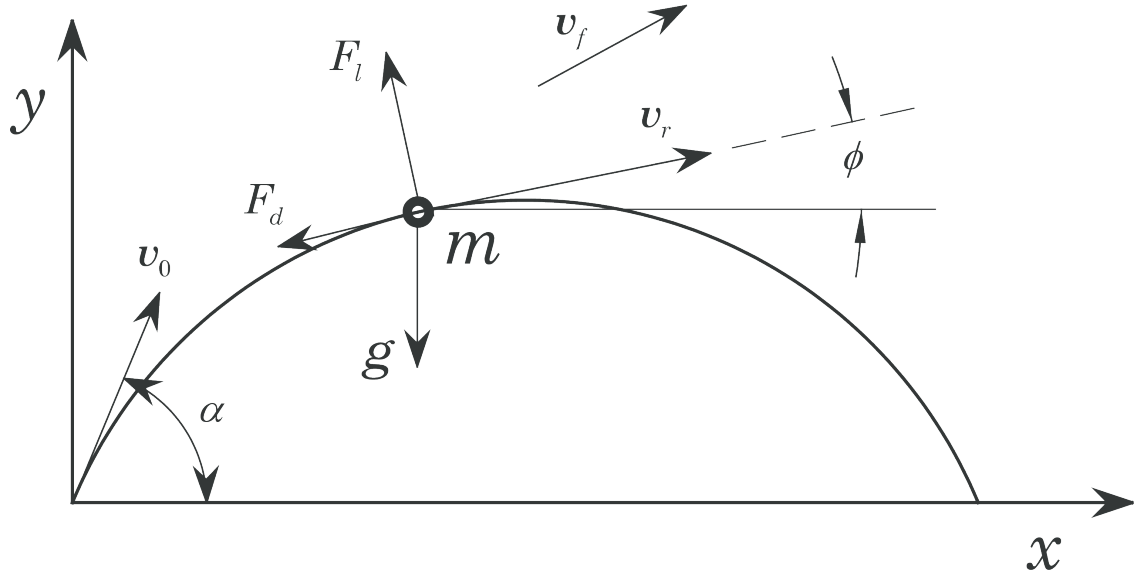
```
# chapter1/src-ch1/FallingSphereEulerHeunRK4.py;ODEschemes.py @ git@lrhgit/tkt4140/allfiles/digital_compendium/cha

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)

font = {'size' : 16}
rc('font', **font)
xlabel('Time [s]')
ylabel('Velocity [m/s]')
savefig('example_sphere_falling_euler_heun_rk4.png', transparent=True)
show()
```

2.9.2 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 (2.3.2)), and then of a golf ball with drag and lift.

The problem is illustrated in the following figure:



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

$$\mathbf{F}_d = \frac{1}{2}\rho_f A C_D v_r^2 \tag{2.75}$$

$$\mathbf{F}_l = \frac{1}{2}\rho_f A C_L v_r^2 \tag{2.76}$$

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

$$\frac{dv_x}{dt} = -\rho_f \frac{A}{2m} v_r^2 (C_D \cdot \cos(\phi) + C_L \sin(\phi)) \tag{2.77}$$

$$\frac{dv_y}{dt} = \rho_f \frac{A}{2m} v_r^2 (C_L \cdot \cos(\phi) - C_D \sin(\phi)) - g \tag{2.78}$$

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 (2.3.2). Here d is the diameter of the sphere and ρ_k the density of the sphere.

Now (2.77) and (2.78) become

$$\frac{dv_x}{dt} = -C \cdot v_r (C_D \cdot v_{rx} + C_L \cdot v_{ry}) \tag{2.79}$$

$$\frac{dv_y}{dt} = C \cdot v_r (C_L \cdot v_{rx} - C_D \cdot v_{ry}) - g \tag{2.80}$$

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

$$\begin{aligned} \frac{dx}{dt} &= v_x \\ \frac{dy}{dt} &= v_y \\ \frac{dv_x}{dt} &= -C \cdot v_r (C_D \cdot v_{rx} + C_L \cdot v_{ry}) \\ \frac{dv_y}{dt} &= C \cdot v_r (C_L \cdot v_{rx} - C_D \cdot v_{ry}) - g \end{aligned} \tag{2.81}$$

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

$$\begin{aligned} \frac{dy_1}{dt} &= y_3 \\ \frac{dy_2}{dt} &= y_4 \\ \frac{dy_3}{dt} &= -C \cdot v_r (C_D \cdot v_{rx} + C_L \cdot v_{ry}) \\ \frac{dy_4}{dt} &= C \cdot v_r (C_L \cdot v_{rx} - C_D \cdot v_{ry}) - g \end{aligned} \tag{2.82}$$

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

$$y_1 = y_2 = 0$$
$$y_3 = v_0 \cos(\alpha)$$
$$y_4 = v_0 \sin(\alpha)$$

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

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

We use the initial velocity $v_0 = 50$ m/s and solve (2.82) 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 (2.82) 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 (2.3.2) to calculate the drag coefficient of the smooth sphere.

The results are shown for some initial angles in Figure 2.11.

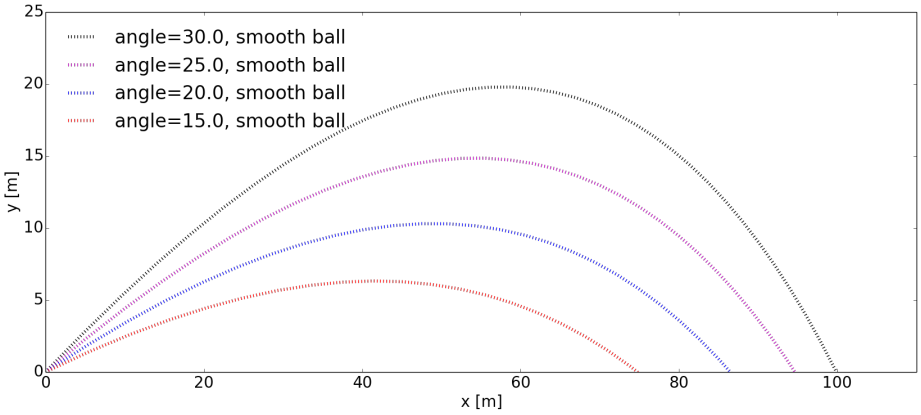


Figure 2.11: Motion of smooth ball with drag.

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 [2] 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 (2.82) 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 2.12. 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.

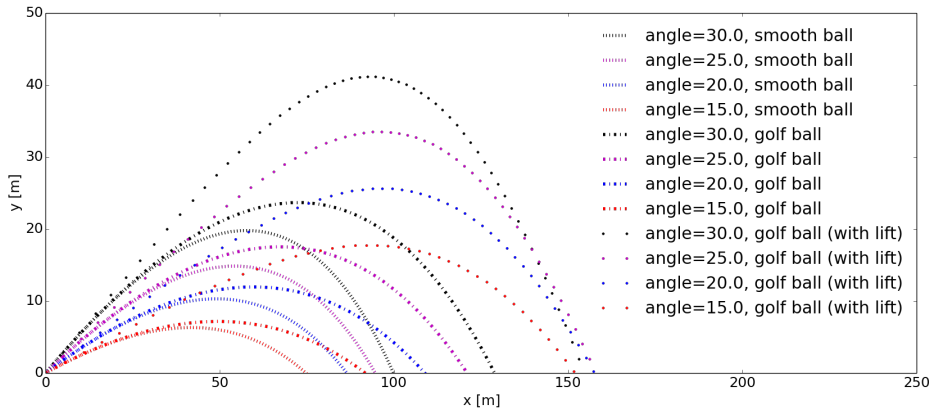


Figure 2.12: Motion of golf ball with drag and lift.

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

```
# chapter1/src-ch1/ParticleMotion2D.py;DragCoefficientGeneric.py @ git@lrhgit/tkt4140/allfiles/digital_compendium/

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()
```

2.9.3 Example: 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 (2.7.2). 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:

$$\begin{aligned} u'(t) &= a\,u + b \\ u(t_0) &= u_0 \end{aligned} \tag{2.83}$$

which has the analytical solutions:

$$u = \begin{cases} \left(u_0 + \frac{b}{a}\right)\, e^{a\,t} - \frac{b}{a}, & a \neq 0 \\ u_0 + b\,t, & a = 0 \end{cases} \tag{2.84}$$

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 (2.46) is $O(h)$, whereas the Heun scheme (2.64) is $O(h^2)$, and Runge-Kutta (2.72) 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 $\Delta t_{i+1} = \frac{\Delta t_i}{2}$. The errors ϵ_{i+1} and ϵ_i for a scheme of order n are then related by:

$$\epsilon_{i+1} = \frac{1}{2^n}\epsilon_i \tag{2.85}$$

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:

$$n \approx \log_2 \frac{\epsilon_i}{\epsilon_{i+1}} \tag{2.86}$$

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 (2.86) with the clause `order_approx.append(previous_max_log_err - ma` 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 2.13, 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).

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])
```

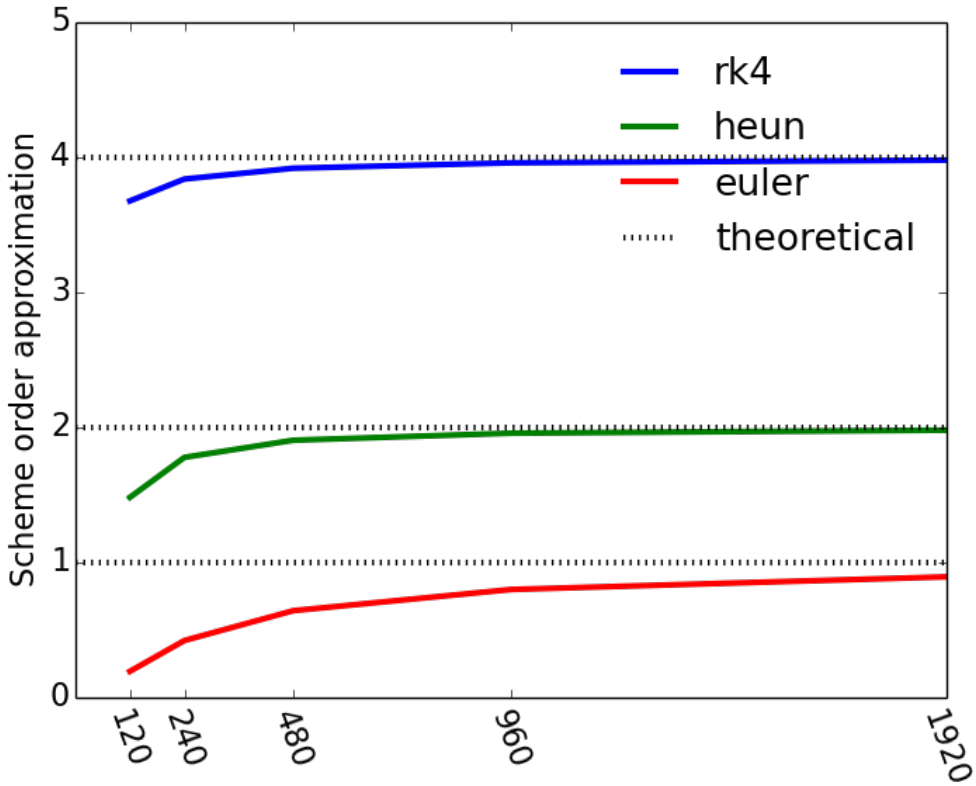


Figure 2.13: The convergence rate for the various ODE-solvers a function of the number of timesteps.

```
            log_error = log2(abs_error[1:]) # Drop 1st elt to avoid log2-problems (1st elt is zero)
            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 zero)
            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 schemes
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: euler(func,
        fanalytic = f(time) # call analytic function f to compute analytical solutions at times: time

        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: "euler":[error


    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_heun_latex, '${\
legend(LegendList, loc='best', frameon=False)
xlabel('-log(h)')
ylabel('-log($\epsilon$)')

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

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

```
# chapter1/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
font = {'size' : 10}; rc('font', **font)


# 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 ."""

    def f_np(z,t):
        """A local function to ensure that the return of func is an np array
        and to avoid lengthy code for implementation of the Heun algorithm"""
        return np.asarray(func(z,t))

    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,:] + f_np(z[i,:],t)*dt   # Predictor step
        z[i+1,:] = z[i,:] + (f_np(z[i,:],t) + f_np(zp,t+dt))*dt/2.0 # Corrector step

    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 ananlytical 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 zero)
            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 zero)
        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 schemes
    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: euler(func,
            fanalytic = f(time) # call analytic function f to compute analytical solutions at times: time

            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: "euler":[error


        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_heun_latex, '${\
    legend(LegendList, loc='best', frameon=False)
    xlabel('-log(h)')
    ylabel('-log($\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()
```

Chapter 3

Shooting Methods for Boundary Value Problems

3.1 Linear equations

Skyteteknikk er en metode til å transformere et randverdiproblem for en ODL til et ekvivalent initialverdiproblem. Betegnelsen "skyteteknikk" stammer fra problemstillingen skissert i Figure 3.1 nedenfor.

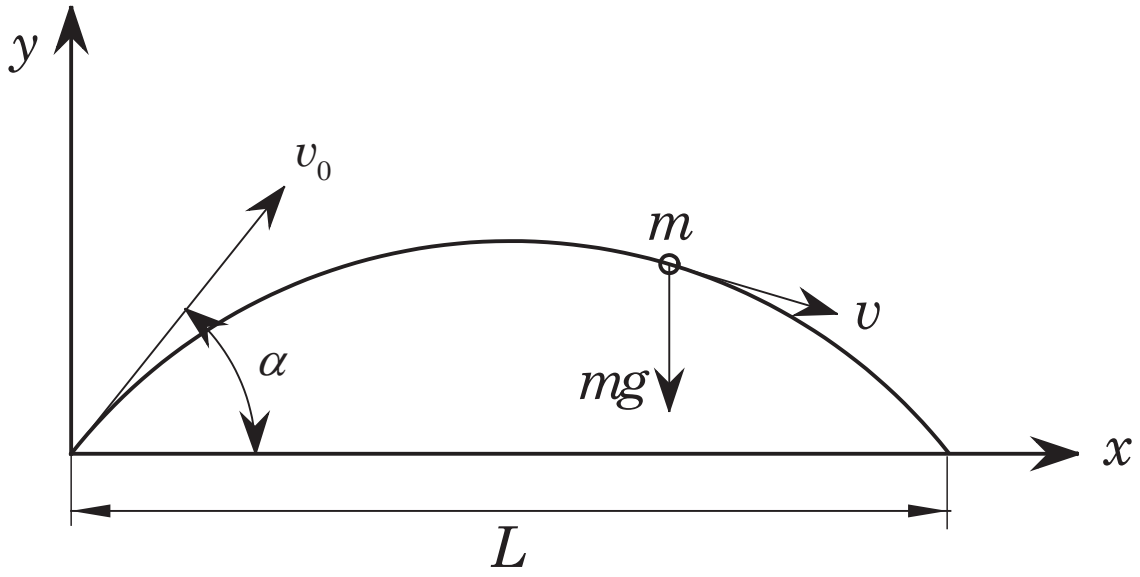


Figure 3.1:

Finn den vinkelen α som gir en gitt skytelengde $x = L$. Dette er et randverdiproblem der en betingelse er gitt for $x = 0$ og en for $x = L$. Ved å variere α, virker det innlysende at dette problemet kan løses når $L \leq L_{maks}$. Vi skal bruke tankegangen ovenfor på problemer som ikke har noe med prosjektiler å gjøre.

La oss se på et eksempel:

$$y'' = y(x) \tag{3.1}$$

med initialbetingelser:

$$y(0) = 0,\ y'(0) = s$$

(3.1) har følgende analytiske løsning:

$$y(x) = s \cdot \sinh(x) \tag{3.2}$$

Vi vil få en ny kurve $y(x)$ for hver s-verdi vi velger. Figure 3.2 viser et eksempel med $s = 0.2$ og 0.7.
Vi ønsker egentlig å løse (3.1) med følgende randbetingelser:

$$y(0) = 0,\ y(1) = 1 \tag{3.3}$$

Fra (3.2) ser vi at dette problemet kan løses ved å velge $s = s^*$ slik at $y(1) = s^* \cdot \sinh(1)$ eller

$$s^* = \frac{1}{\sinh(1)} \tag{3.4}$$

I dette tilfellet er vi istand til å finne den analytiske løsningen av både initial- og randverdiproblemet. Når dette ikke er mulig, blir fremgangsmåten i vårt tilfelle å velge verdier av s helt til betingelsen $y(1) = 1$ er oppfylt. For vilkårlige verdier av s blir da $y(1)$ en funksjon av s.

La oss se på et randverdiproblem for en 2. ordens *lineær* differensialligning:

$$y''(x) = p(x) \cdot y'(x) + q(x) \cdot y(x) + r(x) \tag{3.5}$$

med randbetingelser:

$$y(\alpha) = \alpha,\ y(\beta) = \beta \tag{3.6}$$

Skriver (3.5) som et system:

$$\begin{aligned} y'(x) &= g(x) \\ g'(x) &= p(x) \cdot g(x) + q(x) \cdot y(x) + r(x) \end{aligned} \tag{3.7}$$

med randbetingelser gitt i (3.6)

Velger randbetingelsen $y(\alpha) = \alpha$ som initialbetingelse. Trenger også en initialbetingelse for $y'(\alpha) \equiv g(\alpha)$. Denne mangler, og for å kunne løse (3.7) som et initialverdiproblem, må vi tippe en verdi for $y'(\alpha)$ slik at randbetingelsen $y(b) = \beta$ blir oppfylt. Da (3.5) er en 2. ordens lineær differensialligning, er det tilstrekkelig å beregne to verdier for $s = g(\alpha) \equiv y'(\alpha)$. Den rette verdien finnes ved lineær interpolering. Merk at $y(x)$ alltid er proposjonal med s når ligningen er lineær og homogen.

Setter

$$\phi(s) = y(b; s) - \beta \tag{3.8}$$

der

$$s = g(\alpha) \equiv y'(\alpha)$$

Den rette verdien $s = s^*$ er funnet når

$$\phi(s^*) = 0 \tag{3.9}$$

Fremgangsmåten blir: Vi tipper to verdier s^0 og s^1 og beregner de tilhørende verdiene ϕ^0 og ϕ^1 fra (3.8) ved å løse ligningsystemet i (3.7). Egentlig bør vi skrive $s^{(0)}$ og $s^{(1)}$, men bruker s^0 og s^1 når misforståelser (forhåpentligvis) ikke er mulig.

Den korrekte verdien for s finnes ved lineær interpolering som vist i Figure **??**.

$$\phi = k \cdot s + b, \qquad k = \frac{\phi^1 - \phi^0}{s^1 - s^0}, \qquad b = \frac{s^1 \cdot \phi^0 - \phi^1 \cdot s^0}{s^1 - s^0}$$

$\phi = 0$ for $s^* = -\dfrac{b}{k}$ som gir:

$$s^* = s^1 + \delta s,\ \delta s = -\phi^1 \cdot \left(\frac{s^1 - s^0}{\phi^1 - \phi^0}\right) \rightarrow s^* = \frac{\phi^1 s^0 - \phi^0 s^1}{\phi^1 - \phi^0} \tag{3.10}$$

La oss gå tilbake til eksemplet i (3.1):

$$\begin{gathered} y'' = y(x) \\ y(0) = 0,\ y(1) = 1 \end{gathered} \tag{3.11}$$

med analytisk løsning:

$$y(x) = \frac{\sinh(x)}{\sinh(1)}$$

Skriver først (3.11) som et system:

$$\begin{gathered} y'(x) = g(x) \\ g'(x) = y(x) \end{gathered} \tag{3.12}$$

Setter

$$\phi(s) = y(1; s) - 1 \tag{3.13}$$

der $s = y'(0) = g(0)$
Randbetingelser: $y(0) = 0, y(1) = 1$
Den rette verdien $s = s^*$ er funnet når (3.9) er oppfylt.
Velger $s^0 = 0.2$ og $s^1 = 0.7$.
Med disse verdiene for s og med $\Delta x = 0.1$, gir (3.11) med bruk av RK4 følgende tabell:

m	s^m	$\phi(s^m)$
0	0.2	-0.7650
1	0.7	-0.1774

m som øvre indeks brukes her og senere som nummerteller, f.eks. iterasjonsnummer.

Innsatt for tabellverdier i (3.10): $s^* = 0.8510$ Ved å bruke denne verdien for $s = y'(0)$ får vi $\phi(0.8510) = 0.0001$

Korrekt verdi: $y'(0) = \dfrac{1}{\sinh(1)} = 0.8509$

Framstillingen ovenfor er valgt fordi den lett kan generaliseres til løsning av ikke-lineære differensialligninger. Dersom vi bare er interessert i 2. ordens *lineære* ligninger, kan framstillingen forenkle. I det lineære tilfellet kan man løse følgende to startverdi-delproblemer:

$$y_0''(x) = p(x) \cdot y_0'(x) + q(x) \cdot y_0(x) + r(x) \tag{3.14}$$

$$y_0(a) = \alpha,\ y_0'(a) = 0 \tag{3.15}$$

$$\tag{3.16}$$

$$y_1''(x) = p(x) \cdot y_1'(x) + q(x) \cdot y_1(x) \tag{3.17}$$

$$y_1(a) = 0,\ y_1'(a) = 1 \tag{3.18}$$

Legg merke til at betingelsen $y'(\alpha) = 0$ i (3.15) tilsvarer $s^0 = 0$ og betingelsen $y'(\alpha) = 1$ i (3.18) tilsvarer $s^1 = 1$.

La $y_0(x)$ være løsningen av (3.14) med randbetingelsene (3.15) og $y_1(x)$ løsningen av (3.17) med randbetingelsene (3.18). Den fullstendige løsningen av randverdiproblemet i (3.5) med randbetingelsene (3.6) er da gitt ved:

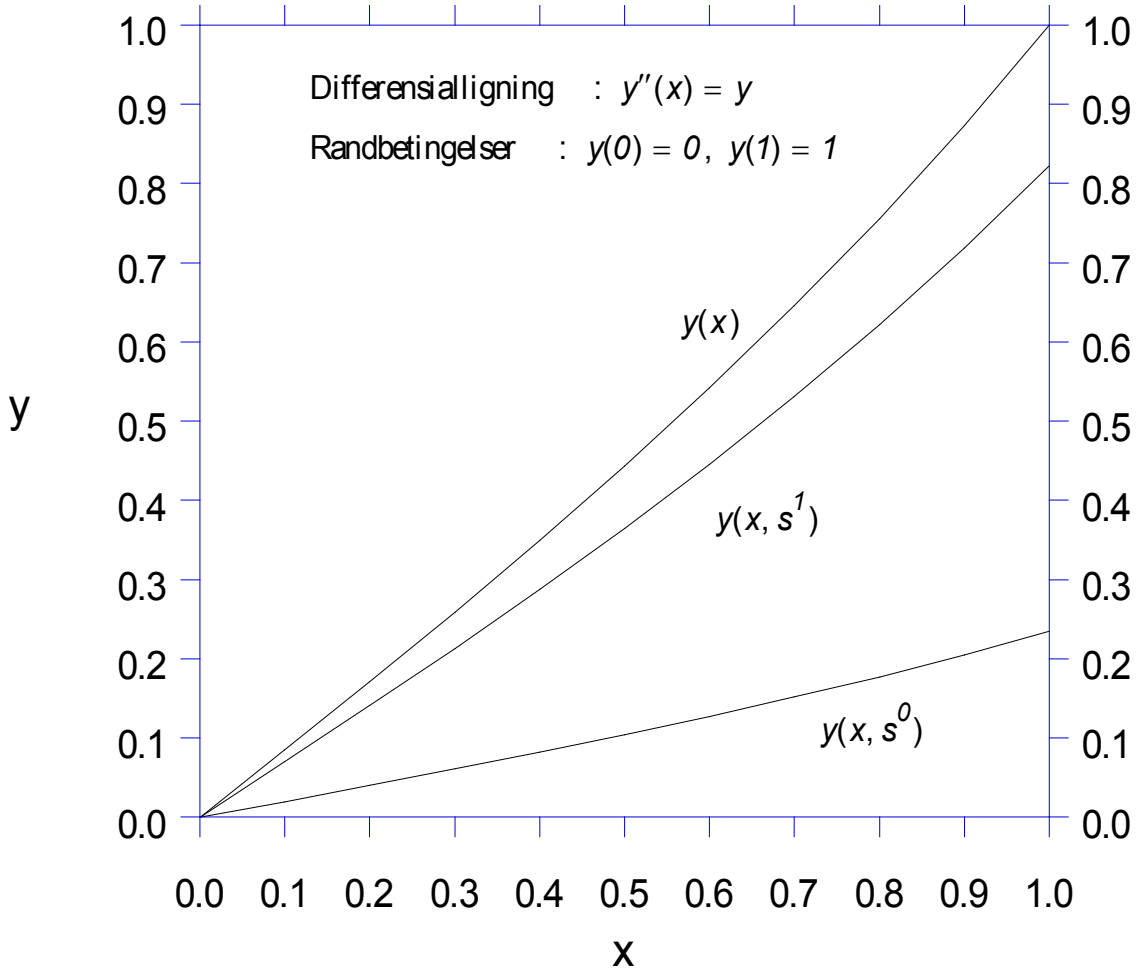


Figure 3.2:

$$y(x) = y_0(x) + \left[\frac{\beta - y_0(b)}{y_1(b)}\right] \cdot y_1(x) = y_0(x) - \left[\frac{\phi^0}{\phi^1 + \beta}\right] \cdot y_1(x) \tag{3.19}$$

med $s^0 = 0$ og $s^1 = 1$.

3.1.1 Example: Couette - Poiseuille strømning

Dette er en strømning mellom to parallelle plater. Den ene plata beveger seg mens den andre er i ro. Dessuten har vi en foreskrevet trykkgradient.

Plata ved $Y = L$ beveger seg med konstant hastighet U_0 . Hastigheten V i Y-retning $= 0$ og trykkgradienten $\frac{\delta p}{\delta x}$ er foreskrevet.

Kontinuitet: $\frac{\partial U}{\partial X} = 0$ da $V = 0$ som impliserer at $U = U(Y)$. Bevegelsesligningen i Y-retning forenkler seg til $\frac{\partial p}{\partial Y} = -\rho g$ Bevegelsesligningen i X-retning:

$$\rho U \cdot \frac{\partial U}{\partial X} = -\frac{\partial p}{\partial X} + \mu \left(\frac{\partial^2 U}{\partial X^2} + \frac{\partial^2 U}{\partial Y^2}\right) \rightarrow \frac{d^2 U}{dY^2} = \frac{1}{\mu}\frac{dp}{dX}$$

Randbetingelser: $U(0) = 0,\ U(L) = U_0$.

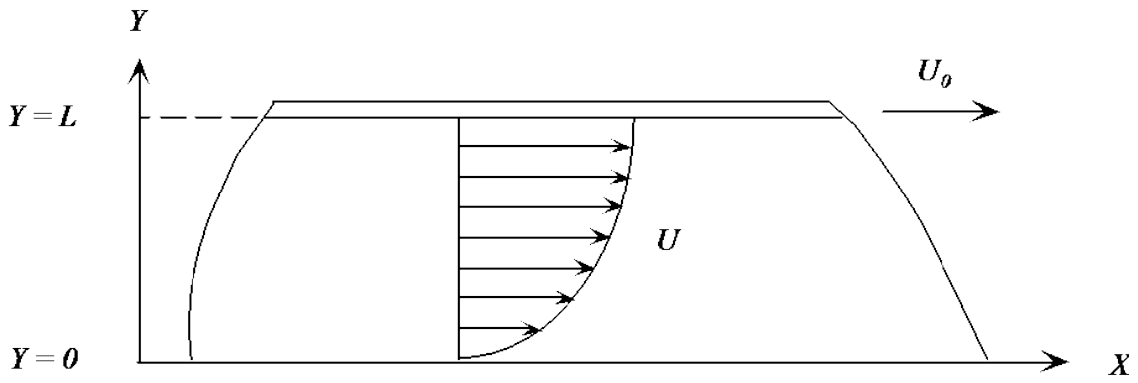


Figure 3.3:

Innfører dimensjonsløse variable: $u = \frac{U}{U_0},\ y = \frac{Y}{L},\ P = -\frac{1}{U_0}(\frac{dp}{dX})\frac{L^2}{\mu}$
Får da følgende ligning:

$$\frac{d^2u}{dy^2} = -P \tag{3.20}$$

med randbetingelser:

$$u = 0 \text{ for } y = 0,\ u = 1 \text{ for } y = 1 \tag{3.21}$$

Den analytiske løsningen av (3.20) med randbetingelser (3.21) er:

$$u = y \cdot \left[1 + \frac{P}{2}(1 - y)\right] \tag{3.22}$$

For $P \leq -2$ får vi tilbakestrømning. Figure 3.4 viser hastighetsfordelingen for forskjellige verdier av P.
Skriver (3.20) som et ligningsystem:

$$\begin{aligned} u'(y) &= u_1(y) \\ u_1'(y) &= -P \end{aligned} \tag{3.23}$$

med randbetingelser:

$$u(0) = 0,\ u(1) = 1 \tag{3.24}$$

Vi må bestemme $s = u'(0) = u_1(0)$ slik at randbetingelsen $u(1) = 1$ blir oppfylt. Dette kan uttrykkes på følgende måte:

$$\phi(s) = u(1; s) - 1, \text{ slik at} \qquad \phi(s) = 0 \text{når} \qquad s = s^*$$

Vi tipper to verdier s^0 og s^1 og beregner den rette s ved lineær interpolering, se ligning (3.10). Programmet nedenfor *Couette_Poiseuille_shoot.py* for beregning og plotting av hastighetsfordelingen for ulike trykkgradienter.

```
# chapter2/src-ch2/Couette_Poiseuille_shoot.py;ODEschemes.py @ git@lrhgit/tkt4140/allfiles/digital_compendium/chap

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
```

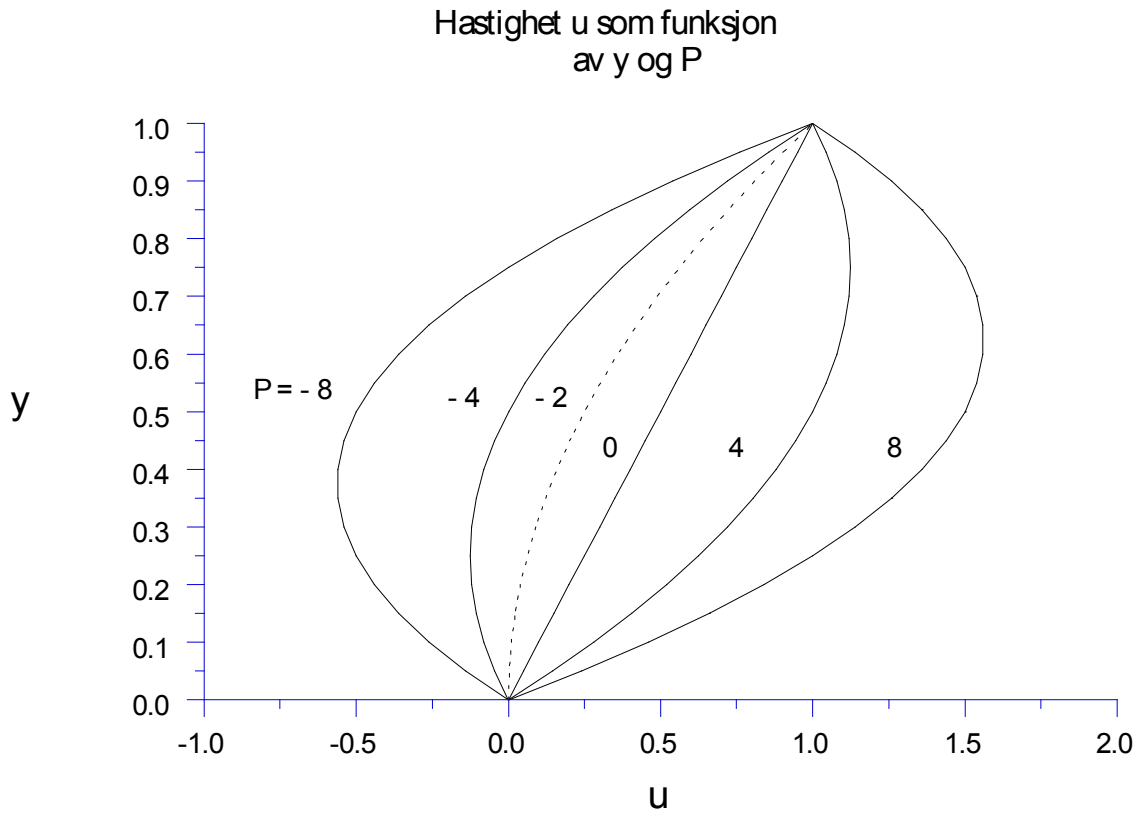


Figure 3.4:

```
rcParams['lines.linewidth'] = LNWDT; rcParams['font.size'] = FNT
font = {'size' : 16}; rc('font', **font)


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: Bjelkesøyle med konstant tverrsnitt

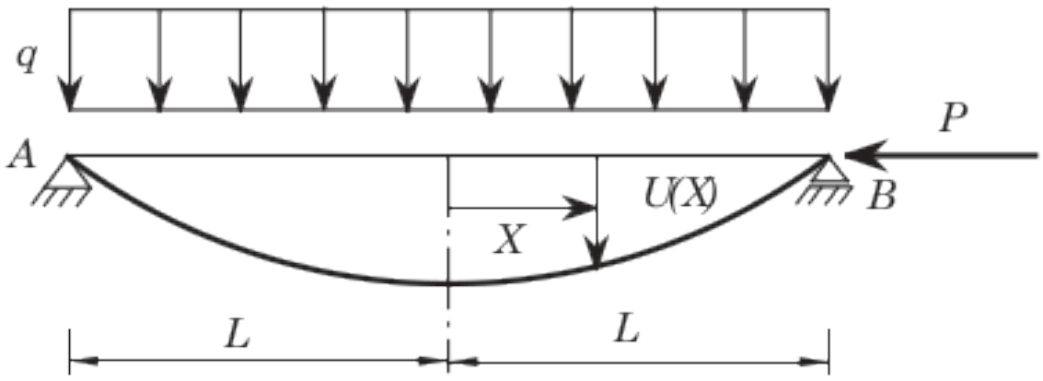


Figure 3.5:

Figuren viser en bjelkesøyle påkjent av en jevnt fordelt last q pr. lengdeenhet samt en horisontal kraft P. Differensial-ligningen for utbøyningen $U(X)$ er gitt ved:

$$\frac{d^2U}{dX^2} + \frac{P}{EI}U = -\frac{q}{2EI}(L^2 - X^2),\ P > 0 \tag{3.25}$$

Randbetingelser:

$$U(-L) = U(L) = 0 \tag{3.26}$$

EI er bjelkestivheten. Antar små utbøyninger. (Lineær ligning).
Alternativt kan vi benytte $\frac{dU}{dX}(0) = 0$ p.g.a symmtrien.
Dimensjonsløse variable:

$$x = \frac{x}{L},\ u = \frac{P}{qL^2} \cdot U,\ \theta^2 = \frac{PL^2}{EI} \tag{3.27}$$

som innført i (3.25) og (3.26) gir:

$$\frac{d^2u}{dx^2} + \theta^2 \cdot u = \theta^2 \frac{(1-x^2)}{2},\ -1 < x < 1 \tag{3.28}$$

med randbetingelser:

$$u(-1) = 0, \quad u(1) = 0 \tag{3.29}$$

(3.28) med randbetingelsene (3.29) kan løses analytisk:

$$u(x) = \frac{1}{\theta^2} \cdot \left[\frac{\cos(\theta x)}{\cos(\theta)} - 1\right] - \frac{(1-x^2)}{2} \tag{3.30}$$

Knekklasta for dette tilfellet er gitt ved $P_k = \dfrac{\pi EI}{4L^2}$ slik at

$$0 \leq \theta \leq \frac{\pi}{2} \tag{3.31}$$

Numerisk løsning

Vi ønsker å løse (3.28) ved bruk av skyteteknikk. Velger nå å bruke fremgangsmåten i forbindelse med lign. (3.19). Må da løse følgende to system:

$$\begin{gathered} u_0''(x) = -\theta^2 \cdot \left[u_0(x) + \tfrac{(1-x^2)}{2}\right] \\ u_0(-1) = 0, \quad u_0'(-1) = 0 \end{gathered} \tag{3.32}$$

$$\begin{gathered} u_1''(x) = -\theta^2 \cdot u_1(x) \\ u_1(-1) = 0, \quad u_1'(-1) = 1 \end{gathered} \tag{3.33}$$

Den fullstendige løsningen er da gitt ved:

$$u(x) = u_0(x) - \frac{u_0(1)}{u_1(1)} \cdot u_1(x) \tag{3.34}$$

Skriver (3.32) og (3.33) som ligningsystem.

System 1 Med $u_0 = y_1$ og $u_o' = y_2$:

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= -\theta^2 \cdot \left[y_1 + \tfrac{(1-x^2)}{2}\right] \end{aligned} \tag{3.35}$$

med startbetingelser

$$y_1(-1) = 0,\ y_2(-1) = 0 \tag{3.36}$$

System 2 Med $u_1 = y_1$ og $u_1' = y_2$:

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= -\theta^2 y_1 \end{aligned} \tag{3.37}$$

med startbetingelser

$$y_1(-1) = 0,\ y_2(-1) = 1 \tag{3.38}$$

Begge system integreres fra til $x = -1$ til $x = 1$ slik at $u_0(1)$ og $u_1(1)$ kan finnes. Den fullstendige løsningen er da gitt av (3.34). Ved hjelp av RK4, og lastparameteren $\theta = 1$ oppnås resultatene i tabellen

under. Intervallet er delt i 20 deler og tabellen viser at overenstemmelsen er god mellom numerisk og analytisk løsning. Dessuten ser vi at symmetribetingelsen er oppfylt.

Marie 1: I kompendiet henvises det til et matlab-program, beamcol. Foreløpig finnes ikke et tilsvarende python-program.

x	$u_{comput.}$	$u_{analyt.}$
-1.000	0.0000e+000	0.0000e+000
-0.900	5.5485e-002	5.5485e-002
-0.800	1.0948e-001	1.0948e-001
-0.700	1.6058e-001	1.6058e-001
-0.600	2.0754e-001	2.0754e-001
-0.500	2.4924e-001	2.4924e-001
-0.400	2.8471e-001	2.8471e-001
-0.300	3.1315e-001	3.1315e-001
-0.200	3.3392e-001	3.3392e-001
-0.100	3.4657e-001	3.4657e-001
0.000	3.5081e-001	3.5082e-001
0.100	3.4657e-001	3.4657e-001
0.200	3.3392e-001	3.3392e-001
0.300	3.1315e-001	3.1315e-001
0.400	2.8471e-001	2.8471e-001
0.500	2.4924e-001	2.4924e-001
0.600	2.0754e-001	2.0754e-001
0.700	1.6058e-001	1.6058e-001
0.800	1.0948e-001	1.0948e-001
0.900	5.5485e-002	5.5485e-002
1.000	0.0000e+000	0.0000e+000

3.1.3 Example: Bjelkesøyler med variabelt tverrsnitt

Vi henviser til samme figuren som i forrige eksempel (3.1.2). Forskjellen er at nå lar vi bjelken ha variabelt tverrsnitt der 2. arealmoment I er funksjon av X. Med I_0 som 2. arealmomet for $X = 0$, lar vi I variere på følgende måte:

$$I(X) = \frac{I_0}{1 + (X/L)^n}, \ n = 2, 4, 6, \ldots \tag{3.39}$$

Dersom vi for eksempel betrakter et rektangulært tverrsnitt med høyde h og konstant bredde b, vil høyden h variere som følger:

$$h(X) = \frac{h_0}{[1 + (X/L)^n]^{1/3}} \tag{3.40}$$

der h_0 er høyden for $X = 0$. Fra lign. (3.25)) i eksempel (3.1.2):

$$\frac{d^2U}{dX^2} + \frac{P}{EI}U = -\frac{q}{2EI}(L^2 - X^2) \tag{3.41}$$

$$U(-L) = U(L) = 0, \ \frac{dU(0)}{dX} = 0 \tag{3.42}$$

La oss først beregne momentfordelingen $M(X)$ i bjelken.
Nå har vi:

$$\frac{d^2U}{dX^2} = -\frac{M}{EI} \tag{3.43}$$

Ved å bruke (3.43) i (3.41) og derivere to ganger, får vi:

$$\frac{d^2M}{dX^2} + \frac{P}{EI}M = -q \tag{3.44}$$

Innfører dimensjonsløse variable

$$x = \frac{X}{L},\ m = \frac{M}{qL^2} \text{ og setter } P = \frac{EI_0}{L^2} \tag{3.45}$$

slik at (3.44) blir:

$$m''(x) + (1 + x^n) \cdot m(x) = -1 \tag{3.46}$$

med randbetingelser:

$$m(-1) = m(1) = 0,\ m'(0) = 0 \tag{3.47}$$

I motsetning til ligningene i forrige eksempel (3.1.2), finnes det ikke en brukbar analytisk løsning av (3.46), selv om ligningen er lineær.

Dersom du f. eks. har programmet *Maple* for hånden, kan du forsøke følgende:

```
eq:= diff(m(x),x,x) + (1+x^2)*m(x) +1;
dsolve(eq,m(x));
```

Når momentfordelingen er funnet, finnes nedbøyningen $u(x)$ fra:

$$u(x) = m(x) - \frac{1}{2}(1 - x^2) \tag{3.48}$$

Den dimensjonelle nedbøyningen U er gitt ved:

$$U = \frac{qL^4}{EI_0}u \tag{3.49}$$

Ved å sette (3.48) inn i (3.46), finner vi diff.ligningen for $u(x)$:

$$u''(x) + (1 + x^n) \cdot u(x) = -\frac{1}{2}(1 - x^2) \cdot (1 + x^n) \tag{3.50}$$

Ved å benytte oss av symmetrien, løser vi nå (3.46) med randbetingelsene $m(1) = 0,\ m'(0) = 0$ ved bruk av skyteteknikk, og beregner deretter $u(x)$ fra (3.48). Nå bruker vi **ode45** som løser istedenfor **rk4c**. **Marie 2**: Henvisninger til Matlab

Skriver (3.46) som et system med $m(x) = y_1(x)$ og $m'(x) = y_2(x)$:

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= -[1 + (1 + x^n) \cdot y_1] \end{aligned} \tag{3.51}$$

Randbetingelser:

$$y_2(0) = y_1(1) = 0 \tag{3.52}$$

Da $y_2(0) \equiv m'(0)$ er gitt, må vi tippe $m(0) \equiv y_1(0)$ slik at betingelsen $m(1) \equiv y_1(1) = 0$ blir oppfylt. Eller uttrykt på standard-måten:

Velger $s = y_1(0)$ og setter $\phi(s) = y_1(1; s)$. Bestemmer $s = s^*$ slik at betingelsen $\phi(s^*) \equiv y_1(1; s^*) = 0$ blir oppfylt. Da ligningen er lineær, finnes s^* fra (3.10):

$$s^* = \frac{\phi^1 s^0 - \phi^0 s^1}{\phi^1 - \phi^0} \tag{3.53}$$

Ved å velge $s^0 = 0$ og $s^1 = 1$:

$$s^* = \frac{\phi^0}{\phi^0 - \phi^1} \tag{3.54}$$

Vær klar over at valget i (3.54) ikke alltid er fornuftig selv om det er rett i prinsippet når ligningen er lineær. Dersom løsningen er av typen $e^{\alpha x}$ der α er stor samtidig som x er stor, kan vi risikere å komme utenfor tallområdet selv for dobbelpresisjon som er ca 10^{308}. I tilfellet ovenfor er ikke dette noe problem, slik at (3.54) kan brukes. Beregningen er utført av programmet **momentdis** **Marie 3**: Henvisning til Matlab-program med tabell og plott nedenfor av både momentet og forskyvningen for $n = 2$.

x	**m**	**u**
0.000	9.3205e-001	4.3205e-001
0.050	9.2964e-001	4.3089e-001
0.100	9.2239e-001	4.2739e-001
0.150	9.1032e-001	4.2157e-001
0.200	8.9342e-001	4.1342e-001
0.250	8.7170e-001	4.0295e-001
0.300	8.4516e-001	3.9016e-001
0.350	8.1382e-001	3.7507e-001
0.400	7.7769e-001	3.5769e-001
0.450	7.3681e-001	3.3806e-001
0.500	6.9122e-001	3.1622e-001
0.550	6.4097e-001	2.9222e-001
0.600	5.8614e-001	2.6614e-001
0.650	5.2681e-001	2.3806e-001
0.700	4.6311e-001	2.0811e-001
0.750	3.9519e-001	1.7644e-001
0.800	3.2323e-001	1.4323e-001
0.850	2.4745e-001	1.0870e-001
0.900	1.6810e-001	7.3104e-002
0.950	8.5503e-002	3.6753e-002
1.000	7.7259e-012	7.7259e-012

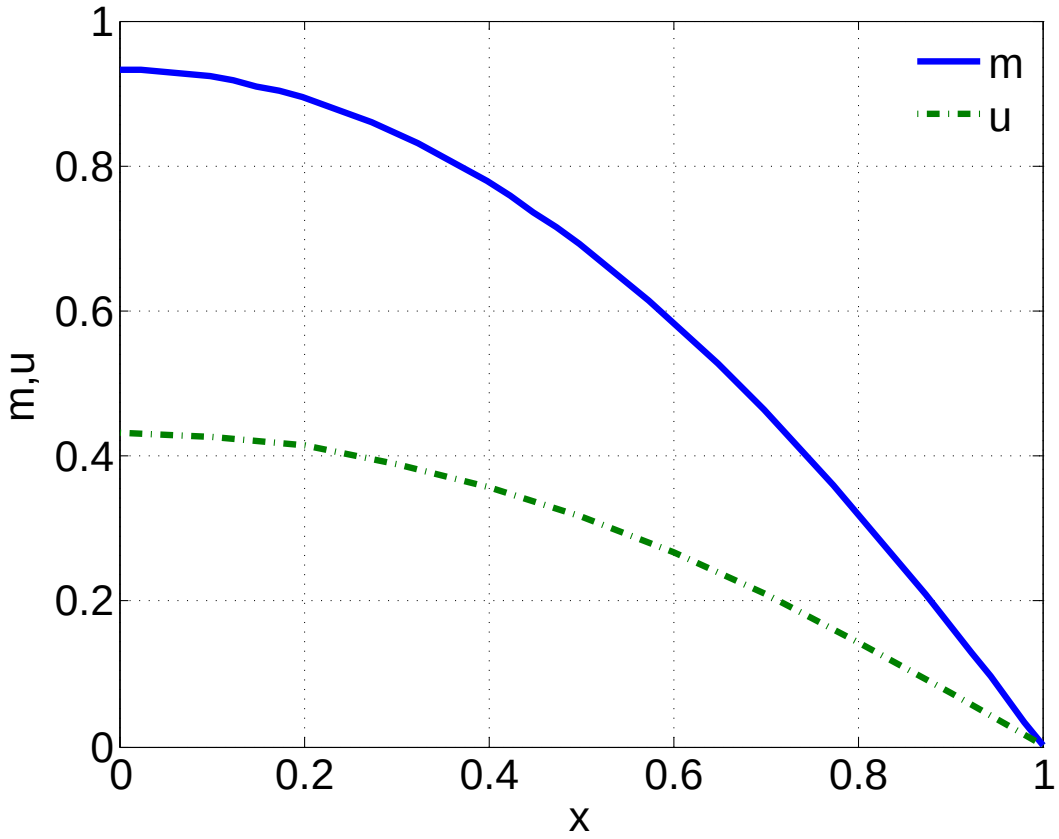


Figure 3.6: Moment m og nedbøyning u.

3.2 Ikke-lineære ligninger

Gitt randverdiproblemet

$$\begin{aligned} y''(x) &= \tfrac{3}{2}y^2 \\ y(0) = 4,\ & y(1) = 1 \end{aligned} \tag{3.55}$$

(3.55) har to løsninger som tilfredstiller de gitte randbetingelsene:
Løsning I:

$$y_I = \frac{4}{(1+x)^2} \tag{3.56}$$

Løsning II kan uttrykkes ved elliptiske Jacobi-funksjoner; se appendiks G del G.3 i kompendiet . Begge løsningene er tegnet i Figure 3.7 . Skriver (3.55) som et ligningsystem med $y \to y_1$ og $y' = y_1' = y_2$:

$$\begin{aligned} y_1'(x) &= y_2(x) \\ y_2'(x) &= \tfrac{3}{2}[y_1(x)]^2 \end{aligned} \tag{3.57}$$

Randbetingelser:

$$y_1(0) = 4,\ y_1(1) = 1 \tag{3.58}$$

Vi bestemmer $s = y'(0) = y_2(0)$ slik at randbetingelsen $y_1(1) = 1$ blir oppfylt. Setter følgelig:

$$\phi(s^m) = y_1(1; s^m) - 1, \ m = 0, 1, \ldots \text{slik at } y_1(1) \to 1 \text{ for } \phi(s^m) \to 0, \ m \to \infty \tag{3.59}$$

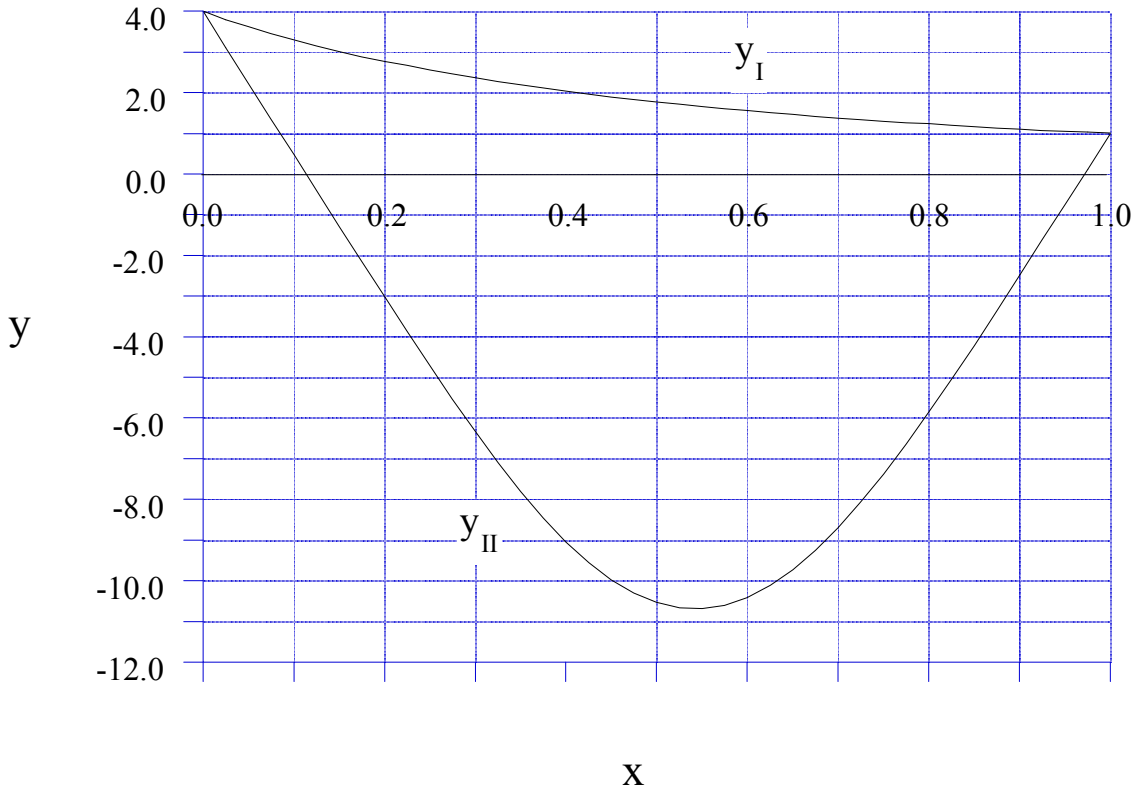


Figure 3.7: De to løsningene y_I og y_{II} av differensialligningen (3.55)

Da ligningen er ikke-lineær, må vi utføre en iterasjonsprosess som indikert i (3.59). Oppgava blir da å bestemme nullpunktet(ene) s^* i funksjonen $\phi(s) = 0$. Vi velger å bruke *sekantmetoden* for dette formålet. (Se C&K [4], avsnitt 3.3) Iterasjonsprosessen vist i Figure **??** starter med at vi tipper to verdier s^0 og s^1 og legger en sekant gjennom $\phi(s^0)$ og $\phi(s^1)$. Skjæringspunktet med s-aksen, $s^{(2)}$, kan da beregnes.

For en vilkårlig iterasjon m:

$$s^{m+1} = s^m + \delta s \tag{3.60}$$

der

$$\delta s = s^{m+1} - s^m = -\phi(s^m) \cdot \left[\frac{s^m - s^{m-1}}{\phi(s^m) - \phi(s^{m-1})} \right], \ m = 1, 2, \ldots$$

Iterasjonsprosessen blir da som følger når $\phi(s^{m-1})$ og s^m antas kjent:

1. Beregn $\phi(s^{m-1})$ og $\phi(s^m)$ ved å løse (3.57) sammen med (3.59).
2. Beregn δs og s^{m+1} fra (3.60)
3. Sett

- $s^{m-1} \leftarrow s^m$
- $s^m \leftarrow s^{m+1}$
- $\phi(s^{m-1}) \leftarrow \phi(s^m)$

Gjenta skritt 1–3 inntil vi har fått konvergens.

Eksempel på konvergenskriterier :

Kontroll av absolutt feil:

$$|\delta s| < \varepsilon_1 \tag{3.61}$$

Kontroll av relativ feil:

$$\left| \frac{\delta s}{s^{m+1}} \right| < \varepsilon_2 \tag{3.62}$$

(3.61) og (3.62) kombineres ofte med:

$$|\phi(s^{m+1})| < \varepsilon_3 \tag{3.63}$$

Vi bruker nå den foreskrevne prosessen til å løse (3.57). Velger startverdier $s^0 = -3.0$ og $s^1 = -6.0$. Figure 3.7 viser at vi sannsynligvis vil finne løsningen y_1 med disse startverdiene. Velger $\Delta x = 0.1$ og bruker RK4. Får følgende tabell:

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

Etter fire iterasjoner er $s = y'(0) = -7.9989$, mens den analytiske verdien er -8.0. Programmet *non_lin_ode.py* viser hvordan problemet kan løses, og sammeligner grafisk den numeriske løsningen med den analytiske.

```
# chapter2/src-ch2/non_lin_ode.py;ODEschemes.py @ git@lrhgit/tkt4140/allfiles/digital_compendium/chapter2/src-ch2/

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)


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=[-34.0,-20]

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()
```

Sekantmetoden er enkel og effektiv. Vi slipper dessuten å finne det analytiske uttrykket(ene) for den(de) deriverte som i Newton-Raphsons metode. Bakdelen er at vi må ha to startverdier for å komme igang. Med kjennskap til fysikken i problemet, er dette vanligvis en overkommelig vanskelighet.

3.2.1 Example: Stor nedbøyning av kragbjelke

Figure 3.8 viser en kragbjelke som er fast innspent ved A og påkjent av en vertikal last P på enden B. Vi vil tillate store nedbøyninger og bruker derfor buelengden l og helningsvinkelen θ som variable. Alle lengder er gjort dimensjonsløse ved divisjon med bjelkelengden L. Buelengden l går derfor fra $l = 0$ i A til $l = 1$ i B. Differensialligningen for den elastiske linja er gitt ved:

$$\kappa = \frac{d\theta}{L \cdot dl} = \frac{M}{EI} \tag{3.64}$$

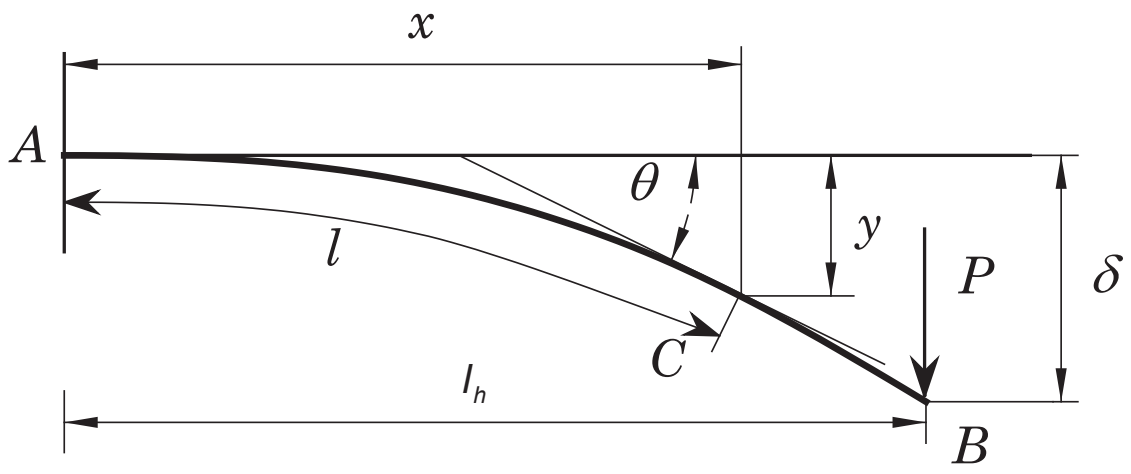


Figure 3.8:

der κ er krumningen, M er momentet og EI bjelkestivheten. Tar momentet om C: $M = P \cdot L(l_h - x)$ som innsatt i (3.64) gir:

$$\frac{d\theta}{dl} = \frac{PL^2}{EI}(l_h - x) \tag{3.65}$$

Fra figuren finner vi følgende geometriske relasjoner:

$$\frac{dx}{dl} = \cos\theta, \ \frac{dy}{dl} = \sin\theta \tag{3.66}$$

Deriverer (3.65) m.h.p. l og setter inn fra (3.66):

$$\frac{d^2\theta}{dl^2} + \frac{PL^2}{EI}\cos\theta = 0 \tag{3.67}$$

Innfører parameteren α definert ved:

$$\alpha^2 = \frac{PL^2}{EI} \tag{3.68}$$

Vi må da løse følgende differensialligninger:

$$\frac{d^2\theta}{dl^2} + \alpha^2\cos\theta = 0 \tag{3.69}$$

$$\frac{dy}{dl} = \sin\theta \tag{3.70}$$

med følgende randbetingelser:

$$y(0) = 0, \ \theta(0) = 0, \ \frac{d\theta}{dl}(1) = 0 \tag{3.71}$$

De to første randbetingelsene betyr at bjelken er fast innspent i A, mens den siste betyr at momentet forsvinner i B. Den analytiske løsningen av dette problemet er gitt i appendiks G, del G.3 i kompendiet.

Numerisk løsning
Skriver (3.69) og (3.70) som et system av første ordens ligninger. Med $\theta = z_1, \ \theta' = z_2$ og $y = z_3$ får vi:

$$z_1' = z_2 \tag{3.72}$$
$$z_2' = -\alpha^2 \cos z_1 \tag{3.73}$$
$$z_3' = \sin z_1 \tag{3.74}$$

Randbetingelser:

$$z_1(0) = 0,\ z_2(1) = 0,\ z_3(0) = 0 \tag{3.75}$$

Vi må tippe $\theta'(0)$ slik at betingelsen $\frac{d\theta}{dl}(1) = 0$ blir oppfylt. Setter $s = \theta'(0)$ og $\phi(s) = \theta'(1; s)$. Må da bestemme $s = s^*$ slik at $\phi(s^*) = 0$. eller med z-variable: $s = z_2(0)$. Bestem $s = s^*$ slik at $\phi(s^*) = z_2(1; s^*) = 0$ Dessuten finner vi:

$$l_h = \frac{s^*}{\alpha^2} \tag{3.76}$$

Bruker sekantmetoden og må da bestemme startverdier s^0 og s^1 for å starte iterasjonsprosessen. Dette gjøres ved å fremstille funksjonen $\phi(s)$ grafisk. Dette kan gjøres med f.eks Python- programmet *phi_plot_beam_deflect_shoot* , som resulterer i Figure 3.9.

```
# chapter2/src-ch2/phi_plot_beam_shoot.py;ODEschemes.py @ git@lrhgit/tkt4140/allfiles/digital_compendium/chapter2/

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)


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()
```

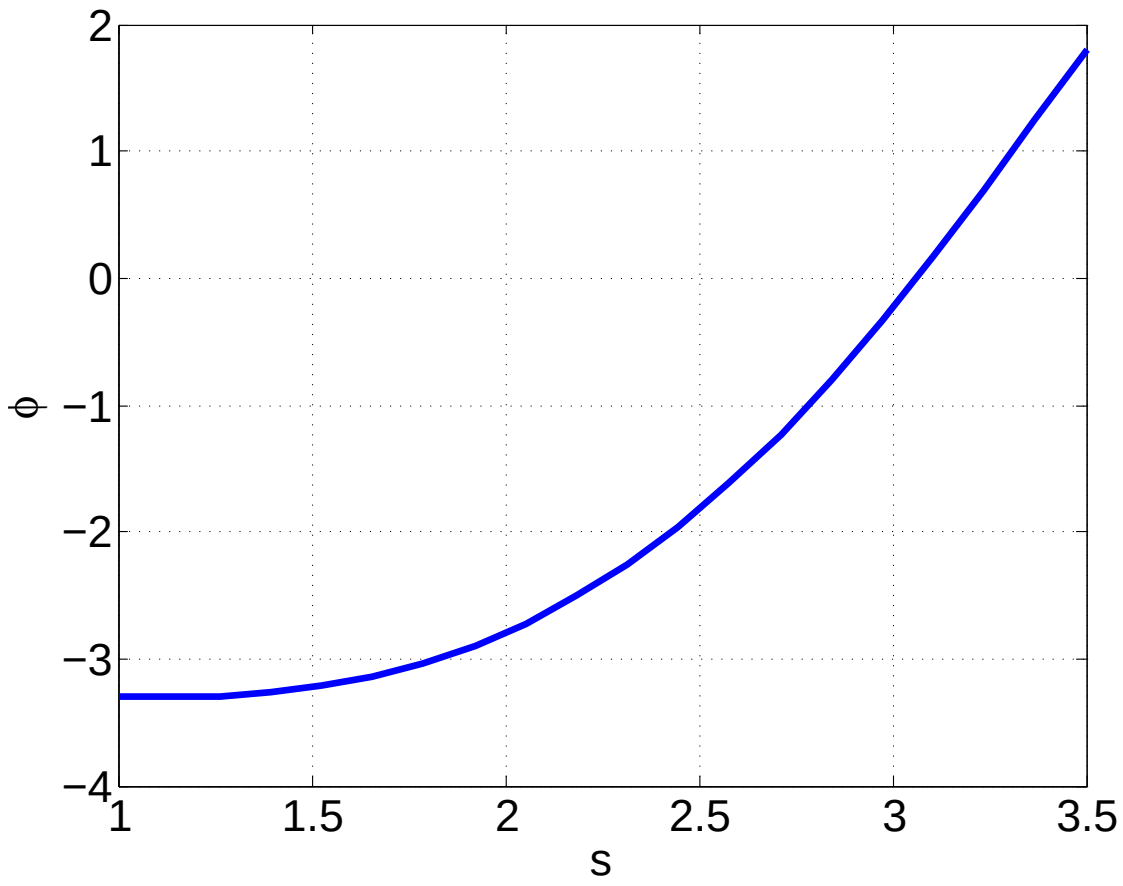


Figure 3.9: Plot av funksjonen $\phi(s)$ for $\alpha^2 = 5$ for identifikasjon av nullpunkt

Vi ser av figuren at vi har et nullpunkt $s^* \approx 3.05$. Merk at vi vil få forskjellige nullpunkt når α forandres. Figure 3.10 viser endenedbøyningen δ og horisontallengden l_h som funksjon av α^2. Figuren er beregnet med Python-programmet *beam_deflect_shoot*.

```
# chapter2/src-ch2/beam_deflect_shoot.py;ODEschemes.py @ git@lrhgit/tkt4140/allfiles/digital_compendium/chapter2/s

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)
```

```
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('theta')

plot(y,z[:,1])
legends.append('dtheta/dl')

plot(y,z[:,2])
legends.append('deflection y')



# Add the labels
legend(legends,loc='best',frameon=False) # Add the legends
ylabel('theta theta')
xlabel('y/L')
#grid(b=True, which='both', axis='both',linestyle='-')
grid(b=True, which='both', color='0.65',linestyle='-')

show()
```

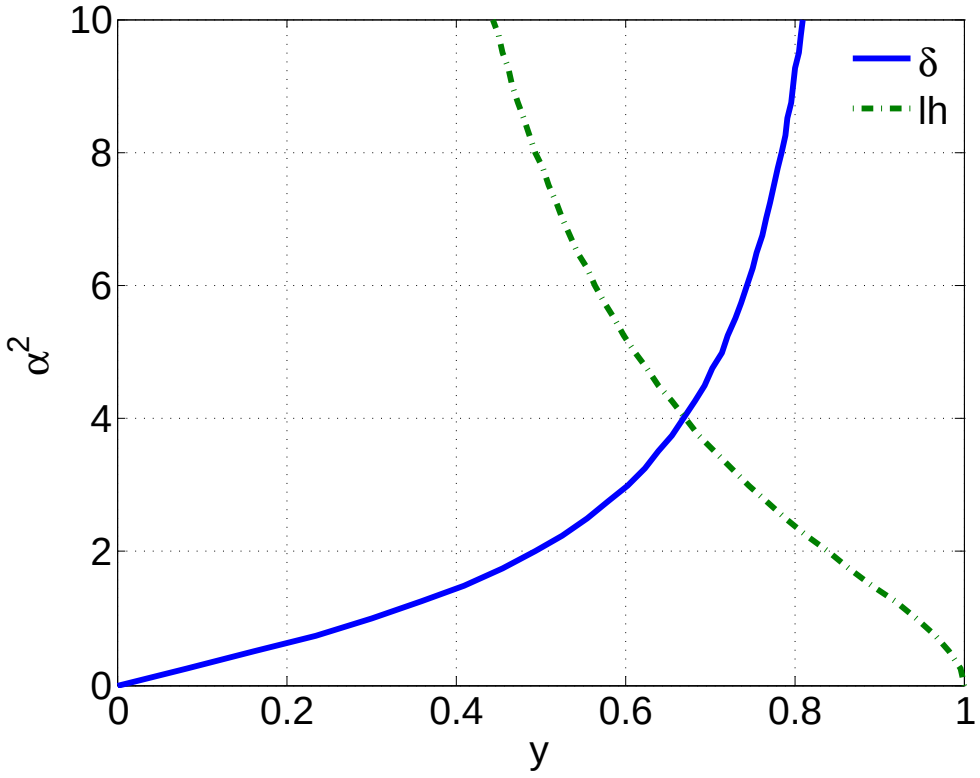


Figure 3.10: Bjelke med stor nedbøying (δ) og endring i horisontal lengde (l_h)

3.3 Litt om likedannhetsløsninger

Den endimensjonale, ikke-stasjonære varmeledningsligningen er gitt ved:

$$\frac{\partial T}{\partial \tau} = \alpha \frac{\partial^2 T}{\partial X^2} \tag{3.77}$$

der τ er tiden, $T = T(X, \tau)$ og α den termiske diffusiviteten. (Se appendiks B i kompendiet for utledning)

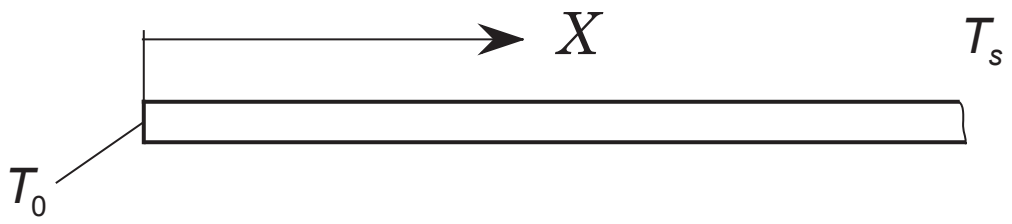


Figure 3.11:

Figure 3.11 viser en stav med halvuendelig utstrekning der $0 \leq X < \infty$. Opprinnelig har staven en temperatur T_s, men ved tiden $\tau = 0$ forandres temperaturen på enden $X = 0$ til T_0 og holdes deretter konstant. Vi ønsker å beregne temperaturfordelingen i staven som funksjon av tiden τ.

Problemet kan beskrives matematisk på følgende måte:
Initialbetingelse:

$$T(X,\tau) = T_s,\ \tau < 0 \tag{3.78}$$

Randbetingelser:

$$T(0,\tau) = T_0,\ T(\infty,\tau) = T_s \tag{3.79}$$

La oss skalere (3.77) ved å innføre følgende dimensjonsløse variable:

$$u = \frac{T - T_0}{T_s - T_0},\ x = \frac{X}{L},\ t = \frac{\tau \cdot \alpha}{L^2} \tag{3.80}$$

der L er en karakteristisk lengde. (3.80) innsatt i (3.77) gir følgende ligning:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2},\ 0 < x < \infty \tag{3.81}$$

Initialbetingelse:

$$u(x,t) = 1,\ t < 0 \tag{3.82}$$

Randbetingelser:

$$u(0,t) = 0,\ u(\infty,t) = 1 \tag{3.83}$$

Det spesielle valget av t i (3.80) er gjort for å fjerne koeffisienten foran det siste leddet i ligningen. Vi har dermed fått redusert antall variable. Det finnes mange analytiske løsninger av diffusjonsligningen på formen gitt i (3.81). Disse løsningene kan derfor brukes på en hel rekke problemer fra forskjellige anvendelsesområder. Den dimensjonsløse tiden gitt i (3.80) er en dimensjonsløs gruppe som kalles Fourier-tallet og skrives vanligvis Fo. Vi vil nå forsøke å transformere (3.81) og (3.82) fra en partiell til en ordinær differensialligning.
Setter:

$$\bar{x} = a \cdot x \text{ og } \bar{t} = b \cdot t \tag{3.84}$$

der a og b er positive konstanter.
(3.84) innført i (3.81) gir følgende ligning:

$$\frac{\partial u}{\partial \bar{t}} = \frac{a^2}{b}\frac{\partial^2 u}{\partial \bar{x}^2} \tag{3.85}$$

Velger $b = a^2$ slik at (3.85) og (3.81) er identiske med samme randbetingelser:

$$u(x,t) = u(\bar{x},\bar{t}) = u(ax, a^2 t),\ \text{med } b = a^2 \tag{3.86}$$

Dersom (3.86) skal være uavhengig av $a > 0$, må $u(x,t)$ skrives på formen

$$u(x,t) = f\left(\frac{x}{\sqrt{t}}\right),\ g\left(\frac{x^2}{t}\right),\ \text{osv.} \tag{3.87}$$

Velger det første alternativet:

$$u(x,t) = f\left(\frac{x}{\sqrt{t}}\right) \tag{3.88}$$

Innfører følgelig en ny variabel η definert ved:

$$\eta = \frac{x}{2\sqrt{t}} \tag{3.89}$$

der faktoren 2 er innført bare for å få et penere sluttresultat.

Transformasjonen gitt i (3.89) reduserer den opprinnelige ligningen fra å være avhengig både av x og t til bare å avhenge av en variabel η. Som funksjon av η, vil ett og samme u-profil gi løsningen for alle x og t. $u(\eta)$ kalles derfor en likedannhetsløsning og (3.89) en likedannhets-transformasjon. (Engelsk: similarity transformation). Transformasjonen i (3.89) kalles ofte Boltzmann-transformasjonen.

Problemet som opprinnelig ble beskrevet av en partiell differensialligning, beskrives nå av en ordinær differensialligning.

Har da fått følgende variable:

$$t = \frac{\tau \cdot \alpha}{L^2},\ \eta = \frac{x}{2\sqrt{t}} = \frac{X}{2\sqrt{\tau\alpha}} \tag{3.90}$$

La oss nå løse (3.81) analytisk.

Innfører

$$u = f(\eta) \tag{3.91}$$

med randbetingelsene:

$$f(0) = 0,\ f(\infty) = 1 \tag{3.92}$$

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial \eta}\left(\frac{\partial n}{\partial t}\right) = f'(\eta)\cdot\left(-\frac{y}{4t\sqrt{t}}\right) = -f'(\eta)\frac{\eta}{2t}$$
$$\frac{\partial u}{\partial x} = \frac{\partial u}{\partial \eta}\left(\frac{\partial \eta}{\partial x}\right) = f'(\eta)\frac{1}{2\sqrt{t}},\ \frac{\partial^2 u}{\partial x^2} = \frac{\partial}{\partial x}\left(f'(\eta)\frac{1}{2\sqrt{t}}\right) = f''(\eta)\frac{1}{4t}$$

Innsatt for $\dfrac{\partial u}{\partial t}$ og $\dfrac{\partial^2 u}{\partial x^2}$ i (3.81):

$$f''(\eta)\frac{1}{4t} + f'(\eta)\frac{\eta}{2t} = 0 \rightarrow f''(\eta) + 2\eta f'(\eta) = 0 \tag{3.93}$$

(3.93) kan løses ved direkte integrasjon:

$$\frac{f''(\eta)}{f'(\eta)} = -2\eta \rightarrow \ln f'(\eta) = -\eta^2 + \ln C_1$$

Som gir

$$f'(\eta) = C_1 e^{-\eta^2}$$

Ny integrasjon:

$$f(\eta) = C_1 \int_0^{\eta} e^{-t^2} dt$$

hvor vi har benyttet randbetingelsen $f(0) = 0$ fra (3.92).

Fra integraltabell:

$$\int_0^x e^{-t^2} dt = \frac{\sqrt{\pi}}{2}\text{erf(x)}$$

der feilfunksjonen erf(x) er definert ved:

$$\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \tag{3.94}$$

Dette gir:

$$f(\eta) = C_1 \frac{\sqrt{\pi}}{2} \text{erf}(\eta) \tag{3.95}$$

erf(x) $\to 1$ for $x \to \infty$ gir: $C_1 = \frac{2}{\sqrt{\pi}}$ og videre innsatt i (3.95):

$$u(x,t) = \text{erf}(\eta) = \text{erf}\left(\frac{x}{2\sqrt{t}}\right) \tag{3.96}$$

Dersom vi ønsker $u(x,t)$ uttrykt i de opprinnelige variable, får vi fra (3.80):

$$\frac{T(X,\tau) - T_0}{T_s - T_0} = \text{erf}\left(\frac{X}{2\sqrt{\tau \cdot \alpha}}\right) \tag{3.97}$$

Noen tabellverdier for erf(x) er vist under:

x	erf(x)	x	erf(x)
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

Figure 3.12 nedenfor viser feilfunksjonen erf(x) samt den komplementære feilfunksjonen erfc(x) som er definert ved:

$$\text{erfc}(x) = 1 - \text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt \tag{3.98}$$

3.3.1 Example: Frysing i vannledning

Et tørt jordlag med opprinnelig temperatur $20°C$, blir i en kuldeperiode utsatt for en overflatetemperatur på $-15°C$ i 30 døgn. Hvor dypt må vi legge en vannledning for å unngå at den fryser ? Den termiske diffusiviteten $\alpha = 5 \cdot 10^{-4}\text{m}^2/\text{time}$.

Med henvisning til (3.78) blir $T_0 = -15°C$ og $T_s = 20°C$. $\tau = 30$ døgn $= 720$ timer. Vannet fryser ved $0°C$.

Fra (3.80):

$$u = \frac{T - T_0}{T_s - T_0} = \frac{0 - (-15)}{20 - (-15)} = \frac{3}{7} = 0.4286$$

Fra tabellverdi for erf(x) og (3.96) finner vi erf(η) $= 0.4286 \to \eta \approx 0.4$ som med (3.90) gir:

$$X = 0.4 \cdot 2\sqrt{\tau \cdot \alpha} = 0.8 \cdot \sqrt{720 \cdot 5 \cdot 10^{-4}} = 0.48\text{m}$$

Vi har regnet med konstant diffusivitet α, mens den kan variere i intervallet $\alpha \in [3 \cdot 10^{-4},\ 10^{-3}]\text{m}^2/\text{time}$. Dessuten vil jorda normalt ikke være tørr.

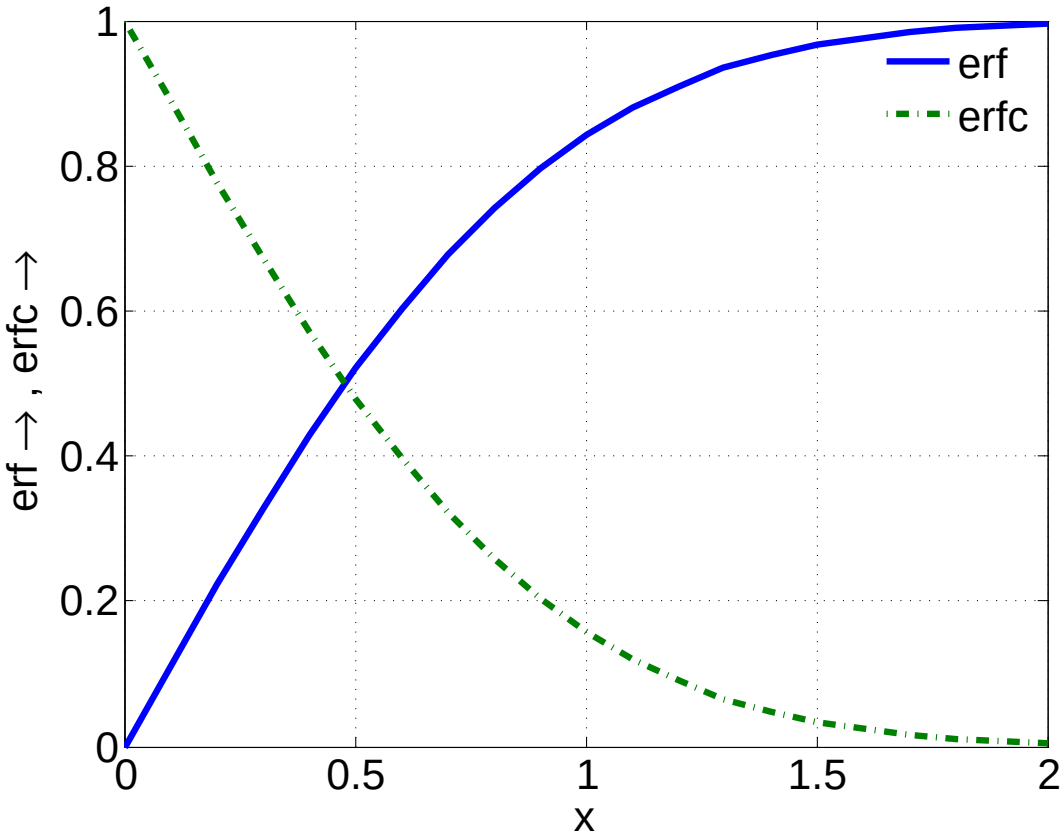


Figure 3.12:

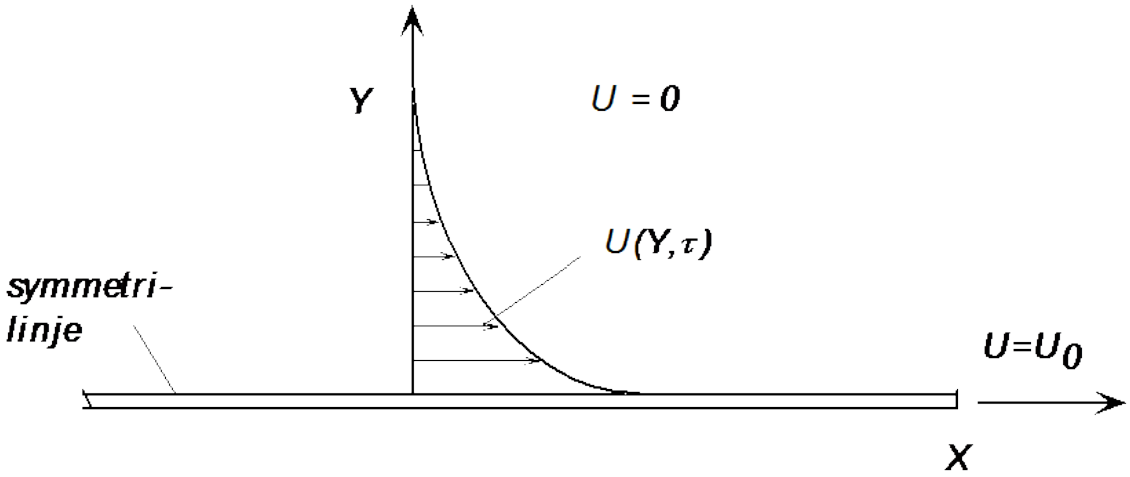


Figure 3.13:

3.3.2 Example: Stokes 1. problem

Vi har en stillestående fluid ved $\tau < 0$. Ved tiden $\tau = 0$ får plata som er parallell med X-aksen, en konstant hastighet $U = U_0$ parallelt med X-aksen. Plata må betraktes som uendelig tynn med uendelig utstrekning i X-retning for en korrekt matematisk beskrivelse i dette tilfellet. Navier-Stokes ligning i X-retning for inkompressibel strømning:

$$\frac{\partial U}{\partial \tau} + U\frac{\partial U}{\partial X} + V\frac{\partial U}{\partial Y} = -\frac{1}{\rho}\frac{\partial p}{\partial X} + \nu\left(\frac{\partial^2 U}{\partial X^2} + \frac{\partial^2 U}{\partial Y^2}\right)$$

(Se utledning i appendiks B i kompendiet)
Antar at løsningen av dette problemet er uavhengig av X, samt at $V = 0$ som betyr at $U = U(Y, \tau)$:

$$\frac{\partial U}{\partial \tau} = \nu\frac{\partial^2 U}{\partial Y^2},\ 0 < Y < \infty \tag{3.99}$$

Initialbetingelse:

$$U(U, 0) = 0 \tag{3.100}$$

Randbetingelser:

$$\begin{gathered} U(0, \tau) = U_0 \text{ (heftbet.)} \\ U(\infty, \tau) = 0 \end{gathered} \tag{3.101}$$

Innfører følgende dimensjonsløse variable:

$$u = \frac{U}{U_0},\ y = \frac{Y}{L},\ t = \frac{\tau \cdot \nu}{L^2} \tag{3.102}$$

der U_0 og L er henholdsvis en karakteristisk hastighet og lengde. (3.102) innsatt i (3.99) gir følgende ligning:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial y^2},\ 0 < y < \infty \tag{3.103}$$

Initialbetingelse:

$$u(y, 0) = 0 \tag{3.104}$$

Randbetingelser:

$$\begin{gathered} u(0, t) = 1 \\ u(\infty, t) = 0 \end{gathered} \tag{3.105}$$

Vi ser at vi har fått samme ligning og problem som i (3.81) og (3.82), bare med en ombytting av 0 og 1 i randbetingelsene. Løsningen av (3.103) og (3.104) med randbetingelser (3.105) blir:

$$u(y, t) = 1 - \text{erf}(\eta) = \text{erfc}(\eta) = \text{erfc}\left(\frac{y}{2\sqrt{t}}\right) \tag{3.106}$$

eller uttrykt ved dimensjonelle variable:

$$U(Y, \tau) = U_0\text{erfc}\left(\frac{Y}{2\sqrt{\tau \cdot \nu}}\right) \tag{3.107}$$

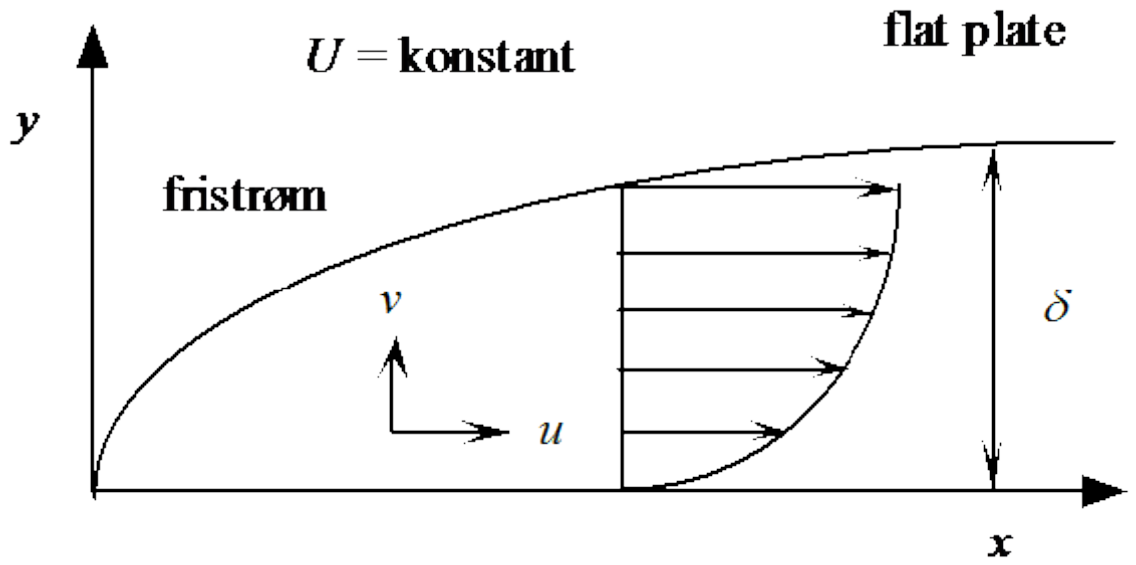


Figure 3.14:

3.3.3 Example: Blasius-ligningen

En detaljert utledning er gitt i appendiks C, del C.2 i kompendiet. Gjengir noe av utledningen her for oversiktens skyld. For en stasjonær og inkompressibel grensesjikt-strømning, får vi fra lign. (C.1.10) i appendiks C i kompendiet:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{3.108}$$

$$u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{1}{\rho}\frac{dp}{dx} + \nu\frac{\partial^2 u}{\partial y^2} \tag{3.109}$$

Ser på det tilfellet at fristrømshastigheten U er konstant $= U_0$. Bernoullis ligning (C.1.8) i appendiks C i kompendiet gir for dette tilfellet $\frac{dp}{dx} = 0$.

(3.109) forenkler seg til:

$$u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = \nu\frac{\partial^2 u}{\partial y^2} \tag{3.110}$$

Randbetingelser:

$$u(0) = 0 \tag{3.111}$$

$$u \to U_0 \text{ for } y \to \delta \tag{3.112}$$

For å oppfylle kontinuitetsligningen (3.108), innfører vi strømfunksjonen $\psi(x, y)$ definert ved:

$$u = \frac{\partial \psi}{\partial y},\ v = -\frac{\partial \psi}{\partial x} \tag{3.113}$$

Vi definerer et Reynoldstall:

$$Re_x = \frac{U_0 x}{\nu} \tag{3.114}$$

Innfører følgende likedannhetsvariable for koordinat og strømfunksjon:

$$\eta = \sqrt{\frac{U_0}{2\nu x}} \cdot y \text{ og } f(\eta) = \frac{\psi}{\sqrt{2U_o\nu x}} \tag{3.115}$$

Transformasjonen (3.115) gir Blasius-ligningen for strømfunksjonen f:

$$f'''(\eta) + f(\eta) \cdot f''(\eta) = 0 \tag{3.116}$$

Notasjon: $\frac{df}{d\eta} \equiv f'(\eta)$ og tilsvarende for høyere deriverte. De fysikalske hastighetskomponentene er gitt ved:

$$\frac{u}{U_0} = f'(\eta),\ \frac{v}{U_0} = \frac{\eta \cdot f'(\eta) - f(\eta)}{\sqrt{2Re_x}} \tag{3.117}$$

Heftbetingelsen $u = 0$ for $y = 0$ blir nå $f'(0) = 0$ da $f'(\eta) = \frac{u}{U_0}$ fra (3.117).

Uten suging eller blåsing gjennom randen $\eta = 0$, blir den andre heftbetingelsen $v = 0$ for $\eta = 0$. Fra (3.117) blir denne betingelsen $f(0) = 0$. Betingelsen $u \to U_0$ fra (3.112), blir nå $f'(\eta) \to 1$ for $\eta \to \infty$. Randbetingelsene for ugjennomstrømmelig sjikt blir da:

$$f(0) = f'(0) = 0,\ f'(\eta_\infty) = 1 \tag{3.118}$$

Skjærspenningen:

$$\tau_{xy} = \mu U_0 \cdot f''(\eta)\sqrt{\frac{U_0}{2\nu x}} \tag{3.119}$$

Numerisk løsning

Vi ønsker å løse (3.116) sammen med randbetingelsene i (3.118) ved bruk av skyteteknikk og skriver derfor ligningen som et sett av tre 1. ordens differensialligninger:

$$f' = f_1 \tag{3.120}$$
$$f_1' = f_2 \tag{3.121}$$
$$f_2' = -f \cdot f_2 \tag{3.122}$$

Randbetingelser:

$$\begin{gathered} f(0) = f_1(0) = 0 \\ f_1(\eta_\infty) = 1 \end{gathered} \tag{3.123}$$

Setter $f''(0) = f_2(0) = s$

s, som i dette tilfellet er skjærspenningen ved veggen, må bestemmes slik at randbetingelsen $f_1(\eta_\infty) = 1$ blir oppfylt. Dette siste kravet kan formuleres på følgende måte:

$$\phi(s) = f_1(\eta_\infty; s) - 1 \tag{3.124}$$

med rett verdi $s = s^*$ når $\phi(s^*) = 0$ Vi velger å bruke sekantmetoden til nullpunktsbestemmelsen i (3.124).

Iterasjonsprosess

Iterasjonsprosessen ved bruk av sekantmetoden, er gitt ved:

$$s^{m+1} = s^m + \delta s,\ \delta s = -\phi(s^m) \cdot \left[\frac{s^m - s^{m-1}}{\phi(s^m) - \phi(s^{m-1})}\right],\ m = 1, 2\ldots. \tag{3.125}$$

Antar at s^{m-1} og s^m er kjent.

1. Beregn $\phi(s^{m-1})$ og $\phi(s^m)$ ved å løse(3.121).
2. Beregn δs og s^{m+1} fra (3.125)
3. Sett
   - $s^{m-1} \leftarrow s^m$
   - $s^m \leftarrow s^{m+1}$
   - $\phi(s^{m-1}) \leftarrow \phi(s^m)$

Gjenta skritt 1–3 inntil konvergens er oppnådd.

I dette tilfellet er korrekt verdi $s^* = 0.46960...$ slik at konvergenskriteriet tilslutt blir $\delta s < \varepsilon$, dvs. en absolutt test. Under prosessen kan vi ha $s > 1$ og da testes $\left|\frac{\delta s}{s}\right|$, altså en relativ test.

La oss se litt nærmere på hvordan vi finner passende startverdier s^0 og s^1 for å komme igang med iterasjonsprosessen. Den vanligvis enkleste måten å få oversikt over nullpunktene for en funksjon på, er å framstille funksjonen grafisk. Figure 3.15 viser $\phi(s)$ der $s \in [0.05,\ 0.8]$. Beregningen er gjort med Pythonprogrammet *phi_plot_blasius_shoot_v2* . Av figuren ser vi at nullpunktet ligger i intervallet $[0.45,\ 0.5]$. Vi ser også at vi har et stort spillerom ved valg av s^0 og s^1.

```
# chapter2/src-ch2/phi_plot_blasius_shoot_v2.py;ODEschemes.py @ git@lrhgit/tkt4140/allfiles/digital_compendium/cha


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()
```

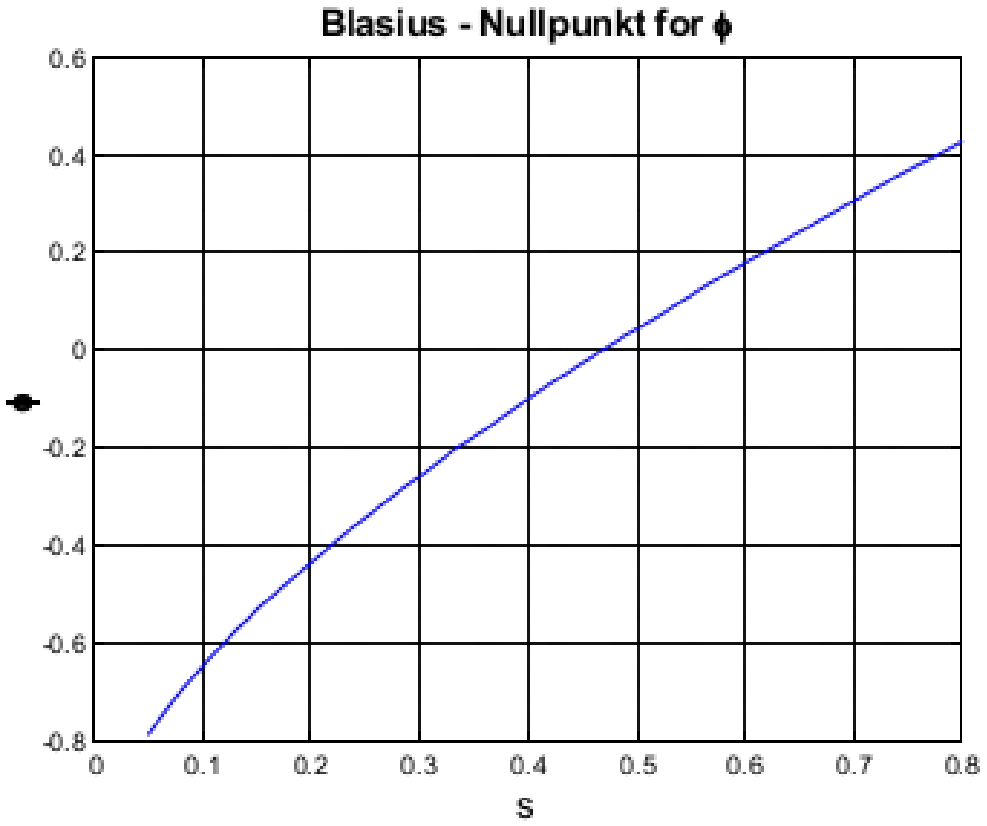


Figure 3.15:

Figure 3.16 viser den dimensjonsløse u-hastigheten $f'(\eta)$ og den dimensjonsløse skjærspenningen $f''(\eta)$ som beregnet i tabellen på neste side. Den maksimale verdien av $f''(\eta)$ inntreffer for $\eta = 0$, som igjen betyr at skjærspenningen er størst ved veggen. Legg merke til at $f'''(0) = 0$ som betyr at $f''(\eta)$ har vertikal tangent på veggen. Tabellen nedenfor er beregnet med programmet *blasius_shoot_v2*:

```
# chapter2/src-ch2/blasius_shoot_v2.py;ODEschemes.py @ git@lrhgit/tkt4140/allfiles/digital_compendium/chapter2/src

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
```

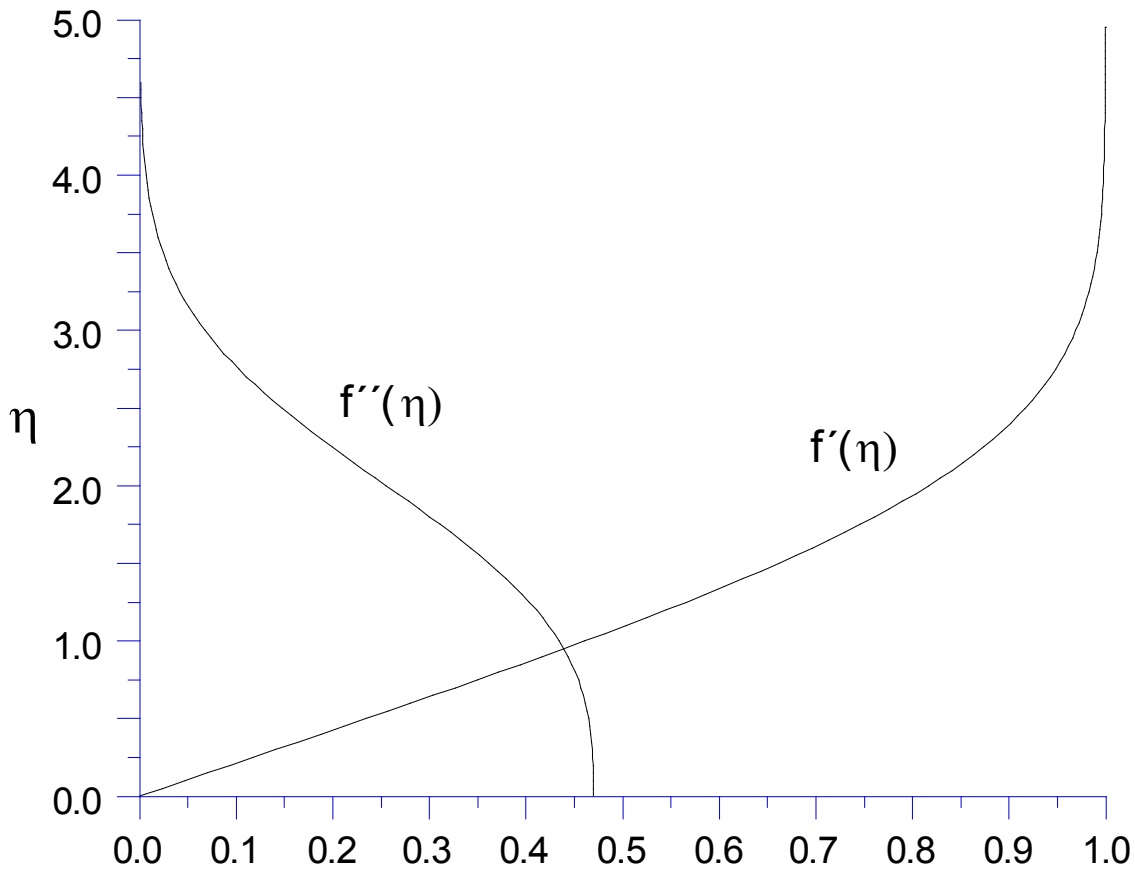


Figure 3.16:

```
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()
```

itr.	s	ds
1	0.471625	- 2.837e-002
2	0.469580	- 2.046e-003
3	0.469601	2.146e-005
4	0.469601	- 1.540e-008

η	f	f'	f''
0.00	0.000000	0.000000	4.69601e-001
0.25	0.014673	0.117364	4.69027e-001
0.50	0.058643	0.234228	4.65032e-001
0.75	0.131642	0.349324	4.54373e-001
1.00	0.232991	0.460634	4.34380e-001
1.25	0.361431	0.565601	4.03495e-001
1.50	0.515032	0.661476	3.61805e-001
1.75	0.691201	0.745765	3.11302e-001
2.00	0.886798	0.816697	2.55671e-001
2.25	1.098374	0.873559	1.99544e-001
2.50	1.322442	0.916810	1.47476e-001
2.75	1.555766	0.947928	1.02930e-001
3.00	1.795573	0.969057	6.77108e-002
3.25	2.039661	0.982573	4.19262e-002
3.50	2.286412	0.990711	2.44148e-002
3.75	2.534723	0.995319	1.33643e-002
4.00	2.783894	0.997773	6.87412e-003
4.25	3.033509	0.999000	3.32201e-003
4.50	3.283340	0.999577	1.50841e-003
4.75	3.533271	0.999832	6.43473e-004
5.00	3.783244	0.999938	2.57831e-004
5.25	4.033235	0.999980	9.70546e-005
5.50	4.283232	0.999995	3.43646e-005
5.75	4.533231	1.000000	1.15317e-005

3.3.4 Example:Falkner-Skan ligningen

Blasius ligning er egentlig et spesialtilfelle av en mer generell ligning som kalles Falkner-Skan ligningen. Falkner-Skan tranformasjonen (1930) overfører grensesjiktligningen til en likedannhetsform gitt ved:

$$f''' + ff'' + \beta \cdot [1 - (f')^2] = 0 \tag{3.126}$$

når fristrømshastigheten $U(x)$ er gitt ved:

$$U(x) = U_0 x^m, \ m = \text{konstant} \tag{3.127}$$

(3.126) kalles også kilestrømsligningen fordi parameteren β har den geometriske betydningen som vist i Figure 3.17:

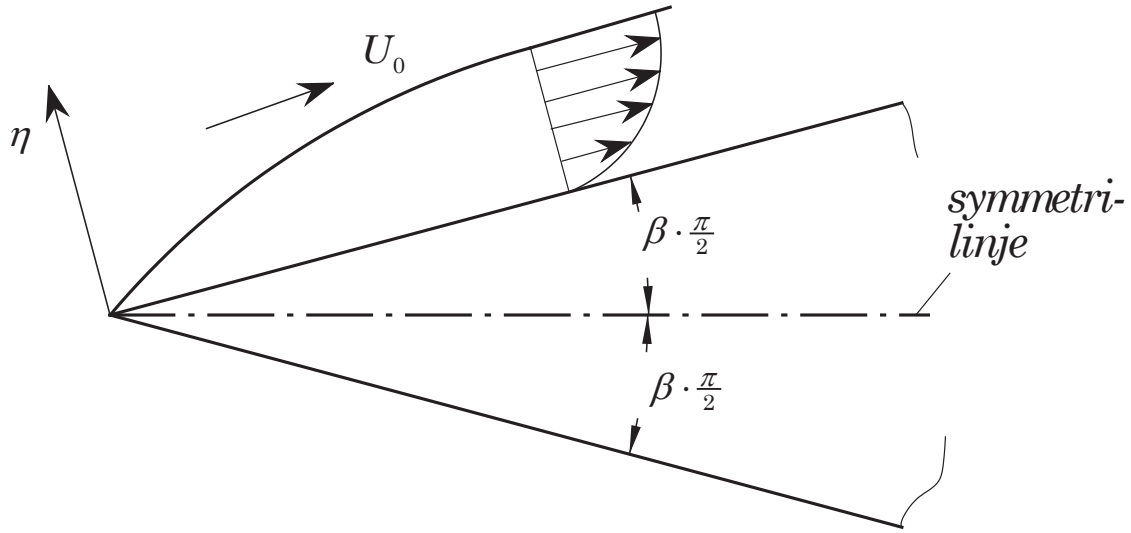


Figure 3.17:

Vi ser at halve kilevinkelen er $\beta \cdot \frac{\pi}{2}$. For $\beta = 0$, får vi følgelig en flat plate, og (3.126) blir da Blasius ligning.

En detaljert utledning er gitt i appendiks C, del C.3 i kompendiet. Vi gjengir noe av utledningen for oversiktens skyld.

Sammenheng mellom m og β: $\beta = \frac{2m}{m+1}$ og $m = \frac{\beta}{2-\beta}$ Vi ser at $\beta = 0$ gir $U(x) = U_0$.

Falkner-Skan transformasjonen er gitt ved:

$$\eta = \sqrt{\frac{U}{(2-\beta)\nu x}} \cdot y, \ f(\eta) = \frac{\psi}{\sqrt{(2-\beta)U\nu x}} \tag{3.128}$$

Dersom vi velger de samme randbetingelsene som i eksempel (3.3.3), får vi:

F-S - ligningen:

$$f''' + ff'' + \beta \cdot [1 - (f')^2] = 0 \tag{3.129}$$

Randbetingelser:

$$\begin{gathered} f(0) = f'(0) = 0 \\ f'(\eta_\infty) = 1 \end{gathered} \tag{3.130}$$

Vi løser ligningen på samme måte som Blasius ligning ved å skrive (3.129) som et sett av 1. ordens differensialligninger:

$$f' = f_1 \tag{3.131}$$
$$f_1' = f_2 \tag{3.132}$$
$$f_2' = -[f f_2 + \beta(1 - f_1^2)] \tag{3.133}$$

Randbetingelser:

$$\begin{array}{c} f(0) = f_1(0) = 0 \\ f_1(\eta_\infty) = 1 \end{array} \tag{3.134}$$

Med $f''(0) = f_2(0) = s$, får vi:

$$\phi(s) = f_1(\eta_\infty; s) - 1 \tag{3.135}$$

med korrekt verdi s^* når $\phi(s^*) = 0$

Flere detaljer om løsningsprosessen er gitt i appendiks C, del C.3 i kompendiet.

Bestemmelse av startverdier.

Vi bruker sekantmetoden for å finne nullpunktet i (3.135) og trenger derfor to startverdier s^0 og s^1 for å starte iterasjonsprosessen. Velger tilfellet med $\beta = 1$ som er strømning mot en flat plate, dvs.: en stagnasjonsstrøm, se Figure 3.18. Dette tilfellet kalles også en Hiemenz-strømning.

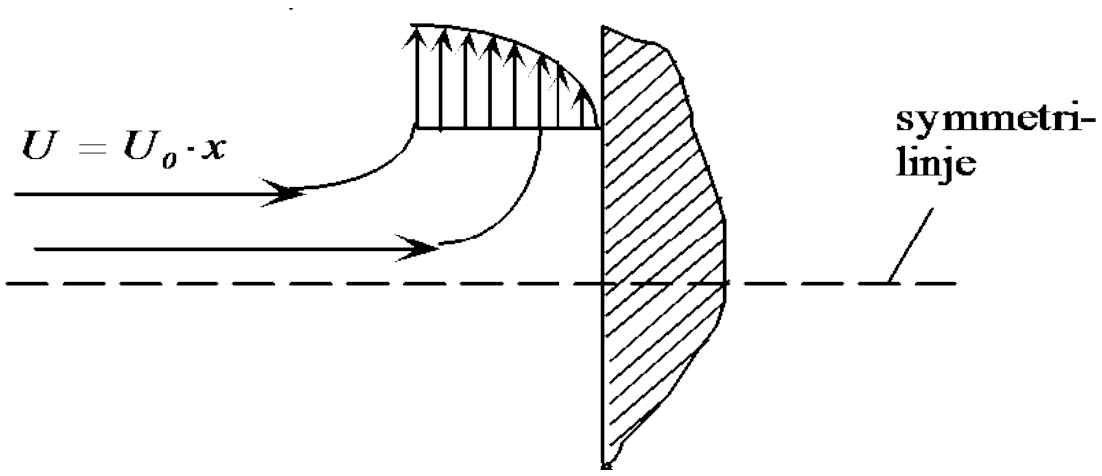


Figure 3.18:

$\phi(s)$ er tegnet i Figure 3.19. Beregningen er utført med bruk av **ode45**. **Marie 4**: Matlab-funksjon

Det korrekte nullpunktet er $s^* = 1.23259$ som er den entydige løsningen av F-S-ligningen for $\eta \to \infty$ med $\beta = 1$. Dersom vi ser på kurven for $\eta_\infty = 5.0$, finner vi et nullpunkt rundt $s = 0.85$ og et rundt $s = 1.23$ der det siste er det korrekte. Årsaken er at F-S-ligningen har to løsninger for endelige verdier av η_∞ der det venstre nullpunktet beveger seg mens det andre som er det korrekte for $\eta \to \infty$, ligger fast. Begge nullpunktene gir korrekte løsninger av diff.ligningen, men det venstre nullpunktet gir en løsning som fører til avløsning og tilbake-strømning, noe som er ufysikalsk i dette tilfellet. Legg merke til at avstanden mellom nullpunktene minsker med økende verdi av η_∞, og med $\eta \to \infty$ vil de to grenene falle sammen.

Dersom vi bruker en løser med konstant skrittlengde, f.eks **RK4**, og samtidig velger en forholdsvis stor skrittlengde, f. eks. $\Delta\eta = 0.2$, vil vi få en rekke andre nullpunkt. Disse nullpunktene gir løsning av differanseligningen, men er ikke løsning av differensialligningen. Ved å la $\Delta\eta \to 0$, vil disse nullpunktene forsvinne også for store η_∞-verdier.

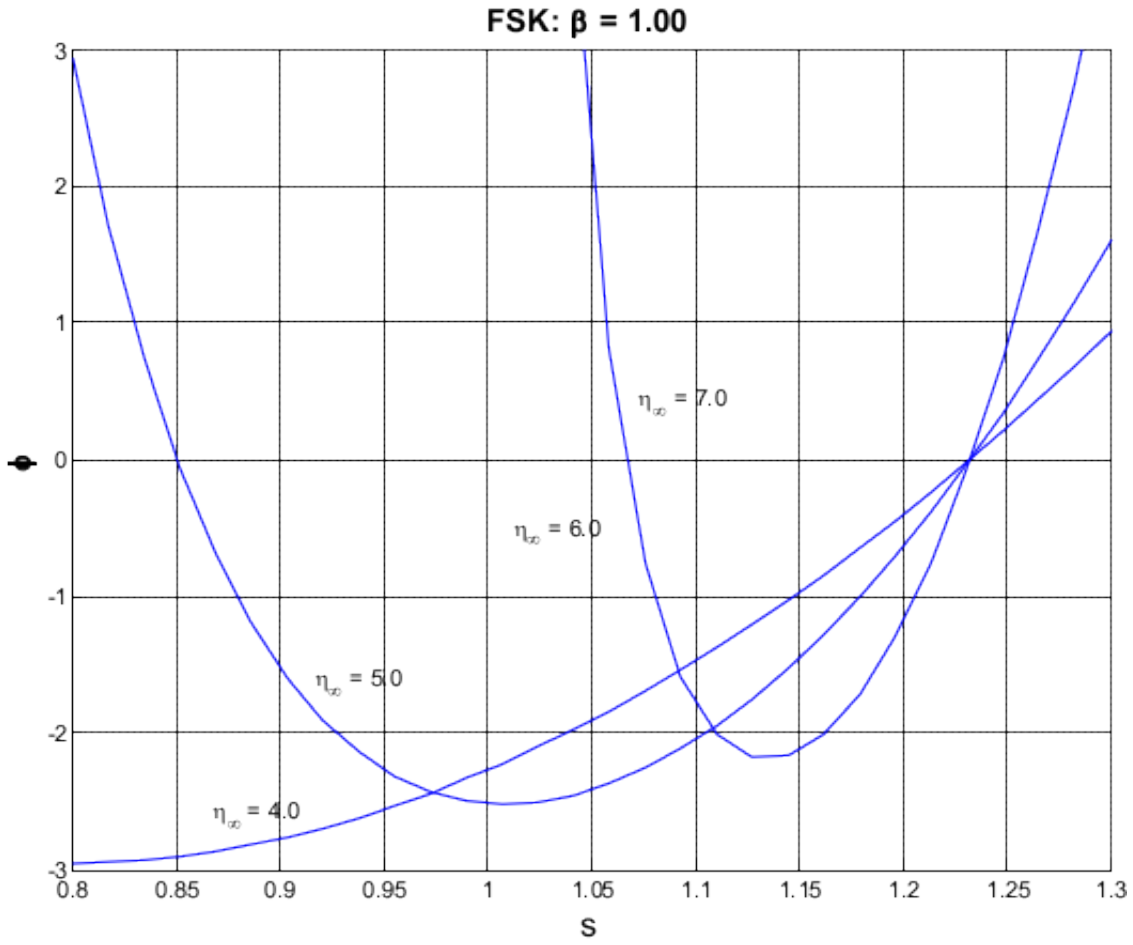


Figure 3.19:

Figure 3.20 og Figure 3.21 viser noen resultater fra Matlab-programmet **fsksec**. **Marie 5**: Matlab-program for ulike verdier av β.

For $\beta = \beta_{sep} = -0.19883768\ldots$ forsvinner skjærspenningen ved veggen. Dette indikerer begynnende tilbakestrømning og separasjon, se Figure 3.22.

Det finnes løsninger av F-S-ligningen for β-verdier som ligger utenfor det området vi fysisk kan knytte til en kilestrøm. For mer informasjon om disse løsningene, henvises til Evans [6], White [21], og Schlichting [19].

Tilslutt bør det nevnes at det finnes en analytisk løsning av F-S-ligningen. Vi tenker oss at det er plassert et sluk i $x = 0$ med styrke K parallelt med platekanten. I dette tilfellet blir $U(x) = -\dfrac{K}{x}$ og $\eta = \dfrac{y}{x}\sqrt{\dfrac{K}{\nu}}$.

Ligningen blir nå:

$$f'''(\eta) - \left(f'(\eta)\right)^2 + 1 = 0 \tag{3.136}$$

med de vanlige randbetingelsene

$$f(0) = f'(0) = 0 \tag{3.137}$$
$$f'(\eta_\infty) = 1 \tag{3.138}$$

Løsningen kan da skrives:

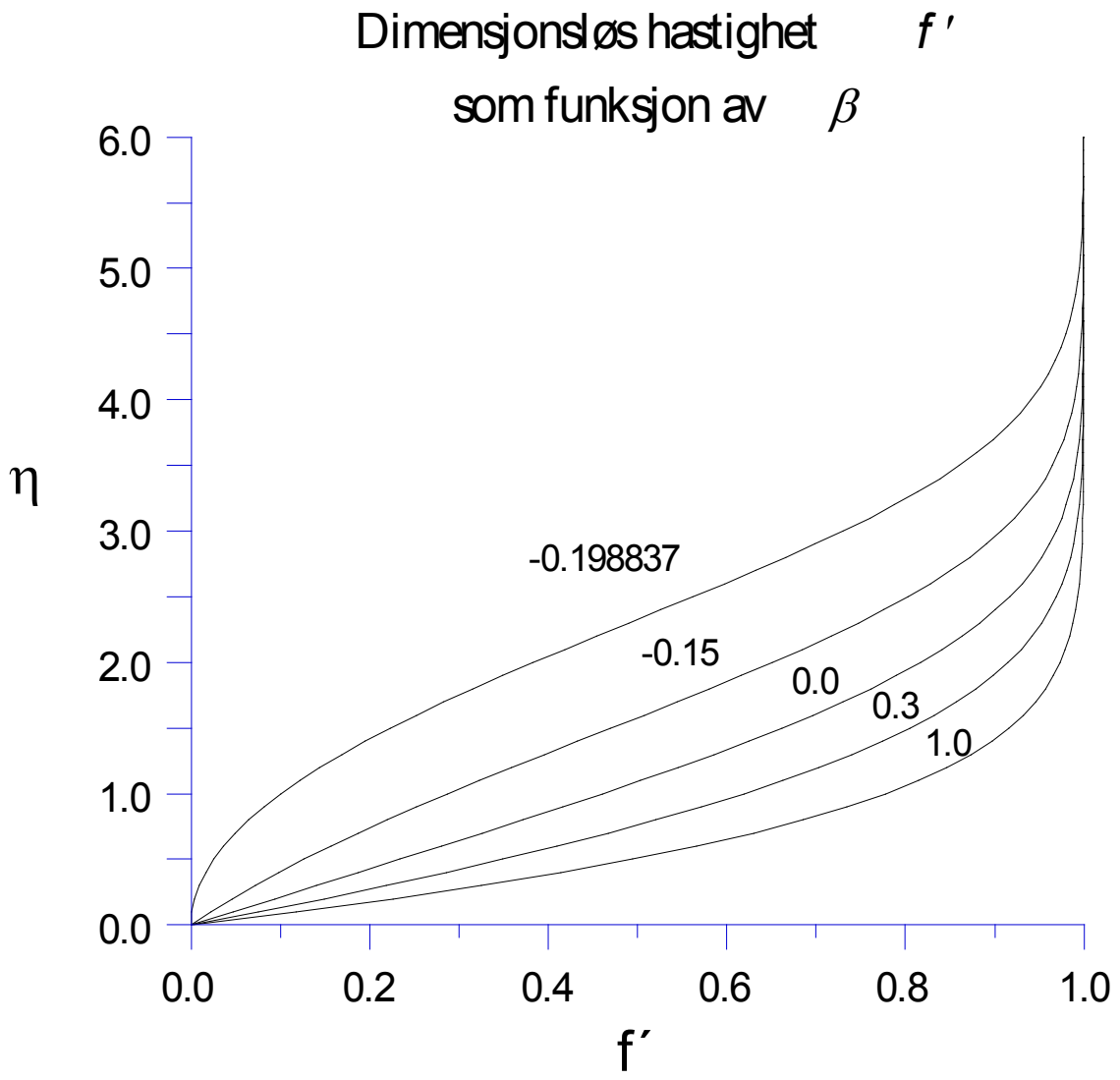


Figure 3.20:

$$f(\eta) = \eta + 2\sqrt{3} - 3\sqrt{2} \cdot \tanh(z) \tag{3.139}$$

$$f'(\eta) = 3\tanh^2(z) - 2 \tag{3.140}$$

$$f''(\eta) = 3\sqrt{2} \cdot \tanh(z) \cdot [1 - \tanh^2(z)] \tag{3.141}$$

Spesielt har vi:

$$f''(0) = \frac{2}{\sqrt{3}} = 1.1547\ldots$$

Merk at en funksjon $f(\eta) = \eta + a - 3\sqrt{2} \cdot \tanh(\eta/\sqrt{2} + b)$ tilfredstiller (3.136) for vilkårlige verdier av a og b med $f'(\infty) = 1$.

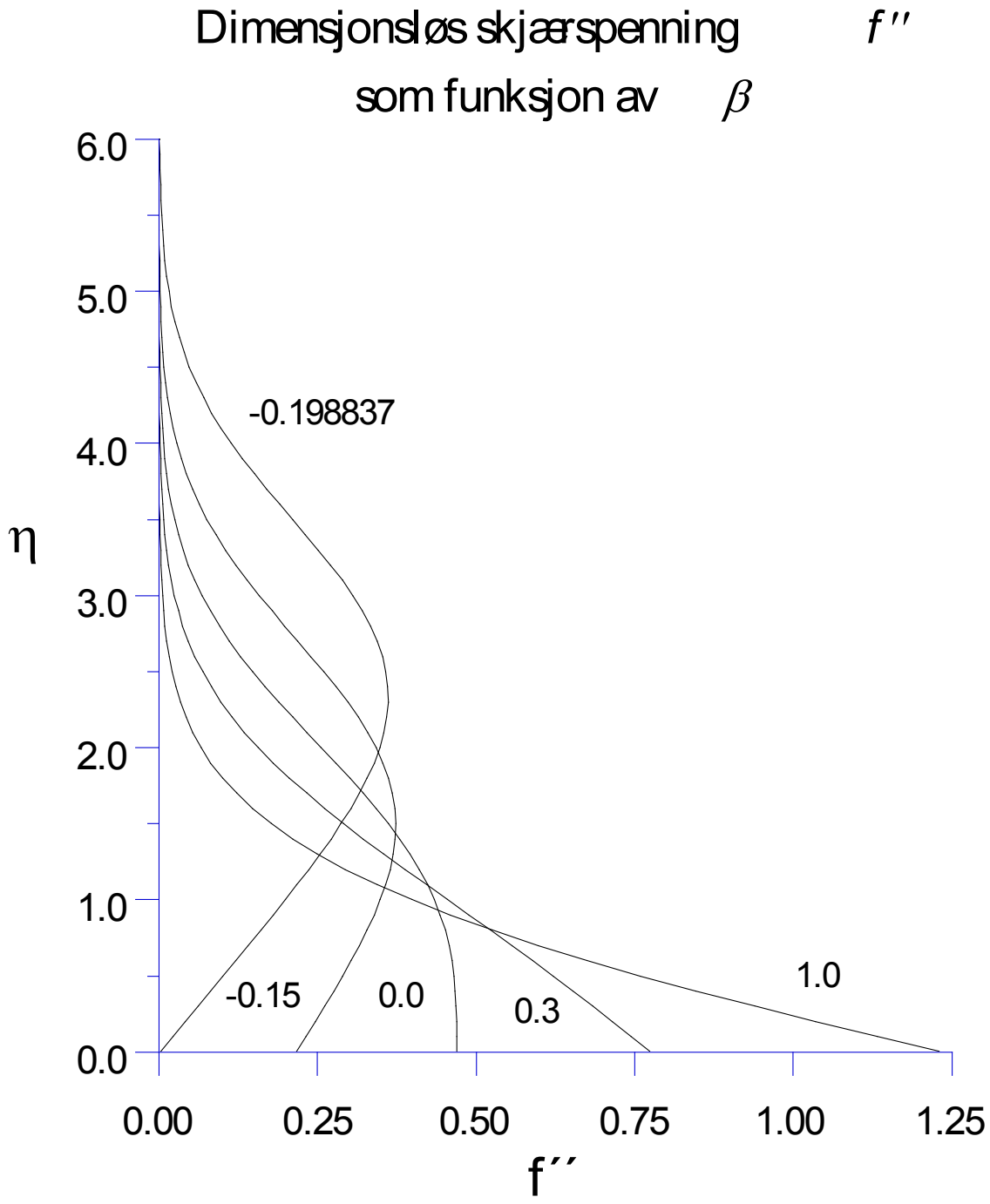

Dimensjonsløs skjærspenning f''
som funksjon av β
6.0
5.0
4.0
3.0
2.0
1.0
0.0
η
-0.198837
-0.15
0.0
0.3
1.0
0.00
0.25
0.50
0.75
1.00
1.25
f''

Figure 3.21:

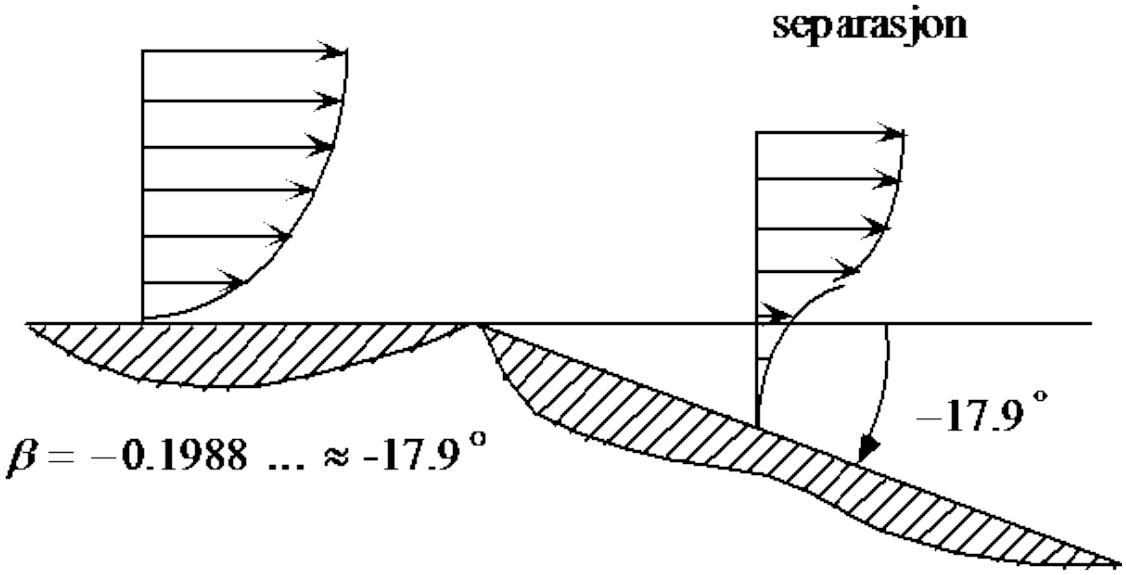


Figure 3.22:

3.4 Skyting med to startbetingelser

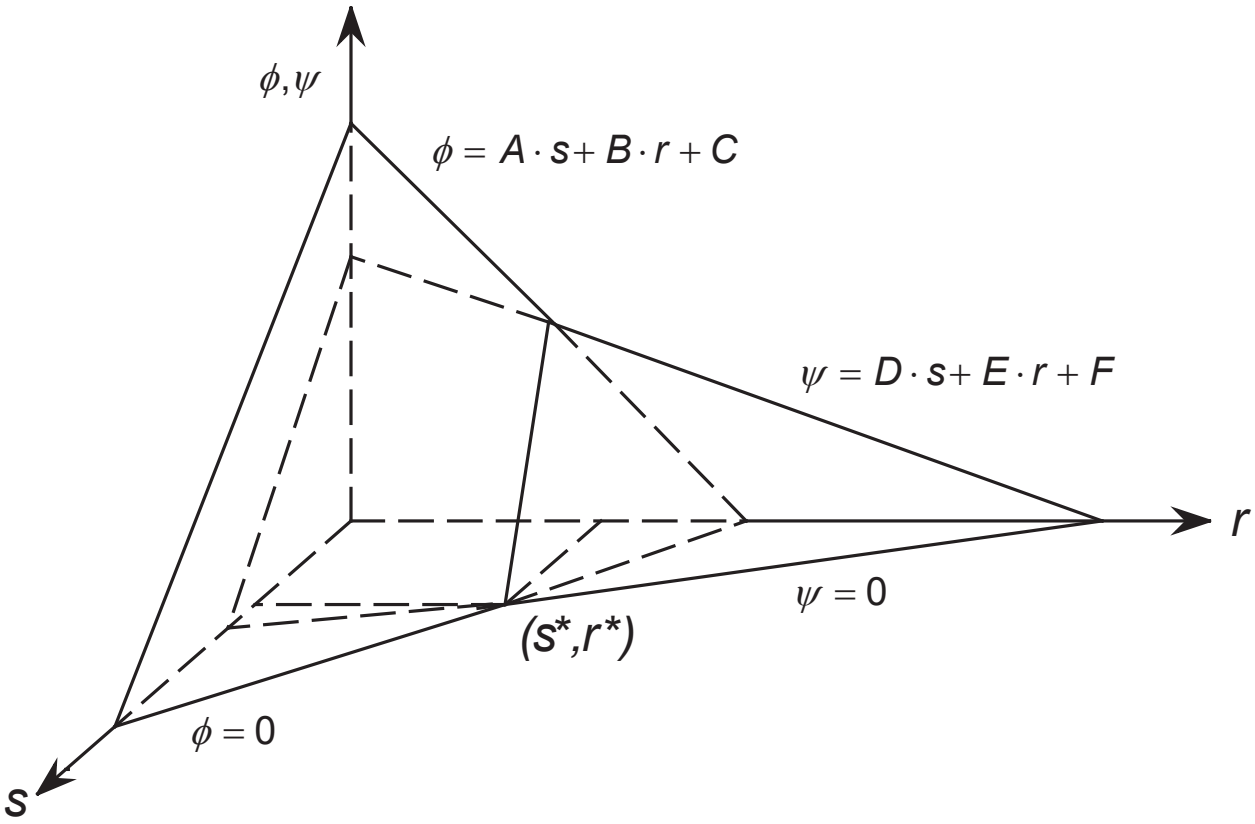


Figure 3.23:

3.4.1 Lineære ligninger

Vi vil nå utvide teknikken som vi brukte i avsnitt (3.1) til to ukjente start-betingelser. I dette tilfellet må vi tippe to startbetingelser r og s med tilhørende funksjoner ϕ og ψ. Dette er vist grafisk i Figure 3.23. Da ligningen som skal løses er lineær, danner ϕ ogψ to plan og kan derfor skrives på formen:

$$\begin{aligned} \phi &= A\cdot s + B\cdot r + C \\ \psi &= D\cdot s + E\cdot r + F \end{aligned} \tag{3.142}$$

der $A, B, C, \cdots, F$ er konstanter som må bestemmes. De rette verdiene r^* og s^* finnes fra $\phi(r^*, s^*) = 0$ og $\psi(r^*, s^*) = 0$. (Se figuren) Vi har seks konstanter som bestemmes fra ligningene:

$$\phi^0 = A\cdot s^0 + B\cdot r^0 + C \tag{3.143}$$
$$\phi^1 = A\cdot s^1 + B\cdot r^1 + C \tag{3.144}$$
$$\phi^{(2)} = A\cdot s^{(2)} + B\cdot r^{(2)} + C \tag{3.145}$$
$$\tag{3.146}$$

$$\psi^0 = D \cdot s^0 + E \cdot r^0 + F \tag{3.147}$$
$$\psi^1 = D \cdot s^1 + E \cdot r^1 + F \tag{3.148}$$
$$\psi^{(2)} = D \cdot s^{(2)} + E \cdot r^{(2)} + F \tag{3.149}$$
$$\tag{3.150}$$

der $s^0,\ s^1,\ s^{(2)},\ r^0,\ r^1$ og $r^{(2)}$ er verdier som vi har tippet for initialbetingelsene.For å forenkle slutt-resultatet, velger vi følgende verdier:

$$1)\ \ s^0 = 0,\ r^0 = 0 \tag{3.151}$$
$$2)\ \ s^1 = 0,\ r^1 = 1 \tag{3.152}$$
$$3)\ \ s^{(2)} = 1,\ r^{(2)} = 0 \tag{3.153}$$

Vi får da følgende uttrykk for konstantene:

$$\begin{array}{c} C = \phi^0,\ B = \phi^1 - \phi^0,\ A = \phi^{(2)} - \phi^0 \\ F = \psi^0,\ E = \psi^1 - \psi^0,\ D = \psi^{(2)} - \psi^0 \end{array} \tag{3.154}$$

De korrekte verdiene r^* og s^* finnes fra $\phi(r^*,\ s^*) = 0$ og $\psi(r^*,\ s^*) = 0$ som her blir $A \cdot s^* + B \cdot r^* + C = D \cdot s^* + E \cdot r^* + C = 0$ med følgende løsning:

$$s^* = \frac{E \cdot C - B \cdot F}{D \cdot B - A \cdot E},\ r^* = \frac{A \cdot F - D \cdot C}{D \cdot B - A \cdot E}$$

som innsatt for $A, B, C, \ldots, F$ gir:

$$\begin{array}{l} s^* = \frac{\psi^1 \cdot \phi^0 - \phi^1 \cdot \psi^0}{(\psi^{(2)} - \psi^0) \cdot (\phi^1 - \phi^0) - (\phi^{(2)} - \phi^0) \cdot (\psi^1 - \psi^0)} \\ r^* = \frac{\phi^{(2)} \cdot \psi^0 - \psi^{(2)} \cdot \phi^0}{(\psi^{(2)} - \psi^0) \cdot (\phi^1 - \phi^0) - (\phi^{(2)} - \phi^0) \cdot (\psi^1 - \psi^0)} \end{array} \tag{3.155}$$

Fremgangsmåten blir da som følger:

Vi løser ligningsystemet for de tre settene av verdier for r og s gitt i (3.152) og finner derved de tilhørende verdiene av ϕ og ψ. De korrekte verdiene r^* og s^* finnes da fra (3.155)). Vi skal bruke denne prosedyren i eksempel (3.4.2)

3.4.2 Example: Sylindrisk tank med væske

DEL 1: Konstant veggtykkelse

Figure 3.24 viser en sylindrisk tank fylt med en væske til en høyde H. W er radiell forskyvning. På detaljen til venstre er V skjærkraft pr. lengdeenhet og M moment pr. lengdeenhet. Pilene angir positiv retning.

Differensialligningen for forskyvningen W er gitt ved:

$$\frac{d^4W}{dX^4} + B \cdot W = -\gamma \frac{H - X}{D} \tag{3.156}$$

der $B = \dfrac{12(1-\nu^2)}{R^2t^2},\ D = \dfrac{Et^3}{12(1-\nu^2)},\ \gamma = \rho g$

Her er ν Poissons tall, E elastisitetsmodelen og ρ væskas tetthet.

Innfører dimensjonsløse størrelser:

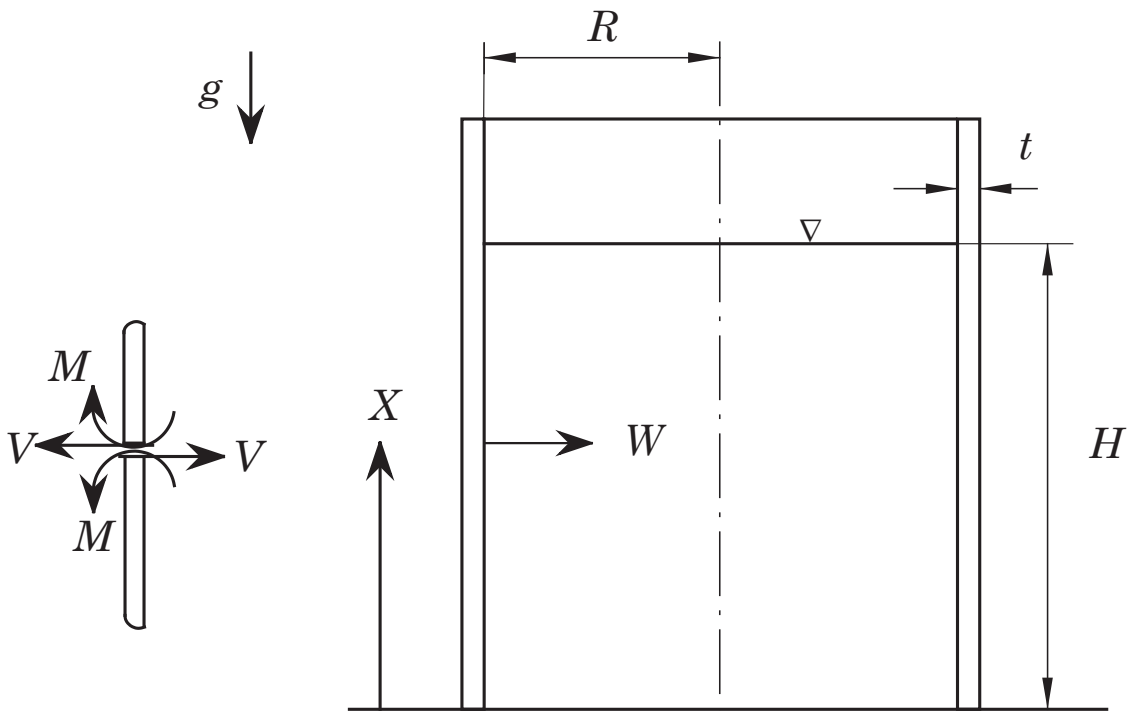


Figure 3.24:

$$x = \frac{X}{H} \text{ og } w = \frac{E}{\gamma H t} \cdot \left(\frac{t}{R}\right)^2 \cdot W \tag{3.157}$$

som innsatt i (3.156) gir følgende differensialligning:

$$\frac{d^4w}{dx^4} + 4\beta^4 \cdot w = -4\beta^4(1-x) \tag{3.158}$$

der

$$\beta^4 = \frac{3(1-\nu^2)H^4}{R^2t^2} \tag{3.159}$$

Randverdier.
For $x = 0$:

$$W = 0,\ \frac{dW}{dX} = 0 \text{ (Fast innspent)}$$

Skjærkraft og moment $= 0$ for $X = H$ som gir:

$$M = -D\frac{d^2W}{dX^2} = 0,\ V = -D\frac{d^3W}{dX^3} = 0$$

Randverdiene uttrykt ved dimensjonsløse størrelser blir da:

$$\begin{array}{c} \text{For } x = 0: \quad w = 0,\ \frac{dw}{dx} = 0 \\ \text{For } x = 1: \quad \frac{d^2w}{dx^2} = 0,\ \frac{d^3w}{dx^3} = 0 \end{array} \tag{3.160}$$

Dimensjonsløst moment $m(x) = -\dfrac{d^2w}{dx^2}$ og dimensjonsløs skjærkraft $v(x) = -\dfrac{d^3w}{dx^3}$. Se appendiks D for flere detaljer.

Velger en en tank av betong med følgende dimensjoner:

$$\begin{gathered} R = 8.5m,\ H = 7.95m,\ t = 0.35m \\ \gamma = 9810N/m^3(\text{vann}),\ \nu = 0.2,\ E = 2 \cdot 10^4\text{MPa} \end{gathered} \tag{3.161}$$

Med disse data blir $\beta = 6.0044$.

Numerisk beregning

Setter $w = y_1,\ w' = y_1' = y_2,\ w'' = y_2' = y_3,\ w''' = y_3' = y_4$ og skriver (3.156) som et system av fire 1. ordens differensialligninger:

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= y_3 \\ y_3' &= y_4 \\ y_4' &= -4\beta^4(y + 1 - x) \end{aligned} \tag{3.162}$$

med følgende randbetingelser:

$$y_1(0) = 0,\ y_2(0) = 0,\ y_3(1) = 0,\ y_4(1) = 0 \tag{3.163}$$

Skal bestemme $s = w''(0) = y_3(0)$ og $r = w'''(0) = y_4(0)$ slik at $y_3(1)$ og $y_4(1) = 0$. Setter følgelig $\phi(r, s) = y_3(1; r, s)$ og $\psi(r, s) = y_4(1; r, s)$. De korrekte verdiene r^* og s^* gir da $\phi(r^*, s^*) = 0$. Løser følgelig (3.162) tre ganger med $y_1(0)$ og $y_2(0) = 0$ mens verdiene for r og s velges fra (3.152). De korrekte verdiene r^* og s^* beregnes deretter fra (3.155).

Utsnitt av programmet **tank1** samt utskrift er vist nedenfor. **Marie 5**: Matlab-program

x	w	dw/dx	m(x)	v(x)
0.000	0.00000e+000	0.00000e+000	6.00946e+001	-7.93776e+002
0.100	-1.89049e-001	-2.86464e+000	4.85241e+000	-3.36907e+002
0.200	-4.57346e-001	-2.19767e+000	-1.36887e+001	-6.60901e+001
0.300	-6.03796e-001	-7.32255e-001	-1.38468e+001	4.15724e+001
0.400	-6.16145e-001	3.93069e-001	-8.41051e+000	5.74987e+001
0.500	-5.43811e-001	9.70524e-001	-3.44813e+000	3.94237e+001
0.600	-4.35148e-001	1.15559e+000	-6.10973e-001	1.81108e+001
0.700	-3.18861e-001	1.15307e+000	4.27833e-001	4.09131e+000
0.800	-2.06030e-001	1.10306e+000	4.68862e-001	-2.15847e+000
0.900	-9.76194e-002	1.07014e+000	1.76515e-001	-2.96401e+000
1.000	8.92555e-003	1.06379e+000	1.20446e-004	6.10070e-004

Vi ser at vi har god overenstemmelse når vi sammenligner med den analytiske løsningen i tabell 1 i appendiks D i kompendiet. Figure **??** nedenfor viser forløpet av det dimensjonsløse momentet samt skjærkrafta.

(3.158) er av typen der den høyeste deriverte er multiplisert med et lite tall ε der $\varepsilon = \frac{1}{4\beta^4}$ i vårt tilfelle. Kalles gjerne en singulær ligning dersom vi tillater at $\beta \to \infty$. Dette gjenspeiler seg i den analytiske løsningen i lign. (D.1.6) og (D.1.8), del D.1, i appendiks D i kompendiet der vi har ledd av typen $e^{\beta x}\sin\beta x$ og $e^{\beta x}\cos\beta x$.

Dette betyr at skytelengden i praksis er lik β. Det vil da bli mer og mer vanskelig å treffe randen $x = 1$ med økende β-verdier. For store β-verdier vil deformasjonen konsentrere seg rundt $x = 0$ slik at det kanskje ikke er nødvendig å skyte helt til $x = 1$, men kan greie oss med en mindre verdi.

Nedenfor har vi beregnet innspenningsmomentet $m_0 = -\left.\dfrac{\partial^2 m}{\partial x^2}\right|_{x=0}$ og skjærkrafta $v_0 = -\left.\dfrac{\partial^2 v}{\partial x^2}\right|_{x=0}$ som funksjon av skytelengden. Merk at $m_0 = -s^*$ og $v_0 = -r^*$.

Skytelengde	m_0	$-v_0$
1.0	60.098	793.809
0.9	60.093	793.715
0.8	60.099	793.670
0.7	60.079	793.598
0.6	59.816	791.835
0.5	58.879	781.541

DEL 2: Lineært variabel veggtykkelse

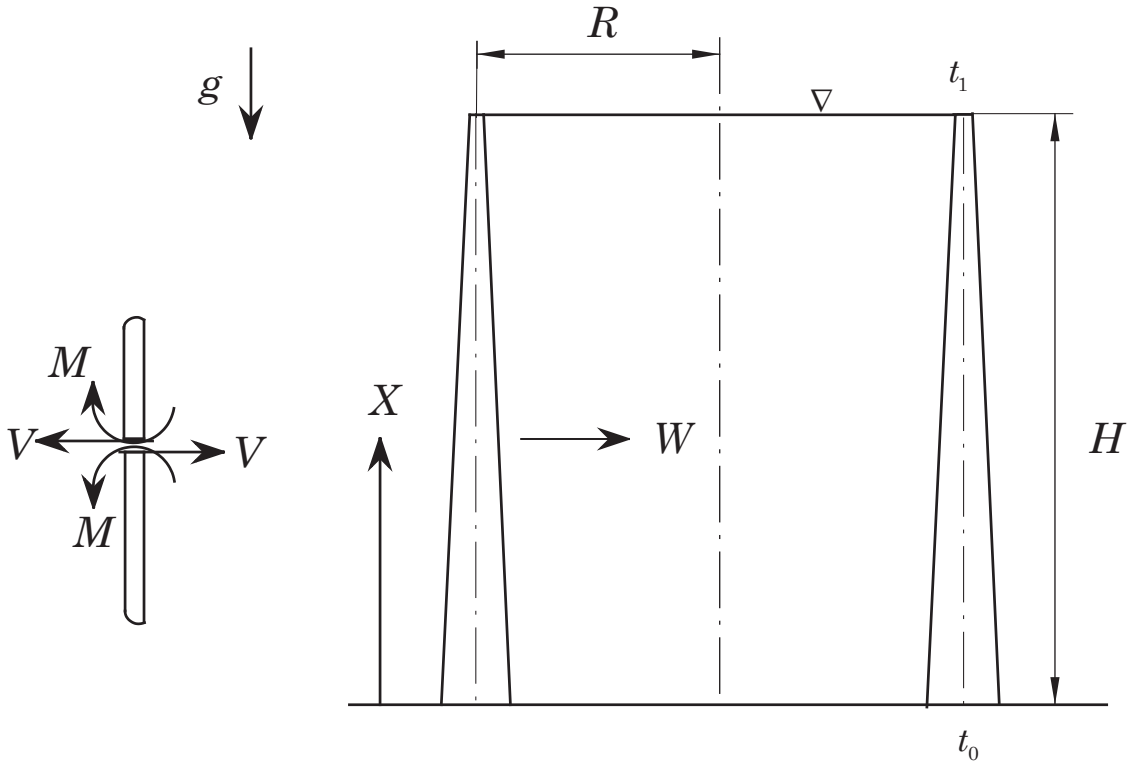


Figure 3.25:

Her varierer veggtykkelsen lineært fra t_0 nederst til t_1 øverst.
Med $\alpha = \dfrac{t_0 - t_1}{t_0}$ kan veggtykkelsen t skrives:

$$t = \left(1 - \alpha \frac{X}{H}\right) \cdot t_0, \ 0 \leq X \leq H \tag{3.164}$$

Differensialligningen for forskyvningen W er nå gitt ved:

$$\frac{d^2}{dX^2}\left(D\frac{d^2W}{dX^2}\right) + E\frac{tW}{R^2} = -\gamma(H - X) \tag{3.165}$$

der $D = \dfrac{Et^3}{12(1-\nu^2)}$, $\gamma = \rho g$
Konstantene har samme betydning som i del 1. (Se appendiks D i kompendiet for flere detaljer).

Dimensjonsløs form

$$x = \frac{X}{H},\ w = \frac{E}{\gamma H t_0}\left(\frac{t_0}{R}\right)^2 \cdot W,\ \beta^4 = \frac{3(1-\nu^2)}{R^2 t_0^2} \cdot H^4 \tag{3.166}$$

(3.166) innsatt i (3.165):

$$\frac{d^2}{dx^2}\left[(1-\alpha x)^3 \frac{d^2 w}{dx^2}\right] + 4\beta^4(1-\alpha x)\cdot w = -4\beta^4(1-x) \tag{3.167}$$

Utderivert:

$$\frac{d^4 w(x)}{dx^4} - \frac{6\alpha}{(1-\alpha x)}\frac{d^5 w(x)}{dx^5} + \frac{6\alpha^2}{(1-\alpha x)^2}\frac{d^2 w(x)}{dx^2} + \frac{4\beta^4}{(1-\alpha x)^2} w(x) = -\frac{4\beta^4(1-x)}{(1-\alpha x)^5} \tag{3.168}$$

Dette systemet kan også løses analytisk, se appendiks D, del D.2 i kompendiet, men matematikken er mer komplisert fordi løsningen blir uttrykt meden spesiell type Besselfunksjoner som kalles Kelvinfunksjoner. Den numeriske fremgangsmåten er derimot uforandret.

Velger samme tank som i del 1 med tillegg for varierende veggtykkelse:

$$\begin{gathered} R = 8.5m,\ H = 7.95m,\ t_0 = 0.35m,\ t_1 = 0.1m \\ \gamma = 9810N/m^3(\text{vann}),\ \nu = 0.2,\ E = 2\cdot 10^4\text{MPa} \end{gathered} \tag{3.169}$$

Her blir $\alpha = \frac{t_0 - t_1}{t_0} = \frac{5}{7}$ og som tidligere blir $\beta = 6.0044$.

Numerisk beregning

Setter $w = y_1,\ w' = y_1' = y_2,\ w'' = y_2' = y_3,\ w''' = y_3' = y_4,\ z = 1-\alpha x$ og skriver (3.164) som et system av fire 1. ordens differensialligninger:

$$\begin{gathered} y_1' = y_2 \\ y_2' = y_3 \\ y_3' = y_4 \\ y_4' = \frac{6\alpha}{z}y_4 - \frac{6\alpha^2}{z^2}y_3 - \frac{4\beta^4}{z^2}y_1 - \frac{4\beta^4(1-x)}{z^3} \end{gathered} \tag{3.170}$$

med følgende randbetingelser:

$$y_1(0) = 0,\ y_2(0) = 0,\ y_3(1) = 0,\ y_4(1) = 0 \tag{3.171}$$

Fremgangsmåten blir som for del 1, bortsett fra at vi nå har parameteren α i tillegg til β. Programmet **tank2** blir derfor nesten identisk med **tank1**. Hovedforskjellen er funksjonen **fcntank2** for diff. ligningen.
Marie 7: Matlab

x	w	dw/dx	m(x)	v(x)
0.000	0.00000e+000	0.00000e+000	6.23338e+001	-7.73773e+002
0.100	-2.15376e-001	-3.45545e+000	9.04151e+000	-3.18887e+002
0.200	-5.66553e-001	-3.16644e+000	-8.14798e+000	-5.86193e+001
0.300	-8.05113e-001	-1.53673e+000	-8.34542e+000	3.32725e+001
0.400	-8.79417e-001	-3.87614e-002	-4.36688e+000	3.81223e+001
0.500	-8.34707e-001	8.24809e-001	-1.47715e+000	1.90313e+001
0.600	-7.31823e-001	1.17417e+000	-3.81333e-001	4.52213e+000
0.700	-6.05528e-001	1.35136e+000	-2.49403e-001	-5.41818e-001
0.800	-4.57651e-001	1.63752e+000	-3.00683e-001	8.66313e-002
0.900	-2.72819e-001	2.06404e+000	-1.83081e-001	2.16427e+000
1.000	-5.26758e-002	2.26301e+000	3.87892e-005	-1.76069e-006

Vi finner god overenstemmelse ved å sammenligne med den analytiske løsningen i tabell 7, del D.2 i appendiks D i kompendiet.

3.4.3 Eksempel på ikke-lineære ligninger

Figure **??** viser et fluidsjikt med initiell hastighet U_1 som strømmer over et annet sjikt med initiell hastighet U_2. Da sjiktene må ha samme hastighet og skjærspenning langs den felles grenseflata, oppstår det et skjærsjikt som vist på figuren. For enkelhets skyld antar vi samme fluid i begge sjiktene. Antar dessuten at Blasius ligning beskriver skjærsjiktet.

For begge sjiktene:

$$f'''(\eta) + f(\eta) \cdot f''(\eta) = 0 \tag{3.172}$$

med randbetingelser:

$$f(0) = 0 \tag{3.173}$$
$$f'(\eta_\infty) = 1 \tag{3.174}$$
$$f'(-\eta_\infty) = \frac{U_2}{U_1} \tag{3.175}$$

Merk her at $\eta \geq 0$ for det øvre sjiktet og $\eta \leq 0$ for det nedre sjiktet. Vi har definert koordinaten η og strømfunksjonen $f(\eta)$ på følgende måte:

$$\begin{array}{c} \eta = \sqrt{\frac{U_1}{2\nu x}} \cdot y \\ f'(\eta) = \frac{U(x,y)}{U_1} \end{array} \tag{3.176}$$

(Sammenlign med lign. (C.2.7), appendiks C, del C.2 i kompendiet)

I dette tilfellet er både hastigheten $f'(0)$ og skjærspenningen $f''(0)$ ukjente, slik at vi må tippe to initialbetingelser når vi skal bruke skyteteknikk.

Får totalt følgende tre initialbetingelser:

$$f(0) = 0,\ f'(0) = r,\ f''(0) = s \tag{3.177}$$

De to ukjente initialbetingelsene må velges slik at randbetingelsene (3.174) og (3.175) blir oppfylt. Vi skriver dette på følgende måte:

$$\phi(\eta_\infty; r, s) = f'(\eta_\infty; r, s) - 1 = 0 \text{ for } s = s^*,\ r = r^* \tag{3.178}$$

$$\psi(-\eta_\infty; r, s) = f'(-\eta_\infty; r, s) - \frac{U_2}{U_1} = 0 \text{ for } s = s^*,\ r = r^* \tag{3.179}$$

Istedenfor å bruke negative verdier for η for det nedre sjiktet, velger vi å innføre koordinaten ζ der $\zeta = -\eta$. Får da følgende system:

I. For det øvre sjiktet:

$$f'''(\eta) + f(\eta) \cdot f''(\eta) = 0 \tag{3.180}$$

med rand - og initialbetingelser:

$$f'(\eta_\infty) = 1 \tag{3.181}$$
$$f(0) = 0,\ f'(0) = r,\ f''(0) = s \tag{3.182}$$
$$\phi(\eta_\infty; r, s) = f'(\eta_\infty; r, s) - 1 = 0 \tag{3.183}$$

II. For det nedre sjiktet:

$$f'''(\zeta) - f(\zeta) \cdot f''(\zeta) = 0 \tag{3.184}$$

med rand - og initialbetingelser:

$$f'(\zeta_\infty) = -\frac{U_2}{U_1} \tag{3.185}$$

$$f(0) = 0,\ f'(0) = -r,\ f''(0) = s \tag{3.186}$$

$$\psi(\zeta_\infty; r, s) = f'(\zeta_\infty; r, s) + \frac{U_2}{U_1} = 0 \tag{3.187}$$

Merk at derivasjon er m. h. p. koordinaten ζ i det nedre sjiktet.
Da $\zeta = -\eta$, får vi for eksempel:

$$\frac{df(\zeta)}{d\zeta} = \frac{df(\eta)}{d\eta} \cdot \frac{d\eta}{d\zeta} = \frac{df(\eta)}{d\eta} \cdot (-1) = -\frac{df(\eta)}{d\eta}$$

Oppgava blir da å løse (3.180), (3.182) og (3.183) simultant med (3.184), (3.186) og (3.187). Dette betyr at vi må finne verdier av r og s slik at (3.183) og (3.187) er tilfredstilt. Dette må gjøres for alle aktuelle verdier av $\frac{U_2}{U_1}$ der $\frac{U_2}{U_1} \in [0, 1]$.

Vi har da det gamle problemet med å finne gode nok startverdier for r og s når vi skal beregne nullpunktene av (3.183) og (3.187). Vi går fram som tidligere ved å beregne $\phi(\eta_\infty; r, s)$ og $\psi(\zeta_\infty; r, s)$ for passende verdier av $\eta_\infty,\ \zeta_\infty,\ r$ og s og deretter tegne funksjonene. La oss se nærmere på framgangsmåten.

Vi skal finne de verdiene av r og s som er nullpunkter både i (3.183) og (3.187). Fra Figure **??** ser vi at dette er punktet (r^*, s^*). Velger verdier for η_∞ og ζ_∞ slik at ϕ og ψ bare er funksjoner av r og s:

$$\begin{aligned} \phi(r^*, s^*) &= 0 \\ \psi(r^*, s^*) &= 0 \end{aligned} \tag{3.188}$$

Velger så en verdi $s = s_0$, se Figure **??**. Setter deretter $r = r_0$ og beregner verdiene av ϕ og ψ langs linja $s = s_0$ med varierende r. Med henvisning til figuren ovenfor, får vi fortløpende:

$$\psi(r_0, s_0) \cdot \psi(r_1, s_0) > 0,\ \psi(r_1, s_0) \cdot \psi(r_2, s_0) > 0,\ \psi(r_2, s_0) \cdot \psi(r_3, s_0) < 0$$

Den negative verdien av $\psi(r_2, s_0) \cdot \psi(r_3, s_0)$ betyr at vi har en rot av $\psi(r, s_0)$ mellom r_2 og r_3. Denne rota som er merket med 1 på figuren, kan finnes med en endimensjonal nullpunktsløser siden ψ nå bare er funksjon av r. Ved å gå videre langs $s = s_0$, finner vi nullpunktet 2 av ϕ på samme måten. Deretter begynner vi på ny s-verdi med $s_1 = s_0 + \Delta s$ og finner nullpunktene langs s_1- linja. Tilslutt har vi beregnet alle nullpunktene merket med en ring på figuren. Dermed kan vi finne en god tilnærmelse $(\bar{r}, \bar{s})$ til det korrekte nullpunktet (r^*, s^*) . Med disse tilnæringsverdiene for nullpunktene, kan vi nå løse (3.188) ved for eks. å bruke en todimensjonal versjon av Newton-Raphsons metode, se avsnitt 14.2 i C&K [4]. Figure **??** gjelder for en bestemt verdi av $\alpha = \frac{U_2}{U_1}$ slik at vi i prinsippet må gjenta søkeprosedyren for hver verdi av α. I praksis er dette sjelden nødvendig, da verdien av $(\bar{r}, \bar{s})$ for en α–verdi vanligvis kan brukes til å finne (r^*, s^*) for en nærliggende verdi av α. Prosedyren ovenfor gir følgende tabell for (r^*, s^*) som funksjon av $\frac{U_2}{U_1}$:

$\frac{U_2}{U_1}$	r^*	s^*
0.0	0.583776	0.284447
0.1	0.613568	0.267522
0.2	0.646947	0.247879
0.3	0.683594	0.225504
0.4	0.723101	0.200460
0.5	0.765049	0.172847
0.6	0.809060	0.142779
0.7	0.854807	0.110363
0.8	0.902021	0.075701
0.9	0.950480	0.038886
1.0	1.000000	0.000000

Med $\alpha = \frac{U_2}{U_1}$ finner vi følgende kurvetilpasninger for r^* og s^* fra tabellen:

$$\begin{aligned} r^* &\approx \bar{r} = 0.1076 \cdot \alpha^2 + 0.3086 \cdot \alpha + 0.5838 \\ s^* &\approx \bar{s} = -0.1224 \cdot \alpha^2 - 0.1620 \cdot \alpha + 0.2844 \end{aligned} \tag{3.189}$$

Istedenfor denne litt omstendelige fremgangsmåten, kan du forsøke programmet **fsolve** fra Matlabs Optimization Toolbox. **Marie 8**: Henviser til Matlab

Chapter 4

Differansemetoder for ordinære differensialligninger

4.1 Tridiagonale algebraiske ligningsystem

Differansemetoder anvendt på både ordinaere og partielle differensialligninger gir vanligvis algebraiske ligningsystem med utpreget bandstruktur. Ved 2. ordens differensialligninger består ofte koeffisientmatrisa for ligningsystemet av tre diagonaler: En hoveddiagonal med en diagonal på hver side av denne. En slik matrise kalles tridiagonal. Et linært, algebraisk ligningsystem med tridiagonal koeffisientmatrise kan skrives på formen:

$$\begin{aligned} & b_1x_1 + c_1x_2 = d_1 \\ & \quad \cdots \\ & a_ix_{i-1} + b_ix_i + c_ix_{i+1} = d_i \\ & \quad \cdots \\ & a_Nx_{N-1} + b_Nx_N = d_N \\ & i = 1, 2, ...N \ , a_1 = c_N = 0 \end{aligned} \tag{4.1}$$

eller på matriseform:

$$\begin{bmatrix} b_1 & c_1 & & & & & & & \\ a_2 & b_2 & c_2 & & & & & & \\ & \cdot & \cdot & \cdot & & & & & \\ & & \cdot & \cdot & \cdot & & & & \\ & & & \cdot & \cdot & \cdot & & & \\ & & & & & a_{N-1} & b_{N-1} & c_{N-1} \\ & & & & & & a_N & b_N \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_{N-1} \\ x_N \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \cdot \\ \cdot \\ \cdot \\ d_{N-1} \\ d_N \end{bmatrix} \tag{4.2}$$

(4.1) kan løses ved Gauss-eliminasjon som for tridiagonale matriser blir en enkel algoritme, ofte kalt Thomas-algoritmen.En detaljert utledning er gitt i appendiks I i kompendiet.

Eliminering:

$$\begin{aligned} & \text{Utfør for } j = 2, 3 \cdots, N \ \ q_j := \tfrac{a_j}{b_{j-1}} \\ & \text{Beregn: } b_j := b_j - q_j \cdot c_{j-1}, \quad d_j := d_j - q_j \cdot d_{j-1} \end{aligned} \tag{4.3}$$

Innsetting:

$$\begin{aligned} & \text{Sett } x_N := \tfrac{d_N}{b_N} \\ & \text{Utfør for } j = N-1, N-2, \cdots, 1 \ \ x_j : \tfrac{d_j - c_j \cdot x_{j+1}}{b_j} \end{aligned}$$

(4.3) er implementert i subprogrammet **tdma** som tilsvarer prosedyren **tri** i C&K [4] , avsnitt 6.3. Programmet tdma er vist under:

```
function x = tdma(a,b,c,d)
#  Solution of a tridiagonal matrix
#  using the Thomas-algorithm.
#  No. of equations are given by the length
#  of the main diagonal b
n = length(b);
x = zeros(size(b));
#  === Elimination ===

for k = 2:n
   q = a(k)/b(k-1);
   b(k) = b(k) - c(k-1)*q;
   d(k) = d(k) - d(k-1)*q;
end
#  === Backsubstitution ===

q = d(n)/b(n);
x(n) = q;
for k = n-1:-1:1
   q = (d(k) - c(k)*q)/b(k);
   x(k) = q;
end
```

Merk at (4.1) ikke bruker $a(1)$ og $c(n)$.

(4.3) er sikker numerisk stabil dersom følgende betingelser er oppfylt:

$$\begin{array}{l} \mid b_1 \mid > \mid c_1 \mid > 0 \\ \mid b_i \mid \geq \mid a_i \mid + \mid c_i \mid, a_i \cdot c_i \neq 0 \ , i = 2, 3, \cdots, N-1 \\ \mid b_n \mid > \mid a_N \mid > 0 \end{array} \tag{4.4}$$

Matriser som oppfyller (4.4) , kalles diagonaldominante. Dersom vi bare har ulikhetstegn i (4.4), betegnes dette som streng diagonaldominans. For diagonal-dominante matriser behøves ikke pivotering ved Gauss-eliminasjon.

Dette sparer tid samtidig som bandstrukturen beholdes. Bevis av(4.4) er gitt i appendiks I i kompendiet.

Legg merke til at i matrisa (4.2) har koeffisientene i hver linje samme indeks. Ligningsystemet i (4.1) kan også skrives:

$$\begin{array}{l} b_1 x_1 + c_2 x_2 = d_1 \\ \qquad \cdots \\ a_{i-1} x_{i-1} + b_i x_i + c_{i+1} x_{i+1} = d_i \\ \qquad \cdots \\ a_{N-1} x_{N-1} + b_N x_N = d_N \\ \\ i = 1, 2, ...N \ , a_1 = c_N = 0 \end{array} \tag{4.5}$$

eller på matriseform:

$$\begin{bmatrix} b_1 & c_2 & & & & & & \\ a_1 & b_2 & c_3 & & & & & \\ & \cdot & \cdot & \cdot & & & & \\ & & \cdot & \cdot & \cdot & & & \\ & & & \cdot & \cdot & \cdot & & \\ & & & & & a_{N-2} & b_{N-1} & c_N \\ & & & & & & a_{N-1} & b_N \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_{N-1} \\ x_N \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \cdot \\ \cdot \\ \cdot \\ d_{N-1} \\ d_N \end{bmatrix} \tag{4.6}$$

Legg merke til at i matrisa (4.6) har koeffisientene i hver *kolonne* samme indeks.

Versjonen i (4.6) fås enkelt fra (4.2) ved å trekke 1 fra a-indeksene og addere 1 til c-indeksene. Matlab lagrer tridiagonale matriser på formen (4.6).

(Se avsnittet *Glisne matriser* i Matlabelfsa).

(4.6) er programmert i funksjonen **tridiag**.

Av gammel vane bruker vi vanligvis funksjonen **tdma** i kurset. Dersom betingelsene i (4.4) ikke er oppfylt, kan f.eks. **tripiv** brukes. (Se appendiks I i kompendiet). I appendiks A, del A.5, står det mer om løsningsnøyaktigheten for lineæreligningsystem, spesielt tridiagonale system.

4.2 Varmeledning

4.2.1 Kjøleribbe med konstant tverrsnitt

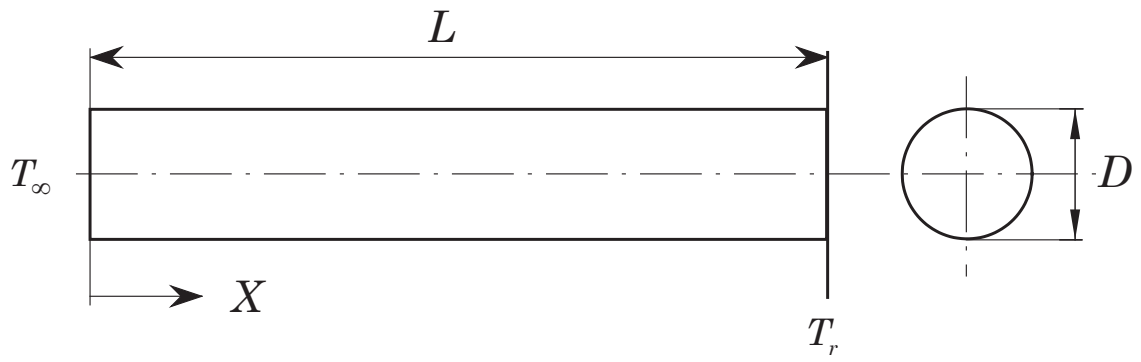


Figure 4.1:

Figure 4.1 viser en sirkulærsylindrisk stav med lengde L, omgivelsestemperatur T_∞ og en konstant temperatur T_r for $X = L$.

Andre data: $\bar{h} = 100W/m^2\ ^\circ C$, $k = 200W/m/^\circ C$

Fra ligning (B.0.22), appendiks B i kompendiet:

$$\frac{d}{dX}\left[A(X)\cdot\frac{d(T-T_\infty}{dX}\right] = \frac{\bar{h}P}{k}(T-T_\infty) \tag{4.7}$$

For konstant tverrsnitt:

$$\frac{d^2}{dX^2}(T-T_\infty) = \frac{\bar{h}P}{kA}(T-T_\infty) \tag{4.8}$$

Innfører dimensjonsløse variable:

$$x = \frac{X}{L}\ ,\ \theta = \frac{T-T_\infty}{T_r-T_\infty}\ ,\ \beta^2 = \frac{\bar{h}P}{kA}L^2 \tag{4.9}$$

der x er en dimensjonsløs lengde (se figur B.5 i kompendiet) og θ en dimensjonsløs temperatur. Temperaturen T_r er en referansetemperatur. Parameteren β^2 kalles ofte Biot-tallet. (Vanlig betegnelse for Biot-tallet er Bi der $Bi = \beta^2$).

Lign. (4.7) kan nå skrives:

$$\frac{d^2\theta}{dx^2} - \beta^2\theta = 0 \tag{4.10}$$

med analytisk løsning:

$$\theta(x) = A\sinh(\beta x) = -B\cosh(\beta x) \tag{4.11}$$

der konstantene A og B bestemmes fra randbetingelsene.
Ser på et tilfelle med følgende to sett av randbetingelser:
I)

$$T = T_\infty \text{ for } X = 0 \ , \ T = T_r \text{ for } X = L$$

II)

$$Q_x = 0 = \frac{dT}{dX} \text{ for } X = 0 \ , \ T = T_r \text{ for } X = L$$

Tilfelle II betyr at staven er isolert for $X = 0$
Randbetingelsene på dimensjonsløs form:
I)

$$\theta = 0 \text{ for } x = 0 \ , \ \theta = 1 \text{ for } x = 1 \tag{4.12}$$

II)

$$\frac{d\theta}{dx} = 0 \text{ for } x = 0 \ , \ \theta = 1 \text{ for } x = 1 \tag{4.13}$$

Randbetingelsene i (4.12) og (4.13) gir følgende uttrykk for temperatur og temperaturgradient:
I)

$$\theta(x) = \frac{\sinh(\beta x)}{\sinh(\beta)} \ , \ \frac{d\theta}{dx} = \beta \cdot \frac{\cosh(\beta x)}{\sinh(\beta)} \tag{4.14}$$

II)

$$\theta(x) = \frac{\cosh(\beta x)}{\cosh(\beta)} \ , \ \frac{d\theta}{dx} = \beta \cdot \frac{\sinh(\beta x)}{\cosh(\beta)} \tag{4.15}$$

For eksemplet ovenfor blir Biot-tallet:

$$\beta^2 = \frac{\bar{h}P}{kA} \cdot L^2 = \frac{2}{D} \cdot L^2 \ , \ L \text{ og } D \text{ i meter} \tag{4.16}$$

Men $D = 0.02m$ og $L = 0.2m$ blir f.eks. $\beta^2 = 4$. En fordobling av lengden gir en 4-dobling av Biot-tallet.

Numerisk løsning

Vi bruker sentraldifferanser for $\frac{d^2\theta}{dx^2}$, og med $\left.\frac{d^2\theta}{dx^2}\right|_i \approx \frac{\theta_{i-1} - 2\theta_i + \theta_{i+1}}{h^2}$ får vi følgende differanseligning:

$$\theta_{i-1} - (2 + \beta^2 h^2) \cdot \theta_i + \theta_{i+1} = 0 \tag{4.17}$$

Tilfelle 1

Vi nummererer punktene som vist på Figure 4.2:
x-koordinatene er her gitt ved: $x_i = i \cdot h \ , \ i = 0, 1, ..., N + 1$ der $h = \frac{1}{N+1}$.
Ligning (4.17) utskrevet for $i = 1$:

$$\theta_0 - (2 + \beta^2 h^2) \cdot \theta_1 + \theta_2 = 0 \rightarrow (2 + \beta^2 h^2) \cdot \theta_1 + \theta_2 = 0$$

da $\theta(0) = \theta_0 = 0$
Ligning (4.17) utskrevet for $i = N$:

$$\theta_{N-1} - (2 + \beta^2 h^2) \cdot \theta_N + \theta_{N+1} = 0$$

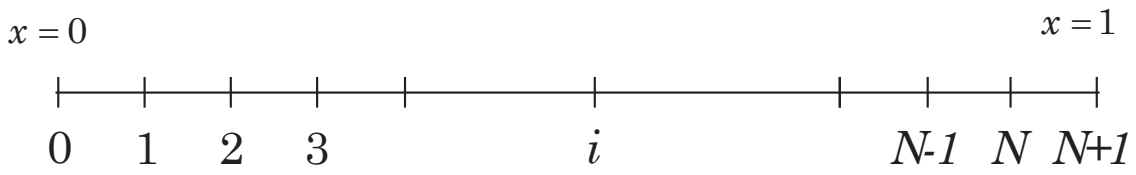


Figure 4.2:

som med $\theta_{N+1} = 1$ gir:

$$\theta_{N-1} - (2 + \beta^2 h^2) \cdot \theta_N = -1$$

Fullstendig ligningsett:

$$\begin{aligned} i = 1: &\quad -(2 + \beta^2 h^2) \cdot \theta_1 + \theta_2 = 0 \\ i = 2, 3, ..., N-1: &\quad \theta_{i-1} - (2 + \beta^2 h^2) \cdot \theta_i + \theta_{i+1} = 0 \qquad (4.18) \\ i = N: &\quad \theta_{N-1} - (2 + \beta^2 h^2) \cdot \theta_N = -1 \end{aligned}$$

(4.19)

I appendiks A, del A.5, eksempel A.15 i kompendiet, er dette systemet behandlet mer inngående. Ved å sammenligne med (4.1), får vi følgende sett med koeffisienter:

$$a_i = 1\ ,\ i = 2, 3, ..., N \qquad (4.20)$$

$$b_i = -(2 + \beta^2 h^2)\ , i = 1, 2, ..., N \qquad (4.21)$$

$$c_i = 1\ ,\ i = 1, 2, ..., N-1 \qquad (4.22)$$

$$d_i = 0\ ,\ i = 1, 2, ..., N-1\ ,\ d_N = -1 \qquad (4.23)$$

Programmet **ribbe1** nedenfor bruker koeffisientene i (4.20) og løser systemet med bruk av **tdma** . Den analytiske løsningen er gitt i (4.14). Minner om at **tdma** ikke bruker a_1 og c_N.

```
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'
```

Resultatene i tabell:

x	numerisk	analytisk	rel. feil
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

Tilfelle 2. Versjon 1

Vi nummererer nå punktene som vist på fig. (4.3). Årsaken er at programmet **tdma** venter at den første ukjente har nummer 1, og her er det temperaturen i $x = 0$ som er den første ukjente, mens vi i tilfelle 1 hadde den første ukjente for $x = h$. (Det er egenltig ikke nødvendig å omnummerere)

x-koordinatene er nå gitt ved: $x_i = (i-1) \cdot h,\ i = 1, 2, ..., N+1$ der $h = \frac{1}{N}$ Bruker her sentraldifferanser for $\frac{d\theta}{dx}$ i forbindelse med randbetingelsen $x = 0$

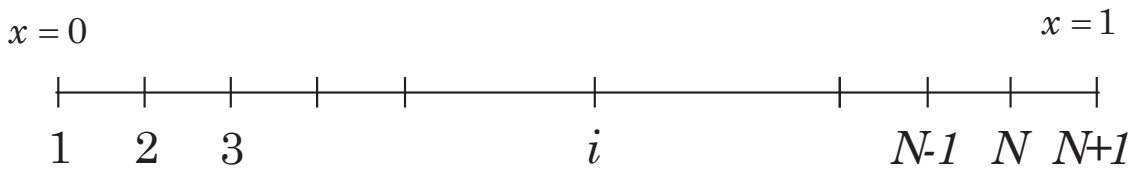


Figure 4.3:

$$\left.\frac{d\theta}{dx}\right|_i \approx \frac{\theta_{i+1} - \theta_{i-1}}{2h} \tag{4.24}$$

Da $\frac{d\theta_1}{dx_1} = 0$: $\frac{\theta_2 - \theta_0}{2h} = 0 \rightarrow \theta_0 = \theta_2$ som insatt i (4.17) gir:

$$-(2 + \beta^2 h^2) \cdot \theta_1 + 2\theta_2 = 0$$

Dette er den første ligningen i settet og er den eneste som er forskjellig fra (4.18). Istedenfor (4.22) , får vi nå:

$$c_1 = 2,\ c_i = 1,\ i = 2, ...N - 1 \tag{4.25}$$

Tilfelle 2. Versjon 2
Istedenfor sentraldifferanser, bruker vi foroverdifferanser gitt ved lign. (4.17) i kap. 1 for den deriverte:

$$\left.\frac{d\theta}{dx}\right|_i \approx \frac{-3\theta_i + 4\theta_{i+1} - \theta_{i+2}}{2h} \tag{4.26}$$

$$\left.\frac{d\theta}{dx}\right|_1 = \frac{-3\theta_1 + 4\theta_2 - \theta_3}{2h} = 0$$

som videre gir: $-3\theta_1 + 4\theta_2 - \theta_3 = 0 \rightarrow \theta_3 = 4\theta_2 - 3\theta_1$
Differanseligningen (4.17) utskrevet for $i = 2$:

$$\theta_1 - (2 + \beta^2 h^2) \cdot \theta_2 + \theta_3 = 0$$

innsatt for θ_3:

$$\theta_1 - (2 + \beta^2 h^2) \cdot \theta_2 + 4\theta_2 - 3\theta_1 = 0 \rightarrow$$

$$2\theta_1 - (2 + \beta^2 h^2) \cdot \theta_2 = 0 \tag{4.27}$$

Dette er den første ligningen i settet og er den eneste som er forskjellig fra (4.18). Istedenfor (4.21) og (4.22) , får vi nå:

$$b_1 = 2\ ,\ b_i = (2 + \beta^2 h^2)\ ,\ i = 2, ..., N \tag{4.28}$$

$$c_1 = -(2 - \beta^2 h^2)\ ,\ c_i = 1\ ,\ i = 2, ..., N - 1 \tag{4.29}$$

For oversiktens skyld skriver vi opp hele settet:

$$\begin{aligned} i = 1 &: \ 2\theta_1 - (2 - \beta^2 h^2) \cdot \theta_2 = 0 \\ i = 2, 3, ..., N - 1 &: \ \theta_{i-1} - (2 + \beta^2 h^2) \cdot \theta_i + \theta_{i+1} = 0 \\ i = N &: \ \theta_{N-1} - (2 + \beta^2 h^2) \cdot \theta_N = -1 \end{aligned} \tag{4.30}$$

Legg merke til at vi her måtte kombinere uttrykket for den deriverte med differanseligningen for å få en tridiagonal koeffisientmatrise. Løser de to versjonene med $\beta = 2$ og $h = 0.1$. Nedenfor er programmet **ribbe2** som løser tilfelle 2 med begge versjonene.

```
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
version1=1
if (version1==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               # all elts in first row is set to 1
diagonals[1,:]= -(2+(beta*h)**2)
diagonals[2,:]= 1.0
diagonals[2,1]= 2.0               # particular value due to derivative bc
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),':')
#plot(x[0:-1],theta2,'-.',x,theta_analytical(beta,x),':')
legend(['sparse','linalg','analytical'])
show()
close()
print 'done'
```

Den relative feilen er beregnet fra $\varepsilon_{rel} = |(\theta_{num} - \theta_{analyt})/\theta_{analyt}|$. Resultatet av beregningen er gitt i tabellen under:

x	Analyt.	Sentr.diff	Rel. feil	Forov.diff	Rel. feil
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

Vi ser at de to uttrykkene for den deriverte randbetingelsen gir omtrent samme nøyaktighet unntatt nær $x = 0$, der foroverdifferansene er en del bedre. I avsnitt (4.6) ser vi nærmere på nøyaktigheten av deriverte randbetingelser.

Istedenfor nummereringen i fig. (4.3), kan vi bruke nummereringen i fig. (4.2) med $x_i = i \cdot h$, $i = 0, 1, ..., N + 1$ der $h = \frac{1}{N+1}$ slik at vi med å sette $i = 1$ i (4.17) får:

$$\theta_0 - (2 + \beta^2 h^2) \cdot \theta_1 + \theta_2 = 0$$

Randbetingelsen i (4.26) blir nå:

$$\left.\frac{d\theta}{dx}\right|_0 = \frac{-3\theta_0 + 4\theta_1 - \theta_2}{2h} = 0$$

som gir:

$$\theta_0 = 4(\theta_1 - \theta_2)/3 \tag{4.31}$$

som innsatt ovenfor gir ligning nr. 1:

$$-(2 + 3\beta^2 h^2) \cdot \theta_1 + 2\theta_2 = 0 \tag{4.32}$$

Etter at systemet er løst, beregnes θ_0 fra (4.31).

4.2.2 Ribbe med variabelt tverrsnitt

Figure 4.4 viser en trapesformet kjøleribbe med lengde L og bredde b. Tykkelsen variere fra d for $X = 0$ til D for $X = L$. Ribba har varmetap for $X = 0$ og har en gitt temperatur T_L for $X = L$. Omgivelsestemperaturen er konstant lik T_∞ .

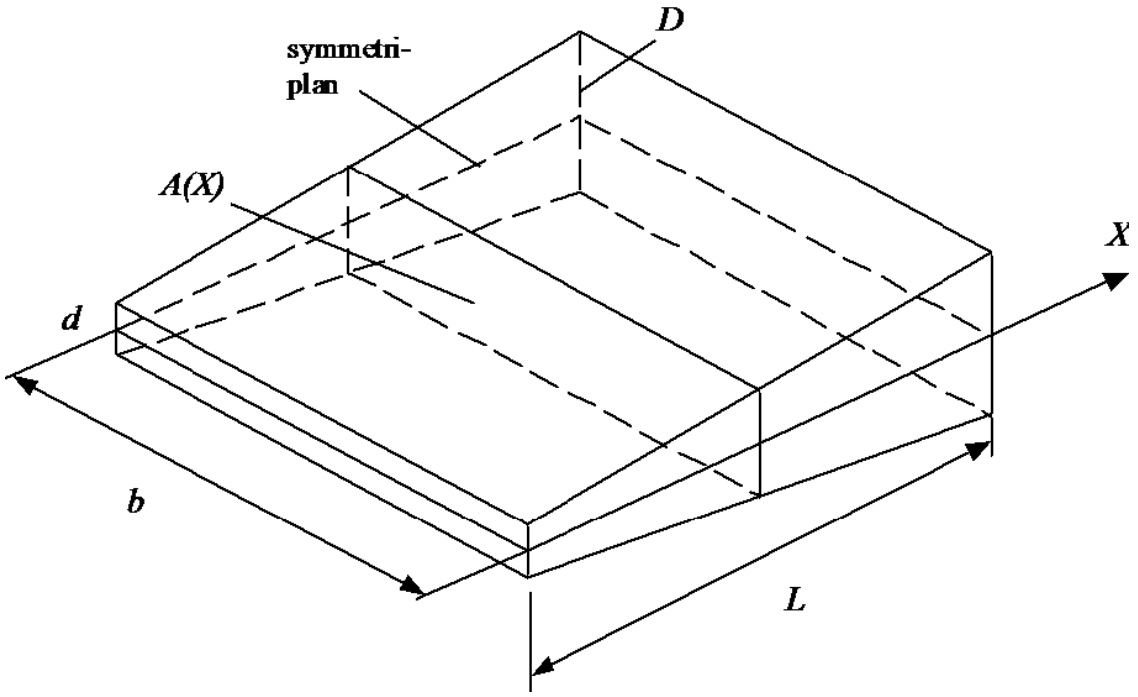


Figure 4.4:

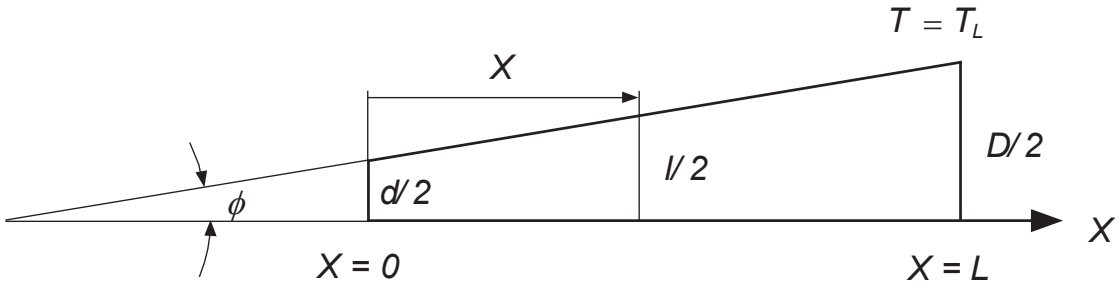


Figure 4.5:

Figure 4.5 viser øvre halvpart av ribba. Av denne figuren finner vi følgende relasjon:

$$\tan(\phi) = \frac{D/2 - d/2}{L} = \frac{l/2 - d/2}{X}$$

som løst m.h.p. l gir:

$$l = d + \left(\frac{D-d}{L}\right) \cdot X = D \cdot \left[\frac{d}{D} + \left(1 - \frac{d}{D}\right) \cdot \frac{X}{L}\right] \tag{4.33}$$

Vi forutsetter at temperaturen varierer hovedsakelig i X-retning slik at utledningen i appendiks B i kompendiet kan brukes. Antar derfor at D er liten, D $\ll$ L samt $0 < \frac{d}{D} < 1$.

Fra appendiks B, lign. (B.0.22) i kompendiet:

$$\frac{d}{dX}\left[A(X) \cdot \frac{d(T - T_\infty)}{dX}\right] = \frac{\bar{h}P(X)}{k}(T - T_\infty) \tag{4.34}$$

randbetingelser:

$$\frac{dT(0)}{dX} = \frac{\bar{h}_0}{k}[T(0) - T_\infty] \ , \ T(L) = T_L \tag{4.35}$$

For trapestverrsnittet i fig. (4.4):

$$P(X) = 2(b + l) \approx 2b \text{ for } l \ll b \text{ og } A(X) = b \cdot l$$

Innfører følgende dimensjonsløse størrelser:

Temperatur $\theta = \dfrac{T - T_\infty}{T_L - T_\infty}$, lengde $x = \dfrac{X}{L}$, $\alpha = \dfrac{d}{D}$, $0 < \alpha < 1$ samt:

$$\text{Biot-tallene } \beta^2 = \frac{2\bar{h} \cdot L^2}{D \cdot k} \text{ og } \beta_0^2 = \frac{\bar{h}_0}{k} \cdot L \tag{4.36}$$

Med bruk av disse dimensjonsløse størrelsene kan (4.34) skrives:

$$\frac{d}{dx}\left[\{\alpha + (1 - \alpha) \cdot x\}\frac{d\theta(x)}{dx}\right] - \beta^2\theta(x) = 0 \tag{4.37}$$

med randbetingelser:

$$\frac{d\theta}{dx}(0) = \beta_0^2 \cdot \theta(0) \ , \ \theta(1) = 1 \tag{4.38}$$

Når $d = 0$ som betyr $\alpha = 0$, går trapesprofilet over til et trekantprofil.
Lign. (4.37) blir nå:

$$x\frac{d^2\theta}{dx^2} + \frac{d\theta}{dx} - \beta^2\theta(x) = 0 \tag{4.39}$$

For $x = 0$:

$$\frac{d\theta}{dx}(0) - \beta^2\theta(0) = 0 \tag{4.40}$$

Den analytiske løsningen av (4.37) og (4.39) er gitt i appendiks G, del G.5 i kompendiet.

4.2.3 Example: Talleksempel for trapesprofilet

$$\begin{aligned} &L = 0.1m, \ D = 0.01m, \ d = 0.005m, \ \bar{h} = 80W/m^2/°C, \bar{h}_0 = \\ &200W/m^2/°C, \text{og } k = 40W/m/°C \text{ som gir } \beta^2 = 4.0 \ , \ \alpha = \tfrac{1}{2} \text{ og } \beta_0^2 = 0.5 \end{aligned} \tag{4.41}$$

(4.37) blir nå:

$$\frac{d}{dx}\left[(1 + x) \cdot \frac{d\theta}{dx}\right] - 8 \cdot \theta(x) = 0 \tag{4.42}$$

Et Matlab-program som beregner temperaturen θ samt gradienten θ' for den analytiske løsningen, der x går fra 0 til 1 med skritt $\Delta x = 0.1$, er gitt i appendiks G, del G.5 i kompendiet .
Programmet gir følgende tabell:

x	θ (x)	θ'(x)
0.00	0.18069	0.09034
0.10	0.19623	0.21842
0.20	0.22415	0.33967
0.30	0.26424	0.46318
0.40	0.31708	0.59556
0.50	0.38383	0.74211
0.60	0.46614	0.90754
0.70	0.56611	1.09629
0.80	0.68632	1.31291
0.90	0.82978	1.56211
1.00	1.00000	1.84901

Numerisk løsning av homogent trapesprofil
Vi vil nå løse lign. (4.42) numerisk med bruk av sentraldifferanser

$$(1+x)\frac{d^2\theta}{dx^2} + \frac{d\theta}{dx} - 8\cdot\theta(x) = 0 \tag{4.43}$$

med randbetingelser:

$$\frac{d\theta}{dx}(0) = \beta_0^2\cdot\theta(0) = \frac{\theta(0)}{2}\ ,\ \theta(1) = 1 \tag{4.44}$$

Fremgangsmåten blir som for tilfelle 2, versjon 2, avsnitt (4.2.1) med samme nummerering.

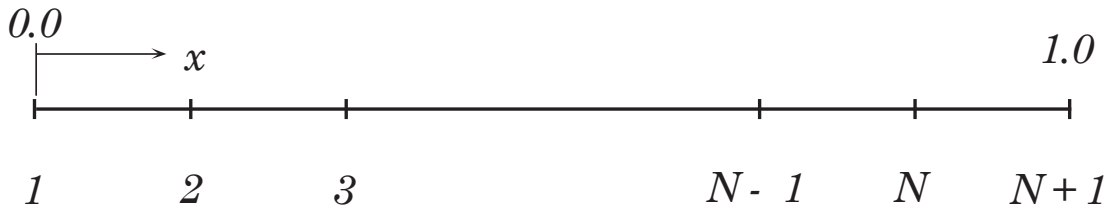


Figure 4.6:

x-koordinatene er gitt ved: $x_i = (i-1)\cdot h$, $i = 1, 2, ..., N+1$ der $h = \frac{1}{N}$.
Diskretisering:

$$(1+x_i)\cdot\frac{\theta_{i+1} - 2\theta_i + \theta_{i-1}}{h^2} + \frac{\theta_{i+1} - \theta_{i-1}}{2h} - 8\theta_i = 0$$

som ordnet blir:

$$-(1-\gamma_i)\cdot\theta_{i-1} + 2\cdot(1+8h\cdot\gamma_i)\cdot\theta_i - (1+\gamma_i)\cdot\theta_{i+1} = 0 \tag{4.45}$$

$$\text{der } \gamma_i = \frac{h}{2(1+x_i)} = \frac{h}{2[1+(i-1)\cdot h]}\ ,\ i = 1, 2, ..., N+1 \tag{4.46}$$

For randbetingelsen i (4.44) får vi: $\frac{\theta_2-\theta_0}{2h} = \frac{\theta_1}{2} \rightarrow \theta_0 = \theta_2 - \theta_1\cdot h$ som innsatt i (4.45) for $i = 1$ gir:

$$[2 + h\cdot(1+15\cdot\gamma_1)]\cdot\theta_1 - 2\theta_2 = 0 \tag{4.47}$$

For $i = N$:

$$-(1-\gamma_N)\cdot\theta_{N-1} + 2\cdot(1+8h\cdot\gamma_N)\cdot\theta_N = 1+\gamma_N \tag{4.48}$$

For $i = 2, 3, ..., N-1$ bruker vi (4.45). Vi har da et tridiagonalt ligningsett som kan løses med Thomas-algoritmen . Tabellen nedenfor viser beregningene med to forskjellige verdier av h. Vi ser at feilen er 100 ganger mindre for $h = 0.01$ enn for $h = 0.1$ som betyr at feilen er av $O(h^2)$; i overenstemmelse med teorien. Den relative feilen er beregnet fra $|(\theta_{numerisk} - \theta_{analytisk})/\theta_{analytisk}|$

$h = 0.1$	x	$\theta(x)$	rel. feil	$h = 0.01$	x	$\theta(x)$	rel. feil
	0.0	0.18054	8.000E-4		0.0	0.18069	4.866E-6
	0.1	0.19634	5.619E-4		0.1	0.19623	1.010E-5
	0.2	0.22442	1.224E-3		0.2	0.22415	1.669E-5
	0.3	0.26462	1.436E-3		0.3	0.264259	1.703E-5
	0.4	0.31752	1.394E-3		0.4	0.31709	1.325E-5
	0.5	0.38430	1.222E-3		0.5	0.38383	9.861E-6
	0.6	0.46660	9.910E-4		0.6	0.46614	6.393E-6
	0.7	0.56653	7.385E-4		0.7	0.56611	2.737E-6
	0.8	0.68665	4.838E-4		0.8	0.68632	4.342E-7
	0.9	0.82998	2.364E-4		0.9	0.82978	1.508E-6
	1.0	1.00000	0		1.0	1.00000	0

Numerisk løsning av sammensatt trekantprofil

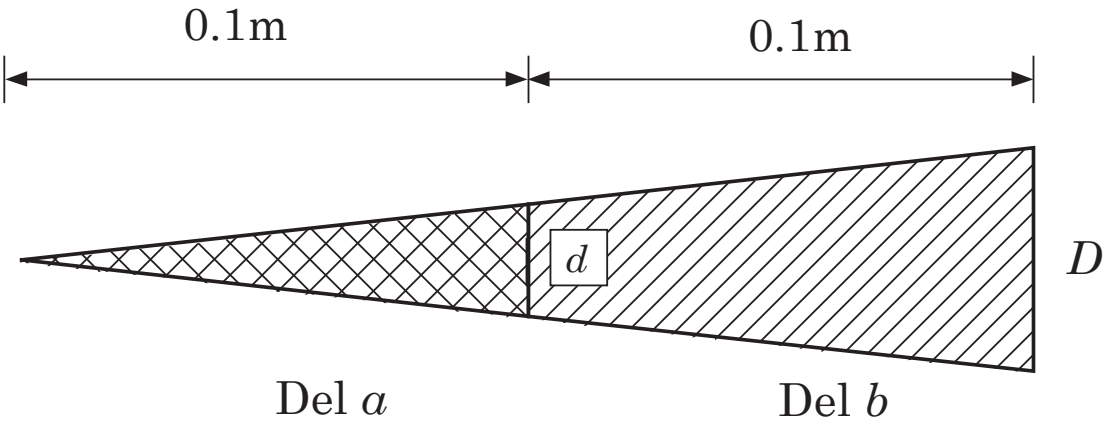


Figure 4.7:

Figure 4.7 viser en trekantformet kjøleribbe sammensatt av en trekant (del a) og et trapes (del b) der del a og b er laget av forskjellige materialer. Hensikten med dette tilfellet er å vise hvordan vi kan behandle diskontinuiteter. Her er temperaturen kontinuerlig, mens temperaturgradienten er diskontinuerlig på grenseflata mellom de to delene p.g.a. forskjellige verdier for varmeledningstallene. Fra appendiks B i kompendiet:

$$-dQ_x = P \cdot h \cdot [T(X) - T_\infty] \cdot dX \text{ der } Q_x = -kA\frac{dT}{dX}$$

For $dX \to 0$ får vi da $dQ \to 0 \Rightarrow Q = konstant$ som betyr at på grenseflata mellom de to legemene gjelder følgende relasjon:

$$Q_a = Q_b \Rightarrow k_a A_a \left(\frac{dT}{dX}\right)_a = k_b A_b \left(\frac{dT}{dX}\right)_b \tag{4.49}$$

Da $A_a = A_b$ på grenseflata, følger: $\left(\frac{dT}{dX}\right)_b = \frac{k_a}{k_b}\left(\frac{dT}{dX}\right)_a$ som på dimensjonsløs form (med bruk av (4.36) blir:

$$\left(\frac{d\theta}{dx}\right)_b = \frac{k_a}{k_b}\left(\frac{d\theta}{dX}\right)_a \tag{4.50}$$

Bruker følgende tallverdier:

$$\begin{aligned} &L = 0.2m\ ,\ D = 0.02m\ ,\ d = 0.01m\ ,\ \bar{h}_a = \bar{h}_b = 80W/m^2/^\circ C\ , \\ &k_a = 160W/m/^\circ C \text{ og } k_b = 40W/m/^\circ C \end{aligned} \tag{4.51}$$

Med disse tallverdiene blir (4.50):

$$\left(\frac{d\theta}{dx}\right)_b = 4 \cdot \left(\frac{d\theta}{dx}\right)_a \tag{4.52}$$

Differensialligning:

$$\frac{d}{dx}\left(\frac{x}{\beta^2}\frac{d\theta}{dx}\right) - \theta(x) = 0 \tag{4.53}$$

$$\beta^2 = \frac{2\bar{h}\cdot L^2}{D\cdot k} \text{ som gir } \beta_a^2 = 2.0 \text{ og } \beta_b^2 = 8.0 \tag{4.54}$$

Ved å skrive (4.53) på den viste formen, har vi oppnådd kontinuiteten i (4.49)

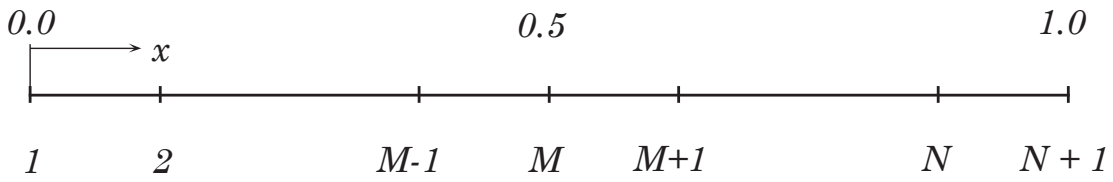


Figure 4.8:

Med henvisning til Figure 4.8 bruker vi følgende nummerering:

$$\begin{aligned} &x_i = (i-1)\cdot h\ ,\ i = 1, 2, ..., N+1 \text{ der } h = \tfrac{1}{N} \\ &M = \tfrac{N}{2} + 1 \text{ der N er et partall og } x_M = 0.5 \end{aligned} \tag{4.55}$$

Med $k = k_a$ for $i = M$ får vi at $\beta = \beta_a$ for $i = 1, 2, ..., M$ og $\beta = \beta_b$ for $i > M$. For $i \neq M$ kan vi skrive (4.53)

$$\frac{d}{dx}\left(x\frac{d\theta}{dx}\right) - \beta^2\cdot\theta(x) = 0 \tag{4.56}$$

Diskretiserer (4.56)) med bruk av (2.36)) i kap. (2):

$$-x_{i-\frac{1}{2}}\cdot\theta_{i-1} + (x_{i+\frac{1}{2}} + x_{i-\frac{1}{2}} + \beta^2h^2)\cdot\theta_i - x_{i+\frac{1}{2}}\cdot\theta_{i+1} = 0$$

Innsatt for $x_{i-\frac{1}{2}} = (i-\frac{3}{2})\cdot h$ og $x_{i+\frac{1}{2}} = (i-\frac{1}{2})\cdot h$ får vi:

$$\left(i-\frac{3}{2}\right)\cdot\theta_{i-1} + [2(i-1) + \beta^2h]\cdot\theta_i - \left(i-\frac{1}{2}\right)\cdot\theta_{i+1} = 0$$

eller dividert med i:

$$-\left(1-\frac{3}{2i}\right)\cdot\theta_{i-1} + \left[2\cdot\left(1-\frac{1}{i}\right) + \frac{\beta^2h}{i}\right]\cdot\theta_i - \left(1-\frac{1}{2i}\right)\cdot\theta_{i+1} = 0 \tag{4.57}$$

(4.57) brukes for alle verdier av $i = 2, 3, ..., N$ unntatt for $i = M = N/2 + 1$
For i = 1
Fra (4.40) får vi for randbetingelsen for $x = 0$:
$\frac{d\theta}{dx}(0) = \beta_a^2 \cdot \theta(0) \rightarrow \frac{-3\theta_1 + 4\theta_2 - \theta_3}{2h} = \beta_a^2 \cdot \theta_1$ hvor vi har benyttet (2.6)
Dette gir:

$$\theta_3 = 4\theta_2 - (3 + 2h\beta_a^2) \cdot \theta_1 \tag{4.58}$$

Skriver ut (4.57) for i = 2:

$$-\left(1 - \frac{3}{4}\right) \cdot \theta_1 + \left[2 \cdot \left(1 - \frac{1}{2}\right) + \frac{\beta_a^2 h}{2}\right] \cdot \theta_2 - \left(1 - \frac{1}{4}\right) \cdot \theta_3$$

som innsatt fra (4.54) og (4.58) gir:

$$\left(1 + \frac{3h}{2}\right) \cdot \theta_1 - \left(1 - \frac{h}{2}\right) \cdot \theta_2 = 0 \tag{4.59}$$

Dette er den første ligningen.
For i = M.

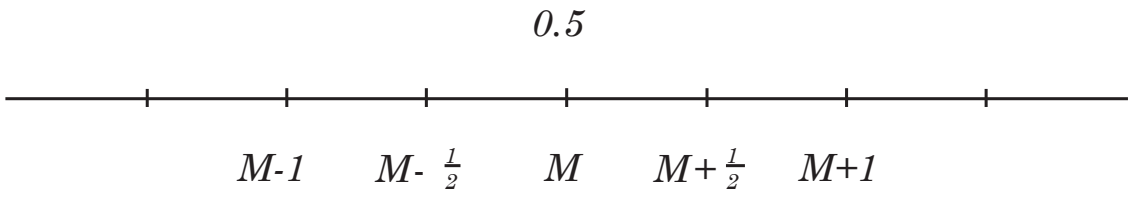


Figure 4.9:

Her bruker vi ligningen på formen gitt i (4.53)) som diskretisert med bruk av (2.36)) fra kap. (2) gir:

$$\beta_{M-\frac{1}{2}}^2 = \beta_a^2 = 2.0 \ , \ \beta_{M+\frac{1}{2}}^2 = \beta_b^2 = 8.0$$
$$x_{M-\frac{1}{2}} = \frac{h}{2}(N-1) = \frac{(1-h)}{2} \ , \ x_{M+\frac{1}{2}} = \frac{h}{2}(N+1) = \frac{(1+h)}{2}$$

som gir:

$$-4 \cdot (1-h) \cdot \theta_{M-1} + [5 - 3h + 16h^2] \cdot \theta_M - (1+h) \cdot \theta_{M+1} = 0 \tag{4.60}$$

For i = N kan vi bruke (4.57) med $\beta^2 = \beta_b^2 = 8.0$:

$$-\left(1 - \frac{3h}{2}\right) \cdot \theta_{N-1} + 2 \cdot [(1-h) + 4h^2] \cdot \theta_N = \left(1 - \frac{h}{2}\right) \tag{4.61}$$

Ligning (4.59)–(4.61) er et lineært ligningsystem på tridiagonal form som kan løses med Thomasalgoritmen. I tillegg til temperaturen ønsker vi også å beregne temperaturgradienten $\theta'(x)$. For $x = 0$ bruker vi (4.40) og sprangverdien $\frac{d\theta}{dx}(0.5+) \equiv \left(\frac{d\theta}{dx}\right)_b$ finnes fra (4.52). De andre verdiene beregnes med vanlige sentraldifferanser, med 2. ordens bakoverdifferanser for x = 0.5 og x = 1.0 . La oss se på bruken av differensial-ligningen som et alternativ.

Vi integrerer (4.56):

$$\frac{d\theta_i}{dx} = \frac{\beta^2}{x_i} \int_{x_1}^{x_i} \theta(x) dx \ , \ i = 2, 3, ..., N+1 \tag{4.62}$$

Integralet $I = \int_{x_1}^{x_i} \theta(x)dx$ beregnes f.eks ved bruk av trapesmetoden . Beregningen av (4.62) er vist i pseudokode nedenfor:

$$\begin{aligned} \theta'_1 &:= \beta_a^2 \cdot \theta_1 \\ s &:= 0 \\ \text{Utfør for } i &:= 2, ..., N+1 \\ x &:= h \cdot (i-1) \\ s &:= s + 0.5h \cdot (\theta_i + \theta_{i-1}) \end{aligned}$$

$$\text{Dersom } (i \leq M) \text{ sett } \theta'_i := \frac{\beta_a^2 \cdot s}{x} \text{ ellers } \theta'_i := \frac{\beta_b^2 \cdot s}{x}$$

I tabellen på neste side har vi brukt (4.62) og trapesmetoden. I dette tilfellet er nøyaktigheten av de to metodene for beregning av $\theta'(x)$ temmelig sammenfallende da både θ og θ' er glatte funksjoner (bortsett fra x = 0.5), men generelt vil integrasjonsmetoden gjerne være mer nøyaktig dersom det er diskontinuiteter.

Løsning av lign. (4.53) med $h = 0.01$

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

4.3 To-punktsmetode. Varmeveksler.

Et system av p første ordens differensialligninger kan skrives:

$$y'_i(x) = f_i(x, y_1, y_2, ..., y_p) \ , \ i = 1, 2, ..., p$$

Dersom randverdiene er foreskrevet for $x = a$ og $x = b$, kan vi skrive randverdiproblemet på vektorform:

$$\mathbf{y'}(x) = \mathbf{f}(x, \mathbf{y}) \tag{4.63}$$

$$\mathbf{A} \cdot \mathbf{y}(a) + \mathbf{B} \cdot \mathbf{y}(b) = \mathbf{c} \tag{4.64}$$

der $\mathbf{A}$ og $\mathbf{B}$ er p x p - matriser.

Randbetingelsene må være lineære, dvs. elementene i $\mathbf{A}$ og $\mathbf{B}$ er konstanter, dersom de skal kunne skrives på formen i (4.64) Det eksisterer selvfølgelig mer generelle randbetingelser; f.eks: $\mathbf{g}(\mathbf{y}(a), \mathbf{y}(b)) = 0$, men (4.64) dekker en lang rekke tilfeller.

4.3.1 Example

Fra lign.(4.10) og (4.12):

$$\frac{d^2\theta}{dx^2} - \beta^2\theta(x) = 0$$
$$\theta'(0) = 0 \ , \ \theta(1) = 1$$

Med notasjonen i (4.63) kan dette systemet skrives:

$$y_1' = y_2$$
$$y_2' = \beta^2 y_1$$

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} y_1(0) \\ y_2(0) \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} y_1(1) \\ y_2(1) \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Vi diskretiserer (4.63) med bruk av sentraldifferanser. Da (4.63) er et sett av 1. ordens ligninger, velger vi å utvikle differanseutrykkene halveis mellom to punkt; i dette tilfellet rundt $x_{j-\frac{1}{2}}$ som vist på Figure 4.10.

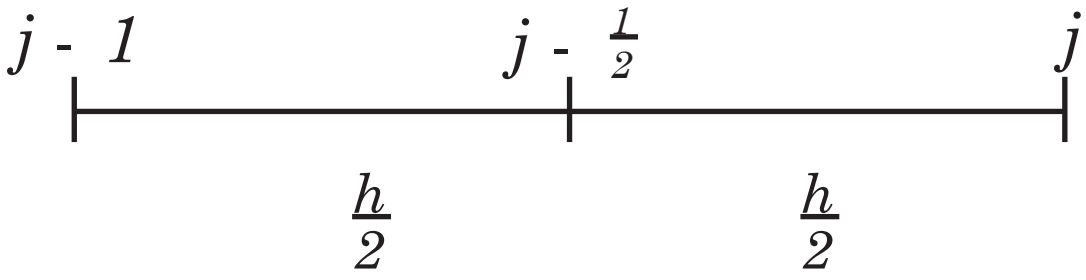


Figure 4.10:

$$\left.\frac{d\mathbf{y}}{dx}\right|_{j-\frac{1}{2}} = \frac{\mathbf{y}_j - \mathbf{y}_{j-1}}{h} + O(h^2) \ , \ \mathbf{y}_{j-\frac{1}{2}} = \frac{1}{2}(\mathbf{y}_j + \mathbf{y}_{j-1}) + O(h^2) \tag{4.65}$$

(4.65) brukt i (4.63) gir:

$$\frac{\mathbf{y}_j - \mathbf{y}_{j-1}}{h} - \mathbf{f}\left(x_{j-\frac{1}{2}}, \frac{\mathbf{y}_j + \mathbf{y}_{j-1}}{h}\right) = 0 \ , \ j = 1, 2, ..., N \tag{4.66}$$

med randbetingelser:

$$\mathbf{A} \cdot \mathbf{y}_0 + \mathbf{B} \cdot \mathbf{y}_N = \mathbf{c} \tag{4.67}$$

Notasjon:

$$x_j = a + j \cdot h, \ j = 0, 1, ...N \ , \ h = \frac{b-a}{N} \tag{4.68}$$

Med denne diskretiseringen kan det vises at følgende relasjon gjelder:

$$\mathbf{y}(x_j) = \mathbf{y}_j(h) + h^2\mathbf{e}(x_j) + O(h^4) \ , \ j = 0, 1, ..., N \tag{4.69}$$

Her er $\mathbf{y}_j(h)$ de diskretiserte verdiene ved skrittlengden h og $\mathbf{e}(x_j)$ er en funksjon som kan relateres til den gitte differensialligningen. (Bevis for de forskjellige teoremene kan finnes i Keller [8] , kapittel 3). **Marie 9**: Nå står G. E. Forsythe som forfatter. Skal det egentlig være H. B. Keller?

På denne måten får vi bare to forskjellige indekser i hver ligning ; derav navnet to-punktsmetoden. Den vesentlige fordelen med denne metoden er normalt i forbindelse med randbetingelsene. En mindre bakdel er

at vi selv for 2. ordens differensialligninger må bruke en generalisert versjon av Thomas-algoritmen, men disse generaliserte versjonene er også effektive. Utvider vi to-punktsmetoden til flere dimensjoner/tidsvariasjon, kalles metoden for en boksmetode. De mest kjente av disse er Kellers boksmetode i grensjikt-teori og Preismanns metode i kanalstrømning. (Wendroffs metode i avsnitt 6.7 i kompendiet er et spesialtilfelle av Preismanns metode). Vi skal nå bruke (6.7.3) på et system av to ligninger der ligningene allerede i utgangspunktet er på formen gitt i (6.7.1).

4.3.2 Example: Varmeveksler

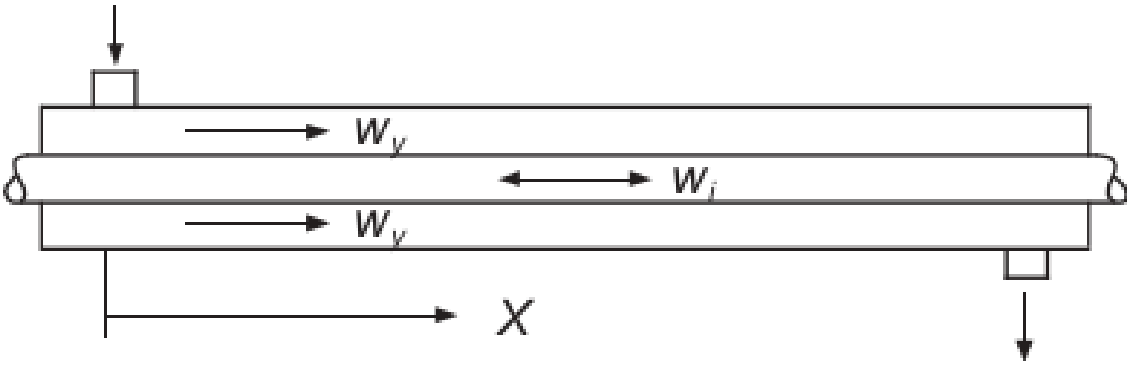


Figure 4.11:

Figure 4.11 viser en varmeveksler som er modellert som to rør. I det ytre røret foregår strømningen fra venstre mot høyre med konstant hastighet w_y og i det indre røret har vi en strømning med konstant hastighet w_i. Vi har likestrøm når w_i er rettet mot høyre mens vi har motstrøm når w_i er rettet mot venstre. Temperaturen i det ytre og det indre røret betegnes henholdsvis T_y og T_i. Vi regner det ytre røret så godt isolert at et eventuelt varmetap til omgivelsene kan neglisjeres.

Noen betegnelser: (Indeks i henviser til det indre røret)

Rørlengde:	L	$[m]$
Lengdekoordinat:	X	$[m]$
Rørdiameter:	d_i	$[m]$
Areal av rørtverrsnitt:	$A_i = \pi \cdot d_i^2/4$	$[m^2]$
Omkrets av indre rør:	$P = \pi \cdot d_i$	$[m]$
Tetthet av medium:	ρ_i	$[kg/m^3]$
Massestrøm:	$\dot{m}_i = \rho_i A_i w_i$	$[kg/s]$
Totalt varmeovergangstall:	$\bar{h}$	$[W/m^2/°C]$
Egenvarme:	c_i	$[J/kg/°C]$
Varmestrøm:	$\dot{Q}_i = \dot{m}_i c_i T_i$	$[W]$

(4.70)

For de indekserte størrelsene (4.70) har vi tilsvarende med indeks y for det ytre røret. $\dot{Q}_{yi}$ er varmestrømmen til det indre røret fra det ytre:

Med henvisning til Figure 4.12:

$$\dot{Q}_{yi} = \bar{h} P \cdot dX[T_y(X) - T_i(X)] \tag{4.71}$$

Varmebalansen for det indre røret:

$$\begin{aligned} &\dot{Q}_i = \dot{Q}_i + d\dot{Q}_i - \dot{Q}_{yi} \text{ som gir:} \\ &d\dot{Q}_i = \dot{Q}_{yi} = -d\dot{Q}_y \end{aligned} \tag{4.72}$$

hvor vi har benyttet oss av at vi ikke har noe varmetap til omgivelsene.

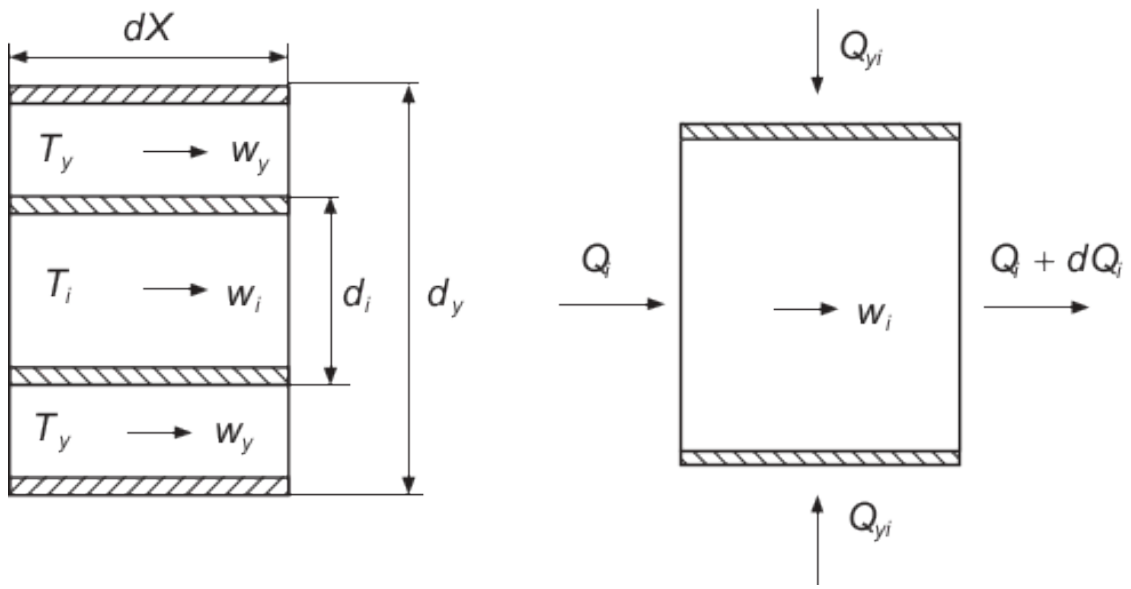


Figure 4.12:

(4.70) - (4.72) gir:

$$\dot{m}_i c_i dT_i(X) = \bar{h}P \cdot dX[T_y(X) - T_i(X)] = -\dot{m}_y c_y dT_y(X) \tag{4.73}$$

Fra (4.73) får vi følgende ligninger:

$$dT_y(X) = -\frac{\bar{h}P}{\dot{m}_y c_y}[T_y(X) - T_i(X)] \cdot dX \tag{4.74}$$

$$dT_i(X) = \frac{\bar{h}P}{\dot{m}_i c_i}[T_y(X) - T_i(X)] \cdot dX \tag{4.75}$$

Vi innfører nå dimensjonsløse størrelser. Lar T_{ir} og T_{yr} være referansetemperaturer som kan gis passende verdier i konkrete tilfeller. De dimensjons-løse temperaturene i ytre og indre rør betegnes nå henholdsvis u og v .

$$\text{Temperatur ytre rør: } u = \frac{T_y - T_{yr}}{T_{ir} - T_{yr}} \tag{4.76}$$

$$\text{Temperatur indre rør: } v = \frac{T_i - T_{yr}}{T_{ir} - T_{yr}} \tag{4.77}$$

$$\text{Lengde: } x = \frac{X}{L} \tag{4.78}$$

$$\text{Koeffisienter: } \alpha_y = \frac{\bar{h}PL}{\dot{m}_y c_y} = \frac{\bar{h}PL}{\rho_y A_y w_y c_y} \qquad \alpha_i = \frac{\bar{h}PL}{\dot{m}_i c_i} = \frac{\bar{h}PL}{\rho_i A_i w_i c_i} \tag{4.79}$$

(4.76) innsatt i (4.74) gir følgende system:

$$\text{Ytre rør: } \frac{du}{dx} = -\alpha_y(u - v) \tag{4.80}$$

$$\text{Indre rør: } \frac{dv}{dx} = \pm\alpha_i(u - v) \tag{4.81}$$

Vi har skrevet $\pm$ foran leddet på høyre side i (4.81) der pluss-tegnet gjelder for strømningsretningen som vist på Figure 4.12, mens negativt fortegn gjelder når w_i skifter retning (motstrøm). Det er underforstått at u og v er funksjoner av x slik at vi slipper å skrive $u(x)$ og $v(x)$.

Analytisk løsning

Dersom vi kjenner temperaturen for en gitt x-verdi både i det ytre og indre røret, kan vi løse (4.80) analytisk.

Ved subtraksjon:

$$-\frac{d}{dx}(u - v) = (\alpha_y \pm \alpha_i) \cdot (u - v)$$

Innfører temperaturdifferansen

$$\theta = u - v \tag{4.82}$$

slik at vi får ligningen $-\frac{d\theta}{dx} = (\alpha_y \pm \alpha_i) \cdot \theta$ med løsning:

$$\theta(x) = \theta_0 \cdot e^{-(\alpha_y \pm \alpha_i)\cdot x} \text{ , der } \theta_0 = \theta(0) \tag{4.83}$$

Får da følgende løsning for u og v:

$$u(x) = u_0 - \theta_0 \cdot \left[\frac{\alpha_y}{\alpha_y \pm \alpha_i}\right] \cdot \left[1 - e^{-(\alpha_y \pm \alpha_i)\cdot x}\right] \tag{4.84}$$

$$v(x) = v_0 + \theta_0 \cdot \left[\frac{\alpha_i}{\alpha_y \pm \alpha_i}\right] \cdot \left[1 - e^{-(\alpha_y \pm \alpha_i)\cdot x}\right] \tag{4.85}$$

Her er $u_0 = u(0)$ og $v_0 = v(0)$. Dersom f. eks. $u(0)$ og $v(1)$ er gitt, har vi et tilfelle med splitta randbetingelser som må løses numerisk.

Tilfelle med splitta randbetingelser.

For et tilfelle med motstrøm blir (4.80):

$$\frac{du}{dx} = -\alpha_y(u - v) \tag{4.86}$$

$$\frac{dv}{dx} = -\alpha_i(u - v) \tag{4.87}$$

Innløpstemperaturen for det ytre røret $= T_y(0)$ og utløpstemperaturen $= T_y(L)$. Tilsvarende betegnelser for det indre røret er $T_i(0)$ og $T_i(L)$. Velger nå referansetemperaturene i (4.76) for dette tilfellet:

$$T_{yr} = T_y(0) \text{ og } T_{ir} = T_i(L)$$

slik at (4.76) nå blir:

$$\text{Temperatur ytre rør: } u = \frac{T_y - T_y(0)}{T_i(L) - T_y(0)} \tag{4.88}$$

$$\text{Temperatur indre rør: } v = \frac{T_i - T_y(0)}{T_i(L) - T_y(0)} \tag{4.89}$$

Dette valget av referansetemperaturer gir følgende randbetingelser:

$$u(0) = 0 \ , \ v(1) = 1 \tag{4.90}$$

(4.86) innsatt i (4.66):

$$\frac{u_j - u_{j-1}}{h} = -\alpha_y \cdot \left[\frac{1}{2}(u_j + u_{j-1}) - \left(\frac{1}{2}(v_j + v_{j-1})\right)\right]$$
$$\frac{v_j - v_{j-1}}{h} = -\alpha_i \cdot \left[\frac{1}{2}(u_j + u_{j-1}) - \left(\frac{1}{2}(v_j + v_{j-1})\right)\right]$$

som ordnet gir følgende ligningsystem:

$$(h \cdot \alpha_y - 2) \cdot u_{j-1} + (2 + h \cdot \alpha_y) \cdot u_j - h \cdot \alpha_y \cdot v_{j-1} - h \cdot \alpha_y \cdot v_f = 0 \tag{4.91}$$

$$h \cdot \alpha_i \cdot u_{j-1} + h \cdot \alpha_i \cdot u_j - (h \cdot \alpha_i + 2) \cdot v_{j-1} + (2 - h \cdot \alpha_i) \cdot v_j = 0 \tag{4.92}$$

(4.91) er et spesialtilfelle av det mer generelle systemet

$$\begin{aligned} a_j^{(1)} u_{j-1} + \alpha_j^{(2)} v_{j-1} + b_j^{(1)} u_j + b_j^{(2)} v_j + c_j^{(1)} u_{j+1} + c_j^{(2)} v_{j+1} &= d_j^{(1)} \\ a_j^{(3)} u_{j-1} + a_j^{(4)} v_{j-1} + b_j^{(3)} u_j + b_j^{(4)} v_j + c_j^{(3)} u_{j+1} + c_j^{(4)} v_{j+1} &= d_j^{(2)} \end{aligned} \tag{4.93}$$

Systemet i (4.93) kalles et bi-tridiagonalt ligningsystem. Algoritmen for løsning av dette systemet kalles ofte Douglas-algoritmen. (Se appendiks I i kompendiet)

Nedenfor har vi skrevet ut (4.93) på matriseform.

$$\begin{bmatrix} \begin{bmatrix} c_1^{(1)} & c_1^{(2)} \\ c_1^{(3)} & c_1^{(4)} \end{bmatrix} & \begin{bmatrix} b_1^{(1)} & b_1^{(2)} \\ b_1^{(3)} & b_1^{(4)} \end{bmatrix} & & & & & & \\ \begin{bmatrix} a_2^{(1)} & a_2^{(2)} \\ a_2^{(3)} & a_2^{(4)} \end{bmatrix} & \begin{bmatrix} b_2^{(1)} & b_2^{(2)} \\ b_2^{(3)} & b_2^{(4)} \end{bmatrix} & \begin{bmatrix} c_2^{(1)} & c_2^{(2)} \\ c_2^{(3)} & c_2^{(4)} \end{bmatrix} & & & & & \\ & \ddots & \ddots & & & & & \\ & & \begin{bmatrix} a_j^{(1)} & a_j^{(2)} \\ a_j^{(3)} & a_j^{(4)} \end{bmatrix} & \begin{bmatrix} b_j^{(1)} & b_j^{(2)} \\ b_j^{(3)} & b_j^{(4)} \end{bmatrix} & \begin{bmatrix} c_j^{(1)} & c_j^{(2)} \\ c_j^{(3)} & c_j^{(4)} \end{bmatrix} & & & \\ & & & \ddots & \ddots & & & \\ & & & & \begin{bmatrix} a_N^{(1)} & a_N^{(2)} \\ a_N^{(3)} & a_N^{(4)} \end{bmatrix} & \begin{bmatrix} b_N^{(1)} & b_N^{(2)} \\ b_N^{(3)} & b_N^{(4)} \end{bmatrix} & \end{bmatrix} \cdot \begin{bmatrix} \begin{bmatrix} u_1 \\ u_1 \end{bmatrix} \\ \begin{bmatrix} u_2 \\ u_2 \end{bmatrix} \\ \vdots \\ \begin{bmatrix} u_j \\ u_j \end{bmatrix} \\ \vdots \\ \begin{bmatrix} u_N \\ u_N \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} d_1^{(1)} \\ d_1^{(2)} \end{bmatrix} \\ \begin{bmatrix} d_2^{(1)} \\ d_2^{(2)} \end{bmatrix} \\ \vdots \\ \begin{bmatrix} d_j^{(1)} \\ d_j^{(2)} \end{bmatrix} \\ \vdots \\ \begin{bmatrix} d_N^{(1)} \\ d_N^{(2)} \end{bmatrix} \end{bmatrix}$$

Vi ser at vi har fått en tridiagonal koeffisientmatrise der hvert element i matrisa er en 2 x 2 - matrise. En slik matrise kalles blokk-tridiagonal der hvert element er en blokk. Vi får ofte blokk-tridiagonale matriser når vi diskretiserer system av første ordens ordinære og partielle ligninger. Hadde vi f.eks. brukt 3 rør, ville hver blokk vært en 3 x 3 - matrise.

Den tridiagonale strukturen blir enda tydeligere når vi setter navn på blokkene:

$$\mathbf{a}_j = \begin{bmatrix} a_j^{(1)} & a_j^{(2)} \\ a_j^{(3)} & a_j^{(4)} \end{bmatrix} \ , \ \mathbf{b}_j = \begin{bmatrix} b_j^{(1)} & b_j^{(2)} \\ b_j^{(3)} & b_j^{(4)} \end{bmatrix} \ , \ \mathbf{c}_j = \begin{bmatrix} c_j^{(1)} & c_j^{(2)} \\ c_j^{(3)} & c_j^{(4)} \end{bmatrix}$$

$$\mathbf{x}_j = \begin{bmatrix} u_j \\ v_j \end{bmatrix} \ , \ \mathbf{d}_j = \begin{bmatrix} d_j^{(1)} \\ d_j^{(2)} \end{bmatrix} \ , j = 1, 2, ..., N$$

slik at systemet ovenfor kan skrives:

$$\begin{bmatrix} \mathbf{b}_1 & \mathbf{c}_1 & & & & & \\ \mathbf{a}_2 & \mathbf{b}_2 & \mathbf{c}_2 & & & & \\ & \cdot & \cdot & & & & \\ & \cdot & \cdot & & & & \\ & \cdot & \cdot & & & & \\ & & \mathbf{a}_j & \mathbf{b}_j & \mathbf{c}_j & & \\ & & & \cdot & \cdot & & \\ & & & \cdot & \cdot & & \\ & & & \cdot & \cdot & & \\ & & & & \mathbf{a}_N & \mathbf{b}_N \end{bmatrix} \cdot \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \cdot \\ \cdot \\ \cdot \\ \mathbf{x}_j \\ \cdot \\ \cdot \\ \cdot \\ \mathbf{x}_N \end{bmatrix} = \begin{bmatrix} \mathbf{d}_1 \\ \mathbf{d}_2 \\ \cdot \\ \cdot \\ \cdot \\ \mathbf{d}_j \\ \cdot \\ \cdot \\ \cdot \\ \mathbf{d}_N \end{bmatrix} \tag{4.94}$$

Å løse dette systemet med Douglas-algoritmen, er det samme som å bruke Thomas-algoritmen når man tar hensyn til at elementene er matriser og vektorer.

Indeksering.

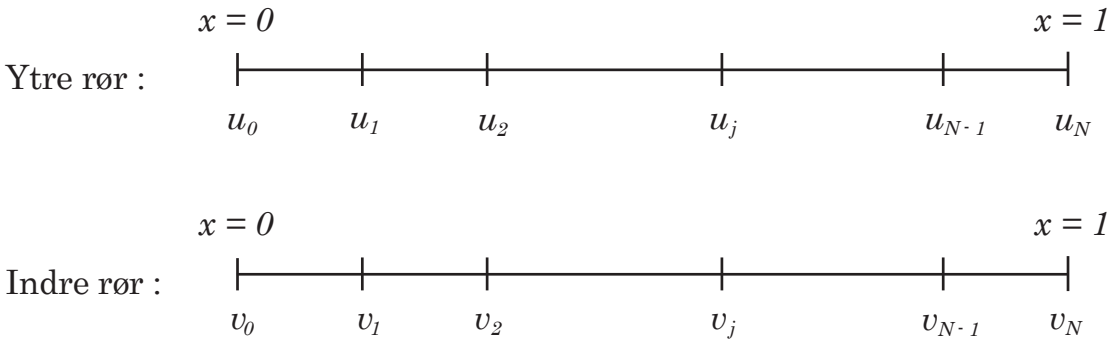


Figure 4.13:

Randbetingelser: $u_0 = 0$, $u_N = 1$.

Algoritmen krever at den første ukjente har indeks 1 og den siste indeks N. Som figuren ovenfor viser, stemmer dette for det ytre røret, men ikke for det indre der v_0 er den første ukjente.

Vi forandrer indeksene som vist nedenfor:

Får da følgende indeksering:

For u:

$$u_j \ , \ j = 0, 1, ..., N \ , \ u_0 = 0 \text{ fra randbetingelsen} \tag{4.95}$$

For v:

$$v_j \ , \ j = 1, 2, ..., N + 1 \ , \ v_{N+1} = 1 \text{ fra randbetingelsen} \tag{4.96}$$

Med den nye indekseringen i (4.96), blir systemet i (4.91) nå:

$$(h \cdot \alpha_y - 2) \cdot u_{j-1} + (2 + h \cdot \alpha_y) \cdot u_j - h \cdot \alpha_y \cdot v_j - h \cdot \alpha_y \cdot v_{j+1} = 0 \tag{4.97}$$

$$h \cdot \alpha_i \cdot u_{j-1} + h \cdot \alpha_i \cdot u_j - (h \cdot \alpha_i + 2) \cdot v_j + (2 - h \cdot \alpha_i) \cdot v_{j+1} = 0 \tag{4.98}$$

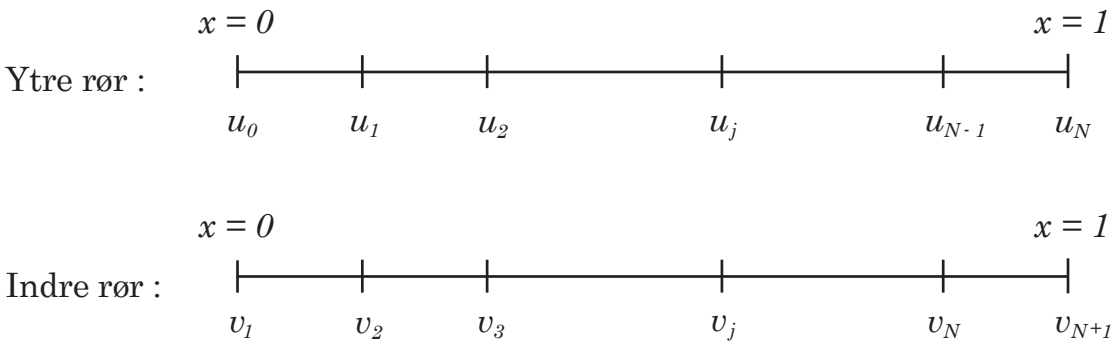


Figure 4.14:

Utskrevet for $j = 1$:

$$\begin{aligned}(2 + h \cdot \alpha_y) \cdot u_1 - h \cdot \alpha_y \cdot v_1 - h \cdot \alpha_y \cdot v_2 &= 0 \\ h \cdot \alpha_i \cdot u_1 - (h \cdot \alpha_i + 2) \cdot v_1 + (2 - h \cdot \alpha_i) \cdot v_2 &= 0\end{aligned}$$

hvor vi har benyttet $u_0 = 0$ fra randbetingelsen.
Utskrevet for $j = N$:

$$\begin{aligned}(h \cdot \alpha_y - 2) \cdot u_{N-1} + (2 + h \cdot \alpha_y) \cdot u_N - h \cdot \alpha_y \cdot v_N &= h \cdot \alpha_y \\ h \cdot \alpha_i \cdot u_{N-1} + h \cdot \alpha_i \cdot u_N - (h \cdot \alpha_i + 2) \cdot v_N &= h \cdot \alpha_i - 2\end{aligned}$$

hvor vi har brukt $v_{N+1} = 1$ fra randbetingelsen. Ved å sammenligne vårt system ovenfor med basissystemet i (4.93), får vi følgende koeffisienter for bruk i Douglas-algoritmen:

$$\begin{array}{llllllll} \alpha_j^{(1)} & = h \cdot \alpha_y - 2\ , & \alpha_j^{(2)} & = 0\ , & b_j^{(1)} & = 2 + h \cdot \alpha_y\ , & b_j^{(2)} & = -h \cdot \alpha_y \\ c_j^{(1)} & = 0, & c_j^{(2)} & = -h \cdot \alpha_y\ , & d_j^{(1)} & = 0\ , & j & = 1, 2, ..., N-1 \\ \alpha_j^{(3)} & = h \cdot \alpha_i\ , & \alpha_j^{(4)} & = 0\ , & b_j^{(3)} & = h \cdot \alpha_i\ , & b_j^{(4)} & = -(h \cdot \alpha_i + 2) \\ c_j^{(3)} & = 0\ , & c_j^{(4)} & = 2 - h \cdot \alpha_i\ , & d_j^{(2)} & = 0\ , & j & = 1, 2, ..., N-1 \end{array} \tag{4.99}$$

hvor $\alpha_1^{(1)}$ og $\alpha_1^{(2)}$ ikke brukes.

For $j = N$ blir utrykkene som i (4.99) bortsett fra $d_N^{(1)} = h \cdot \alpha_y$ og $d_N^{(2)} = h \cdot \alpha_i - 2$. $c_N^{(2)}$ og $c_N^{(4)}$ brukes ikke. Et eksempel med $N = 6$ er gitt vist nedenfor.

$$\begin{bmatrix} b_1^{(1)} & b_1^{(2)} & 0 & c_1^{(2)} & & & & & & & & \\ b_1^{(3)} & b_1^{(4)} & 0 & c_1^{(4)} & 0 & & & & & & & \\ a_2^{(1)} & 0 & b_2^{(1)} & b_2^{(2)} & 0 & c_2^{(2)} & & & & & & \\ a_2^{(3)} & 0 & b_2^{(3)} & b_2^{(4)} & 0 & c_2^{(4)} & 0 & & & & & \\ & 0 & a_3^{(1)} & 0 & b_3^{(1)} & b_3^{(2)} & 0 & c_3^{(2)} & & & & \\ & & a_3^{(3)} & 0 & b_3^{(3)} & b_3^{(4)} & 0 & c_3^{(4)} & 0 & & & \\ & & & 0 & a_4^{(1)} & 0 & b_4^{(1)} & b_4^{(2)} & 0 & c_4^{(2)} & & \\ & & & & a_4^{(3)} & 0 & b_4^{(3)} & b_4^{(4)} & 0 & c_4^{(4)} & 0 & \\ & & & & & 0 & a_5^{(1)} & 0 & b_5^{(1)} & b_5^{(2)} & 0 & c_5^{(2)} \\ & & & & & & a_5^{(3)} & 0 & b_5^{(3)} & b_5^{(4)} & 0 & c_5^{(4)} \\ & & & & & & & 0 & a_6^{(1)} & 0 & b_6^{(1)} & b_6^{(2)} \\ & & & & & & & 0 & a_6^{(3)} & 0 & b_6^{(3)} & b_6^{(4)} \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \\ u_4 \\ v_4 \\ u_5 \\ v_5 \\ u_6 \\ v_6 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ d_6^{(1)} \\ d_6^{(2)} \end{bmatrix} \tag{4.100}$$

Sammenlignet med det generellle systemet ser vi at fire av halv-diagonalene forsvinner i vårt tilfelle: $a_j^{(2)}$, $a_j^{(4)}$, $c_j^{(1)}$, og $c_j^{(3)}$. Dersom det er behov for å spare denne plassen, kan vi skrive en spesiell versjon som ikke bruker disse vektorene. (Se Fortran-funksjonen **bitris** i appendiks I i kompendietsom også finnes i Matlab-versjon)

4.3.3 Example: Talleksempel

Vi skal avkjøle smøreoljen fra et gassturbinanlegg. Oljen har en temperatur på 100° C. Til disposisjon har vi vann med temperatur på 30° C. Andre data:

Rørlengde:	L	$= 61m$
Rørdiameter:	d	$= 0.025m$
Omkrets av rør:	P	$= \pi \cdot d = 0.07854m$
Massestrøm:	$\dot{m}_y$	$= 0.1$ og $\dot{m}_i = 0.23kJ/s$
Totalt varmeovergangstal:	$\bar{h}$	$= 40W/m^2/°C$
Egenvarme:	c_y	$= 2130$ og $c_i = 4180J/Kg/°C$

Med disse data, får vi følgende verdier for koeffisientene α_y og α_i:

$$\alpha_y = \frac{\bar{h}PL}{\dot{m}_y c_y} \approx 0.9 \ , \ \alpha_i = \frac{\bar{h}PL}{\dot{m}_i c_i} \approx 0.2 \tag{4.101}$$

Referansetemperaturene i (4.71) og (4.82):

$$T_{ir} = T_i(L) = 30°C \ , \ T_{yr} = T_y(0) = 100°C \tag{4.102}$$

som gir forbindelsen mellom dimensjonsløs og dimensjonelle temperaturer:

$$T_y(x) = 100 - 70 \cdot u(x) \ , \ T_i(x) = 100 - 70 \cdot v(x) \tag{4.103}$$

Legg merke til at vi bruker den dimensjonsløse koordinaten x i (4.103).

Tabellen viser resultatene ved å løse (4.100) med koeffisienter gitt i (4.99) ved bruk av bitris gitt i appendiks I i kompendiet. Vi ser av tabellen at smøroljen blir avkjølt fra 100° C til 60.4° C, mens vannets temperatur stiger fra 30° C til 38.8° C. Temperaturforløpet i tabellen er vist i Figure 4.15.

Selv om vi ikke har noen direkte analytisk løsning å sammenligne med, indikerer den små forskjellen mellom løsningene for h = 0.1 og h = 0.01 at tabellverdiene sannsynligvis er temmelig nøyaktige. En delvis bekreftelse kan fås ved å bruke den analytiske løsningen i (4.84) som innsatt for tallverdiene for dette eksemplet blir:

$$\begin{aligned} u(x) &= v_0 \tfrac{9}{7}[1 - e^{-0.7 \cdot x}] \\ v(x) &= v_0[1 + \tfrac{2}{7}(1 - e^{-0.7 \cdot x})] \end{aligned} \tag{4.104}$$

$h = 0.1$	x	T_y(° C)	T_i(° C)	$h = 0.01$	x	$T_y(°)C$	T_i(° C)
	0.0	100.00000	38.80441		0.0	100.00000	38.80226
	0.1	94.67864	37.62189		0.1	94.68054	37.62016
	0.2	89.71719	36.51934		0.2	89.72070	36.31798
	0.3	85.09129	35.49137		0.3	85.09618	35.49031
	0.4	80.77825	34.53291		0.4	80.78431	34.53211
	0.5	76.75692	33.63928		0.5	76.76395	33.63870
	0.6	73.00756	32.80609		0.6	73.01539	32.80569
	0.7	69.51178	32.02925		0.7	69.52026	32.02899
	0.8	66.25243	31.30495		0.8	66.26142	31.30480
	0.9	63.21352	30.62964		0.9	63.22291	30.62958
	1.0	60.38014	30.00000		1.0	60.38981	30.00000

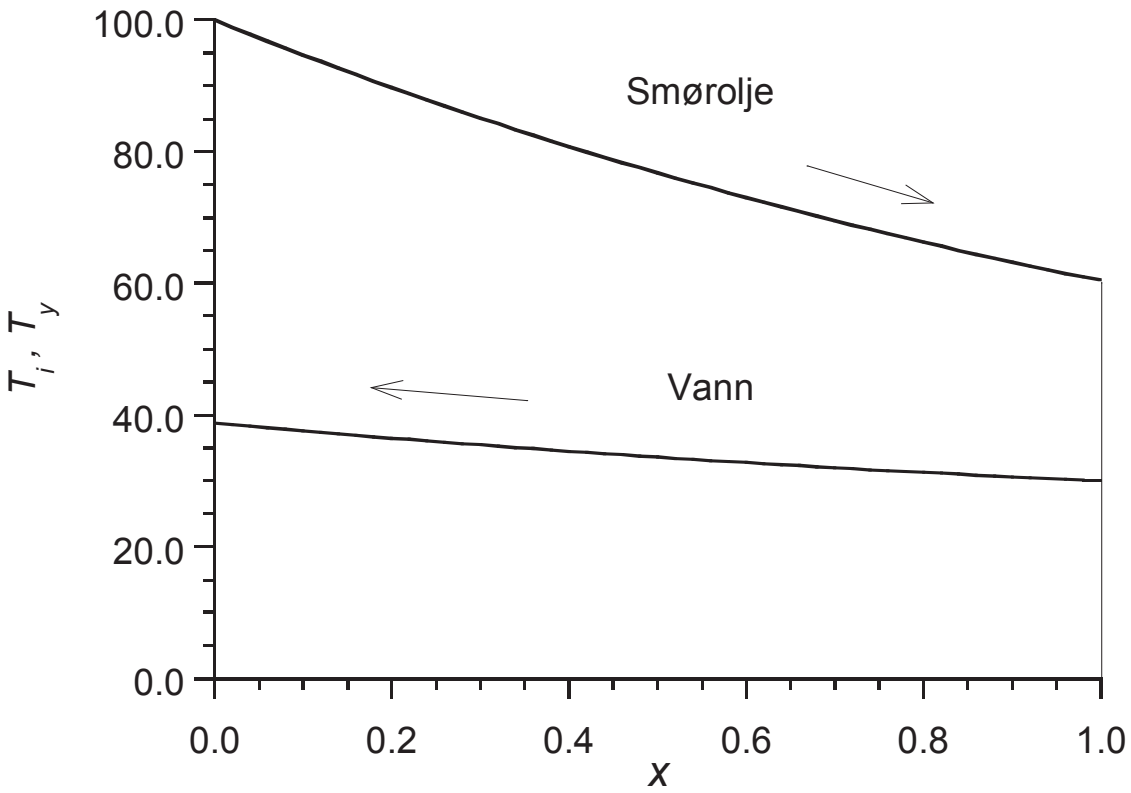


Figure 4.15:

Løsning av lign. (4.100)

Marie 10: figuren har norske forklaringer.

Verdien av $v_0 \equiv v(0)$ er ukjent og må beregnes numerisk. Ved å utføre beregningen for forskjellige verdier av h, får vi følgende tabell:

h	v_0
0.100	0.87422266
0.050	0.87424592
0.010	0.87425336
0.005	0.87425359
0.001	0.87425359

Dersom vi aksepterer v_0- verdien for $h = 0.001$ som den "rette", kan vi bruke denne i (4.104) og deretter sammenligne med tabellverdiene. Gjør vi det, finner vi at den relative feilen for $h = 0.1$ ligger i intervallet $[10^{-4},\ 10^{-5}]$ mens den for h = 0.01 ligger i intervallet $[10^{-6},\ 10^{-7}]$. Dette viser at differanseskjemaet er av $O(h^2)$, i overenstemmelse med teorien.

I kapittel 6 avsnitt 7 i kompendiet behandler vi det ikke-stasjonære tilfellet av dette eksemplet.

Bruk av Richardson ekstrapolering

Denne teknikken som er velkjent ved Romberg-integrasjon, kan ofte brukes til å forbedre verdier som er funnet ved bruk av differanse-metoder. (Se avsnitt 4.3 i C&K [4]). La oss anta at den korrekte verdien er y, mens vi har funnet en tilnærmelse y_1 med bruk av skritt lengden h_1. Dessuten antar vi at forbindelsen mellom y og y_1 er gitt ved:

$$y_1 = y + A \cdot h_1^k + B \cdot h_1^{k+1} + \ldots \tag{4.105}$$

der A og B er konstanter.

Dersom vi nå beregner to tilnærmelser y_1 og y_2 med to forskjellige verdier h_1 og h_2, får vi to ligninger av typen (4.105) og kan dermed eliminere A med følgende resultat:

$$\frac{h_2^k \cdot y_1 - h_1^k \cdot y_2}{h_2^k - h_1^k} = y + B \cdot \left(\frac{h_2^k h_1^{k+1} - h_1^k h_2^{k+1}}{h_2^k - h_1^k} \right) + \dots \tag{4.106}$$

(4.106) kan kalles h^k- ekstrapoleringsformelen. Spesielt for k = 2 og $h_2 = \frac{1}{2}h_1$ får vi fra (4.106):

$$y \approx y_2 + \frac{1}{3}(y_2 - y_1) \tag{4.107}$$

med en feil $\frac{1}{6}Bh_1^3 + \dots$

I noen tilfeller er vi så heldige at B = 0. Da blir feilen av $O(h_1^4)$. Dette er nettopp tilfelle for to-punkts metoden. For denne metoden får vi fra (4.69) med $h = h_1$:

$$\mathbf{y}(x_j) = \mathbf{y}_j(h) + h^2\mathbf{e}(x_j) + O(h^4)\ ,\ j = 0, 1..., N \tag{4.108}$$

En tilstrekkelig betingelse for at (4.108) gjelder er at (4.63) har kontinuerlig femte derivert på det aktuelle intervallet. (Se Keller [8]). Med vår notasjon blir (4.107):

$$\mathbf{y}(x_j) = \mathbf{y}_j\left(\frac{h}{2}\right) + \frac{1}{3}\left[\mathbf{y}_j\ \left(\frac{h}{2}\right) - \mathbf{y}_j(h)\right] + \mathbf{O}(h^4) \tag{4.109}$$

T_{ytre}	x	$h = 0.2$	$h = 0.1$	Ekstrapol.	$h = 2 \cdot 10^3$
	0.0	100.0000	100.00000	100.00000	100.00000
	0.1		94.67864		
	0.2	89.70651	89.71719	89.72075	89.72073
	0.3		85.09129		
	0.4	80.75984	80.77825	80.78439	80.78437
	0.5		76.75692		
	0.6	72.98376	73.00756	73.01549	73.01547
	0.7		69.51178		
	0.8	66.22511	66.25243	66.26154	66.26151
	0.9		63.21352		
	1.0	60.35076	60.38014	60.38993	60.38990

I tabellen ovenfor har vi beregnet temperaturen T_y i det ytre røret ; først med h=0.2 , så med h=0.1 og deretter ekstrapolert verdiene etter formelen i (4.109) I følge denne formelen skal vi da få verdier med nøyaktighet av $O(h^4)$ som i dette tilfellet blir $O(0.2^4) = O(10^{-5}) \approx 2 \cdot 10^{-3}$ som eksempel. I den siste kolonnen har vi utført beregningen med $h = 2 \cdot 10^{-3}$ med verdier som viser at teorien stemmer.

4.4 Linearinsering av ikke-linære algebraiske ligninger

Som eksempel, bruker vi problemet i avsnitt (4.2.2):

$$\begin{aligned} y''(x) &= \tfrac{3}{2}y^2 \\ y(0) = 4\ ,\ & y(1) = 1 \end{aligned} \tag{4.110}$$

der en av løsningene er gitt ved:

$$y = \frac{4}{(1+x)^2} \tag{4.111}$$

Vi diskretiserer(4.110) med bruk av sentraldifferanser for $y''(x)$:

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} = \frac{3}{2}y_i^2$$

eller ordnet:

$$y_{i-1} - \left(2 + \frac{3}{2}h^2 y_i\right) y_i + y_{i+1} = 0 \tag{4.112}$$

med h = Δx

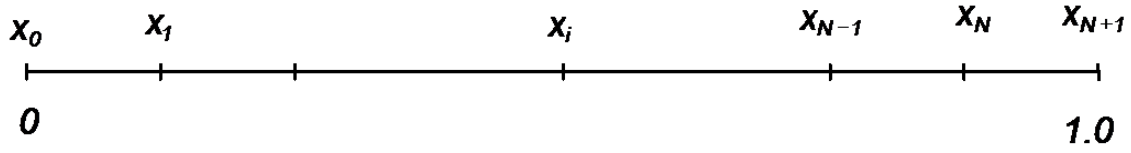


Figure 4.16:

Vi har delt intervallet $[0, 1]$ i $N + 1$ deler der $h = 1/(N + 1)$ og $x_i = h \cdot i$, $i = 0, 1, ..., N + 1$. Da $y_0 = 4$ og $y_{N+1} = 1$, blir (4.112):

$$\begin{aligned} -\left(2 + \tfrac{3}{2}h^2 y_1\right) y_1 + y_2 &= -4 \\ &\dots \\ y_{i-1} - \left(2 + \tfrac{3}{2}h^2 y_i\right) y_i + y_{i+1} &= 0 \\ &\dots \\ y_{N-1} - \left(2 + \tfrac{3}{2}h^2 y_N\right) y_N &= -1 \\ \text{der } i = 2, 3, ...N - 1 & \end{aligned} \tag{4.113}$$

Koeffisientmatrisa for (4.113) er tridiagonal, men systemet er ikke-lineært. (Et system av 2. grads ligninger). Vi har ikke formler som kan løse slike system, og settet må derfor lineariseres. La oss skal se nærmere på to metoder for å utføre denne lineariseringen.

4.4.1 Metoden med etterslep

Da ligningsystemet er ikke-lineært, må vi utføre en iterasjonsprosess. La y_i^{m+1} og y_i^m være løsningen av den diskretiserte ligningen (4.112) eller (4.113) for henholdsvis iterasjon $m + 1$ og m . Metoden med etterslep går ut på å erstatte avhengige variable ved iterasjon $m + 1$ med tilsvarende variable fra iterasjon m helt til vi bare har lineære ledd. I (4.112) og (4.113) er det y^2-leddet som forårsaker ikke-linearitetet og som derfor må lineariseres. Skriver ut (4.113):

$$\begin{aligned} -\left(2 + \tfrac{3}{2}y_1^{m+1}h^2\right) y_1^{m+1} + y_2^{m+1} &= -4 \\ y_{i-1}^{m+1} - \left(2 + \tfrac{3}{2}y_i^{m+1}h^2\right) y_i^{m+1} + y_{i+1}^{m+1} &= 0 \\ {}^{m+1}_{N-1} - \left(2 + \tfrac{3}{2}y_N^{m+1}h^2\right) y_N^{m+1} &= -1 \end{aligned} \tag{4.114}$$

der $i = 2, 3, ...N - 1$, $m = 0, 1, 2, ...$

Lineariseringen består nå i å erstatte $\frac{3}{2}h^2 y_1^{m+1}$, $\frac{3}{2}h^2 y_i^{m+1}$ og $\frac{3}{2}h^2 y_N^{m+1}$ med $\frac{3}{2}h^2 y_1^m$, $\frac{3}{2}h^2 y_i^m$ og $\frac{3}{2}h^2 y_N^m$ slik at vi får følgende lineære system:

$$\begin{aligned} -\left(2 + \tfrac{3}{2}y_1^m h^2\right) y_1^m + y_2^m &= -4 \\ y_{i-1}^m - \left(2 + \tfrac{3}{2}y_i^m h^2\right) y_i^m + y_{i+1}^m &= 0 \\ {}^m_{N-1} - \left(2 + \tfrac{3}{2}y_N^m h^2\right) y_N^m &= -1 \end{aligned} \tag{4.115}$$

der $i = 2, 3, ...N - 1$, $m = 0, 1, 2, ...$

Vi har nå fått et lineært, tridiagonalt system som kan løses med Thomas-algoritmen. For å starte iterasjonsprosessen, må vi tippe verdier for y_i^0 , $i = 1, 2, ...N$. Dersom vi ikke har spesielle opplysninger, kan

vi f.eks. tippe $y_i^0 = 0$, $i = 1, 2, ...N$. Gode startverdier vil generelt gi raskere konvergens. Vi må også huske å teste for diagonal-dominans når vi bruker **tdma**. For (4.115) blir kravet:

$$\left| 2 + \frac{3}{2} y_i^m h^2 \right| \geq 2 \ , \ \ i = 1, 2, ..., N$$

som er oppfylt dersom alle $y_i^m > 0$. Av Figure 3.7 ser vi at dette kravet er oppfylt for den ene løsningen, men ikke for den andre. Når denne betingelsen ikke er oppfylt, kan det være fornuftig å bruke en løsningsrutine med pivotering, f.eks. **tripiv** istedenfor **tdma**. Når vi ikke vet hvoran iterasjonsprosessen vil foreløpe, er det lurt å bare bruke antall iterasjoner som et stoppkrav. Etter vi har fått innsikt i iterasjonsforløpet, kan vi legge til andre stoppkriterier. Endel eksempler er vist i avsnitt (4.4.6)

Matlabprogrammet **delay34** gir følgende resultat med $h = 0.05$:

Itr.	max. avvik
1	1.000e+000
2	5.525e-001
3	1.104e-001
4	2.632e-002
5	5.892e-003
6	1.328e-003
7	2.980e-004
8	6.690e-005
9	1.502e-005
10	3.370e-006
11	7.562e-007
12	1.697e-007

Max. avvik betyr her det maksimale relative avviket. Løsningen av differanseligningen konvergerer langsomt mot løsningen $y_I = \dfrac{4}{(1+x)^2}$ som vist på (3.7) i avsnitt (3.2) . Den maksimale relative feilen i løsningen er $\approx 5.6 \cdot 10^{-4}$) som ikke kan minskes uten å minske skrittlengden h.

Fordelen med denne metoden er at lineariseringsprosessen er enkel. Bakdelen er at den konvergerer ofte langsomt, samt at vi gjerne må ha gode startverdier for å få konvergens. Når vi vet hvordan iterasjonen forløper, kan vi som nevnt ovenfor, legge inn et stoppkriterium basert på et iterasjonsavvik. Iterasjonsløkka kan nå f.eks. se ut som vist nedenfor:

```
it = 0; itmax = 10 ; dymax = 1.0; RelTol = 1.0e-5;
d = 0 ; % h\T1\o yre side
d(1) =  - 4.0; d(n) = - 1.0;
while (dymax > RelTol) & (it < itmax)
        it = it + 1;     b = -(2.0 + fac*ym); % hoveddiagonal
        ym1 = tdma(a,b,c,d); % L\T1\o ser ligningsystemet
        dymax = max(abs((ym1-ym)./ym1));% Beregner relativ avvik
        ym = ym1; % Oppdatering    fprintf(' %10d     %12.3e \n',it,dymax);
end
```

Legg merke til at for å starte iterasjonsløkka, må *dymax* være større enn *RelTol*

4.4.2 Newton-linearisering

Før vi setter opp den formelle utviklingen, viser vi en variant som er enkel å bruke når ikke-linearitetene er rene produkt.

$$\text{Setter } y_i^{m+1} = y_i^m + \delta y_i \tag{4.116}$$

der δy_i er avviket (residuet) mellom y_i -verdiene for de to iterasjonene. Ved bruk av (4.116) :

$$\begin{aligned}\left(y_i^{m+1}\right)^2 &= (y_i^m + \delta y_i)^2 = (y_i^m)^2 + 2y_i^m \cdot \delta y_i + (\delta y_i^m)^2 \\ &\approx (y_i^m)^2 + 2y_i^m \cdot \delta y_i = y_i^m(2y_i^{m+1} - y_i^m)\end{aligned} \tag{4.117}$$

Lineariseringen består i å neglisjere $(\delta y)^2$ som liten i forhold til de andre leddene.
(4.117) innsatt i (4.113) gir følgende system:

$$\begin{aligned}-(2+3y_1^m h^2)y_1^{m+1} + y_2^{m+1} &= -\tfrac{3}{2}(y_i^m h)^2 - 4 \\ y_{i-1}^{m+1} - (2+3y_i^m h^2)y_i^{m+1} + y_{i+1}^{m+1} &= -\tfrac{3}{2}(y_i^m h)^2 \\ y_{N-1}^{m+1} - (2+3y_N^m h^2)y_N^{m+1} &= -\tfrac{3}{2}(y_N^m h)^2 - 1\end{aligned} \tag{4.118}$$

der $i = 2, 3, ..., N-1$, $N-1, m = m = 0, 1, 2, ...$
Vi har igjen fått et lineært, tridiagonalt system som kan løses med Thomas-algoritmen. Velger samme startverdier som i foregående versjon. Matlabprogrammet **taylor34** gir følgende resultat med $h = 0.05$:

Itr.	max. avvik
1	1.000e+000
2	3.765e-001
3	2.479e-002
4	9.174e-005
5	1.175e-009
6	2.405e-015

Vi ser at iterasjonsprosessen nå går mye raskere enn ved etterslep-metoden. Riktignok langsomt i starten, men fra tredje iterasjon har vi rask konvergens. Linearisering i (4.116) og (4.117) er egentlig en Taylorutvikling der vi rekkeutvikler rundt iterasjon m og bare beholder de to første leddene. Kaller det ikke-lineære leddet som skal lineariseres for F og antar at F er funksjon av den avhengige variable z i punktet i ved iterasjon $m+1$: Vi skal linearisere leddet $F(z_i)\Big|^{m+1} \equiv F(z_i)_{m+1}$. Bruker den siste notasjonen i fortsettelsen.

Rekkeutvikling:

$$\begin{gathered}F(z_i)_{m+1} \approx F(z_i)_m + \delta z_i \left(\frac{\partial F}{dz_i}\right)_m \\ \text{der } \delta z_i = z_i^{m+1} - z_i^m\end{gathered} \tag{4.119}$$

I tilfellet ovenfor:

$$\delta z_i \to \delta y_i \ , \ z_i^{m+1} \to y_i^{m+1} \ , \ z_i^m \to y_i^m \ , \ F(y_i)_{m+1} = (y_i^{m+1})^2 \ , \ F(y_i)_m = (y_i^m)^2$$

som innsatt i (4.119) gir: $(y_i^{m+1})^2 \approx (y_i^m)^2 + 2y_i^m \cdot \delta y_i$ som er identisk med (4.117) . Fremgangsmåten i (4.119) kalles ofte *Newton-linearisering*.

I mange tilfeller inngår flere indekser i det ikke-lineære leddet. Anta f.eks at vi har et ledd der både z_i og z_{i+1} inngår. Vi bruker da Taylorutvikling for to variable:

$$\begin{gathered}F(z_i, z_{i+1})_{m+1} \approx F(z_i, z_{i+1})_m + \delta z_i \left(\frac{\partial F}{\partial z_i}\right)_m + \delta z_{i+1} \left(\frac{\partial F}{\partial z_{i+1}}\right)_m \\ \text{der } \delta z_i = z_i^{m+1} - z_i^m \ , \ \delta z_{i+1} = z_{i+1}^{m+1} - z_{i+1}^m\end{gathered} \tag{4.120}$$

Dersom både z_{i-1}, z_i og z_{i+1} inngår, får vi:

$$F(z_{i-1}, z_i, z_{i+1})_{m+1} \approx F(z_{i-1}, z_i, z_{i+1})_m$$

$$+\delta z_{i-1}\left(\frac{\partial F}{\partial z_{i-1}}\right)_m + \delta z_i\left(\frac{\partial F}{\partial z_i}\right)_m + \delta z_{i+1}\left(\frac{\partial F}{\partial z_{i+1}}\right)_m \tag{4.121}$$

der $\delta z_{i-1} = z_{i-1}^{m+1} - z_{i-1}^m$, $\delta z_i = z_i^{m+1} - z_i^m$, $\delta z_{i+1} = z_{i+1}^{m+1} - z_{i+1}^m$

Tilsvarende for flere indekser.

4.4.3 Example:

Gitt differensialligningen $y''(x) + y(x)\sqrt{y(x)} = 0$ som diskretisert med sentral-differanser blir:

$$y_{i+1}^{m+1} - 2y_i^{m+1} + y_{i-1}^{m+1} + h^2 y_i^{m+1}\sqrt{y_i^{m+1}} = 0 \text{ ved iterasjon } m+1.$$

Det er leddet $y_i^{m+1} \cdot \sqrt{y_i^{m+1}}$ som er ikke-lineært og må lineariseres. I dette tilfellet har vi bare et indeks og vi bruker (4.119) med $z_i \to y_i$ og $F(y_i) = y_i^{\frac{3}{2}}$:

$$F(y_i)_{m+1} \approx F(y_i)_m + \delta y_i\left(\frac{\partial F}{\partial y_i}\right)_m = (y_i^m)^{\frac{3}{2}} + \delta y_i \cdot \tfrac{3}{2} \cdot (y_i^m)^{\frac{1}{2}} \text{ eller}$$

$$y_i^{m+1}\sqrt{y_i^{m+1}} \approx y_i^m\sqrt{y_i^m} + \tfrac{3}{2} \cdot \sqrt{y_i^m} \cdot \delta y_i \text{ der } \delta y_i = y_i^{m+1} - y_i^m$$

Merk at uttrykket nå er lineært da y_i^{m+1} bare inngår i 1. potens.
Innsatt i (4.4.3) får vi følgende differanseligning:

$$y_{i-1}^{m+1} + \left(\frac{3}{2}h^2\sqrt{y_i^m} - 2\right)y_i^{m+1} + y_{i+1}^{m+1} = \frac{h^2}{2}y_i^m\sqrt{y_i^m} \tag{4.122}$$

4.4.4 Example:

$y''(x) + \sin(y(x)) = 0$ som diskretisert med sentral-differanser blir:

$$y_{i+1}^{m+1} - 2y_i^{m+1} + y_{i-1}^{m+1} + h^2\sin\left(y_i^{m+1}\right) = 0 \tag{4.123}$$

Det er leddet y_i^{m+1} som er ikke-lineært og må lineariseres. Vi har bare ett indeks og vi bruker (4.119) med $z_i \to y_i$ og $F(y_i) = \sin(y_i)$:

$$F(y_i)_{m+1} \approx F(y_i)_m + \delta y_i\left(\frac{\partial F}{\partial y_i}\right)_m = \sin(y_i^m) + \delta y_i \cdot \cos(y_i^m) \text{ eller}$$

$$\sin(y_i^{m+1}) \approx \sin(y_i^m) + \cos(y_i^m) \cdot \delta y_i \text{ der } \delta y_i = y_i^{m+1} - y_i^m$$

Innsatt i (4.123) får vi følgende differanseligning:

$$y_{i-1}^{m+1} + (h^2\cos(y_i^m) - 2)y_i^{m+1} + y_{i+1}^{m+1} = h^2(y_i^m\cos(y_i^m) - \sin(y_i^m)) \tag{4.124}$$

4.4.5 Example:

Gitt differensialligningen $y''(x) + y(x)\sqrt{y'(x)} = 0$ som diskretisert med sentral-differanser blir:

$$y_{i+1}^{m+1} - 2y_i^{m+1} + y_{i-1}^{m+1} + \alpha \cdot y_i^{m+1}\sqrt{y_{i+1}^{m+1} - y_{i-1}^{m+1}} = 0 \ , \ \text{der } \alpha = h\sqrt{h/2} \tag{4.125}$$

Det er leddet $y_i^{m+1}\sqrt{y_{i+1}^{m+1} - y_{i-1}^{m+1}}$ som er ikke-lineært og må lineariseres. I dette tilfellet har vi tre indekser og vi bruker (4.121) med $z_{i-1} \to y_{i-1}$, $z_i \to y_i$ og $z_{i+1} \to y_{i+1}$ samt $F(y_{i-1}, y_i, y_{i+1})_m = y_i\sqrt{y_{i+1} - y_{i-1}}$. Finner de enkelte leddene i (4.121) :

$$F(y_{i-1}, y_i, y_{i+1})_m = y_i^m\sqrt{y_{i+1}^m - y_{i-1}^m} \ , \ \left(\frac{\partial F}{\partial y_{i-1}}\right)_m = -\frac{y_i^m}{2\sqrt{y_{i+1}^m - y_{i-1}^m}} \ ,$$

$$\left(\frac{\partial F}{\partial y_i}\right)_m = \sqrt{y_{i+1}^m - y_{i-1}^m} \text{ og tilslutt } \left(\frac{\partial F}{\partial y_{i+1}}\right)_m = \frac{y_i^m}{2\sqrt{y_{i+1}^m - y_{i-1}^m}}$$

Totalt:

$$y_i^{m+1}\sqrt{y_{i+1}^{m+1} - y_{i-1}^{m+1}} \approx y_i^m\sqrt{y_{i+1}^m - y_{i-1}^m} - \frac{y_i^m}{2\sqrt{y_{i+1}^m - y_{i-1}^m}}\delta y_{i-1}$$

$$+\sqrt{y_{i+1}^m - y_{i-1}^m} \cdot \delta y_i + \frac{y_i^m}{2\sqrt{y_{i+1}^m - y_{i-1}^m}}\delta y_{i+1}$$

Merk at uttrykket nå er lineært da y_{i-1}^{m+1}, y_i^{m+1} og y_{i+1}^{m+1} bare inngår i 1. potens. Vi setter dette inn i (4.125) som nå blir lineær:

$$\left(1 - \alpha\frac{y_i^m}{2g^m}\right) \cdot y_{i-1}^{m+1} - (2 - \alpha g^m) \cdot y_i^{m+1} + \left(1 + \alpha\frac{y_i^m}{2g^m}\right) \cdot y_{i+1}^{m+1}$$

$$= \alpha\frac{y_i^m}{2g^m}(y_{i+1}^m - y_{i-1}^m) \tag{4.126}$$

$$\text{der } g^m = \sqrt{y_{i+1}^m - y_{i-1}^m}$$

I neste avsnitt viser vi hvordan vi kan operere direkte med delta-størrelsene $\delta y_{i-1}^{m+1}, \delta y_i^{m+1}$ og δy_{i+1}^{m+1}

Hvordan vil disse eksemplene bli dersom vi bruker metoden med etterslep? Lign. (4.4.3)) i eksempel (4.4.3) skrives: $y_{i-1}^{m+1} + (h^2\sqrt{y_i^{m+1}} - 2) \cdot y_i^{m+1} + y_{i-1}^{m+1} = 0$

Koeffisienten foran y_i^{m+1} -leddet inneholder $\sqrt{y_i^{m+1}}$ som blir erstattet med $\sqrt{y_i^m}$ slik at ligningen blir: $y_{i-1}^{m+1} + (h^2\sqrt{y_i^m} - 2) \cdot y_i^{m+1} + y_{i-1}^{m+1} = 0$

I lign. ligning (4.123)) i eksempel (4.4.4) erstattes med slik at ligningen blir:
$y_{i-1}^{m+1} - 2y_i^{m+1} + y_{i+1}^{m+1} = -h^2\sin(y_i^m) = 0$

Lign. (4.125)) i eksempel (4.4.5) skrives:
$y_{i-1}^{m+1} + (\alpha \cdot \sqrt{y_{i+1}^{m+1} - y_{i-1}^{m+1}} - 2) \cdot y_i^{m+1} + y_{i+1}^{m+1} = 0$

Koeffisienten foran y_i^{m+1} -leddet inneholder $\sqrt{y_{i+1}^{m+1} - y_{i-1}^{m+1}}$ som erstattes med $\sqrt{y_{i+1}^m - y_{i-1}^m}$ slik at ligningen blir:

$$y_{i-1}^{m+1} + (\alpha \cdot \sqrt{y_{i+1}^m - y_{i-1}^m} - 2) \cdot y_i^{m+1} + y_{i+1}^{m+1} = 0$$

Når vi bruker denne metoden, må vi først skrive systemet på en form som viser koeffisientene i ligningsystemet. Dersom koeffisientene inneholder avhengige variable ved iterasjon $m+1$, må disse erstattes med verdier fra iterasjon m. Men dette er ofte det samme som å Taylor-utvikle koeffisientene rundt iterasjon m og bruke bare 1. ledd i utviklingen.

Vi kan derfor vente at metoden med etterslep generelt konvergerer langsommere enn når vi bruker to ledd av Taylor-utviklingen. Metoden med etterslep brukes helst for partielle differanseligninger der f. eks. materialparametrene er funksjon av de avhengige variable, f.eks. temperaturavhengige varmeledningstall.

4.4.6 Eksempler på stoppkriterier

Vi setter:

$$\delta y_i = y_i^{m+1} - y_i^m \ , \ i = 1, 2, ..., N \ , \ m = 0, 1, ... \\ \text{der N er antall beregningspunkt} \tag{4.127}$$

1. Test for absolutte størrelser.

$$t_{a1} = \max(|\delta y_i|) < tol_a \tag{4.128}$$

$$t_{a2} = \frac{1}{n} \sum_{i=1}^{N} |\delta y_i| < tol_a \tag{4.129}$$

$$t_{a3} = \frac{1}{n} \sqrt{\sum_{i=1}^{N} (\delta y_i)^2} < tol_a \tag{4.130}$$

2. Test for relative størrelser.

$$t_{r1} = \max \left(\left| \frac{\delta y_i}{y_i^{m+1}} \right| \right) < tol_r \ , \ y_i^{m+1} \neq 0 \tag{4.131}$$

$$t_{r2} = \frac{\sum_{i=1}^{N} |\delta y_i|}{\sum_{i=1}^{N} |y_i^{m+1}|} < tol_r \tag{4.132}$$

$$t_{r3} = \frac{\max(|\delta y_i|)}{\max(|y_i^{m+1}|)} < tol_r \tag{4.133}$$

Dersom de størrelsene vi beregner er av størrelsesorden 1, er det likegyldig om vi bruker en absolutt eller en relativ test. Vanligvis bruker vi en relativ test da denne samsvarer med uttrykket "antall korrekte siffer". Dersom den størrelsen vi beregner er liten i hele sitt beregningsområde, bruker vi vanligvis en absolutt test.

La oss bruke testene ovenfor på problemet i avsnitt (3.2) der vi skal beregne løsningen y_{II}) gitt på fig (3.7)

Vi gjentar ligningen fra (4.110) :

$$y''(x) = \frac{3}{2} y^2 \tag{4.134}$$

$$y(0) = 4 \ , \ y(1) = 1 \tag{4.135}$$

Vi tipper et startprofil gitt ved parabelen $y_s = 20(x - x^2)$, $0 < x < 1$. Denne parabelen, samt løsningen y_{II} , er plottet i figuren nedenfor.

I dette tilfellet er det naturlig å bruke et relativt stoppkriterium da y_{II} ligger i intervallet $[4, -10.68]$. Med ni iterasjoner får vi følgende 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	$2.26 \cdot 10^{-11}$	$2.55 \cdot 10^{-11}$

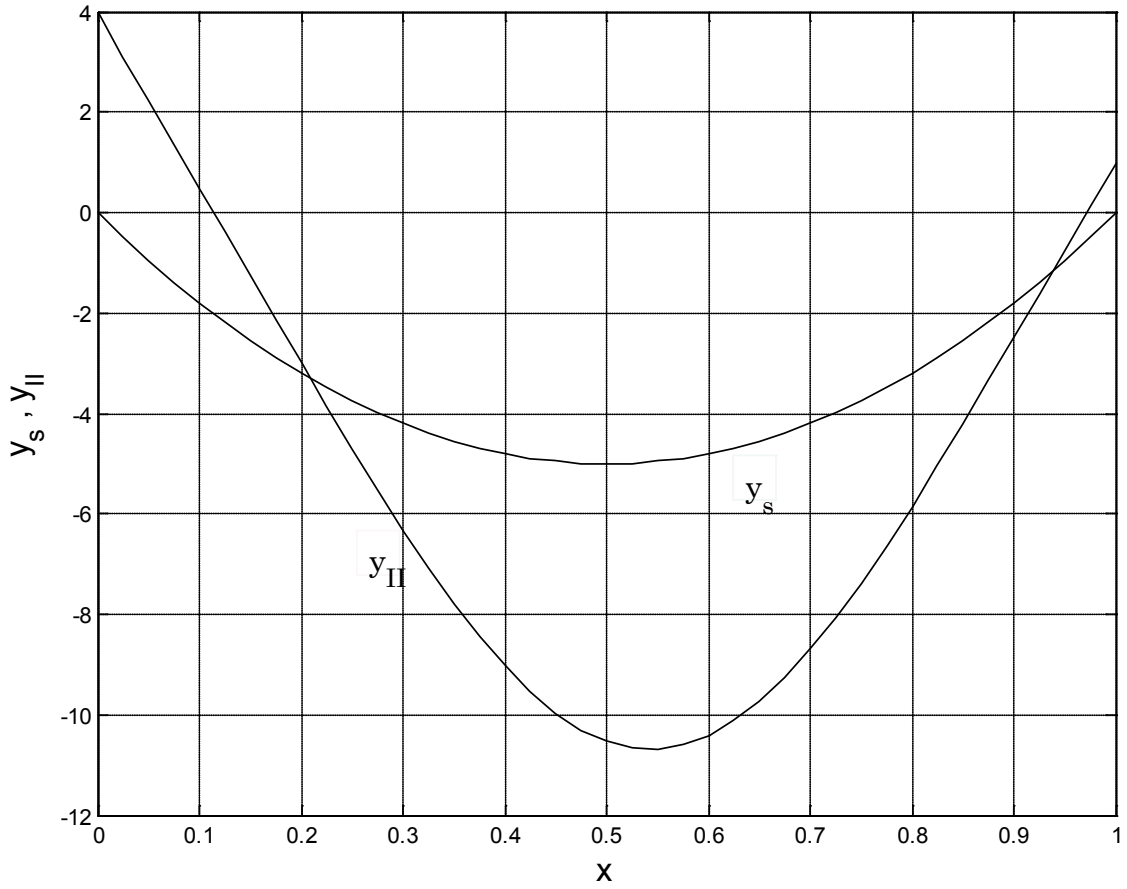


Figure 4.17:

Iterasjonsforløpet er typisk for Newton-iterasjon da startverdiene våre ligger et godt stykke fra de korrekte verdiene. For de seks første iterasjonene minker feilen langsomt, mens vi har rask konvergens for de siste tre iterasjonene. Vi ser også at det er liten forskjell mellom de to kriteriene i dette tilfellet.

Nedenfor vises listing av programmet **avvikr**.

```
clear
h = 0.025; % skrittlengde
ni = 1/h; % Antall intervall
#  h maa velges slik at ni er et heltall
n = ni-1; % Antall ligninger
fac = 3.0*h*h;
a = ones(n,1) ; % underdiagonal
c = a; % overdiagonal
#  a og c blir ikke \T1\o delagt under eliminasjons-prosessen og
#  kan derfor legges utenfor iterasjonsl\T1\o kka.
x = (h:h:1.0-h)';
ym = -20*x.*(1 - x); % Startverdier
b = zeros(n,1); d = b; dy = b; % allokering
fprintf('        Itr.       \n');
for it = 1:9
   b = -(2.0 + fac*ym); % hoveddiagonal
   d = -(fac*0.5)*ym.^2 ; % h\T1\o yre side
   d(n) = d(n)- 1.0;
   d(1) = d(1) - 4.0;
   ym1 = tdma(a,b,c,d); % L\T1\o ser ligningsystemet
   dy = abs(ym1 - ym);
```

```
    tr3 = max(dy)/max(abs(ym1));
    tr2 = sum(dy)/sum(abs(ym1));
    ym = ym1; % Oppdatering av y-verdier
    fprintf(' %10d   %9.2e   %9.2e  \n',it,tr2, tr3);
end
```

Programmet **avvika** som beregner de absolutte kriteriene, er temmelig identisk med **avvikr** bortsett av vi erstatter beregningen av t_{r2} og t_{r3} med

```
    ta1 = max(dy);
    ta2 = sum(dy)/n;
    ta3 = sqrt(dot(dy,dy))/n;
```

4.4.7 Ligninger på delta-form

Fra lign. (4.117)

$$(y_i^{m+1})^2 = (y_i^m + \delta y_i)^2 = (y_i^m)^2 + 2y_i^m \cdot \delta y_i + (\delta y_i)^2$$
$$\approx (y_i^m)^2 + 2y_i^m \cdot \delta y_i = y_i^m(2y_i^{m+1} - y_i^m)$$

Vi har her innført $\delta y_i = y_i^{m+1} - y_i^m$ slik at ligningsystemet løses m.h.p y_i^{m+1}. Dette er ikke nødvendig. Vi kan beholde δy_i som ukjent og løse systemet m.h.p. delta-størrelsene. Ved å innføre $y_k^{m+1} = y_k^m + \delta y_k$, $k = i-1\ ,\ i\ ,\ i+1$ samt $(y_i^{m+1})^2 \approx (y_i^m)^2 + 2y_i^m \cdot \delta y_i$ i (4.114) eventuelt (4.118), får vi følgende ligningsystem

$$\begin{aligned} -(2+3y_1^m h^2)\delta y_1 + \delta y_2 &= -(y_2^m - 2y_1^m + 4) + \tfrac{3}{2}(y_1^m h)^2 \\ \delta y_{i-1} - (2+3y_i^m h^2)\delta y_i + \delta y_{i+1} &= -(y_{i+1}^m - 2y_i^m + y_{i-1}^m) + \tfrac{3}{2}(y_i^m h)^2 \\ \delta y_{N-1} - (2+3y_N^m h^2)\delta y_N &= -(1 - 2y_N^m + y_{N-1}^m) + \tfrac{3}{2}(y_N^m h)^2 \end{aligned} \tag{4.136}$$

der $i = 2, 3, ..., N-1$, $m = 0, 1, 2, ...$

Vi har benyttet at $y_0 = 4$, $\delta y_0 = 0$, $y_{N+1} = 1$, $\delta y_{N+1} = 0$ fra randbetingelsene. For hver iterasjon oppdateres y-verdiene ved:

$$y_i^{m+1} = y_i^m + \delta y_i\ ,\ i = 1, 2, ..., N \tag{4.137}$$

Når vi har funnet ut hvordan iterasjonen forløper, kan vi f. eks. legge inn stopp-kriterier som $\max|\delta y_i| < \varepsilon_1$ eller $\max|\delta y_i / y_i^{m+1}| < \varepsilon_2$, $i = 1, 2, ..., N$ Iterasjonsløkka i programmet **delta34** ser slik ut med bruk av (4.133):

```
it = 0; itmax = 10; dymax = 1.0; RelTol = 1.0e-5;
while (dymax > RelTol) & (it < itmax)
    it = it + 1;    b = -(2.0 + fac*y); % hoveddiagonal
    d = (fac*0.5)*y.^2 ; % h\T1\o yre side
   for j = 2:n-1
      d(j) = d(j) -(y(j+1)-2*y(j) + y(j-1));
   end
    d(n) = d(n)- (1.0- 2*y(n) + y(n-1));
    d(1) = d(1) - (y(2)-2*y(1) + 4.0);
    dy = tdma(a,b,c,d); % L\T1\o ser ligningsystemet
    y = y + dy; % Oppdatering av y-verdier
    dymax = max(abs((dy)./y));% Beregner relativ avvik
    fprintf(' %10d     %12.3e \n',it,dymax);
end
```

4.4.8 Kvasilinearisering

I det foregående har vi først diskretisert ligningen og deretter linearisert den. Det er fullt mulig å linearisere ligningen først. Dette blir gjerne kalt kvasi-linearisering. La oss se på en generell, ikke-lineær 2. ordens ligning:

$$y''(x) = f(x, y, y') \tag{4.138}$$

Skriver (4.138) på formen:

$$g(x, y, y', y'') \equiv y''(x) - f(x, y, y') = 0 \tag{4.139}$$

Setter

$$\delta y = y_{m+1} - y_m \ , \ \delta y' = y'_{m+1} - y'_m \ , \ \delta y'' = y''_{m+1} - y''_m \tag{4.140}$$

der m og $m+1$ som vanlig betyr iterasjonsnummer. (I storparten av dette avsnittet skriver vi iterasjonsnumrene som subindekser)

Ved rekkeutvikling av (4.139) rundt iterasjon m :

$$\begin{aligned} g(x, y_{m+1}, y'_{m+1}, y''_{m+1}) \quad &\approx g(x, y_m, y'_m, y''_m) \\ &+ \quad \left(\frac{\partial g}{\partial y}\right)_m \delta y + \left(\frac{\partial g}{\partial y'}\right)_m \delta y' + \left(\frac{\partial g}{\partial y''}\right)_m \delta y'' \end{aligned} \tag{4.141}$$

Anta at vi har iterert så mange ganger at

$$g(x, y_{m+1}, y'_{m+1}, y''_{m+1}) \approx g(x, y_m, y'_m, y''_m) \approx 0$$

som innsatt i (4.141) gir:

$$\left(\frac{\partial g}{\partial y}\right)_m \delta y + \left(\frac{\partial g}{\partial y'}\right)_m \delta y' + \left(\frac{\partial g}{\partial y''}\right)_m \delta y'' = 0 \tag{4.142}$$

Ved derivasjon av (4.139):

$$\frac{\partial g}{\partial y} = -\frac{\partial f}{\partial y} \ , \ \frac{\partial g}{\partial y'} = -\frac{\partial f}{\partial y'} \ , \ \frac{\partial g}{\partial y''} = 1 \tag{4.143}$$

som innsatt i (4.142) gir:

$$\delta y'' = \left(\frac{\partial f}{\partial y}\right)_m \delta y + \left(\frac{\partial f}{\partial y'}\right)_m \delta y' \tag{4.144}$$

Ved å sette inn fra (4.140) i (4.144) samt bruk av (4.138), får vi:

$$\begin{aligned} &y''_{m+1} - \left(\frac{\partial f}{\partial y'}\right)_m \cdot y'_{m+1} - \left(\frac{\partial f}{\partial y}\right)_m \cdot y_{m+1} \\ &= f(x, y_m, y'_m) - \left(\frac{\partial f}{\partial y}\right)_m \cdot y_m - \left(\frac{\partial f}{\partial y'}\right)_m \cdot y'_m \end{aligned} \tag{4.145}$$

Skriver tilslutt (4.145) med vår vanlige notasjon med iterasjonsnummer oppe:

$$\begin{aligned} &(y'')^{m+1} - \left(\frac{\partial f}{\partial y'}\right)_m \cdot (y')^{m+1} - \left(\frac{\partial f}{\partial y}\right)_m \cdot y^{m+1} \\ &= f(x, y^m, (y')^m) - \left(\frac{\partial f}{\partial y'}\right)_m \cdot (y')^m - \left(\frac{\partial f}{\partial y}\right)_m \cdot y^m \end{aligned} \tag{4.146}$$

Vi har brukt en 2. ordens ligning som eksempel, men (4.144) – (4.146) lar seg umiddelbart generalisere til en n'te ordens ligning, f.eks. en 3. ordens:

$$(y''')^{m+1} - \left(\frac{\partial f}{\partial y''}\right) \cdot (y'')^{m+1} - \left(\frac{\partial f}{\partial y'}\right)_m \cdot (y')^{m+1} - \left(\frac{\partial f}{\partial y}\right)_m \cdot y^{m+1}$$

$$= f(x, y^m, (y')^m, (y'')^m) - \left(\frac{\partial f}{\partial y''}\right)_m \cdot (y'')^m \tag{4.147}$$

$$- \left(\frac{\partial f}{\partial y'}\right)_m \cdot (y')^m - \left(\frac{\partial f}{\partial y}\right)_m \cdot (y)^m$$

4.4.9 Example:

1)

$$\text{Ligningen } y''(x) = \frac{3}{2}y^2$$

Her blir $\frac{\partial f}{\partial y'} = 0$, $\frac{\partial f}{\partial y} = 3y$ som innsatt i (4.146) gir:

$$(y'')^{m+1} - 3y^m \cdot y^{m+1} = -\frac{3}{2}(y^m)^2$$

Ved å diskretisere med sentraldifferanser $y_i'' \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}$: får vi følgende differanseligning:

$$y_{i-1}^{m+1} - (2 + 3h^2 y_i^m) y_i^{m+1} + y_{i+1}^{m+1} = -\frac{3}{2}(h y_i^m)^2$$

som er i overenstemmelse med (4.118).

2)

$$\text{Falkner-Skan-ligningen } y'' + y \cdot y'' + \beta \cdot [1 - (y')^2] = 0$$

Vi skriver ligningen på formen:

$$y'' = -\left(y \cdot y'' + \beta \cdot [1 - (y')^2]\right) = f(x, y, y', y'')$$

Lign. (4.147) gir med $\frac{\partial f}{\partial y''} = -y$, $\frac{\partial f}{\partial y'} = 2\beta y'$ og $\frac{\partial f}{\partial y} = -y''$:

$$(y''')^{m+1} + y^m \cdot (y'')^{m+1} - 2\beta (y')^m \cdot (y')^{m+1} + (y'')^m \cdot y^{m+1}$$
$$= -\beta \left(1 + [(y')^m]^2\right) + y^m \cdot (y'')^m$$

Bruk sentraldifferanser og verifiser lign.(F.O.16) i appendiks F i kompendiet
!split

4.5 Løsning av Blasius ligning ved bruk av differansemetode

Test av kapittel 3 avsnitt 5

4.6 Deriverte randbetingelser

Test av kapittel 3 ansnitt 6

4.7 Iterasjonsmetoder ved løsning av ODL

Test av kapittel 3 seksjon7

Chapter 5

Elliptic partial differential equations

5.1 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:

$$\nabla^2 u = 0 \tag{5.1}$$

- Poisson:

$$\nabla^2 u = q \tag{5.2}$$

- Helmholtz:

$$\nabla^2 u + c \cdot u = q \tag{5.3}$$

$$\text{where } \nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \text{ in 3D}$$

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:

$$\frac{\partial}{\partial x}\left(a\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(b\frac{\partial u}{\partial y}\right) + c \cdot u = q \tag{5.4}$$

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:

$$\frac{\partial T}{\partial t} = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} \right) \tag{5.5}$$

which is a parabolic PDE, but at steady state when $\left(\frac{\partial T}{\partial t} = 0 \right)$, we get:

$$\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} \right) = 0 \tag{5.6}$$

which is an elliptic PDE.

From the classification approach in Chapter **??** (see also Classification of linear 2nd order PDEs) we see that (5.1) to (5.3) are elliptic, meaning no real characteristics. Therefore, elliptic PDEs must be treated as boundary value problems (see **??** and **??**) for a discussion about the conditions for a well posed problem).

For the solution of a generic elliptic PDE like (5.4), in a domain in two-dimensions bounded by the boundary curve C (5.1), we reiterate the most common boundary value conditions from **??**:

- 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

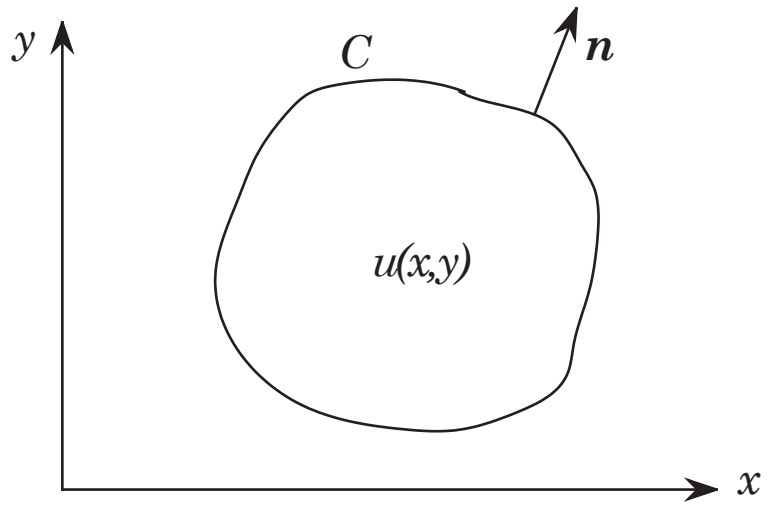


Figure 5.1: A solution domain in two-dimensions bounded by a boundary curve C.

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 (5.2)

Possion problems.

- The stress equation for torsion of an elastic rod
- The displacement of an membrane under constant pressure.
- Stokes flow (low Reynolds number flow)

- Numerous analytical solutions (also in polar coordinantes) may be found in [21] section 3-3.

5.2 Direct numerical solution

In situations where the elliptic PDE correspons to the stationary solution of a parabolic problem (5.5), 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 5.2:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{5.7}$$

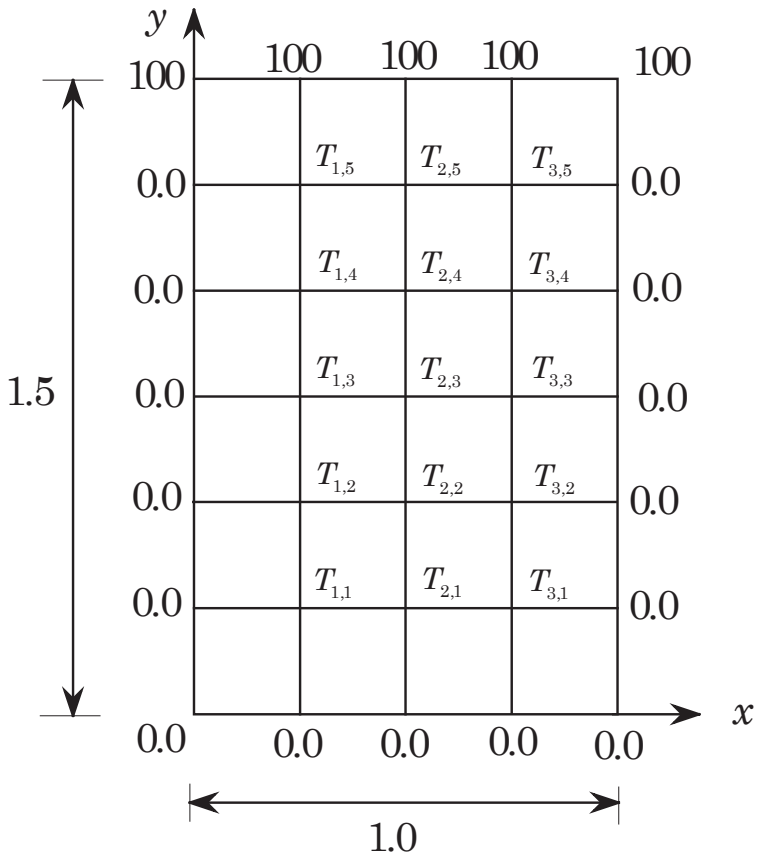


Figure 5.2: Rectangular domain with prescribed values at the boundaries (Dirichlet).

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 (5.3).

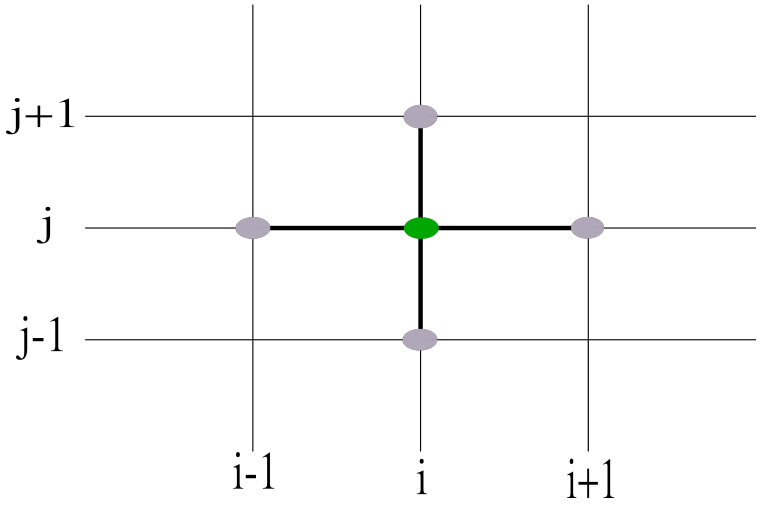


Figure 5.3: Illustration of the numerical stencil.

By approximation the second order differentials in (5.7) by central differences we get the following numerical stencil:

$$T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1} - 4T_{i,j} = 0 \tag{5.8}$$

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 (5.8) reads:

$$T_e + T_w + T_n + T_s - 4T_m = 0 \tag{5.9}$$

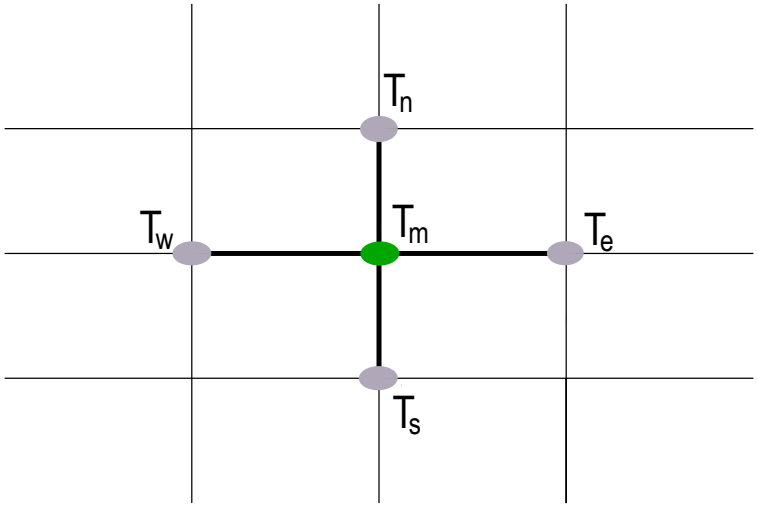


Figure 5.4: Illustration of the numerical stencil with compass notation.

The smoothing nature of elliptic problems may be seen even more clearly by isolating the $T_{i,j}$ in (5.9) on the left hand side:

$$T_m = \frac{T_e + T_w + T_n + T_s}{4} \tag{5.10}$$

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:

$$\begin{aligned} &T = 0.0 \quad \text{at} \quad y = 0 \\ &T = 0.0 \quad \text{at} \quad x = 0 \quad \text{and} \quad x = 1 \quad \text{for} \quad 0 \leq y < 1.5 \\ &T = 100.0 \quad \text{at} \quad y = 1.5 \end{aligned} \tag{5.11}$$

Our mission is now to find the temperature distribution over the plate by using (5.8) and (5.11) with $\Delta x = \Delta y = 0.25$. In each discretized point in (5.3) the temperatures need to satisfy (5.8), 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 (5.8) and (5.11) in each. All unknown temperatures close to any of the boundaries (left, right, top, bottom) in Figure 5.2 will be influenced by the prescribed and known temperatures the wall via the 5-point stencil (5.8). 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 5.2, 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 (5.8). 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:

$$\begin{aligned} -4 \cdot T_{11} + T_{12} + T_{21} &= 0 \\ T_{11} - 4 \cdot T_{12} + T_{13} + T_{22} &= 0 \\ T_{12} - 4 \cdot T_{13} + T_{14} + T_{23} &= 0 \\ T_{13} - 4 \cdot T_{14} + T_{15} + T_{24} &= 0 \\ T_{14} - 4 \cdot T_{15} + T_{25} &= -100 \\ T_{11} - 4 \cdot T_{21} + T_{22} + T_{31} &= 0 \\ T_{12} + T_{21} - 4 \cdot T_{22} + T_{23} + T_{32} &= 0 \\ T_{13} + T_{22} - 4 \cdot T_{23} + T_{24} + T_{33} &= 0 \\ T_{14} + T_{23} - 4 \cdot T_{24} + T_{25} + T_{34} &= 0 \\ T_{15} + T_{24} - 4 \cdot T_{25} + T_{35} &= 100 \\ T_{21} - 4 \cdot T_{31} + T_{32} &= 0, \\ T_{22} + T_{31} - 4 \cdot T_{32} + T_{33} &= 0 \\ T_{23} + T_{32} - 4 \cdot T_{33} + T_{34} &= 0 \\ T_{24} + T_{33} - 4 \cdot T_{34} + T_{35} &= 0 \\ T_{25} + T_{34} - 4 \cdot T_{35} &= -100 \end{aligned} \tag{5.12}$$

The equations in (5.12) represent a linear, algebraic system of equations with $5 \times 3 = 15$ unknowns, which has the more convenient and condensed symbolic representation:

$$\mathbf{A} \cdot \mathbf{T} = \mathbf{b} \tag{5.13}$$

where **A** denotes the coefficient matrix, **T** holds the unknown tempeartures, and **b** the prescribed boundary temperatures. Notice that the structure of the coefficient matrix **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 **A** and the temperature vector **T** are given by:

$$\begin{pmatrix} -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 \end{pmatrix} \cdot \begin{pmatrix} T_{11} \\ T_{12} \\ T_{13} \\ T_{14} \\ T_{15} \\ T_{21} \\ T_{22} \\ T_{23} \\ T_{24} \\ T_{25} \\ T_{31} \\ T_{32} \\ T_{33} \\ T_{34} \\ T_{35} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -100 \\ 0 \\ 0 \\ 0 \\ 0 \\ -100 \\ 0 \\ 0 \\ 0 \\ 0 \\ -100 \end{pmatrix} \tag{5.14}$$

The analytical solution of (5.7) and (5.11) may be found to be:

$$T(x,y) = 100 \cdot \sum_{n=1}^{\infty} A_n \sinh(\lambda_n y) \cdot \sin(\lambda_n x) \tag{5.15}$$

$$\text{where } \lambda_n = \pi \cdot n \text{ and } A_n = \frac{4}{\lambda_n \sinh(\frac{3}{2}\lambda_n)}$$

The analytical solution of the temperature field $T(x,y)$ in (5.15) may be proven to be symmetric around $x = 0.5$ (see 1).

We immediately realize that it may be inefficient to solve (5.14) as it is, due to the presence of all the zero-elements. The coefficient matrix **A** has $15 \times 15 = 225$ elementes, out of which on 59 are non-zero. Further, a symmetric, band structure of **A** is evident from (5.14). Clearly, these properties may be exploited to construct an efficent scheme which does not need to store all non-zero elements of **A**.

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 (see SciPy sparse example).

```
# chapter7/src-ch7/laplace_Diriclhet1.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

# 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 (5.7) 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:

```
# chapter7/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

#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=30
Tbottom=0.0
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()
```

Vi får følgende resultat der tallene i parantes er beregnet fra den analytiske løsningen i (5.15):

$$
\begin{aligned}
&T_{11} = T_{31} = 1.578\ (1.406),\ T_{12} = T_{32} = 4.092\ (3.725)\\
&T_{13} = T_{33} = 9.057\ (8.483),\ T_{14} = T_{34} = 19.620\ (18.945)\\
&T_{15} = T_{35} = 43.193\ (43.483),\ T_{21} = 2.222\ (1.987),\ T_{22} = 5.731\ (5.261)\\
&T_{23} = 12.518\ (11.924),\ T_{24} = 26.228\ (26.049),\ T_{25} = 53.154\ (54.449)
\end{aligned}
$$

The structure of the coefficient matrix will not necessarily be so regular 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.

5.2.1 von Neumann boundary conditions

The problem illustrated in Figure 5.15 has prescribed temperatures on the boundaries (i.e. Dirichlet boundaray conditions). In this section we will consider a problem with identical governing equation (5.7) as for the problem in 5.15. The only difference being that we consider von Neuman conditions, i.e. zero derivatives, at $x = 0$ og $y = 0$, as illustrated in figure 5.5.

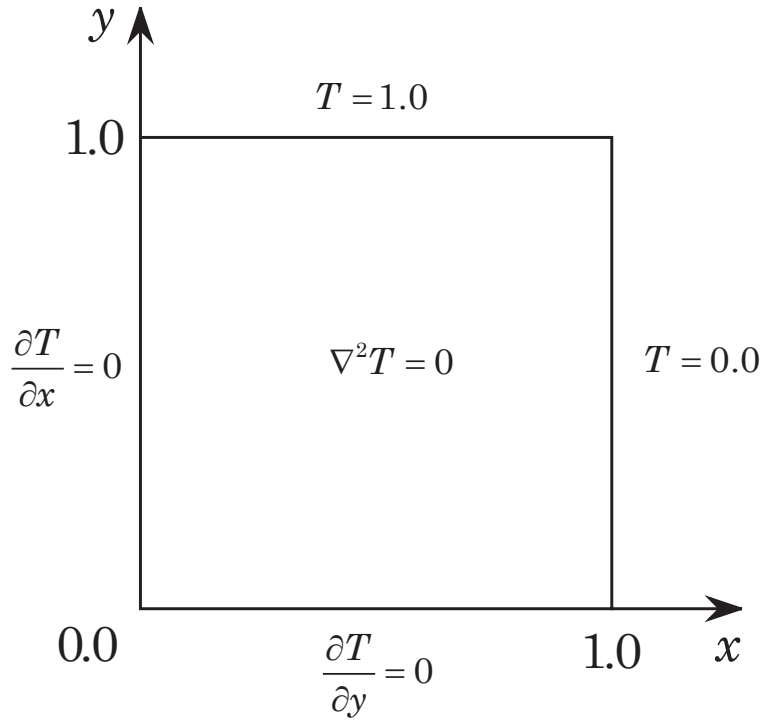


Figure 5.5: Laplace-equation for a quadratic domain with von Neumann boundary conditions.

Mathematically the problem in Figure 5.5, is specified with the governing equation (5.7) which we reiterate for convenience:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{5.16}$$

with corresponding boundary conditions (von Neuman)

$$\frac{\partial T}{\partial x} = 0 \quad \text{for } x = 0, \text{ and } 0 < y < 1 \tag{5.17}$$

$$\frac{\partial T}{\partial y} = 0 \quad \text{for } y = 0, \text{ and } 0 < x < 1 \tag{5.18}$$

and prescribed values 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 (5.8). However, at the boundaries we need to take into account the von Neumann conditions. We will take a closer look at $\dfrac{\partial T}{\partial x} = 0$ for $x = 0$. The other von Neuman boundary 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.

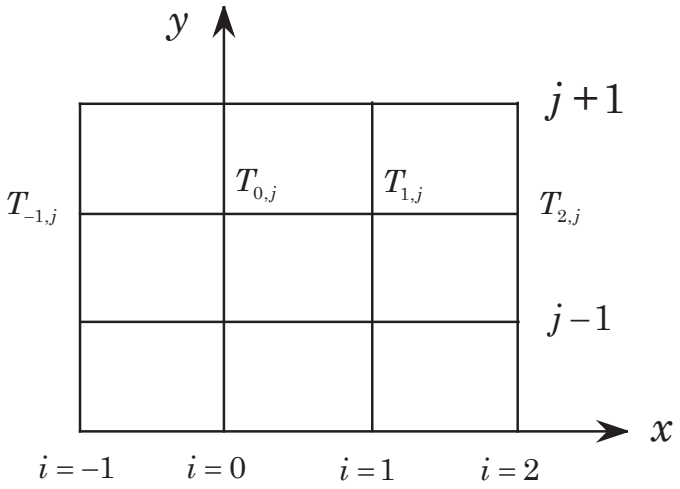


Figure 5.6: Illustration of how ghostcells with negative indices may be used to implement von Neumann boundary conditions.

A central difference approximation (see Figure 5.6) of $\dfrac{\partial T}{\partial x} = 0$ at $i = 0$ yields:

$$\frac{T_{1,j} - T_{-1,j}}{2\Delta x} = 0 \rightarrow T_{-1,j} = T_{1,j} \tag{5.19}$$

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 (6.9)) for a generic discrete point (i, j):

$$\left.\frac{\partial T}{\partial x}\right|_{i,j} = \frac{-3T_{i,j} + 4T_{i+1,j} - T_{i+2,j}}{2\Delta x}$$

which for $\dfrac{\partial T}{\partial x} = 0$ for $x = 0$ reduce to:

$$T_{0,j} = \frac{4T_{1,j} - T_{2,j}}{3} \tag{5.20}$$

For the problem at hand, illustrated in Figure 5.5, central difference approximation (5.19) and forward difference approximation (5.20) yields fairly similar results, but (5.20) results in a shorter code when the methods in 5.3 are employed.

To solve the problem in Figure 5.5 we employ the numerical stencil (5.9) for the unknows in the field (not influenced by the boundaries)

$$T_w + T_e + T_s + T_n - 4T_m = 0 \tag{5.21}$$

where we used the same ordering as given in Figurw 5.7. For the boundary conditions we have chosen to implement the by means of (5.19) which is illustrated in Figure 5.7 by the dashed lines.

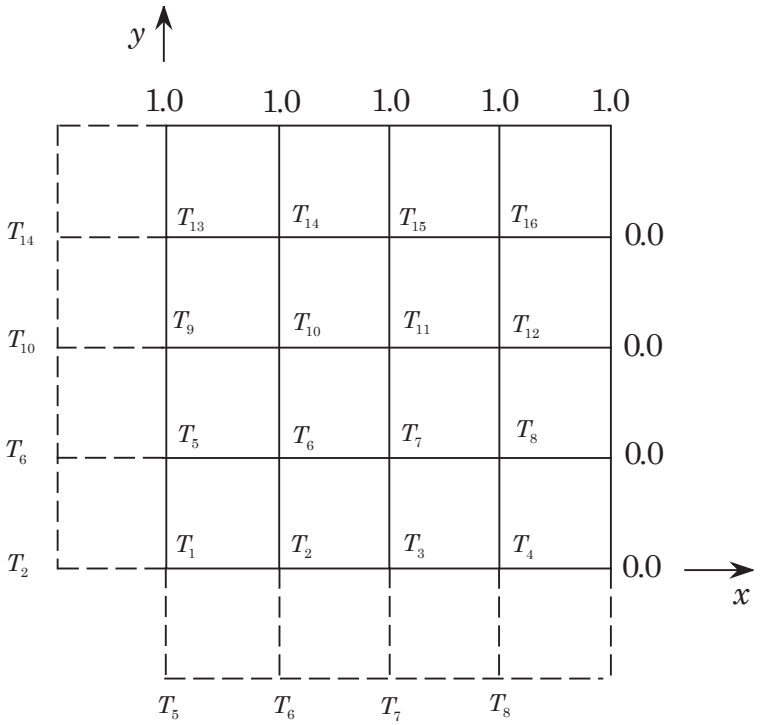


Figure 5.7: Von Neumann boundary conditions with ghost cells.

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 (5.21) along with the boundary conditions in (5.18) yield:

$$
\begin{aligned}
2T_2 + 2T_5 - 4T_1 &= 0 \\
T_1 + 2T_6 + T_3 - 4T_2 &= 0 \\
T_2 + 2T_7 + T_4 - 4T_3 &= 0 \\
T_3 + 2T_8 - 4T_4 &= 0
\end{aligned}
$$

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 5.5) 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:

$$\begin{pmatrix}
-4 & 2 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -4 & 1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -4 & 1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -4 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & -4 & 2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 2 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -4 & 2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4
\end{pmatrix}
\cdot
\begin{pmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \\ T_6 \\ T_7 \\ T_8 \\ T_9 \\ T_{10} \\ T_{11} \\ T_{12} \\ T_{13} \\ T_{14} \\ T_{15} \\ T_{16} \end{pmatrix}
=
\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -1 \\ -1 \\ -1 \\ -1 \end{pmatrix}
\tag{5.22}$$

```
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 central diffe
        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 element
    d1_upper[nx-1::nx] = 0
    d1_upper[::nx] = 2 # every nx element on first upper diagonal is two; stating from the first element.
                # 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], format='csc')

    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:

$$T(x,y) = \sum_{n=1}^{\infty} A_n \cosh(\lambda_n y) \cdot \cos(\lambda_n x), \tag{5.23}$$

$$\text{der } \lambda_n = (2n-1) \cdot \frac{\pi}{2}, \; A_n = 2\frac{(-1)^{n-1}}{\lambda_n \cosh(\lambda_n)}, \; n = 1, 2, \ldots \tag{5.24}$$

The solution for the problem illustrated Figure 5.5 is computed and visualized the the python code above. The solution is illustrated in Figure 5.11.

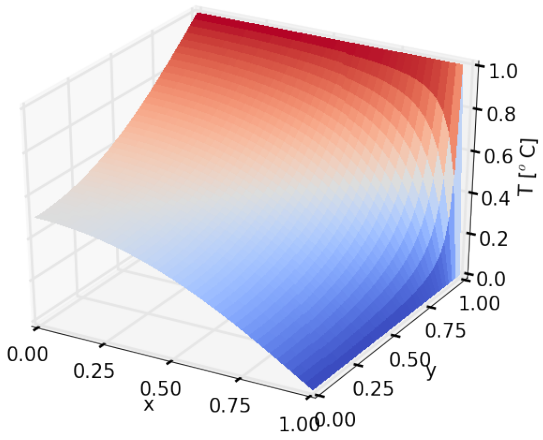


Figure 5.8: Solution of the Laplace equation with von Neuman boundary conditions.

5.3 Iteration methods for linear algebraic equation systems

5.3.1 EKSEMPLER

In **??** we introduced the classical iteration methods, Jacobi, Gauss-Seidel og SOR, for the solution of ordinary differential equations (ODEs).

Vi skal her se nærmere på noen av disse metodene, hovedsakelig brukt på lineære, elliptiske differensialligninger.

Konvergenskriterier er behandlet i 5.3.2.

Anta at vi har et system av lineære, algebraiske ligninger $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ som utskrevet for et sett av tre ligninger blir:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1 \tag{5.25}$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2 \tag{5.26}$$
$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \tag{5.27}$$

Skriver (5.27) på følgende måte:

$$x_1 = x_1 + \frac{1}{a_{11}}[b_1 - (a_{11}x_1 + a_{12}x_2 + a_{13}x_3)] \tag{5.28}$$
$$x_2 = x_2 + \frac{1}{a_{22}}[b_2 - (a_{21}x_1 + a_{22}x_2 + a_{23}x_3)] \tag{5.29}$$
$$x_3 = x_3 + \frac{1}{a_{33}}[b_3 - (a_{31}x_1 + a_{32}x_2 + a_{33}x_3)] \tag{5.30}$$

Lager oss en iterasjonsprosess ved å skrive:

$$x_1^{m+1} = x_1^m + \frac{1}{a_{11}}[b_1 - (a_{11}x_1^m + a_{12}x_2^m + a_{13}x_3^m)] \tag{5.31}$$

$$x_2^{m+1} = x_2^m + \frac{1}{a_{22}}[b_2 - (a_{21}x_1^m + a_{22}x_2^m + a_{23}x_3^m)] \tag{5.32}$$

$$x_3^{m+1} = x_3^m + \frac{1}{a_{33}}[b_3 - (a_{31}x_1^m + a_{32}x_2^m + a_{33}x_3^m)] \tag{5.33}$$

der m og $m+1$ er iterasjonsnummer. (5.33) kan skrives kompakt for et system av n ligninger:

$$x_i^{m+1} = x_i^m + \delta x_i \tag{5.34}$$

$$\delta x_i = \frac{1}{a_{ii}}\left[b_i - \sum_{j=1}^{n} a_{ij}x_j^m\right],\ i = 1, 2, \ldots, n,\ m = 0, 1, \ldots \tag{5.35}$$

Iterasjonsprosessen starter ved å velge verdier $x = x_0$ ved iterasjon $m = 0$.

Prosessen i (5.35) kalles Jacobis metode . Den kan også utledes direkte fra lign. (**??**) i **??**. Vi ser av (5.33) at metoden kan forbedres ved å sette inn for x_1^{m+1} i den andre ligningen og for x_1^{m+1} og x_2^{m+1} i den tredje ligningen, slik at vi får:

$$x_1^{m+1} = x_1^m + \frac{1}{a_{11}}[b_1 - (a_{11}x_1^m + a_{12}x_2^m + a_{13}x_3^m)] \tag{5.36}$$

$$x_2^{m+1} = x_2^m + \frac{1}{a_{22}}[b_2 - (a_{21}x_1^m + a_{22}x_2^m + a_{23}x_3^m)] \tag{5.37}$$

$$x_3^{m+1} = x_3^m + \frac{1}{a_{33}}[b_3 - (a_{31}x_1^m + a_{32}x_2^m + a_{33}x_3^m)] \tag{5.38}$$

Dette er Gauss-Seidels metode.

Ved å multiplisere med en faktor ω, får vi enda en variant:

$$x_1^{m+1} = x_1^m + \frac{\omega}{a_{11}}[b_1 - (a_{11}x_1^m + a_{12}x_2^m + a_{13}x_3^m)] \tag{5.39}$$

$$x_2^{m+1} = x_2^m + \frac{\omega}{a_{22}}[b_2 - (a_{21}x_1^m + a_{22}x_2^m + a_{23}x_3^m)] \tag{5.40}$$

$$x_3^{m+1} = x_3^m + \frac{\omega}{a_{33}}[b_3 - (a_{31}x_1^m + a_{32}x_2^m + a_{33}x_3^m)] \tag{5.41}$$

Generelt kan (5.41) skrives:

$$x_i^{m+1} = x_i^m + \delta x_i \tag{5.42}$$

$$\delta x_i = \frac{\omega}{a_{ii}}\left[b_i - \left(\sum_{k=1}^{i-1} a_{ik}x_k^{m+1} + \sum_{k=1}^{n} a_{ik}x_k^m\right)\right],\ i = 1, 2, \ldots, n, \tag{5.43}$$

Faktoren ω kalles *relaksasjonsparameteren* eller *relaksasjonsfaktoren*. Metoden i (5.43) kalles suksessiv overrelaksasjon når $\omega > 1$, vanligvis forkortet til SOR. $\omega = 1$ gir Gauss-Seidels metode. Det kan vises at ω ligger i intervallet $(0, 2)$ for Laplace - og Poissonligninger, men $\omega > 1$ er mest effektivt. Vi skal ikke bruke SOR på ligningsystem gitt på formen (5.27), men isteden bruke differanseligningene direkte, like in Chapter **??**.

La oss se på en Poissonligning:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y) \tag{5.44}$$

som diskretisert med $\Delta x = \Delta y = h$ gir følgende differanseligning:

$$u_{i-1,j} + u_{i+1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j} = h^2 \cdot f_{i,j} \tag{5.45}$$

Som vanlig:

$$x_i = x_0 + i \cdot h,\ i = 0, 1, 2, \ldots, i_{max} \tag{5.46}$$
$$y_j = y_0 + j \cdot h,\ j = 0, 1, 2, \ldots, j_{max} \tag{5.47}$$

(Se 5.2)
Dersom vi bruker (5.43) på (5.45) og (5.47) får vi:

$$u_{i,j}^{m+1} = u_{i,j}^m + \delta u_{i,j} \tag{5.48}$$
$$\delta u_{i,j} = \frac{\omega}{4}\left[u_{i-1,j}^{m+1} + u_{i,j-1}^{m+1} + u_{i+1,j}^m + u_{i,j+1}^m - 4u_{i,j}^m - h^2 \cdot f_{i,j}\right] \tag{5.49}$$

Kan også skrive:

$$u_{i,j}^{m+1} = u_{i,j}^m + \frac{\omega}{4} R_{i,j} \tag{5.50}$$
$$R_{i,j} = \left[u_{i-1,j}^{m+1} + u_{i,j-1}^{m+1} + u_{i+1,j}^m + u_{i,j+1}^m - 4u_{i,j}^m - h^2 \cdot f_{i,j}\right] \tag{5.51}$$

$R_{i,j}$ kalles residuet (avviket) i punkt (i,j).
Løser nå eksemplet i figur 5.2, 5.2, med bruk av (5.49).

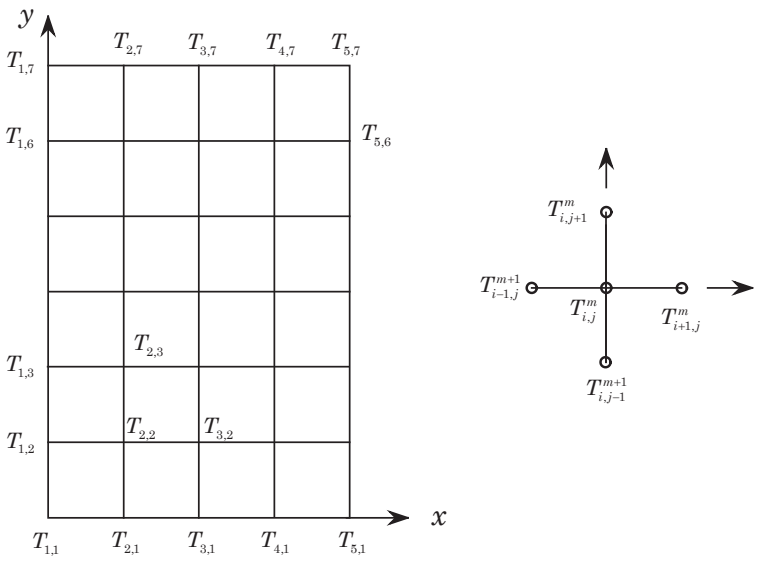


Figure 5.9: Caption goes here.

Figur 5.9 viser figur 5.2 med ny indeksering. Vi tar ikke hensyn til symmetrien. i-indeks henviser til x-retningen, j-indeks til y-retningen.

Randverdier:

$$\text{Langs } x = 0 : T_{1,j} = 0.0,\ j = 1, 2, \ldots, 6 \tag{5.52}$$
$$\text{Langs } x = 1 : T_{5,j} = 0.0,\ j = 1, 2, \ldots, 6 \tag{5.53}$$
$$\text{Langs } y = 0 : T_{i,1} = 0.0,\ i = 2, 3, \ldots, 4 \tag{5.54}$$
$$\text{Langs } y = 1.5 : T_{i,7} = 100,\ i = 1, 2, \ldots, 5 \tag{5.55}$$

Velger startverdier lik 0 for de andre temperaturene. Bruker $\omega = 1.5$ og utfører 20 iterasjoner som vist i programmet **lapsor1v1** nedenfor. $h = 0.25$.

```
% program lapsor1v1
clear
nx = 4 ; % parts in x-direction
ny = 6;  % parts in y-direction
imax = nx + 1; % points in x-direction
jmax = ny + 1; % points in y-direction
T = zeros(imax,jmax); % temperatures
T(1:imax,jmax) = 100; %  boundary values
omega = 1.5; % relaxation factor
% --- Start iteration ---
for it = 1: 20
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;
T(i,j) = T(i,j) + dT;
end
end
end
```

Utskrift av T-matrise:

0	0	0	0	0	0	100.0000
0	1.5784	4.0918	9.0575	19.6196	43.1933	100.0000
0	2.2220	5.7310	12.5185	26.2279	53.1537	100.0000
0	1.5784	4.0917	9.0575	19.6197	43.1933	100.0000
0	0	0	0	0	0	100.0000

Resultatet stemmer godt overens med verdiene gitt i 5.2. I dette programmet har vi valgt vilkårlig $\omega = 1.5$ og antall iterasjoner lik 20. Vi ønsker selvfølgelig et program med en selvstoppende iterasjonsprosess. Dessuten er det ønskelig med en ω-verdi som gir færrest mulig iterasjoner for en gitt nøyaktighet.

Stoppkriterier. Eksempelvis kan vi bruke $\max(\delta T_{i,j}) < \varepsilon_a$ eller $\max\left(\dfrac{\delta T_{i,j}}{T_{i,j}}\right) < \varepsilon_r$. Eventuelt kan vi bruke residuet isteden. Andre alternativ:

$$\frac{1}{N}\sum_i\sum_j |\delta T_{i,j}| < tol_a,\ \frac{1}{N}\sum_i\sum_j \left|\frac{\delta T_{i,j}}{T_{i,j}}\right| < tol_r,\ |T_{i,j}| \neq 0 \tag{5.56}$$

N er antall beregningspunkt. I (5.56) kan residuet brukes istedet for $\delta T_{i,j}$. I den første uttrykket bruker vi en absolutt toleranse, mens vi bruker en relativ toleranse i det siste. Velger å bruke følgende alternativ til (5.56):

$$\frac{\sum_i\sum_j |\delta T_{i,j}|}{\sum_i\sum_j |T_{i,j}|} < tol_r,\ \frac{\max\left(|\delta T_{i,j}|\right)}{\max\left(|T_{i,j}|\right)} < tol_r \tag{5.57}$$

(5.57) gir en slags midlere relativ feiltest. Vi summerer over alle beregnings-punktene i disse formlene. Fra figur 4.10 i **??**, ser vi at antall iterasjoner er en funksjon både av ω og h, der vi bruker $\Delta x = \Delta y = h$.

Lager derfor et program der vi kan variere skrittlengden h. I programmet **lapsor1v2** på neste side, kan vi forandre h i trinn med $h_n = \frac{h}{2^n},\ n = 0, 1, \ldots$ Bruker $\frac{\sum_i \sum_j |\delta T_{i,j}|}{\sum_i \sum_j |T_{i,j}|} < tol_r$, som stoppkriterium. Figur 5.10 viser antall iterasjoner som funksjon av ω og h med $tol_r = 10^{-5}$

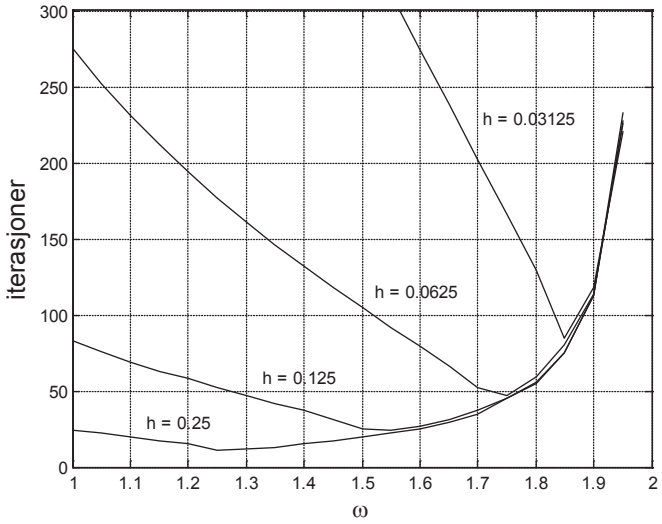


Figure 5.10: Caption goes here.

```
% program lapsor1v2
clear
n = 0;
fac = 2^(n+1);
imax = 2*fac + 1; % points in x-direction
jmax = 3*fac + 1; % points in y-direction
T = zeros(imax,jmax); % temperatures

T(1:imax,jmax) = 100; % boundary values
reltol = 1.0e-5; % relative iteration error
omega = 1.95; % relaxation factor
relres = 1.0; it = 0;
% --- Start iteration ---
while relres > reltol
it = it + 1;
Tsum = 0.0; dTsum = 0.0;
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
relres = dTsum/Tsum;
end
```

Dersom vi isteden bruker $\frac{\max(|\delta T_{i,j}|)}{\max(|T_{i,j}|)} < tol_r$ som stoppkriterium, vil iterasjonløkka bli:

```
while relres > reltol
it = it + 1;
Tmax = 0.0; dTmax = 0.0;
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;
dTmax = max(dTmax, abs(dT));
T(i,j) = T(i,j) + dT;
Tmax = max(Tmax, abs(T(i,j));
end
end
relres = dTmax/Tmax;
end
```

Resultatet blir stort sett det samme for dette tilfellet.

Optimal relaksasjonsparameter. Med optimal menes her den verdien som gir færrest mulig iterasjoner når ω holdes konstant under hele beregningen. For Laplace- og Poisson-ligninger på rektangulære områder er det mulig å beregne en slik optimal ω som vi vil kalle den teoretisk optimale.

La L_x og L_y være utstrekningen av rektanglet i henholdsvis x- og y-retning. Med den samme skrittlengden h i begge retningene, setter vi:
$n_x = \dfrac{L_x}{h},\ n_y = \dfrac{L_y}{h}$ der n_x og n_y blir antall deler i henholdsvis x- og y-retning. n_x og n_y skal være heltall. Den teoretisk optimale ω er da gitt ved:

$$\rho = \frac{1}{2}[\cos(\pi/n_x) + \cos(\pi/n_y)] \tag{5.58}$$

$$\omega = \frac{2}{1+\sqrt{1-\rho^2}} \tag{5.59}$$

Dersom skrittlengden h er forskjellig i x- og y- retning med $h = h_x$ i x-retning og $h = h_y$ i y-retning, får vi isteden for (5.59):

$$\rho = \frac{\cos(\pi/n_x) + (h_x/h_y)^2 \cdot \cos(\pi/n_y)}{1 + (h_x/h_y)^2} \tag{5.60}$$

Dersom området ikke er rektangulært, kan vi bruke Garabedians estimat:

$$\omega = \frac{2}{1 + 3.014 \cdot h/\sqrt{A}},\ A \text{ er arealet av området} \tag{5.61}$$

La oss regne ut ω fra disse formlene for eksemplet der $L_x = 1,\ L_y = 1.5$ og $A = L_x \cdot L_y = 1.5$.

h	(5.59)	(5.61)
0.25	1.24	1.24
0.125	1.51	1.53
0.0625	1.72	1.74
0.03125	1.85	1.86

Vi ser at Garabedians estimat stemmer godt overens med de teoretisk eksakte verdiene i dette tilfellet. Figur 5.10 stemmer også godt overens med verdiene i denne tabellen.

Eksempel på bruk av SOR. Vi løser nå temperaturproblemet i figur 5.5 (se 5.2.1). Nummereringen blir som vist nedenfor i figur 5.11.

Vi bruker også her falske punkt som indikert med de stiplede linjene. For $T_{1,1}$ får vi spesielt:

$$T_{1,1} = \frac{1}{4}(T_{1,2} + T_{1,2} + T_{2,1} + T_{2,1}) = \frac{1}{2}(T_{1,2} + T_{2,1}) \tag{5.62}$$

Beregningen startes ved iterere langs $y = 0$ med start i $T_{2,1}$. Deretter itereres langs $x = 0$ med start i $T_{1,2}$. Etterpå itereres i en dobbel løkke over de indre punktene. Tilslutt beregnes $T_{1,1}$ fra (5.62). Beregningen er

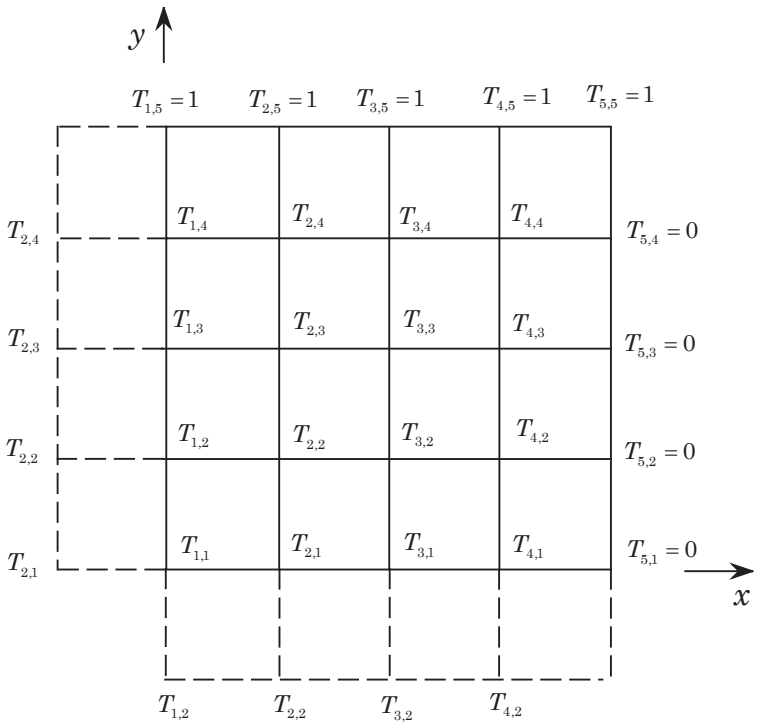


Figure 5.11: Caption goes here.

vist i programmet **lapsor2** på neste side. Vi har brukt stoppkriteriet $\frac{\sum_i \sum_j |\delta T_{i,j}|}{\sum_i \sum_j |T_{i,j}|} < tol_r$ samt optimal ω fra (5.59) og tipper $T = 0.5$ som startverdier for hele feltet unntatt for de gitte randbetingelsene. Nøyaktigheten er som i utskriften for **lap2v3** i 5.2.1.

```
% 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
```

Startverdier og randbetingelser. Vi venter å få raskere konvergens dersom vi tipper startverdier som ligger nær den korrekte løsningen. Dette er typisk for ikke-lineære ligninger, mens vi står mer fritt til å tippe startverdier når vi løser lineære ligninger uten at det går utover konvergenshastigheten. For temperaturproblemet i figur 5.2 for eksempel, er det lite forskjell i antall iterasjoner om vi starter iterasjonen med å tippe $T = 0$ i hele feltet eller om vi starter med $T = 100$. Den optimale ω for dette tilfellet er også uavhengig av startverdiene. Situasjonen er helt forskjellig for tilfellet i figur 5.5. Vi løser også her en lineær ligning, men har mer kompliserte randbetingelser der vi foreskriver temperaturen langs to render (Dirichlet-betingelser) og den deriverte av temperaturen langs de to andre rendene (Neumann-betingelser) . Dessuten er temperaturen diskontinuerlig i hjørnet $x = 1,\ y = 1$. I hjørnet $x = 0,\ y = 0$ er den korrekte løsningen $T = 0.5$ fra den analytiske løsningen. Dersom vi tipper $T = 0.5$ som startverdier i hele feltet, får vi rask konvergens med optimal ω lik den teoretiske fra (5.59). Dersom vi avviker litt fra $T = 0.5$ som startverdi, er ikke lenger den optimale ω lik den teoretisk optimale. Situasjonen blir som vist i figur **??** nedenfor.

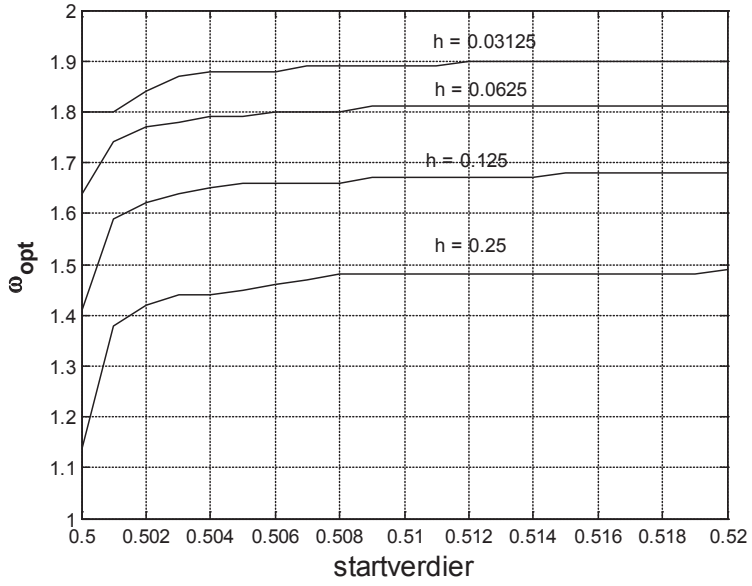


Figure 5.12: Caption goes here.

Tabellen nedenfor viser den teoretisk optimale ω etter formel (5.59) og (5.61)

h	(5.59)	(5.61)
0.25	1.17	1.14
0.125	1.45	1.45
0.0625	1.67	1.68
0.03125	1.82	1.83

Vi ser at tabell-verdiene stemmer godt med verdiene i figur 5.12 når startverdien for iterasjonsprosessen er 0.5.

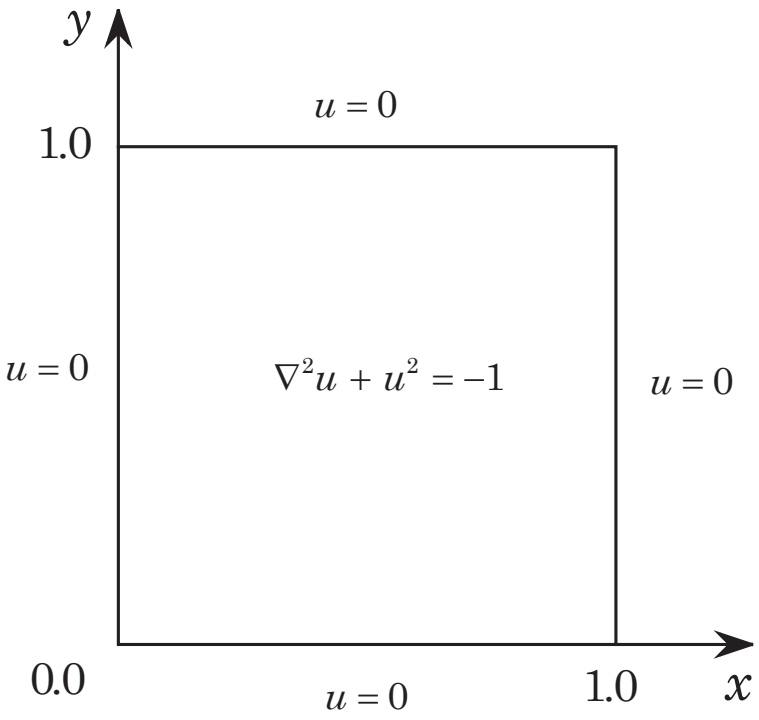


Figure 5.13: Caption goes here.

Eksempel på en ikke-lineær ligning. Vi ønsker å løse en ikke-lineær Poisson-ligning på kvadratet i figur 5.13:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + u^2 = -1 \tag{5.63}$$

der $u = u(x, y)$ og $u = 0$ på rendene.

Denne ligningen er bare svakt ikke-lineær, kalt semi-lineær i, men den illustrerer fremgangsmåten.
Vi diskretiserer på vanlig måte med skrittlengde h i begge retningene:

$$\nabla^2 u_{i,j} + u_{i,j}^2 + 1 = 0 \rightarrow$$
$$\frac{1}{h^2}[u_{i+1,j} + u_{i-1,j} + u_{i,j-1} - 4u_{i,j}] + u_{i,j}^2 + 1 = 0$$

eller:

$$f_{i,j} = u_{i+1,j} + u_{i-1,j} + u_{i,j-1} - 4u_{i,j} + h^2(u_{i,j}^2 + 1) = 0 \tag{5.64}$$

Her er $x_i = h \cdot (i-1),\ i = 1, 2, \ldots$ og $y_j = h \cdot (j-1),\ j = 1, 2, \ldots$

Nummereringen er vist i figur 5.14 for tilfellet $h = 0.25$.

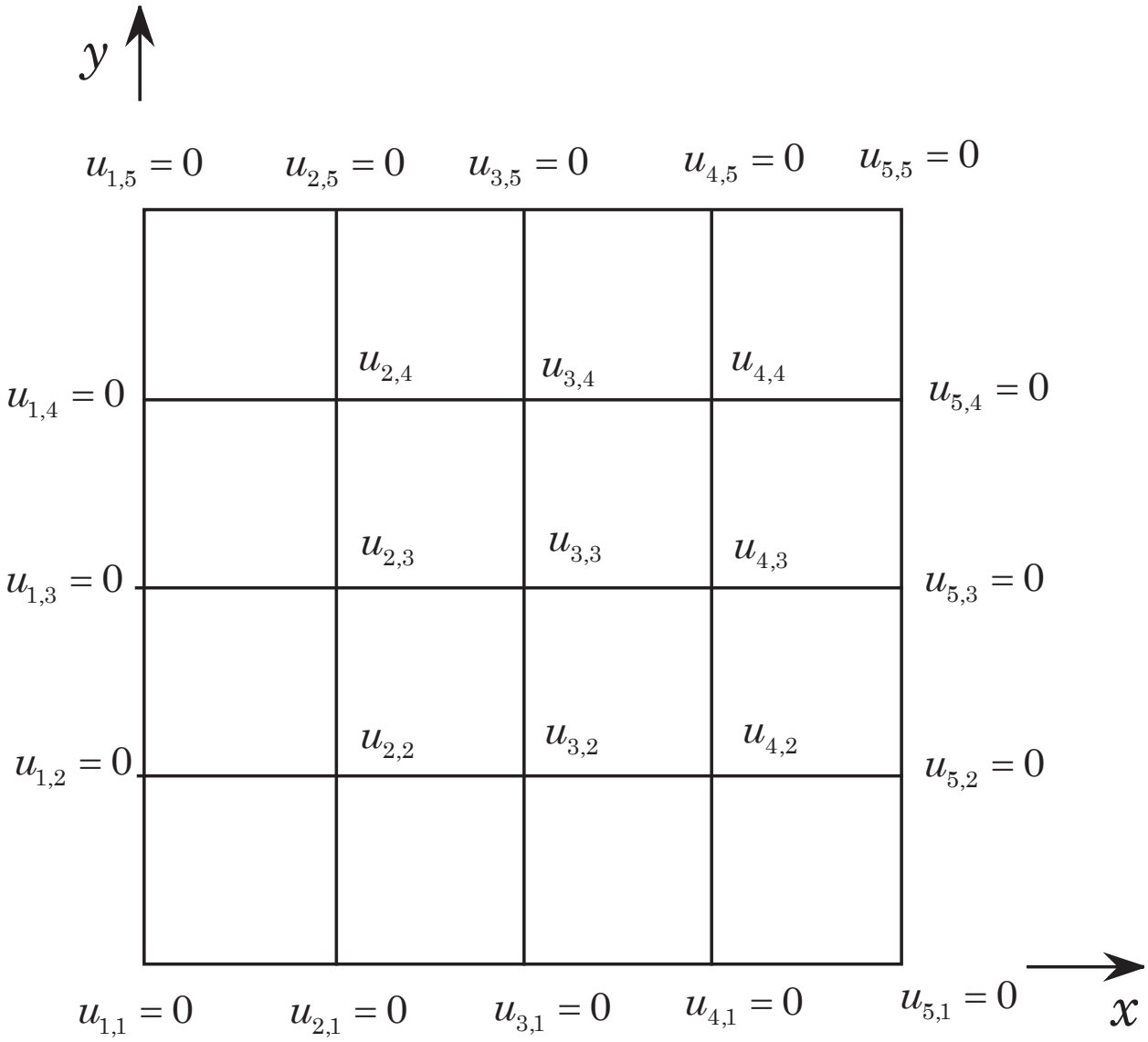


Figure 5.14: Caption goes here.

Vi løser (5.64) med Newtons metode ved å se på den lokalt som en funksjon der vi forandrer en variabel etter tur.

I dette tilfellet blir iterasjonsprosessen:

$$u_{i,j}^{m+1} = u_{i,j}^{m} + \delta u_{i,j} \tag{5.65}$$

$$\delta u_{i,j} = -\omega \frac{f(u_{k,l})}{\frac{\partial f(u_{k,l})}{\partial u_{i,j}}} \tag{5.66}$$

Her blir:

$$u_{k,l} = u_{k,l}^{m+1} \text{ for } k < i,\ l < j \tag{5.67}$$

$$u_{k,l} = u_{k,l}^{m} \text{ ellers} \tag{5.68}$$

$$\frac{\partial f}{\partial u_{i,j}} = -4 + 2h^2 \cdot u_{i,j} \text{ og } \delta u_{i,j} = \omega \frac{f}{4 - 2h^2 u_{i,j}} \tag{5.69}$$

Programmet **npoisor** på neste side bruker (5.64) og (5.66) –(5.69)

```
% program npoisor
% We may select different nets
% by specifying the parameter net.
%
% net = 1 -> h = 0.25, net = 2 -> h = 0.25/2
% giving hn = h/2^(net -1)
clear
net = 2;
h = 0.25;
hn = h/2^(net -1);
nx = 1/hn; ny = nx;
hn2 = hn*hn;
imax = nx + 1; % points in x-direction
jmax = ny + 1; % points in y-direction
% --- Initial values including the boundaries ---
u = zeros(imax,jmax);
%
% --- Compute optimal omega ---
ro = cos(pi/nx);
omega = 2/(1 + sqrt(1 - ro^2));
reltol = 1.0e-5; % relative iteration error
relres = 1.0; it = 0;
% --- Start iteration ---
while relres > reltol
it = it + 1;
usum = 0.0; dusum = 0.0;
for i = 2: imax-1
for j = 2: jmax-1
fac1 = 4 - 2*hn2*u(i,j) ;
fac2 = hn2*(u(i,j)^2 + 1);
resid = u(i-1,j)+u(i,j-1)+u(i+1,j)+u(i,j+1)-
4*u(i,j)+fac2
du = omega*resid/fac1;
dusum = dusum + abs(du);
u(i,j) = u(i,j) + du;
usum = usum + abs(u(i,j));
end
end
relres = dusum/usum;
end
u
```

```
u =
 0         0         0         0         0         0         0         0         0
```

```
0    0.0178    0.0278    0.0330    0.0346    0.0330    0.0278    0.0178         0
0    0.0278    0.0448    0.0539    0.0568    0.0539    0.0448    0.0278         0
0    0.0330    0.0539    0.0654    0.0691    0.0654    0.0539    0.0330         0
0    0.0346    0.0568    0.0691    0.0730    0.0691    0.0568    0.0346         0
0    0.0330    0.0539    0.0654    0.0691    0.0654    0.0539    0.0330         0
0    0.0278    0.0448    0.0539    0.0568    0.0539    0.0448    0.0278         0
0    0.0178    0.0278    0.0330    0.0346    0.0330    0.0278    0.0178         0
0         0         0         0         0         0         0         0         0
```

Utskriften ovenfor er for $h = 0.125$ og vi har brukt stoppkriteriet $\frac{\sum_i \sum_j |\delta TT_{i,j}|}{\sum_i \sum_j |T_{i,j}|} < tol_r$ med $tol_r = 10^{-5}$. Startverdien $u = 0$ for hele feltet er naturlig i dette tilfellet. Vi har brukt (5.59) til å beregne den optimale ω. Tabellen nedenfor viser også den virkelig optimale ω for starverdien $u = 0$.

h	(5.59)	Virkelig
0.25	1.17	1.19
0.125	1.45	1.46
0.0625	1.67	1.69
0.03125	1.82	1.83

I dette tilfellet får vi god overenstemmelse selv om ligningen er svakt ikke-lineær.

5.3.2 KONVERGENSKRITERIER

Vi går tilbake til et lineært ligningsystem $\mathbf{Ax} = \mathbf{b}$ som for et tilfelle med 4 ukjente blir:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 &= b_3 \\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 &= b_4 \end{aligned}$$

Innfører følgende matriser:

$$\mathbf{D} = \begin{pmatrix} a_{11} & 0 & 0 & 0 \\ 0 & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & 0 \\ 0 & 0 & 0 & a_{44} \end{pmatrix}, -\mathbf{L} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ a_{21} & 0 & 0 & 0 \\ a_{31} & a_{32} & 0 & 0 \\ a_{41} & a_{42} & a_{43} & 0 \end{pmatrix} -\mathbf{U} = \begin{pmatrix} 0 & a_{12} & a_{13} & a_{14} \\ 0 & 0 & a_{23} & a_{24} \\ 0 & 0 & 0 & a_{34} \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{5.70}$$

$\mathbf{D}$ er en diagonalmatrise og $\mathbf{L}$ og $\mathbf{U}$ er henholdsvis en nedre og en øvre trekantmatrise. Koeffisientmatrisa $\mathbf{A}$ kan da skrives: $\mathbf{A} = \mathbf{D} - \mathbf{L} - \mathbf{U}$ slik at ligningsystemet blir:

$$\mathbf{Dx} = (\mathbf{L} + \mathbf{U})\mathbf{x} + \mathbf{b} \tag{5.71}$$

Med bruk av notasjonen i (5.71), kan Jacobis iterasjonsmetode i (5.33) skrives:

$$\mathbf{Dx^{m+1}} = (\mathbf{L} + \mathbf{U})\mathbf{x^m} + \mathbf{b}$$

som gir:

$$\mathbf{x^{m+1}} = \mathbf{D^{-1}}(\mathbf{L} + \mathbf{U})\mathbf{x^m} + \mathbf{D^{-1}b} \tag{5.72}$$

Setter:

$$\mathbf{G_J} = \mathbf{D^{-1}}(\mathbf{L} + \mathbf{U}) \tag{5.73}$$

$\mathbf{G_J}$ kalles iterasjonsmatrisa for Jacobis metode. Når ligningene er skrevet på matriseform, gjelder de selvfølgelig for et vilkårlig antall ukjente.

Tilsvarende for Gauss-Seidels metode i (5.38):

$$\mathbf{D}\mathbf{x}^{\mathbf{m+1}} = \mathbf{L}\mathbf{x}^{\mathbf{m+1}} + \mathbf{U}\mathbf{x}^{\mathbf{m}} + \mathbf{b}$$

som gir:

$$\mathbf{x}^{\mathbf{m+1}} = (\mathbf{D} - \mathbf{L})^{-\mathbf{1}}\mathbf{U}\mathbf{x}^{\mathbf{m}} + (\mathbf{D} - \mathbf{L})^{-\mathbf{1}}\mathbf{b} \tag{5.74}$$

Setter:

$$\mathbf{G_{GS}} = (\mathbf{D} - \mathbf{L})^{-\mathbf{1}}\mathbf{U} \tag{5.75}$$

$\mathbf{G_{GS}}$ kalles iterasjonsmatrisa for Gauss-Seidels metode.

For SOR- metoden i (5.43):

$$(\mathbf{I} - \omega\mathbf{D}^{-1}\mathbf{L})\mathbf{x}^{m+1} = [(1-\omega)\mathbf{I} + \omega\mathbf{D}^{-1}\mathbf{U}]\mathbf{x}^{m} + \omega\mathbf{D}^{-1}\mathbf{b}$$

som gir:

$$\mathbf{x}^{m+1} = (\mathbf{I} - \omega\mathbf{D}^{-1}\mathbf{L})^{-1}[(1-\omega)\mathbf{I} + \omega\mathbf{D}^{-1}\mathbf{U}]\mathbf{x}^{m} + (\mathbf{I} - \omega\mathbf{D}^{-1}\mathbf{L})^{-1}\omega\mathbf{D}^{-1}\mathbf{b} \tag{5.76}$$

Setter

$$\mathbf{G}_{SOR} = (\mathbf{I} - \omega\mathbf{D}^{-1}\mathbf{L})^{-1}[(1-\omega)\mathbf{I} + \omega\mathbf{D}^{-1}\mathbf{U}] \tag{5.77}$$

$\mathbf{G}_{SOR}$ kalles iterasjonsmatrisa for SOR-metoden. Gauss-Seidel fås som et spesialtilfelle ved å sette $\omega = 1$.

Vi har delvis fulgt fremstillingen som er gitt i Smith [20], side 266.

Vi ser at alle tre metodene kan skrives:

$$\mathbf{x}^{m+1} = \mathbf{G}\mathbf{x}^{m} + \mathbf{c} \tag{5.78}$$

$$\mathbf{G} \text{ står her for } \mathbf{G}_J,\ \mathbf{G}_{GS} \text{ og } \mathbf{G}_{SOR} \tag{5.79}$$

Ved konvergens blir $\mathbf{x}^{m+1} = \mathbf{x}^{m} = \mathbf{x}$ slik at det eksakte systemet kan skrives:

$$\mathbf{x} = \mathbf{G}\mathbf{x} + \mathbf{c} \tag{5.80}$$

Vi kaller feil-vektoren i den m'te iterasjonen for $\mathbf{e}^m$ slik at

$$\mathbf{e}^{m} = \mathbf{x} - \mathbf{x}^{m} \tag{5.81}$$

Ved å trekke (5.79) fra (5.80) får vi:

$$\mathbf{e}^{m+1} = \mathbf{G}\mathbf{e}^{m} \tag{5.82}$$

Derav følger:

$$\mathbf{e}^m = \mathbf{G}\mathbf{e}^{m-1} = \mathbf{G}^2\mathbf{e}^{m-2} = \mathbf{G}^3\mathbf{e}^{m-3} = \cdots = \mathbf{G}^m\mathbf{e}^0 \tag{5.83}$$

der $\mathbf{e}^m$ er feilvektoren i den første iterasjonen, altså forskjellen mellom startverdiene og de rette verdiene.

Dersom prosessen i (5.82) og (5.83) skal konvergere for vilkårlige startverdier $\mathbf{x^0}$, må følgende betingelse være oppfylt:

$$\lim_{m\to\infty} \mathbf{G}^m = 0 \tag{5.84}$$

En nødvendig og tilstrekkelig betingelse for å oppfylle (5.84), er at tallverdien for den største egenverdien i $\mathbf{G}$ er mindre enn 1.

Dette skrives:

$$\rho = |\lambda_{max}| < 1 \tag{5.85}$$

Absoluttverdien av den største egenverdien av en matrise $\mathbf{A}$ kalles spektralradien og betegnes ofte med ρ. Skrives gjerne $\rho(\mathbf{A})$ når det er nødvendig å henvise til den underliggende matrisa. (Se **??**).

I Smith [20] er både nødvendigheten og tilstrekkeligheten vist under forutsetningen av at alle egenvektorene av $\mathbf{G}$ er uavhengige. Et generelt bevis er mer kronglete og kan finnes i mer avansert litteratur, f.eks Hageman &Young [9].

Følger derfor Smith og antar at iterasjonsmatrisa $\mathbf{G}$ har dimensjon $n \times n$ og har n uavhengige egenvektorer $\mathbf{v}_k,\ k = 1, 2, \ldots, n$.

Feilvektoren $\mathbf{e}^0$ i den første iterasjonen kan da utrykkes i egenvektor-rommet ved:

$$\mathbf{e}^0 = c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_n\mathbf{v}_n = \sum_{k=1}^{n} c_k\mathbf{v}_k$$

der $c_k,\ k = 1, 2, \ldots, n$ er skalarer

Nå får vi:

$$\mathbf{e}^1 = \mathbf{G}\mathbf{e}^0 = c_1\mathbf{G}\mathbf{v}_1 + c_2\mathbf{G}\mathbf{v}_2 + \cdots + c_n\mathbf{G}\mathbf{v}_n = \sum_{k=1}^{n} c_k\mathbf{G}\mathbf{v}_k \tag{5.86}$$

La λ_k være den k'te egenverdien slik at med $\mathbf{v}_k$ som den k'te egenvektoren, får vi $\mathbf{G}\mathbf{v}_k = \lambda_k\mathbf{v}_k$ som innsatt i (5.86) gir:

$$\mathbf{e}^1 = \sum_{k=1}^{n} c_k\lambda_k\mathbf{v}_k$$

Dette er for den første iterasjonen. For den m'te iterasjonen:

$$\mathbf{e}^m = \sum_{k=1}^{n} c_k(\lambda_k)^m\mathbf{v}_k \tag{5.87}$$

(5.87) gir da følgende:

Nødvendig betingelse: Dersom $\lim_{m\to 0} \mathbf{e}^m = 0$ skal gjelde for vilkårlige startvektorer $\mathbf{x}^0$ og da også for vilkårlige feilvektorer $\mathbf{e}^0$, må $|\lambda_k| < 1,\ k = 1, 2, \ldots, n$.

Tilstrekkelig betingelse: Dersom $|\lambda_k| < 1,\ k = 1, 2, \ldots, n$, følger umiddelbart at $\mathbf{e}^m \to 0$ for en vilkårlig $\mathbf{e}^0$.

Dette betyr at at de klassiske iterasjonsmetodene konvergerer hvis og bare hvis spektralradien for iterasjonsmatrisa er mindre enn 1.

Vi kan også bruke (5.87) til å si noe om konvergenshastigheten. Til det trenger vi vektor- og matrisenormer. Henviser her til **??**. La oss se på et egenverdiproblem $\mathbf{Ax} = \lambda\mathbf{x},\ \mathbf{x} \neq 0,$. Med bruk av lign. (**??**)

$$||\mathbf{Ax}|| = ||\lambda\mathbf{x}|| = |\lambda|||\mathbf{x}||$$
$$||\mathbf{Ax}|| \leq ||\mathbf{A}|| \cdot ||\mathbf{x}||$$

som gir:

$$|\lambda| \cdot ||\mathbf{x}|| \leq ||\mathbf{A}|| \cdot ||\mathbf{x}||$$

eller:

$$|\lambda| \leq ||\mathbf{A}|| \tag{5.88}$$

Merk at λ står for alle egenverdiene slik at spesielt $\rho(\mathbf{A}) \leq ||\mathbf{A}||$
Fra (5.82):

$$\mathbf{e}^{m+1} = \mathbf{G}\mathbf{e}^m \rightarrow ||\mathbf{e}^{m+1}|| = ||\mathbf{G}|| \cdot ||\mathbf{e}^m||$$

Anta nå at etter at vi har utført m iterasjoner, gjør k ekstra:

$$||\mathbf{e}^{m+k}|| = ||\mathbf{G}^k|| \cdot ||\mathbf{e}^m|| \tag{5.89}$$

Sorterer egenverdiene etter størrelse: $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n|$.
Her er $\lambda_1 = \lambda_{max}$ og $\rho = |\lambda_{max}|$. Merk at vi har antatt at λ_1 og λ_2 ikke faller sammen. Fra (5.87):

$$\begin{aligned}\mathbf{e}^m =& c_1\lambda_1^m\mathbf{v}_1 + c_2\lambda_2^m\mathbf{v}_2 + \cdots + c_n\lambda_n^m\mathbf{v}_n \rightarrow ||\mathbf{e}^m|| \leq ||c_1\lambda_1^m\mathbf{v}_1|| + ||c_2\lambda_2^m\mathbf{v}_2|| \\ &+ \cdots + ||c_n\lambda_n^m\mathbf{v}_n|| \\ =& |\lambda_1^m| \cdot \left[||c_1\mathbf{v}_1|| + \left|\frac{\lambda_2}{\lambda_1}\right|^m ||c_2\mathbf{v}_2|| + \cdots + \left|\frac{\lambda_n}{\lambda_1}\right|^m ||c_n\mathbf{v}_n||\right]\end{aligned}$$

For tilstrekkelig store verdier av m:

$$||\mathbf{e}^m|| \approx |\lambda_1|^m||c_1\mathbf{v}_1|| = \rho^m||c_1\mathbf{v}_1|| \tag{5.90}$$

Fra (5.88), (5.89) og (5.90) følger:

$$||\mathbf{e}^{m+k}|| \approx \rho^k||\mathbf{e}^m|| \rightarrow \frac{||\mathbf{e}^{m+k}||}{||\mathbf{e}^m||} \approx \rho^k \tag{5.91}$$

Vi ønsker spesielt å finne for hvilken verdi av k er $||\mathbf{e}^{m+k}||$ en tiendepart av $||\mathbf{e}^m||$:

$$\frac{||\mathbf{e}^{m+k}||}{||\mathbf{e}^m||} = \frac{1}{10} \approx \rho^k \tag{5.92}$$

Ved bruk av den Briggske logaritmen i (5.92):

$$k \approx -\frac{1}{\log_{10}(\rho)} \tag{5.93}$$

k angir hvor mange iterasjoner vi må utføre for å vinne ett desimalsiffer. Størrelsen

$$R = -\log_{10}(\rho) \tag{5.94}$$

kalles ofte det midlere konvergenstallet. Istedenfor den Briggske logaritmen brukes gjerne den naturlige.

Ved bruk av matrisenormer kan vi finne tilstrekkelige betingelser for konvergens. La oss f. eks. velge Jacobis metode, lign.(5.71) med iterasjonsmatrisa gitt i (5.73):

$$\mathbf{G}_J = \mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})$$

For systemet i (5.70):

$$\mathbf{G}_J = -\begin{pmatrix} \frac{1}{a_{11}} & 0 & 0 & 0 \\ 0 & \frac{1}{a_2} & 0 & 0 \\ 0 & 0 & \frac{1}{a_3} & 0 \\ 0 & 0 & 0 & \frac{1}{a_{44}} \end{pmatrix} \cdot \begin{pmatrix} 0 & a_{12} & a_{13} & a_{14} \\ a_{21} & 0 & a_{23} & a_{24} \\ a_{31} & a_{32} & 0 & a_{34} \\ a_{41} & a_{42} & a_{43} & 0 \end{pmatrix}$$

$$= -\begin{pmatrix} 0 & \frac{a_{12}}{a_{11}} & \frac{a_{13}}{a_{11}} & \frac{a_{14}}{a_{11}} \\ \frac{a_{21}}{a_{22}} & 0 & \frac{a_{23}}{a_{22}} & \frac{a_{24}}{a_{22}} \\ \frac{a_{31}}{a_{33}} & \frac{a_{32}}{a_{33}} & 0 & \frac{a_{34}}{a_{33}} \\ \frac{a_{41}}{a_{44}} & \frac{a_{42}}{a_{44}} & \frac{a_{43}}{a_{44}} & 0 \end{pmatrix}$$

En matrise $\mathbf{A}$ kalles strengt diagonaldominant dersom:

$$|a_{i,j}| > \sum_{j=1,\ j\neq i}^{n} |a_{i,j}| \text{ for alle } 1 \leq i \leq n \tag{5.95}$$

Anta nå at $\mathbf{A}$ er strengt diagonaldominant. Da blir tallverdien av alle leddene i $\mathbf{G}_j$ mindre enn 1. Bruker matrisenormen $||\mathbf{A}||_\infty$ samt (5.88):

$$|\lambda| < ||\mathbf{G}_J||_\infty = \max_{1\leq i\leq n} \sum_{j=1}^{n} |\mathbf{G}_{i,j}| < 1$$

Med andre ord: Dersom iterasjonsmatrisa er strengt diagonaldominant, konvergerer Jacobis metode uavhengig av startvektoren. Det kan vises at dette også gjelder både Gauss-Seidel og SOR også.

Eksempel ved bruk av Gauss-Seidels metode. La oss løse følgende system $\mathbf{Ax} = \mathbf{b}$ med fire ukjente der vi bruker Gauss-Seidels metode:

$$\begin{pmatrix} 2 & -1 & 0 & 0 \\ -1 & 3 & -1 & 0 \\ 0 & -1 & 3 & -1 \\ 0 & 0 & -1 & 2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} -1 \\ 4 \\ 7 \\ 0 \end{pmatrix}$$

Dette systemet har løsningen $x_1 = 1,\ x_2 = 3,\ x_3 = 4,\ x_4 = 2$. I praksis ville vi selvfølgelig ikke bruke Gauss-Seidels metode på et slik system, men systemet er så enkelt at det er mulig å analysere det. Lign. (5.70) blir i dette tilfellet:

$$\mathbf{D} = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}, -\mathbf{U} = \begin{pmatrix} 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 \end{pmatrix} -\mathbf{L} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Videre får vi:

$$\mathbf{D}-\mathbf{L} = \begin{pmatrix} 2 & 0 & 0 & 0 \\ -1 & 3 & 0 & 0 \\ 0 & -1 & 3 & 0 \\ 0 & 0 & -1 & 2 \end{pmatrix},\ (\mathbf{D}-\mathbf{L})^{-1} = \begin{pmatrix} \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{6} & \frac{1}{3} & 0 & 0 \\ \frac{1}{18} & \frac{1}{9} & \frac{1}{3} & 0 \\ \frac{1}{36} & \frac{1}{18} & \frac{1}{6} & \frac{1}{2} \end{pmatrix}$$

Iterasjonsmatrisa $\mathbf{G}_{GS}$ i (5.75) blir nå:

$$\mathbf{G_{GS}} = (\mathbf{D}-\mathbf{L})^{-\mathbf{1}}\mathbf{U} = \begin{pmatrix} 0 & \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{6} & \frac{1}{3} & 0 \\ 0 & \frac{1}{18} & \frac{1}{9} & \frac{1}{3} \\ 0 & \frac{1}{36} & \frac{1}{18} & \frac{1}{6} \end{pmatrix}$$

Beregner egenverdiene av iterasjonsmatrisa:

$$\det(\mathbf{G_{GS}} - \lambda\mathbf{I}) = \det \begin{pmatrix} -\lambda & \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{6}-\lambda & \frac{1}{3} & 0 \\ 0 & \frac{1}{18} & \frac{1}{9}-\lambda & \frac{1}{3} \\ 0 & \frac{1}{36} & \frac{1}{18} & \frac{1}{6}-\lambda \end{pmatrix} = 0$$

Vi får en enkel 4. grads ligning: $\lambda^2(\lambda^2 - \frac{4}{9}\lambda + \frac{1}{36}) = 0$ med løsning:

$$\rho = \lambda_1 = 0.3692,\ \lambda_2 = 0.0752,\ \lambda_3 = \lambda_4 = 0$$

Vi kan selvfølgelig bruke Matlab direkte til å finne egenverdiene:

```
> Ggs = [0 1/2 0 0; 0 1/6 1/3 0; 0 1/18 1/9 1/3; 0 1/36 1/18 1/6];
>> eig(Ggs)
ans = 0 0.0752  0.3692 0.0000
>>
```

Da tallverdien for alle egenverdiene er mindre enn 1, følger at Gauss-Seidels metode konvergerer for dette tilfellet. (Systemet er strengt diagonaldominant). Konvergenstallet $-\log_{10}(\rho)$ fra (5.94) blir $-\log_{10}(0.3692) = 0.4327$ slik at $k \approx 2.3$ som betyr at vi må utføre litt mer enn to iterasjoner for hvert desimalsiffer vi ønsker.

Gauss-Seidel systemet i (5.74) blir i vårt tilfelle:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}^{m+1} = \begin{pmatrix} 0 & \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{6} & \frac{1}{3} & 0 \\ 0 & \frac{1}{18} & \frac{1}{9} & \frac{1}{3} \\ 0 & \frac{1}{36} & \frac{1}{18} & \frac{1}{6} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}^{m} + \begin{pmatrix} -\frac{1}{2} \\ \frac{7}{6} \\ \frac{49}{18} \\ \frac{49}{36} \end{pmatrix},\ m = 0, 1, \ldots$$

Vi skriver et lite Matlabprogram som utfører iterasjonsprosessen:

```
% Program GStest
Ggs = [0 1/2 0 0; 0 1/6 1/3 0; 0 1/18 1/9 1/3 ; 0 1/36 1/18 1/6];
c = [ -1/2; 7/6 ; 49/18; 49/36 ];
x = zeros(4,1); % Startverdier
for k = 1:12
x = Ggs*x + c;
fprintf(' $%6.4f %6.4f  %6.4f  %6.4f \n',x')
end
```

```
>> GStest
-0.5000 1.1667  2.7222  1.3611
0.0833 2.2685  3.5432  1.7716
0.6343 2.7258  3.8325  1.9162
0.8629 2.8985  3.9382  1.9691
0.9492 2.9625  3.9772  1.9886
0.9812 2.9861  3.9916  1.9958
0.9931 2.9949  3.9969  1.9984
0.9974 2.9981  3.9989  1.9994
0.9991 2.9993  3.9996  1.9998
0.9997 2.9997  3.9998  1.9999
0.9999 2.9999  3.9999  2.0000
1.0000 3.0000  4.0000  2.0000
>>
```

Vi ser at i dette tilfellet konvergerer Gauss-Seidels metode raskt, noe som skyldes at spektralradien er liten og godt separert fra λ_2. Desverre er nok ikke forholdene så gunstige for de tilfellene der vi ønsker å bruke iterasjonsmetoder.

Anta at vi skal løse en Poisson-ligning $\nabla^2 u = f(x, y)$ på et enhetskvadrat, se f.eks. figur 5.13, og la skrittlengden være h i begge koordinat-retningene. For dette tilfellet er det mulig å finne spektralradien analytisk, se. f.eks Hageman & Young [9]. Vi finner følgende uttrykk:

Jacobis metode:

$$\rho = \cosh(\pi h) \approx 1 - \frac{(\pi h)^2}{2} \text{ for små } h \tag{5.96}$$

Gauss-Seidels metode:

$$\rho = \cosh^2(\pi h) \approx 1 - (\pi h)^2 \text{ for små } h \tag{5.97}$$

SOR:

$$\rho = \frac{1 - \sinh(\pi h)}{1 + \sinh(\pi h)} \approx 1 - 2\pi h \text{ for små } h \tag{5.98}$$

Tilfelle (c) er basert på optimal ω.

Vi husker at betingelsen for konvergens er at spektralradien $\rho < 1$. For små h nærmer spektralradien seg raskt 1 for alle metodene, men særlig er dette tydelig for Jacobi- og Gauss-Seidels metode. Tabellen nedenfor viser dette klart.

Spektral radius	$h = \frac{1}{32}$	$h = \frac{1}{64}$	$h = \frac{1}{128}$
ρ_J	0.9952	0.9988	0.9997
ρ_{GS}	0.9904	0.9976	0.9994
ρ_{SOR}	0.8215	0.9065	0.9521

Dette vises enda tydligere når vi beregner antall iterasjoner som behøves for å vinne ett desimalsiffer, se lign. (5.93).

Iterasjoner	$h = \frac{1}{32}$	$h = \frac{1}{64}$	$h = \frac{1}{128}$
k_J	477	1910	7644
k_{GS}	239	955	3822
k_{SOR}	12	23	47

Konklusjon: Det er bare SOR som er praktisk brukbare av disse metodene.

På grunn av denne konklusjonen, har vi bare brukt SOR på eksemplene i dette avsnittet med unntak av et demo-eksempel med Gauss-Seidel. Det er mulig å forbedre SOR betraktelig. Vi kan f.eks. variere ω for hver iterasjon etter bestemte skjema istedenfor å la den være konstant i hele iterasjonsprosessen. Legg også merke til at SOR, med Gauss-Seidel som spesialtilfelle, er avhengig av nummereringen. Vi kan forbedre konvergensen ved f.eks å bruke sjakkbrett-nummerering, dvs: Først gjennomløpe $1, 3, 5, \ldots$ og deretter $2, 4, 6, \ldots$. Den SOR-versjonen vi har brukt kalles punkt-SOR fordi vi går fra punkt til punkt. Her er det forbedringspotensiale ved heller å operere på hele blokker, eventuelt hele linjer. For dem som er interessert i disse variantene, henvises til Press [17] og Hageman & Young [9] der det også finnes programmer. Det er sjelden bryet verdt å bruke disse mer avanserte versjonene fordi det idag finnes mer effektive iterasjons-metoder. Fordelen med den enkle SOR-metoden vi har brukt, er at den er lett å programmere både for lineære- og ikke-lineære ligninger. Legg merke til at når vi brukte direkte-løsere (se 5.2), måtte vi sette opp hele matrisa på forhånd. Dette var forholdsvis enkelt i de viste eksemplene, noe som ofte ikke er tilfelle ellers. Samtidig var vi avhengige av de innebygde løserne i Matlab.

Dersom du ønsker å bruke iterasjonsmetoder på et fint nett, bør du bruke noen av de nyere metodene. Disse går under betegnelsene Flernett-metoder (Multigrid) og Krylov-metoder. En god introduksjon til Flernett-metoder finnes i Briggs [3]. En rekke av Krylov-metodene finnes tilgjengelig i Matlab. Stikkord er her $bicg, cgs, bicgstab, pcg, gmresogqmr$. Skriv f.eks *doc gmres* og se på eksemplene. Istedenfor matrisene som er gitt i eksemplene der, kan du bruke koeffisient-matrisene fra eksemplene i avsnitt 5.2. Se Saad [18] for mer om Krylov-metoder. Mange av disse metodene er også behandlet av Kelley [13] med nedlastbare Matlabprogram.

5.3.3 UTNYTTELSE AV SYMMETRI

Ved å utnytte eventuell symmetri for et problem, kan vi redusere antall ukjente. Dersom vi ser på figur 5.15, har vi symmetri om linja $x = 0.5$.

Formelt får vi da betingelsen $\frac{\partial T}{\partial x} = 0$ langs denne linja, men det er selvfølgelig ikke nødvendig å utføre noe diskretisering da det følger direkte at
$T_{11} = T_{31},\ T_{12} = T_{32}$ osv. Dette fører til at at systemet i (5.14) i dette tilfellet blir:

$$\begin{pmatrix} -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 0 & 1 \\ 2 & 0 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 1 & -4 & 1 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 1 & -4 \end{pmatrix} \cdot \begin{pmatrix} T_{11} \\ T_{12} \\ T_{13} \\ T_{14} \\ T_{15} \\ T_{21} \\ T_{22} \\ T_{23} \\ T_{24} \\ T_{25} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -100 \\ 0 \\ 0 \\ 0 \\ 0 \\ -100 \end{pmatrix} \tag{5.99}$$

Løsningen av (5.99) er gitt i Matlab-programmet **lap1s** nedenfor.

```
% program lap1s
clear
n = 10;
d = ones(n,1); % diagonal
b = zeros(n,1); % right hand side
% --- Update b ---
```

```
b(5) = -100; b(10) = -100;
% --- Generate A-matrix ---
A = spdiags([2*d d -4*d d d],[-5 -1 0 1 5], n,n);
% --- Update A ---
A(5,6) = 0; A(6,5) = 0;
% --- Solve system ---
T = A\b;
```

Poisson-ligning. La oss se på lign. (**??**) for et kvadratisk tverrsnitt av en kanal

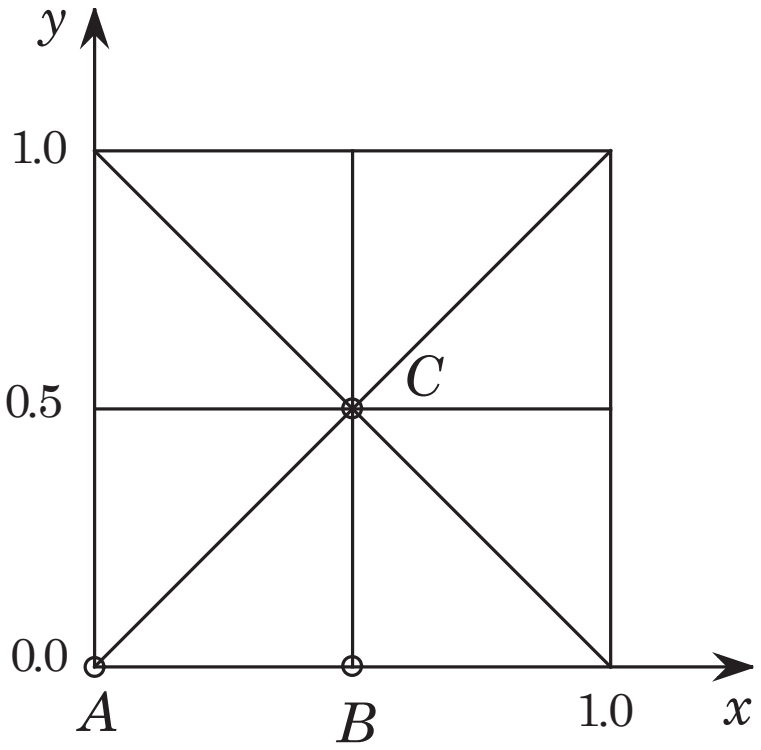


Figure 5.15: Caption goes here.

Dersom vi ikke utnytter symmetrien, blir problemet fra (**??**)

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -1 \tag{5.100}$$

Randbetingelser: $u = 0$ langs hele randkurven.

Dette er et problem med Dirichlet-betingelser. Dersom vi utnytter symmetrien om $x = 0.5$ og $y = 0.5$, blir formelt symmetri-betingelsene:

$$\frac{\partial u}{\partial x} = 0 \text{ for } x = 0.5$$
$$\frac{\partial u}{\partial y} = 0 \text{ for } y = 0.5$$

Vi ser at vi også kan benytte oss av symmetrien om diagonalene, slik at det f.eks er tilstrekkelig å løse ligningen i trekanten ABC da AC er en symmetrilinje. Figur 5.16 på neste side viser trekanten ABC med nummerering der vi har utnyttet symmetrien. Vi setter $u_1 = u_{1,1}$, $u_2 = u_{1,2}$, $u_3 = u_{1,3}$ osv. Langs AB har vi avmerket randverdien $u = 0$.

Den diskretiserte versjonen av (5.100) blir:

$$u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j} = -h^2 \tag{5.101}$$

med $\Delta x = \Delta y = h$

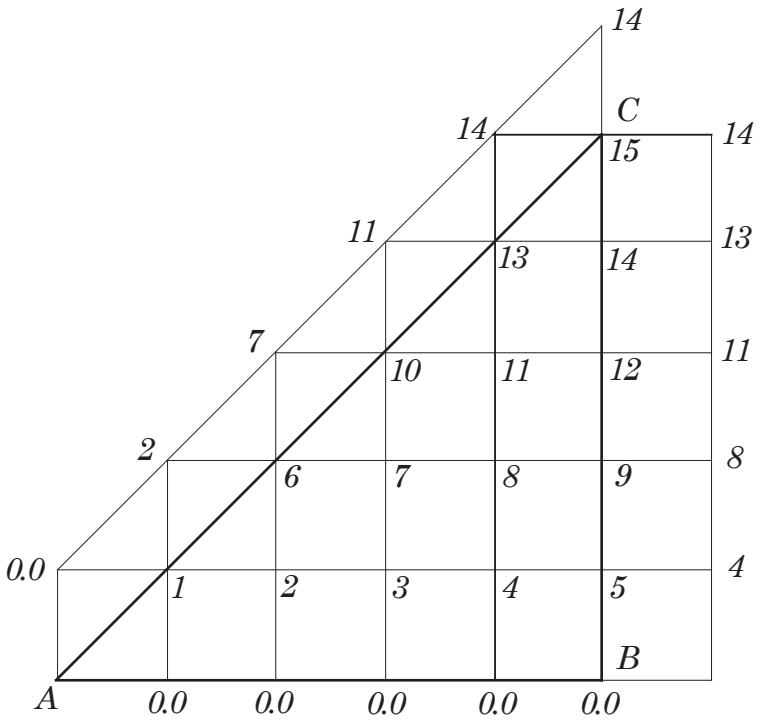


Figure 5.16: Caption goes here.

Vi har valgt $h = 0.1$ i figur 5.16. Beregningsmolekylet med enkle indekser:
slik at (5.101) blir:

$$u_a + u_b + u_c + u_d - 4u_m = -h^2 \tag{5.102}$$

Noen eksempler på bruk av (5.102):

$$\begin{aligned} 2u_2 - 4u_1 &= -h^2 \\ 2u_4 + u_9 - 4u_5 &= -h^2 \\ u_7 + u_4 + u_9 + u_{11} - 4u_8 &= -h^2 \end{aligned}$$

med $m = 1$, 5 og 8. Det endelige ligningsystemet blir som gitt i (5.103):

$$\begin{pmatrix} -4 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & -4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & -4 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 2 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & -4 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 2 & -4 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & -4 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & -4 \end{pmatrix} \cdot \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \\ u_9 \\ u_{10} \\ u_{11} \\ u_{12} \\ u_{13} \\ u_{14} \\ u_{15} \end{pmatrix} = - \begin{pmatrix} h^2 \\ h^2 \\ h^2 \\ h^2 \\ h^2 \\ h^2 \\ h^2 \\ h^2 \\ h^2 \\ h^2 \\ h^2 \\ h^2 \\ h^2 \\ h^2 \\ h^2 \end{pmatrix} \tag{5.103}$$

Selv om bandstrukturen er tydelig, blir den mer rotet fordi vi har utnyttet symmetrien. Det blir mer oppdatering av koeffisientmatrisa. Programmet **poisson** løser (5.103):

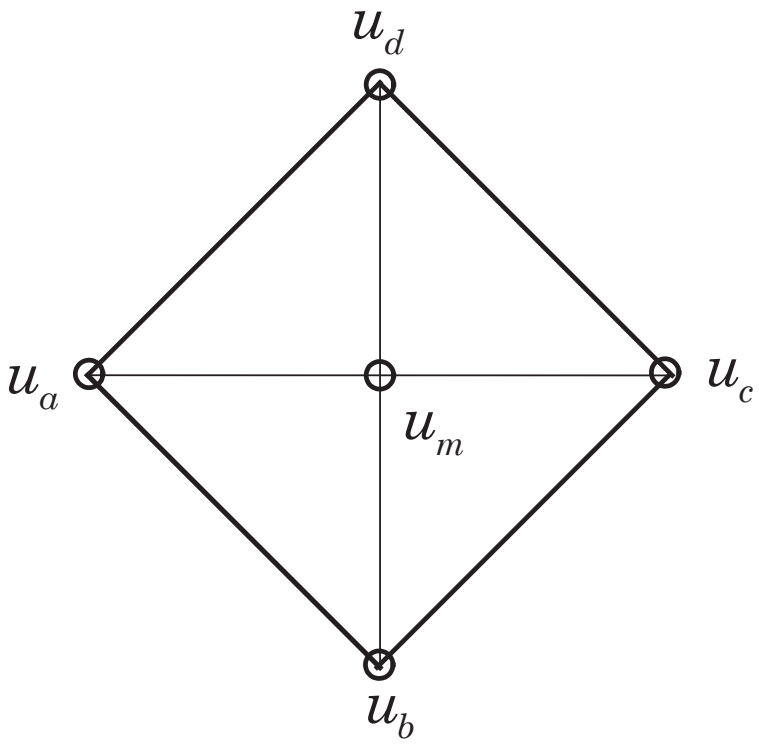


Figure 5.17: Caption goes here.

```
% program poisson
clear
n = 15;
h = 0.1; h2 = h*h;
d0 = zeros(n,1); % diagonal
d = ones(n,1); % diagonal
b = -h2*ones(n,1); % right hand side
% --- generate A-matrix ---
A = spdiags([d0 d0 d0 d -4*d d d0 d0 d0],[-4 -3 -2  -1 0 1 2 3 4], n,n);
% === Update A ===

% --- sub-diagonals ---
A(6,2) = 2; A(7,3) = 1; A(8,4) = 1; A(9,5) = 1;
A(10,7) = 2; A(11,8) = 1; A(12,9) = 1;
A(13,11) = 2; A(14,12) = 1;
A(5,4) = 2; A(6,5) = 0; A(9,8) = 2; A(10,9) = 0; A(12,11) = 2;
A(13,12) = 0; A(14,13) = 2; A(15,14) = 4;
% --- super-diagonals ---
A(1,2) = 2; A(5,6) = 0; A(6,7) = 2; A(9,10) = 0;
A(10,11) = 2; A(12,13) = 0; A(13,14) = 2;
A(11,13) = 1; A(12,14) = 1;
A(7,10 ) = 1; A(8,11) = 1; A(9,12) = 1;
A(2,6) = 1; A(3,7) = 1; A(4,8) = 1; A(5,9) = 1;
% --- solve system ---
u = A\b;
```

Tabellen nedenfor viser de numeriske verdiene fra programmet. De analytiske verdiene er gitt i siste kolonne. Senterverdien u_{15} har en feil på 0.8%. Koordinatene refererer til figur 5.15.

$u_1 = u(0.1, 0.1)$	0.0128	0.0131
$u_2 = u(0.2, 0.1)$	0.0206	0.0209
$u_3 = u(0.3, 0.1)$	0.0254	0.0256
$u_4 = u(0.4, 0.1)$	0.0280	0.0282
$u_5 = u(0.5, 0.1)$	0.0288	0.0290
$u_6 = u(0.2, 0.2)$	0.0343	0.0346
$u_7 = u(0.3, 0.2)$	0.0430	0.0433
$u_8 = u(0.4, 0.2)$	0.0478	0.0482
$u_9 = u(0.5, 0.2)$	0.0493	0.0497
$u_{10} = u(0.3, 0.3)$	0.0544	0.0548
$u_{11} = u(0.4, 0.3)$	0.0608	0.0613
$u_{12} = u(0.5, 0.3)$	0.0629	0.0634
$u_{13} = u(0.4, 0.4)$	0.0682	0.0687
$u_{14} = u(0.5, 0.4)$	0.0706	0.0712
$u_{15} = u(0.5, 0.5)$	0.0731	0.0737

De analytiske verdiene er beregnet fra:

$$u = \zeta \cdot \frac{(1-\zeta)}{2} - \frac{4}{\pi^3} \sum_{n=1,3,5,\ldots}^{\infty} \frac{1}{n^3} \frac{\cosh(n\pi\bar{y})}{\cosh\left(\frac{n\pi}{2}\right)} \cdot \sin(n\pi\zeta) \tag{5.104}$$

$$\text{der } \bar{x} = \left| x - \frac{1}{2} \right|, \ \bar{y} = \left| y - \frac{1}{2} \right|, \zeta = \frac{1}{2} - \bar{x}, \ (x, y) \in [0, 1].$$

$$\text{Bytt om } \bar{x} \text{ og } \bar{y} \text{ dersom } \bar{y} > \bar{x} \tag{5.105}$$

Exercise 1: Symmetric solution

Prove that the analytical solution of the temperature field $T(x, y)$ in (5.15) is symmetric around $x = 0.5$.

Chapter 6

Diffusjonsproblemer

6.1 Differanser. Notasjon

I avsnitt ((2.6)) brukte vi Taylor-utviklingen for en funksjon av en uavhengig variabel til å utlede differanseformler. Tilsvarende kan gjøres for funksjoner av flere uavhengige variable. Dersom uttrykkene ikke har noen kryssderiverte, kan vi direkte bruke formlene som vi utledet i avsnitt (2.6). Vi må bare huske å holde indekset for den andre variable konstant.

Følgende notasjon brukes for de diskrete verdiene i x- og y-retning:

$$x_i = x_0 + i \cdot \Delta x, \ i = 0, 1, 2, \ldots$$
$$y_j = y_0 + j \cdot \Delta y, \ j = 0, 1, 2, \ldots$$

Som tidligere forutsetter vi at Δx og Δy er konstante dersom intet annet blir spesifisert. Figure 6.1 nedenfor gir eksempel på notasjonen i to dimensjoner.

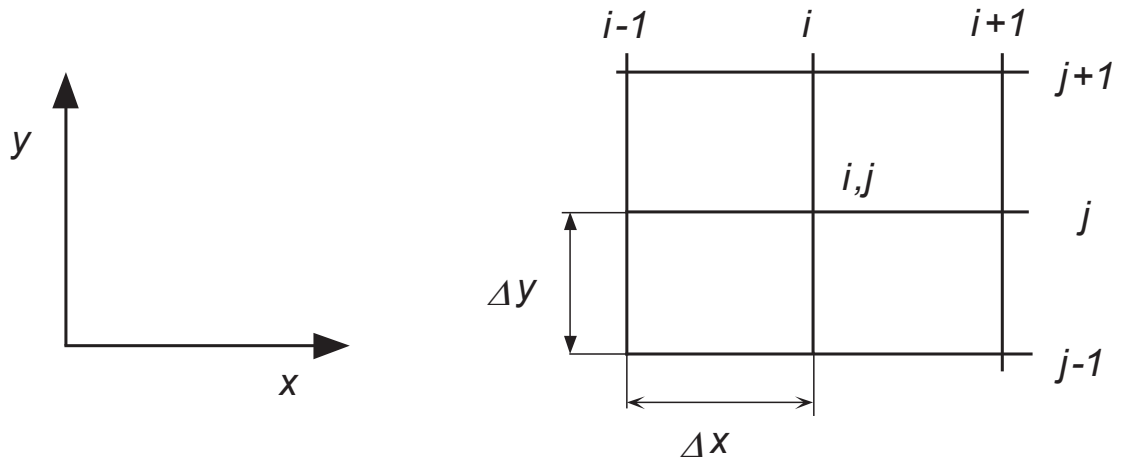


Figure 6.1:

Vi skriver nå opp en rekke formler direkte fra avsnitt ((2.6)).
Foroverdifferanser:

$$\left.\frac{\partial u}{\partial x}\right|_{i,j} = \frac{u_{i+1,j} - u_{i,j}}{\Delta x} + O(\Delta x) \tag{6.1}$$

Bakoverdifferanser:

$$\left.\frac{\partial u}{\partial x}\right|_{i,j} = \frac{u_{i,j} - u_{i-1,j}}{\Delta x} + O(\Delta x) \tag{6.2}$$

Sentraldifferanser

$$\left.\frac{\partial u}{\partial x}\right|_{i,j} = \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} + O\left[(\Delta x)^2\right] \tag{6.3}$$

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{i,j} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2} + O\left[(\Delta x)^2\right] \tag{6.4}$$

Tilsvarende formler for $\frac{\partial u}{\partial y}$ og $\frac{\partial^2 u}{\partial y^2}$ følger direkte:
Foroverdifferanser:

$$\left.\frac{\partial u}{\partial y}\right|_{i,j} = \frac{u_{i,j+1} - u_{i,j}}{\Delta y} \tag{6.5}$$

Bakoverdifferanser:

$$\left.\frac{\partial u}{\partial y}\right|_{i,j} = \frac{u_{i,j} - u_{i,j-1}}{\Delta y} \tag{6.6}$$

Sentraldifferanser

$$\left.\frac{\partial u}{\partial y}\right|_{i,j} = \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta y} \tag{6.7}$$

$$\left.\frac{\partial^2 u}{\partial y^2}\right|_{i,j} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{(\Delta y)^2} \tag{6.8}$$

6.1.1 Example:

Vi diskretiserer Laplace-ligningen $\frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial y^2} = 0$ ved å bruke (6.4)og (6.8) og velger $\Delta x = \Delta y$.
Dette gir følgende differanseligning:

$$u_{i+1,j} + u_{i-1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j} = 0$$

Det resulterende beregningsmolekylet er illustrert i Figure 6.2.
Vi ser av formelen at senterverdien er middelverdien av verdiene i de andre puntene. Denne formelen er velkjent og kalles gjerne 5-punkts-formelen (Brukes i kapittel (5)).

Ved render er vi ofte nødt til å bruke bakover eller foroverdifferanser. Vi gir her noen formler med 2. ordens nøyaktighet.
Foroverdifferanser:

$$\left.\frac{\partial u}{\partial x}\right|_{i,j} = \frac{-3u_{i,j} + 4u_{i+1,j} - u_{i+2,j}}{2\Delta x} \tag{6.9}$$

Bakoverdifferanser:

$$\left.\frac{\partial u}{\partial x}\right|_{i,j} = \frac{3u_{i,j} - 4u_{i-1,j} + u_{i-2,j}}{2\Delta x} \tag{6.10}$$

Formlene for (2.6) , (2.6) og (2.6) differanser gitt i kapittel (2) kan brukes nå også bare vi utstyrer dem med to indekser. En fyldig samling av differanseformler finnes i Anderson [1]. Detaljerte utledninger er gitt i Hirsch [11].

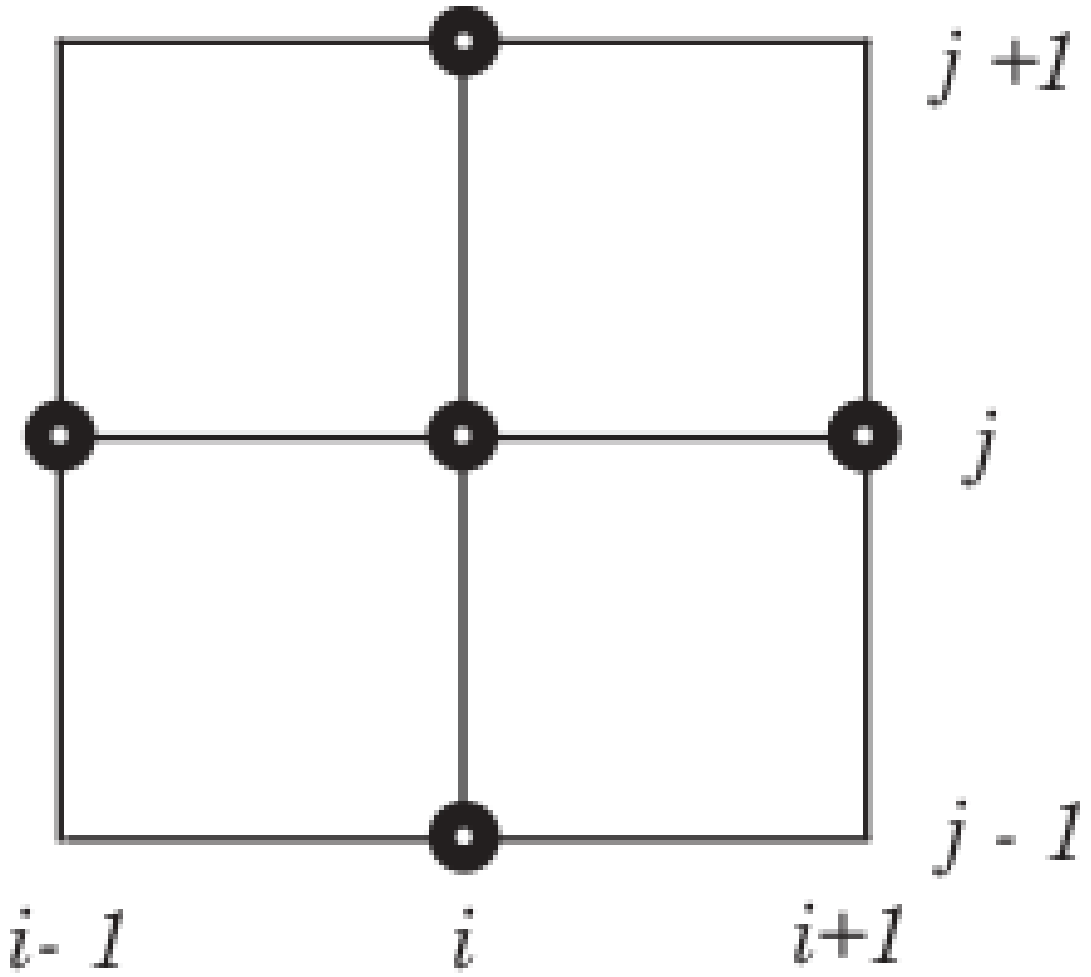


Figure 6.2: 5-punkts beregningsmolekyl for diffusjonsligningen

6.2 Diffusjonsligningen

6.2.1 Introduction

Den endimensjonale diffusjonsligningen er gitt ved:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} \tag{6.11}$$

t kalles som tidligere nevnt, den evolusjonsvariable og kan være både tid-og romkoordinat.
Typiske diffusjonsproblemer:
1) Varmeledning:

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2}$$

2) Ikke-stasjonært grensesjikt: (Stokes problem)

$$\frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial y^2}$$

3) Strømning i porøse media:

$$\frac{\partial u}{\partial t} = c \frac{\partial^2 u}{\partial x^2}$$

4) Linearisert grensesjiktligning:

$$\frac{\partial u}{\partial x} = \frac{\nu}{U_0} \frac{\partial^2 u}{\partial y^2}$$

(Her er x den evolusjonsvariable.)
Sammenligner (6.11) med klassifikasjonsligningen (6.12): **Marie 11**: Sikkert greit å henvise til kapittel 4, seksjon 3 i kompendiet

$$A\frac{\partial^2\phi}{\partial x^2} + B\frac{\partial^2\phi}{\partial x\partial y} + C\frac{\partial^2\phi}{\partial y^2} + f = 0 \tag{6.12}$$

$$A\cdot(dy)^2 - B\cdot dy\cdot dx + C\cdot(dx)^2 = 0 \tag{6.13}$$

$$\lambda_{1,2} = \frac{B \pm \sqrt{B^2 - 4AC}}{2A} \tag{6.14}$$

$B = C = 0,\ A = 1$ som innsatt i (6.13) og (6.14) gir:

$$dt = 0,\ B^2 - 4AC = 0$$

(6.11) er følgelig parabolsk, og karakteristikken er gitt ved $t =$ konstant.
Dersom vi dividerer dt med dx:

$$\frac{dt}{dx} = 0 \rightarrow \frac{dx}{dt} = \infty \tag{6.15}$$

Derav: Uendelig signalforplantningshastighet langs den karakteristiske kurva $t =$ konstant.
Marie 12: Mer henvisning til kapittel 4.

6.2.2 Ikke-stasjonær couette strømning

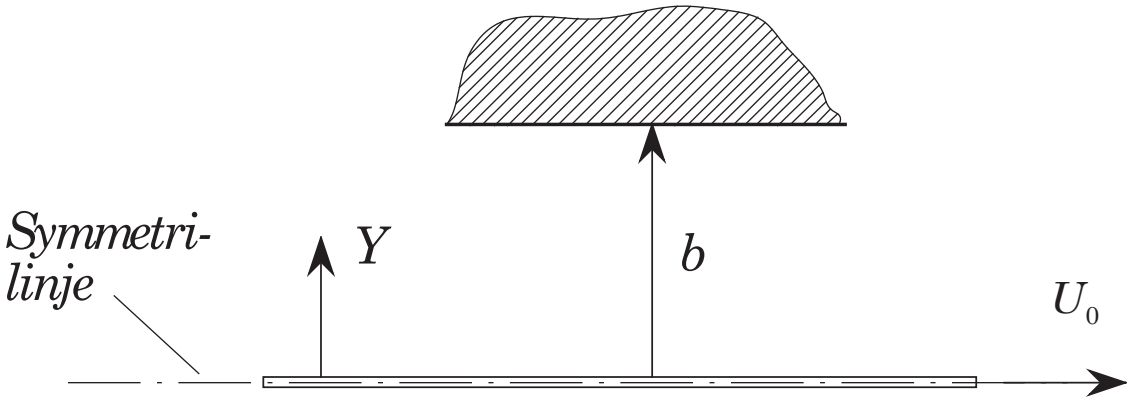


Figure 6.3:

Den klassiske versjonen av dette problemet med $b = \infty$ har vi tidligere behandlet i avsnitt (3.3) som Stokes 1. problem.
Ligning:

$$\frac{\partial U}{\partial \tau} = \nu\frac{\partial^2 U}{\partial Y^2},\ 0 < Y < b \tag{6.16}$$

Initialbetingelse:

$$U(Y,\ \tau) = 0,\ \tau < 0 \tag{6.17}$$

Randbetingelser:

$$\left.\begin{array}{l} U(0,\ \tau) = \quad U_0 \\ U(b,\ \tau) = \quad 0 \end{array}\right\} = \tau \geq 0 \tag{6.18}$$

I avsnitt (3.3) har vi sett på flere måter å gjøre (6.16)) dimensjonsløs.
Innfører følgende dimensjonsløse variable:

$$y = \frac{Y}{b},\ u = \frac{U}{U_0},\ t = \frac{\tau\nu}{b^2} \tag{6.19}$$

(6.16) kan da skrives:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial y^2},\ 0 < y < 1 \tag{6.20}$$

Initialbetingelse:

$$u(y,\ t) = 0,\ t < 0 \tag{6.21}$$

Randbetingelser:

$$\left.\begin{array}{l} u(0,\ t) = \quad 1 \\ u(1,\ t) = \quad 0 \end{array}\right\},\ t \geq 0 \tag{6.22}$$

Som i avsnitt (3.3), kan dette eksemplet også formuleres som et varmeledningsproblem.
Analytisk løsning:

$$u(y,t) = 1 - y - \frac{2}{\pi} \cdot \sum_{n=1}^{\infty} \frac{1}{n} \exp[-(n\pi)^2 t] \sin(n\pi y) \tag{6.23}$$

Utledningen av (6.23) er gitt i detalj i appendiks G.6 i kompendiet.
Vi diskretiserer (6.20) ved å benytte foroverdifferanser for tiden t og sentraldifferanser for romkoordinaten y:

$$\left.\frac{\partial u}{\partial t}\right|_j^n \approx \frac{u_j^{n+1} - u_j^n}{\Delta t} \tag{6.24}$$

$$\frac{\partial^2 u}{\partial y^2} \approx \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta y)^2} \tag{6.25}$$

$$\text{der } t_n = n \cdot \Delta t,\ n = 0, 1, 2, \ldots,\ y_j = j \cdot \Delta y,\ j = 0, 1, 2, \ldots$$

(6.24) innsatt i (6.20) gir følgende differanseligning:

$$u_j^{n+1} = r(u_{j+1}^n + u_{j-1}^n) + (1 - 2r)u_j^n \tag{6.26}$$

$$\text{der } r = \frac{\Delta t}{(\Delta y)^2} = \nu \frac{\Delta \tau}{(\Delta Y)^2} \tag{6.27}$$

r er en dimensjonsløs gruppe. I varmeledning brukes betegnelsen det numeriske Fourier-tallet om denne gruppa.(Lign. (6.26) er behandlet i avsnitt 13.1 i C&K [4])). I avsnitt (6.5) viser vi at skjemaet i (6.26)) er av 2. ordens nøyaktighet i t og 4. ordens nøyaktighet i y dersom vi bruker $r = 1/6$. I strømningsmekanikken er det vanlig å skrive u_j^n istedenfor $u_{j,n}$ slik at indekset for den evolusjonsvariable skrives som et øvre indeks. Vi adopterer denne skrivemåten generelt.

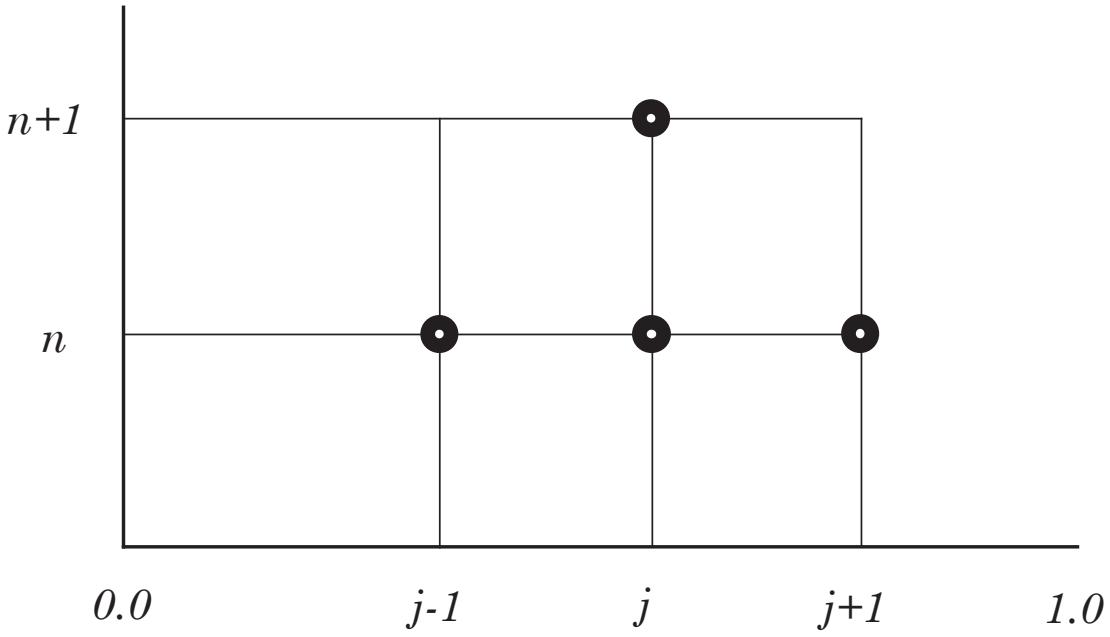


Figure 6.4: FTCS-skjemaet

Et skjema der vi bruker foroverdifferanser for den tidslignende variable og sentraldifferanser for romkoordinaten, betegnes gjerne på engelsk som et FTCS-skjema (Forward Time Central Space). Når vi bruker betegnelsen FTCS , menes 1. ordens nøyaktighet i t og 2. ordens nøyaktighet i romkoordinaten(e). Det er også vanlig å kalle dette skjemaet for Euler-skjemaet.

Figure 6.4 viser at skjemaet er eksplisitt; verdien ved tiden $n+1$ kan finnes direkte av formelen uten å løse et ligningsystem. Velger $\Delta y = 0.1$ og prøver med $r = 0.3$ og $r = 0.6$

Tilfelle A, r = 0.3

Fra (6.26) får vi:

$$\Delta t = (\Delta y)^2 \cdot r = 3.0 \cdot 10^{-3}, \text{ som gir:}$$
$$t_n = n \cdot 3.0 \cdot 10^{-3}, \ n = 0, 1, 2, \ldots$$

Tabell 5.1

y	$t_n = 0.03$	% ε	$t_n = 0.06$	% ε	$t_n = 0.12$	% ε	$t_n = 0.45$	% ε
0.0	1.0000	0.0	1.0000	0.0	1.0000	0.0	1.0000	0.00
0.1	0.6917	1.3	0.7761	0.4	0.8394	0.1	0.8978	0.01
0.2	0.4266	3.0	0.5692	1.0	0.6851	0.3	0.7958	0.02
0.3	0.2310	4.7	0.3927	1.6	0.5427	0.5	0.6942	0.03
0.4	0.1080	5.5	0.2537	2.2	0.4163	0.8	0.5931	0.04
0.5	0.0428	3.8	0.1528	2.6	0.3084	1.0	0.4928	0.06
0.6	0.0140	-2.3	0.0854	2.7	0.2191	1.3	0.3931	0.07
0.7	0.0036	-15.2	0.0440	2.1	0.1472	1.5	0.2942	0.08
0.8	0.0007	-35.7	0.0206	0.9	0.0896	1.6	0.1958	0.08
0.9	0.0001	-60.9	0.0078	-0.6	0.0422	1.7	0.0978	0.09
1.0	0.0000	0.0	0.0000	0.0	0.0000	0.0	0.0000	0.00

% er prosent relativ feil $= \left(\frac{u_{num} - u_{analytisk}}{u_{analytisk}} \right) \cdot 100$ der $u_{analytisk}$ er gitt i (6.23).

For $t_n = 0.03$ ser vi at Δy er for stor til å gi tilstrekkelig oppløsning. Med $t_n = 0.45$ har vi praktisk talt fått den stasjonære løsningen $u_s = 1 - y$.

For å få en bedre oppløsning for tilfellet $t_n = 0.03$, utfører vi en beregning med $\Delta y = 0.05$ og en med $\Delta y = 0.01$. For $\Delta y = 0.05$ får vi 40 tidskritt og for $\Delta y = 0.01$ får vi 1000.

Table 5.2: Resultater for $r = 0.3$. Venstre side av tabellen er for $\Delta y = 0.05$ og høyre side for $\Delta y = 0.01$.

y	$t_n = 0.03$	% ε	$t_n = 0.06$	% ε
0.0	1.0000	0.00	1.0000	0.00
0.1	0.6853	0.32	0.6832	0.01
0.2	0.4173	0.75	0.4143	0.03
0.3	0.2232	1.17	0.2208	0.05
0.4	0.1038	1.33	0.1025	0.05
0.5	0.0416	0.92	0.0412	0.04
0.6	0.0142	-0.5	0.0143	-0.02
0.7	0.0041	-3.44	0.0043	-0.13
0.8	0.0010	-8.39	0.0011	-0.33
0.9	0.0002	-15.2	0.0002	-0.61
1.0	0.0000	0.00	0.0000	0.00

Tilfelle B, r = 0.6

Med $\Delta y = 0.1$ får vi nå $t_n = n \cdot 6.0 \cdot 10^{-3}, \ n = 0, 1, 2, \ldots$. Resultatet av denne beregningen er vist i tabellen under:

Tabell 5.3: Resultater for r = 0.6.

y	$t_n = 0.03$	% ε	$t_n = 0.06$	% ε
0.0	1.0000	0.0	1.0000	0.0
0.1	0.7939	16.2	0.5797	-25.0
0.2	0.2995	-27.7	0.9186	63.0
0.3	0.3715	68.4	0.0027	-99.3
0.4	0.0259	-74.7	0.6239	151.4
0.5	0.0778	88.6	-0.1241	-183.3
0.6	0.0000		0.2663	220.0
0.7	0.0000		-0.0551	-227.7
0.8	0.0000		0.0625	206.4
0.9	0.0000		-0.0081	-202.3
1.0	0.0000		0.0000	0.0

Tabellen indikerer tydelig en instabilitet. Ser vi på siste leddet i (6.26), finner vi at alle leddene er positive for $2r \leq 1$, dvs. $r \leq \frac{1}{2}$, som er en tilstrekkelig betingelse for å hindre oscillasjoner. I Figure 6.5 har vi tegnet opp løsningen fra tabell (6.2.2) og i Figure 6.6 løsningen fra tabell (6.2.2. Figure 6.7 og (6.8 viser løsningen for $r = 0.514$ med Δy lik henholdsvis 0.1 og 0.05.

Betingelsen $r \leq \frac{1}{2}$ forhindrer altså økende oscillasjoner i differanseligningen for den gitte initialbetingelsen. Vi skal nå vise at vi kan utvide dette resultatet slik at det også kan brukes på andre differanseligninger.

Resultater FTCS for ulike verdier av r.

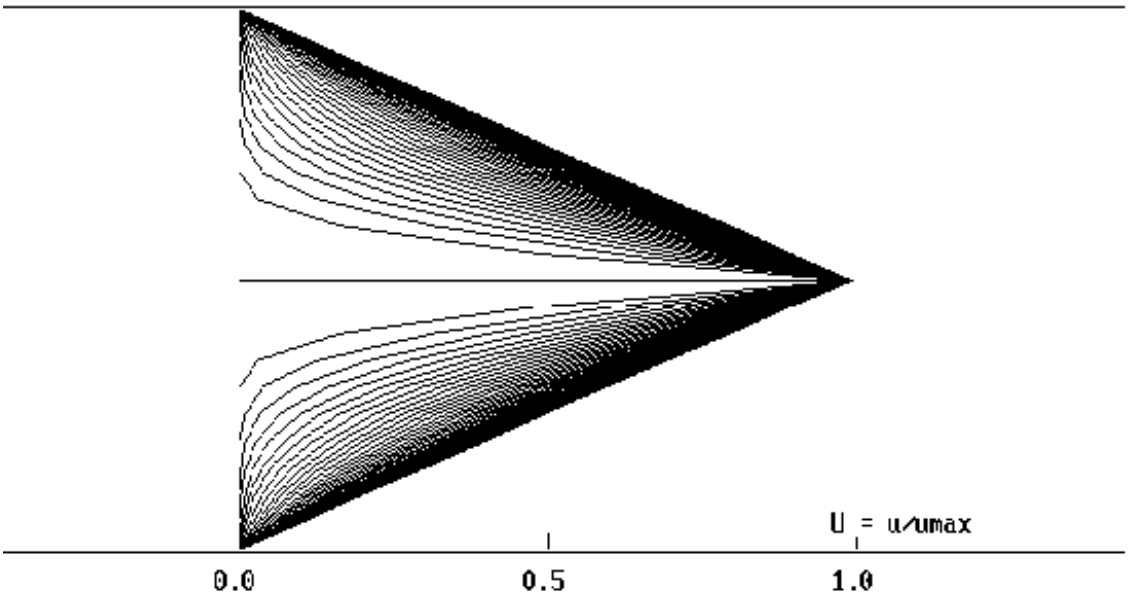


Figure 6.5: $r = 0.3$ og $\Delta y = 0.1$.

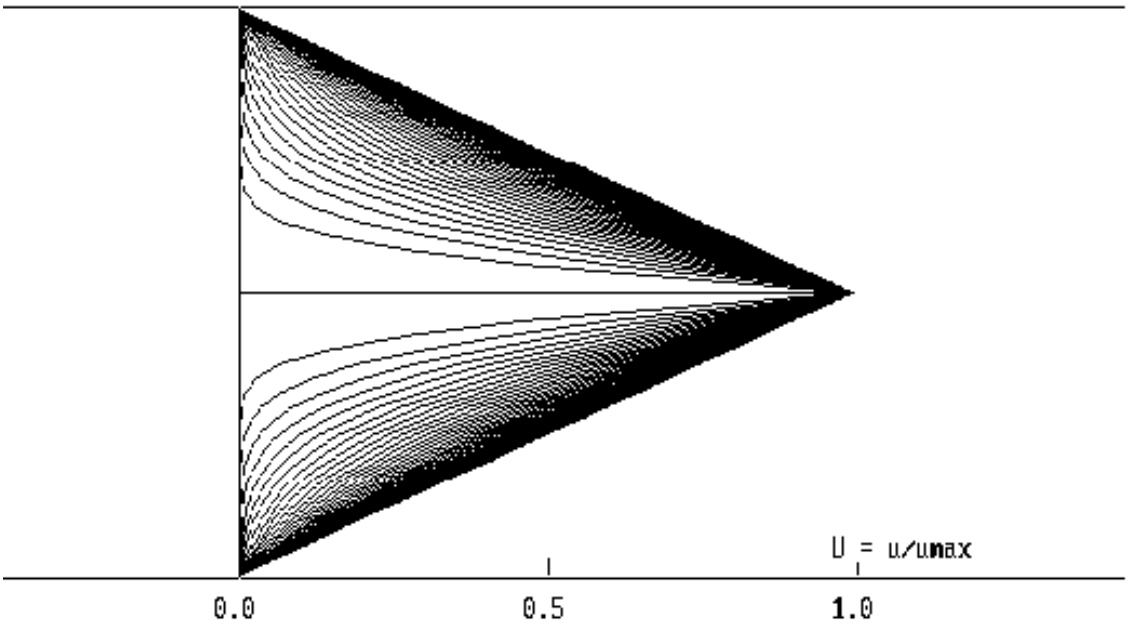


Figure 6.6: $r = 0.3$ og $\Delta y = 0.05$.

6.2.3 PK-kriteriet: Kriteriet om positive koeffisienter

La $s = a_1x_1 + a_2x_2 + \cdots + a_kx_k$ være en sum av k ledd der tallene $a_1, a_2, \ldots, a_k$ er positive.
Setter: $x_{min} = \min(x_1, x_2, \ldots, x_k)$ og $x_{max} = \max(x_1, x_2, \ldots, x_k)$
Derav:

$$x_{min} \cdot (a_1 + a_2 + \cdots + a_k) \leq s \leq x_{max} \cdot (a_1 + a_2 + \cdots + a_k) \tag{6.28}$$

Merk at dette bare gjelder dersom $a_1, a_2, \ldots, a_k$ er positive. La oss se på to tilfeller:

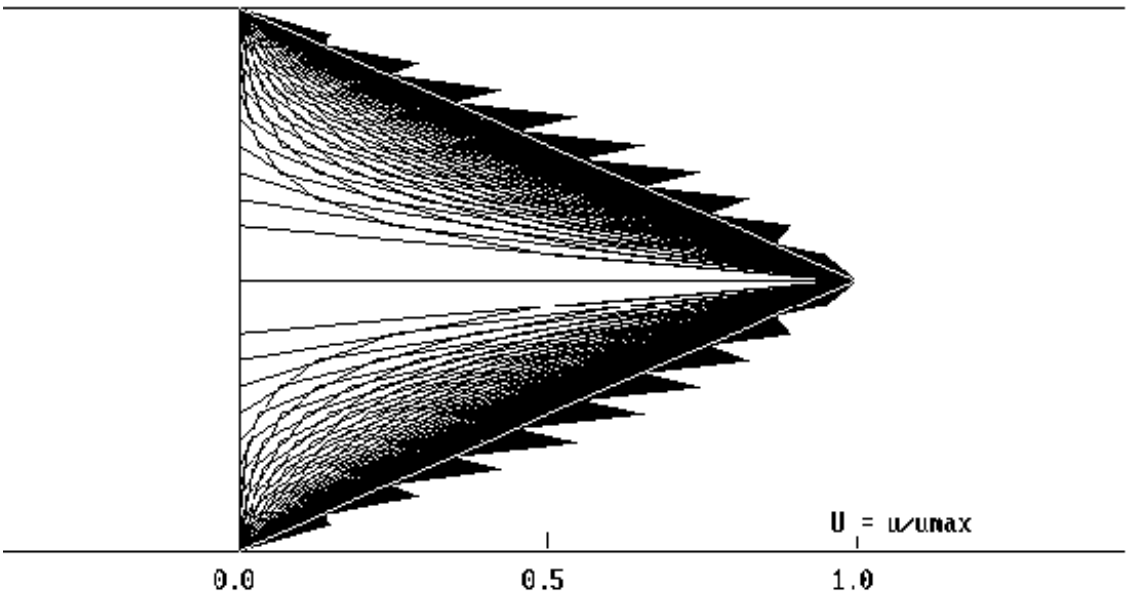


Figure 6.7: $r = 0.541$ og $\Delta y = 0.1$.

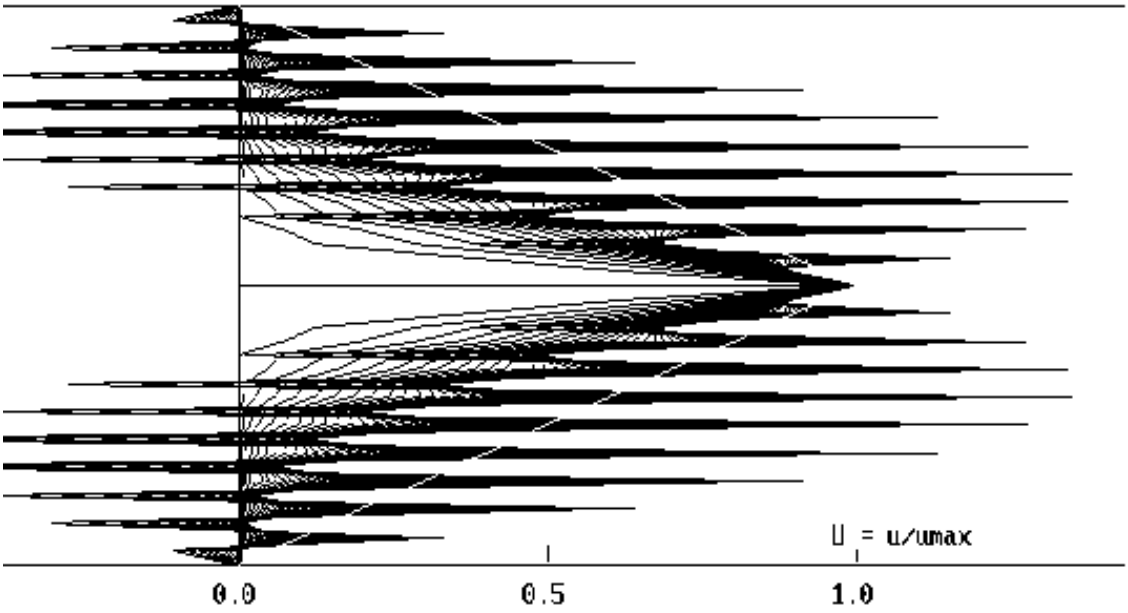


Figure 6.8: $r = 0541$ og $\Delta y = 0.05$.

Tilfelle 1: $a_1 + a_2 + \cdots + a_k = 1$
Innsatt i (6.28):

$$x_{min} \leq s \leq x_{max} \tag{6.29}$$

Likhetstegnene i (6.29) gjelder for $x_1 = x_2 = \cdots = x_k$.
La oss nå anvende (6.29) på differanseligningen i (6.26):

$$u_j^{n+1} = r(u_{j+1}^n + u_{j-1}^n) + (1 - 2r)u_j^n$$

Her har vi $a_1 = r,\ a_2 = r,\ a_3 = 1 - 2r$ slik at $a_1 + a_2 + a_3 = 1$.
(6.29) gir da:

$$\min(u_{j+1}^n, u_j^n, u_{j-1}^n) \leq u_j^{n+1} \leq \max(u_{j+1}^n, u_j^n, u_{j-1}^n)$$

Betingelsen forat dette gjelder, er at alle koeffisientene $a_1, a_2 \ldots, a_k$ er positive. Da $r > 0$, betyr dette at bare $a_3 = 1 - 2r$ kan bli negativ. Betingelsen for at a_3 skal være positiv, blir da: $1 - 2r > 0$ som gir $r < \frac{1}{2}$. Når $r = \frac{1}{2}$, forsvinner a_3, slik at $a_1 + a_2 = 1$ og betingelsen også er oppfylt med $r = \frac{1}{2}$. Dette tilfellet kalles Bender - Schmidt formelen.

Det betyr at dersom vi velger $r \leq \frac{1}{2}$, vil u-verdiene ved tidskrittet $n + 1$ ligge mellom den største og den minste u-verdien ved tidskritt n. Dersom skjemaet er eksplisitt og homogent og $u = u_0$ er konstant er en løsning, vil summen av koeffisientene ofte være lik 1. (Sett f.eks $u = u_0$ inn i (6.26)). Dette skyldes formen differanseutrykkene får fra Taylor-utviklingen. (Se (2.6) , (2.6) og (2.6) differanser gitt i kapittel (2)).

Tilfelle 2: $a_1 + a_2 + \cdots + a_k < 1$
Definerer $b = 1 - (a_1 + a_2 + \cdots + a_k) > 0$ slik at $a_1 + a_2 + \cdots + a_k + b = 1$
Kan da skrive følgende sum: $a = a_1 x_1 + a_2 x_2 + \cdots + a_k + b \cdot 0$ Vi ser da at disse utrykkene oppfyller alle betingelsene for tilfelle 1 slik at vi får:

$$\min(0, x_1, x_2, \ldots, x_k) \leq s \leq \max(0, x_1, x_2, \ldots, x_k) \tag{6.30}$$

Til forskjell fra (6.29) har vi nå fått inn tallet 0 ; ellers som før. La oss se på et eksempel:

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2} + bT,\ b = \text{konstant},\ t < t_{maks}$$

Dersom vi bruker FTCS-skjemaet, får vi:

$$T_j^{n+1} = r(T_{j+1}^n + T_{j-1}^n) + (1 - 2r + \Delta t \cdot b) T_j^n,\ r = \alpha \frac{\Delta t}{(\Delta x)^2}$$

Med $a_1 = a_2 = r$ og $a_3 = 1 - 2r + \Delta t \cdot b$ får vi:

$$a_1 + a_2 + a_3 = 1 + \Delta t \cdot b \leq 1 \text{ bare med negativ verdi av } b$$

Kravet om positive koeffisienter blir nå:

$$1 - 2r + \Delta t \cdot b > 0 \text{ som blir: } 0 < r < \frac{1}{2} + \frac{\Delta t \cdot b}{2} \text{ der } b \text{ er negativ}$$

Kriteriet sier da at T-verdiene fra differanseligningen ikke øker når b er negativ. Dette stemmer med fysikken, da en negativ b betyr at leddet bT kan oppfattes som et varmesluk.
(6.29) og (6.30) er egentlig ikke stabilitetskriterium. Setningene gir en *tilstrekkelig* betingelse for å hindre at voksende oscillasjoner oppstår, noe som kan være adskillig mer restriktivt enn det som er nødvendig for stabilitet. Men i mange tilfeller er vi tilfreds med en slik betingelse. PK-kriteriet er mye anvendt fordi den ofte kan brukes på differanseligninger der en mer eksakt analyse er vanskelig å gjennomføre. Legg merke til at det bare kan brukes for eksplisitte skjema dersom vi ikke har ekstra opplysninger. For parabolske ligninger har vi gjerne slike ekstra opplysninger i form av det vi kaller maks.-min. prinsipp.(Se avsnitt (6.4.3)). Kriteriet må også modifiseres når vi har problemer med økende amplitude.
Det beste ville selvfølgelig være å ha et kriterium som gir en nødvendig og tilstrekkelig betingelse for numerisk stabilitet. For generelle differanseligninger eksisterer det ikke noe slikt kriterium, noe som neppe er overraskende. Et kriterium som ofte gir nødvendige og i noen tilfeller også tilstrekkelige betingelser for stabilitet, er von Neumanns metode. Denne metoden består i en Fourier-analyse av den lineariserte differanseligningen og kan også brukes for implisitte skjema. Vi skal se på denne metoden i det neste avsnittet.

6.3 Stabilitetsanalyse med von Neumanns medtode

Vi har differensialligningen:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \tag{6.31}$$

La oss repetere hvordan vi kan løse denne ved bruk av separasjon av variable. Antar at vi kan skrive $u(x,t)$ som produkt av to funksjoner $f(t)$ og $g(x)$:

$$u(x,t) = f(t) \cdot g(x)$$

Ved derivasjon: $\frac{\partial u}{\partial t} = \frac{df(t)}{dt} \cdot g(x),\ \frac{\partial^2 u}{\partial x^2} = f(t) \cdot \frac{d^2 g(x)}{dx^2} \cdot \frac{1}{g(x)}$ som innsatt ovenfor gir:

$$\frac{df(t)}{dt} \cdot g(x) = f(t) \cdot \frac{d^2 g(x)}{dx^2} \rightarrow \frac{df(t)}{dt} \cdot \frac{1}{f(t)} = \frac{d^2 g(x)}{dx^2} \cdot \frac{1}{g(x)}$$

Den ene siden av ligningen er bare funksjon av t, mens den andre bare er funksjon av x. Da de to sidene skal være like for vilkårlige verdier av t og x, er den eneste muligheten at hver av sidene er lik en konstant. Denne konstanten kalles separasjonskonstanten og vi velger den lik $-\beta^2$:

$$\frac{df(t)}{dt} \cdot \frac{1}{f(t)} = -\beta^2 \rightarrow \frac{df(t)}{dt} + \beta^2 \cdot f(t) = 0$$
$$\frac{d^2g(x)}{dx^2} \cdot \frac{1}{g(x)} = -\beta^2 \rightarrow \frac{d^2g(x)}{dx^2} + \beta^2 \cdot g(x) = 0$$

Den første ligningen har løsning $f(t) = e^{-\beta^2 t}$ og den andre

$$g(x) = A\sin(\beta x) + B\cos(\beta x) \text{ slik at vi totalt får:}$$
$$u(x,t) = e^{-\beta^2 t} \cdot [A\sin(\beta x) + B\cos(\beta x)]$$

Dette er en partikulær løsning, og da ligningen er lineær, vet vi at summen av slike løsninger også er en løsning:

$$u(x,t) = \sum_{m=0}^{m=\infty} e^{-\beta_m^2 t} \cdot [A_m \sin(\beta_m x) + B_m \cos(\beta_m x)]$$

I et gitt tilfelle bestemmes A_m, B_m og β_m av startbetingelsene og randbetingelsene. Dette er vist i detalj i appendiks G.6 i kompendiet.

Vi har ikke bruk for den generelle løsningen, men greier oss med to partikulære løsninger:

$$u(x,t) = \begin{cases} e^{-\beta^2 t}\sin(\beta x) \\ e^{-\beta^2 t}\cos(\beta x) \end{cases} \tag{6.32}$$

(6.32) kan skrives mer kompakt ved å bruke Eulers formel:

$$e^{i\cdot x} = \cos(x) + i \cdot \sin(x),\ i = \sqrt{-1} \tag{6.33}$$

Følgende uttrykk er da en løsning av (6.31):

$$u(x,t) = e^{-\beta^2 t} \cdot e^{i\cdot\beta x} = e^{-\beta^2 t + i\cdot\beta x} \tag{6.34}$$

Merk. Egentlig skal vi ta realdelen eller imaginærdelen av høyre side av (6.34) da u er reell. Men da både realdelen og imaginærdelen tilfredstiller (6.31), tillater vi oss nå og senere litt juks i notasjonen.

Med notasjonen $x_j = j \cdot \Delta x,\ j = 0, 1, 2, \ldots$ og $t_n = n \cdot \Delta t,\ n = 0, 1, 2, \ldots$ får vi fra (6.34):

$$u(x_j, t_n) = e^{-\beta^2 t_n} \cdot e^{i\cdot\beta x_j} = e^{-\beta^2 \cdot n \cdot \Delta t} \cdot e^{i\beta x_j} = (e^{-\beta^2 \cdot \Delta t})^n \cdot e^{i\beta x_j} \tag{6.35}$$

$$u(x_j, t_{n+1}) = e^{-\beta^2 t_{n+1}} \cdot e^{i\cdot\beta x_j} = e^{-\beta^2 \cdot (n+1) \cdot \Delta t} \cdot e^{i\beta x_j} = (e^{-\beta^2 \cdot \Delta t})^{n+1} \cdot e^{i\beta x_j} \tag{6.36}$$

Vi får amplitudeforholdet ved å dividere de to uttrykkene på hverandre:

$$G_a = \frac{u(x_j, t_{n+1})}{u(x_j, t_n)} = e^{-\beta^2 \Delta t} \tag{6.37}$$

G_a kalles ofte den *analytiske* forsterkningsfaktoren. (Se avsnitt 1.6.1 i kompendiet) I dette tilfellet ser vi at $G_a < 1$.
(6.35) kan nå skrives:

$$u(x_j, t_n) = (G_a)^n \cdot e^{i\cdot\beta x_j} \equiv G_a^n \cdot e^{i\cdot\beta x_j} \tag{6.38}$$

Da $G_a < 1$, vil $G_a^n \rightarrow 0$ for $n \rightarrow \infty$ for dette problemet.
Fra (6.26) har vi følgende differanseligning for løsning av (6.31):

$$u_j^{n+1} = r(u_{j+1}^n + u_{j-1}^n) + (1-2r)u_j^n,\ r = \frac{\Delta t}{(\Delta x)^2} \tag{6.39}$$

For endelig verdier av Δx og Δt vet vi at løsningen av differanseligningen nødvendigvis må avvike fra den analytiske løsningen av differensialligningen.Avviket vil selvfølgelig bli større for økende verdier av Δx og Δt og vi har sett at når forholdet $r > \frac{1}{2}$, blir differanseligningen ustabil: Stadig økende amplituder med vekslende fortegn. Da (6.38) er en løsning av differensialligningen, forsøker vi et tilsvarende uttrykk for differanseligningen (6.39):

$$u_j^n \rightarrow E_j^n = G^n \cdot e^{i\cdot\beta x_j} \tag{6.40}$$

Her er den *numeriske* forsterkningsfaktoren G gitt ved:

$$G = \frac{E_j^{n+1}}{E_j^n} \tag{6.41}$$

som gir:

$$E_j^n = G^n \cdot E_j^0,\ E_j^0 = e^{i\cdot\beta\cdot x_j} \tag{6.42}$$

der G er en funksjon av Δt og β og kan være kompleks. Den numeriske forsterkningsfaktoren G vil selvfølgelig være forskjellig fra den analytiske faktoren G_a. Merk: $G^n = G$ i n'te potens.

Dersom amplitudeforholdet skal være begrenset, må G tilfredstille følgende betingelse:

$$|G| \leq 1 \tag{6.43}$$

(6.43) kalles von Neumanns strenge stabilitetskriterium. Det kalles det strenge kriteriet fordi det ikke tillater at amplitudeforholdet øker. Vi skal fjerne denne restriksjonen i avsnitt (6.4.4). Selv om vi har brukt den enkle diffusjonsligningen som eksempel, gjelder fremgangsmåtem også i mer generelle tilfeller: Vi setter (6.40) inn i den aktuelle differanseligningen og forlanger at (6.43) skal være oppfylt.

Noen egenskaper for kriteriet:

1. Den lineære differanseligningen må ha konstante koeffisienter. Ved variable koeffisienter, kan kriteriet brukes ved å "fryse" koeffisientene lokalt. Praksis viser at kriteriet da gir en nødvendig betingelse for stabilitet.
2. Kriteriet tar ikke hensyn til randverdier da det er utledet for periodiske initialdata. Dersom vi ønsker å undersøke randverdienes innvirkning på stabiliteten, må koeffisientmatrisa for ligningsystemet undersøkes. Eventuelt en mer generell teori.
3. Kriteriet tar ikke hensyn til randverdier da det er utledet for periodiske initialdata. Dersom vi ønsker å undersøke randverdienes innvirkning på stabiliteten, må koeffisientmatrisa for ligningsystemet undersøkes. Eventuelt en mer generell teori.

6.3.1 Bruk av von Neumann kriteriet

Vi vil nå analysere det eksplisitte skjemaet i (6.26) ved bruk av von Neumanns metode. Skriver differanseligningen på følgende form:

$$E_j^{n+1} = r(E_{j+1}^n + E_{j-1}^n) + (1-2r)E_j^n \tag{6.44}$$

Fra (6.40): $E_j^n = G^n \cdot e^{i\cdot\beta\cdot y_j}$ (G^n betyr G i n-te potens) som innsatt i (6.44) gir:

$$G^{n+1}e^{i\cdot\beta y_j} = r\cdot(G^n\cdot e^{i\cdot\beta y_{j+1}} + G^n\cdot e^{i\cdot\beta y_{j-1}}) + (1-2r)G^n e^{i\cdot\beta y_j}$$

Dividerer med $G^n \cdot e^{i\cdot\beta y_j}$:

$$G = r(e^{i\cdot\beta h} + e^{-i\cdot\beta h}) + (1-2r) = r(e^{i\delta} + e^{-i\delta}) + (1-2r) \tag{6.45}$$

$$\text{der } \delta = \beta h \tag{6.46}$$

Merk. Dersom vi bruker terminologien for periodiske funksjoner, er β et bølgetall (vinkelfrekvens) , og δ en fasevinkel. (Se appendiks A.3 i kompendiet)

Noen standardformler som brukes ofte:

$$\begin{aligned} 2\cos(x) &= e^{ix} + e^{-ix} \\ i\cdot 2\sin(x) &= e^{ix} - e^{-ix} \\ \cos(x) &= 1 - 2\sin^2(\tfrac{x}{2}) \end{aligned} \tag{6.47}$$

(6.47) brukt i (6.45) gir:

$$G = 1 - 2r(1-\cos(\delta)) = 1 - 4r\sin^2\left(\frac{\delta}{2}\right) \tag{6.48}$$

Betingelsen $|G| \leq 1$ betyr, da G er reell:

$$-1 \leq G \leq 1 \text{ eller } -1 \leq 1 - 4r\sin^2\left(\frac{\delta}{2}\right) \leq 1$$

Høyre side er tilfredstilt med $r \geq 0$.

Venstre side:

$$r \leq \frac{1}{2\sin^2(\frac{\delta}{2})}$$

som er oppfylt for alle δ med $r \leq \frac{1}{2}$.

Generelt har vi $-\pi \leq \delta \leq \pi$

For lineære, homogene differanseligninger blir $G = 1$ for $\delta = 0$. Kan brukes som en sjekk på innsettingen.

Betingelsen for stabilitet for dette skjemaet blir da:

$$0 < r < \frac{1}{2} \tag{6.49}$$

med $r = \frac{\Delta t}{(\Delta y)^2}$ fra (6.27)

Da vi har et to-nivå skjema med konstante koeffisienter, er (6.49) en nødvendig og tilstrekkelig betingelse for stabilitet. Dette bekreftes av resultatet som vi fant tidligere ved bruk av det tilstrekkelige kriteriet i (6.29). Resultatet betyr en kraftig begrensning på størrelsen av Δt, noe som selvfølgelig vil influere sterkt på tidsforbruket. Et røft overslag for regnetiden ved bruk av FTCS-skjemaet gir:

$$\frac{T_2}{T_1} \approx \left(\frac{h_1}{h_2}\right)^3 \text{ med samme } r\text{-verdi}$$

der T_1 er regnetiden for $\Delta y = h_1$ og T_2 er regnetiden for $\Delta y = h_2$.

Med $h_1 = 0.1$ og $h_2 = 0.01$ får vi $\frac{T_2}{T_1} = 1000$.

Bruk av derivasjon

Et alternativ er å finne for hvilken verdi av δG har maksimum og minimum ved å beregne $\frac{dG}{d\delta}$ og sette $\frac{dG}{d\delta} = 0$. Deretter brukes betingelsen $|G| < 1$.

Fra (6.48):

$$G = 1 - 2r(1 - cos(\delta)) \rightarrow \frac{dG}{d\delta} = -2r\sin(\delta)$$

som gir max-min for $\delta = 0$,
$\delta = \pm\pi$, $\delta = 0$ gir $G = 1$, mens $\delta = \pm\pi$ gir $G = 1 - 4r$ som med betingelsen $|G| \leq 1$ blir

$$-1 \leq 1 - 4r \leq 1$$

Høyre side av ulikheten er alltid oppfylt, mens venstre side gir $r \leq \frac{1}{2}$ som før. Vi vil ofte finne at $\delta = \pm\pi$ er kritisk, slik at det uansett kan være lurt å sjekke disse verdiene. Men husk at dette nødvendigvis ikke er tilstrekkelig til å påvise stabilitet. Derimot kan det være tilstrekkelig til å påvise instabilitet da betingelsen $|G| \leq 1$ må være oppfylt for alle δ - verdier i intervallet $[-\pi,\ \pi]$.

Sammenligning av forsterkningsfaktorene.

Det kan være interessant å sammenligne forsterkningsfaktoren G i (6.48) med den analytiske forsterkningsfaktoren G_a i (6.37).

Ved bruk av (6.39) og (6.46) kan (6.37) skrives:

$$G_a = \exp(-\delta^2 \cdot r) \tag{6.50}$$

Stabilitetsgrensa er $0 < r \leq 0.5$ for G med følgende verdier for spesielle verdier av r:

$r = 0.025$

$$G = \cos^2\left(\frac{\delta}{2}\right) \geq 0 \text{ med } G = 0 \text{ for } \delta = \pi \tag{6.51}$$

$r = 0.5$

$$G = \cos(\delta) \text{ med } G = 0 \text{ for } \delta = \pi/2 \text{ og } G = -1 \text{ for } \delta = \pi \tag{6.52}$$

Figure 6.9 viser G og G_a som funksjon av $\delta \in [0,\ \pi]$ for forskjellige verdier av r. Merk at de heltrukne linjene er den numeriske forsterknings-faktoren.

For $\delta \in [0°, 90°]$ gir differanseskjemaet større dempning enn den analytiske løsningen; liten forskjell for små δ, men øker kraftig for økende δ.

For $\delta \in [90°,\ 180°]$ ser vi at amplituden endatil får feil fortegn i tillegg til stor avvik i størrelse. Når løsningen likevel er brukbar, skyldes dette at den analytiske løsningen har en amplitude G_a som avtar sterkt med økende frekvens; se den analytiske løsningen i (6.23). Denne glattingseffekten er typisk for parabolske ligninger. Likevel er effekten merkbar fordi vi i dette tilfellet har en diskontinuitet i randbetingelsen for $y = 0$, slik at løsningen inneholder mange høyfrekvente komponenter.

Feil i amplituden defineres ofte ved $\varepsilon_D = \left|\frac{G}{G_a}\right|$ og kalles diffusjonsfeil eller dissipasjonsfeil. (Ingen feil for $\varepsilon_D = 1$). Begrepet dissipativt skjema brukes gjerne om skjema der amplituden avtar med økende t.

I vårt tilfelle:

$$\varepsilon_D = \left|1 - 4r\sin^2(\delta/2)\right| \cdot \exp(\delta^2 \cdot r) \tag{6.53}$$

Rekkeutviklet for små δ:

$$\varepsilon_D = 1 - r^2\delta^4/2 + r\delta^4/12 + O(\delta^6)$$

som for $r \leq 1/2$ viser at feilen er liten for lave frekvenser.

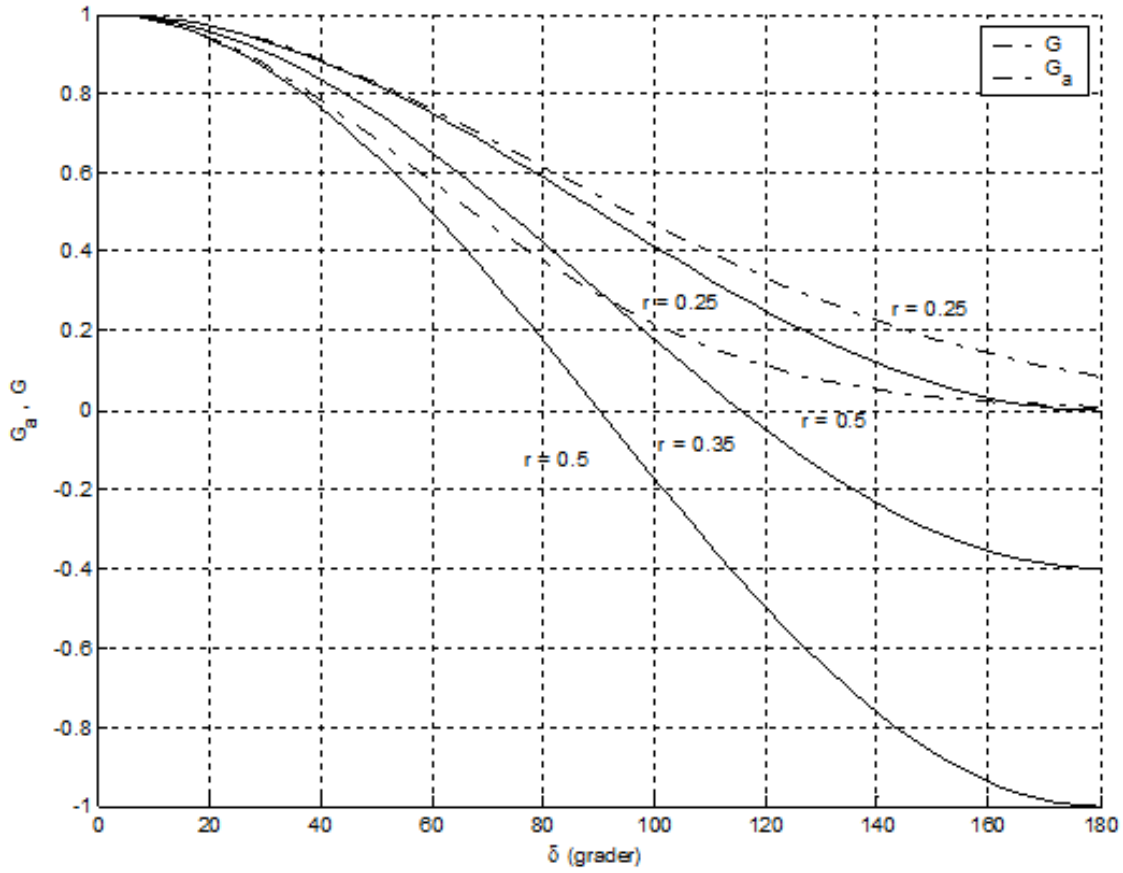


Figure 6.9: Forsterkningsfaktorer for diffusjonsligningen.

6.4 Flere skjema for parabolske ligninger

6.4.1 Richardson-skjemaet (1910)

FTCS-skjemaet er av 1. ordens nøyaktighet i t og 2. orden i y. Vi ønsker et skjema som også er av av 2. ordens nøyaktighet i t. Dette oppnår vi ved å benytte sentraldifferanser for leddet $\frac{\partial u}{\partial t}$:

$$\left.\frac{\partial u}{\partial t}\right|_j^n \approx \frac{u_j^{n+1} - u_j^{n-1}}{2\Delta t}$$

som gir følgende differanseligning:

$$u_j^{n+1} = u_j^{n-1} + 2r\left(u_{j-1}^n - 2u_j^n + u_{j+1}^n\right) \tag{6.54}$$

Dette er et eksplisitt, 3-nivå skjema som kalles *Richardson-skjemaet*; se Figure 6.10.

Stabilitetsanalyse.

La oss først forsøke det tilstrekkelige kriteriet i (6.29). Betingelsen om bare positive koeffisienter lar seg ikke oppfylle da koeffisienten foran u_j^n-leddet alltid er negativt for $r > 0$. Vi prøver derfor med von Neumanns metode. (6.40) innsatt i (6.54) gir:

$$G^{n+1}e^{i\cdot\beta y_j} = G^{n-1}e^{i\cdot\beta y_j} + 2r[G^n e^{i\cdot\beta y_{j-1}} - 2G^n e^{i\cdot\beta y_j} + G^n e^{i\cdot\beta y_{j+1}}]$$

Dividerer med $G^{n-1}e^{i\beta y_j}$ der $y_j = j\cdot h$:

$$\begin{aligned} G^2 &= 1 + 2rG\cdot\left(e^{-i\delta} + e^{i\delta} - 2\right) = 1 + 4rG\cdot(\cos(\delta) - 1) \\ &= 1 - 8Gr\cdot\sin^2\left(\tfrac{\delta}{2}\right) \end{aligned}$$

hvor vi har brukt (6.46) og (6.47). Vi har fått en 2. gradsligning fordi vi har et 3-nivå skjema:

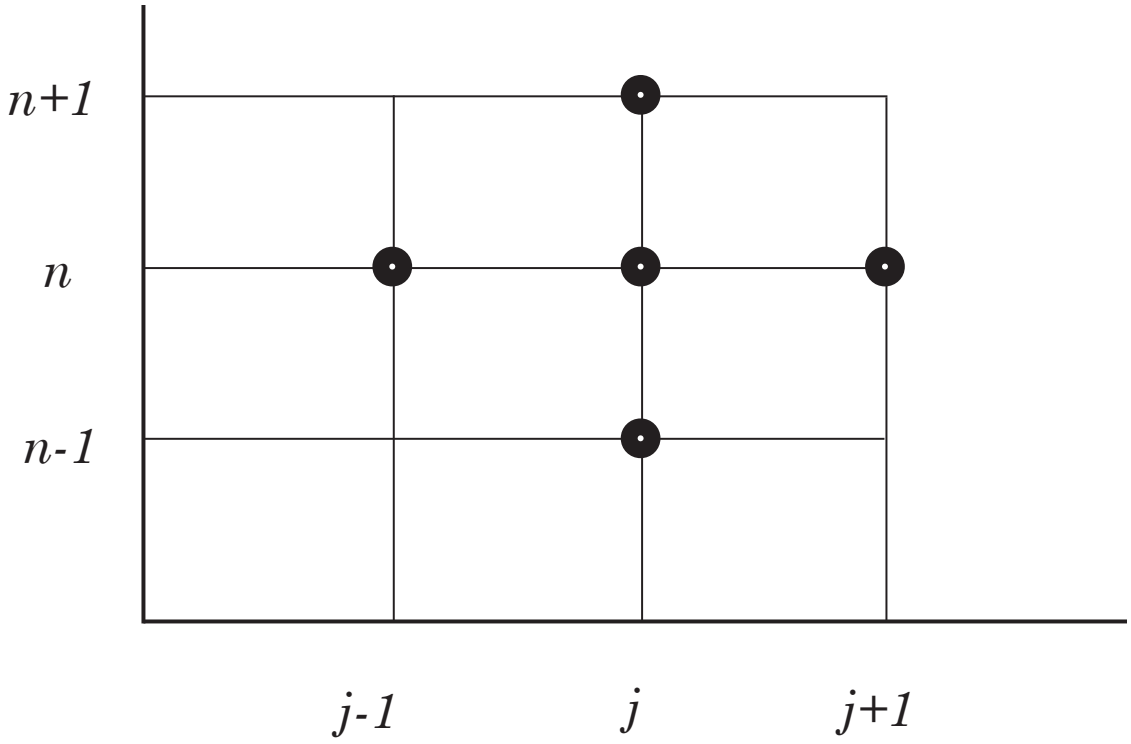


Figure 6.10:

$G^2 + 2bG - 1 = 0$ med løsning
$G_{1,2} = -b \pm \sqrt{b^2+1},\ b = 4r\sin^2\left(\frac{\delta}{2}\right) \geq 0$

$|G| = 1$ for $b = 0$. For alle andre verdier av b har vi $|G_2| > 1$. Skjemaet er følgelig ustabilt for alle aktuelle verdier av r. Et slikt skjema betegnes som ubetinget ustabilt. Dette tilfellet viser at det ikke er noen sammenheng mellom nøyaktighet og stabilitet.

Bruk av derivasjon

Vi kan bruke derivasjon her også:

$G^2 = 1 + 4rG \cdot (\cos(\delta) - 1)$
$2G\frac{dG}{d\delta} = 4r\left[(\cos(\delta) - 1)\frac{dG}{d\delta} - G\sin(\delta)\right]$, som med $\frac{dG}{d\delta} = 0$
gir max-min for $\delta = 0$, $\delta = \pm\pi$ som for FTCS-skjemaet.
$\delta = 0$ gir $G_{1,2} = \pm 1$ mens $\delta = \pm\pi$ gir $G_{1,2} = -4r \pm \sqrt{1 + (4r)^2}$

med instabilitet for $|G_2| > 1$ som tidligere.

6.4.2 Dufort-Frankel skjemaet (1953)

Richardson-skjemaet i (6.54) kan gjøres stabilt ved følgende modifikasjon:

$$u_j^n = \frac{1}{2}(u_j^{n+1} + u_j^{n-1}) \tag{6.55}$$

som innsatt i (6.54) gir:

$$u_j^{n+1} = \frac{1}{1+2r}\left[(1-2r)u_j^{n-1} + 2r(u_{j+1}^n + u_{j-1}^n)\right] \tag{6.56}$$

Dette er et eksplisitt 3-nivå skjema som kalles DuFort-Frankel-skjemaet, se Figure 6.11

3-nivå skjema der leddet u_j^n-leddet mangler, kalles skjema av "Leap-frog"-typen. (Leap-frog: hoppe bukk). Det tilstrekkelige kriteriet om positive koeffisienter fra (6.29) krever $r \leq \frac{1}{2}$ for et stabilt skjema. Stabilitetsanalysen er her litt mer komplisert p.g.a. at vi også må drøfte komplekse verdier av G.

Innsatt fra (6.40) og divisjon med $G^{n-1}e^{e^{i\delta j}}$:

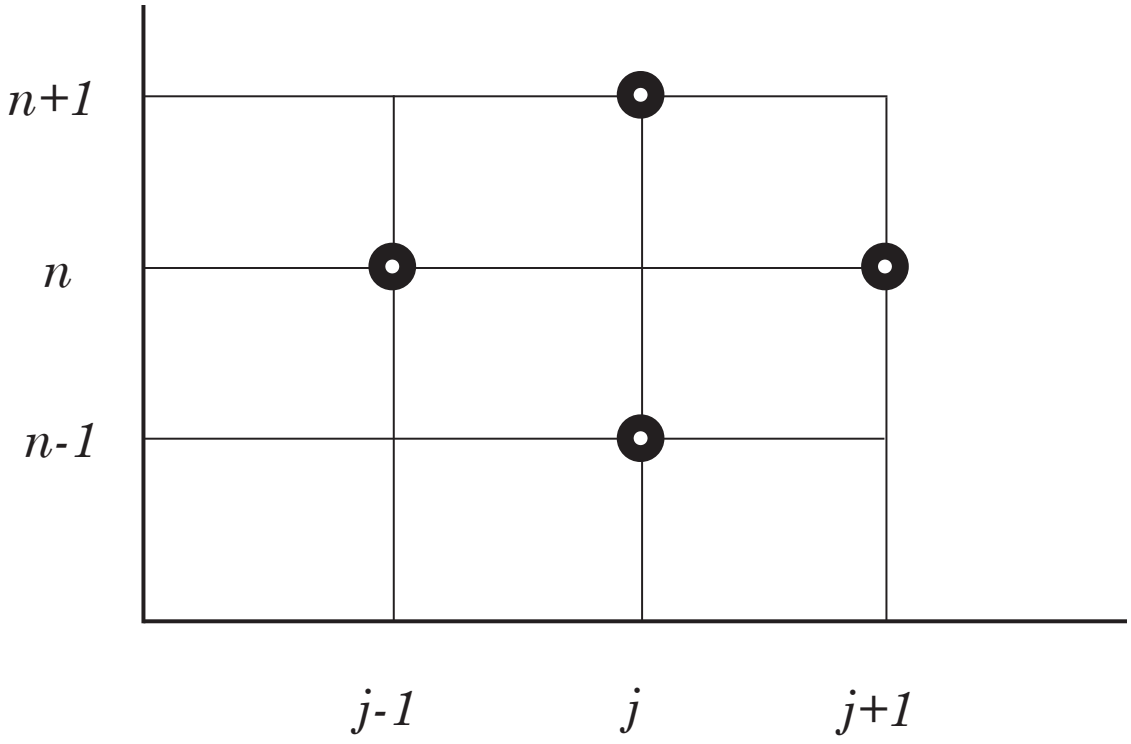


Figure 6.11:

$$G^2 = \frac{1}{1+2r}\left[(1-2r) + 2r\cdot G\cdot(e^{i\delta}+e^{-i\delta})\right] = \frac{1}{1+2r}\left[(1-2r) + 4r\cdot G\cdot\cos(\delta)\right]$$

som gir følgende 2. grads ligning:

$$(1+2r)\cdot G^2 - 4r\cdot G\cos(\delta) - (1-2r) = 0 \text{ med løsning:}$$

$$G_{1,2} = \frac{4r\cos(\delta) \pm \sqrt{(4r\cos(\delta))^2 + 4(1+2r)\cdot(1-2r)}}{2(1+2r)}$$

$$= \frac{2r\cos(\delta) \pm \sqrt{1-r^2\sin^2(\delta)}}{1+2r}$$

For stabilitet må begge røttene oppfylle betingelsen $|G| \leq 1$. Generelt må vi dessuten skille mellom reelle og komplekse røtter for å ta vare på det tilfellet at $G \leq 0$ når G er reell.

1. Reelle røtter: $1 - 4r^2\sin(\delta) \geq 0$

   $$|G_{1,2}| \leq \frac{2r\cdot|\cos(\delta)| + \sqrt{1-4r^2\sin^2(\delta)}}{1+2r} \leq \frac{1+2r}{1+2r} \leq 1$$

2. Komplekse røtter: $1 - 4r^2\sin^2(\delta) < 0 \rightarrow \sqrt{1-4r^2\sin^2(\delta)} = i\cdot\sqrt{4r^2\sin^2(\delta)-1}$

   $$|G_{1,2}|^2 = \left|\frac{\left(2r\cos(\delta)\right)^2 + 4r^2\sin^2(\delta) - 1}{(1+2r)^2}\right| = \left|\frac{4r^2-1}{4r^2+4r+1}\right| = \left|\frac{2r-1}{2r+1}\right| < 1$$

Analysen viser at (6.56) faktisk er ubetinget stabilt. DuFort-Frankel skjemaet er det eneste enkle kjente eksplisitte skjemaet med 2. ordens nøyaktighet som har denne egenskapen. Det har derfor vært en del brukt ved løsning av Navier-Stokes ligninger. I avsnitt (6.5) skal vi se at forholdene ikke er så fullt så rosenrøde som analysen ovenfor kan tyde på. For å starte beregningen, kan FTCS-skjemaet brukes.

6.4.3 Crank-Nicolson skjemaet. θ-skjemaet

En av bakdelene ved DuFort-Frankel skjemaet er at det behøves et spesielt skjema for å starte regneprosessen. Vi forsøker derfor å finne en approksimasjon for $\frac{\partial u}{\partial t}$ av 2. ordens nøyaktighet der bare to tidsnivå inngår.

Bruker sentraldifferanser for halve tidsintervallet:

$$\left.\frac{\partial u}{\partial t}\right|_j^{n+\frac{1}{2}} = \frac{u_j^{n+1} - u_j^n}{\Delta t} + O(\Delta t)^2 \tag{6.57}$$

Problemet blir nå å approksimere $\left.\frac{\partial u}{\partial t}\right|_j^{n+\frac{1}{2}}$ uten at nivået $n+\frac{1}{2}$ eksplisitt inngår i skjemaet. Dette oppnås ved Crank-Nicolson approksimasjonen (1947):

$$\left.\frac{\partial^2 u}{\partial t^2}\right|_j^{n+\frac{1}{2}} = \frac{1}{2}\left[\frac{u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}}{(\Delta x)^2} + \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2}\right] + O(\Delta x)^2 \tag{6.58}$$

Differanseligningen blir nå:

$$u_{j-1}^{n+1} - 2(1 + \frac{1}{r})u_j^{n+1} + u_{j+1}^{n+1} = -u_{j-1}^n + 2(1 - \frac{1}{r})u_j^n - u_{j+1}^n \tag{6.59}$$

der $r = \dfrac{\Delta t}{(\delta x)^2}$ som før.

Skjemaet er illustrert i Figure 6.12.

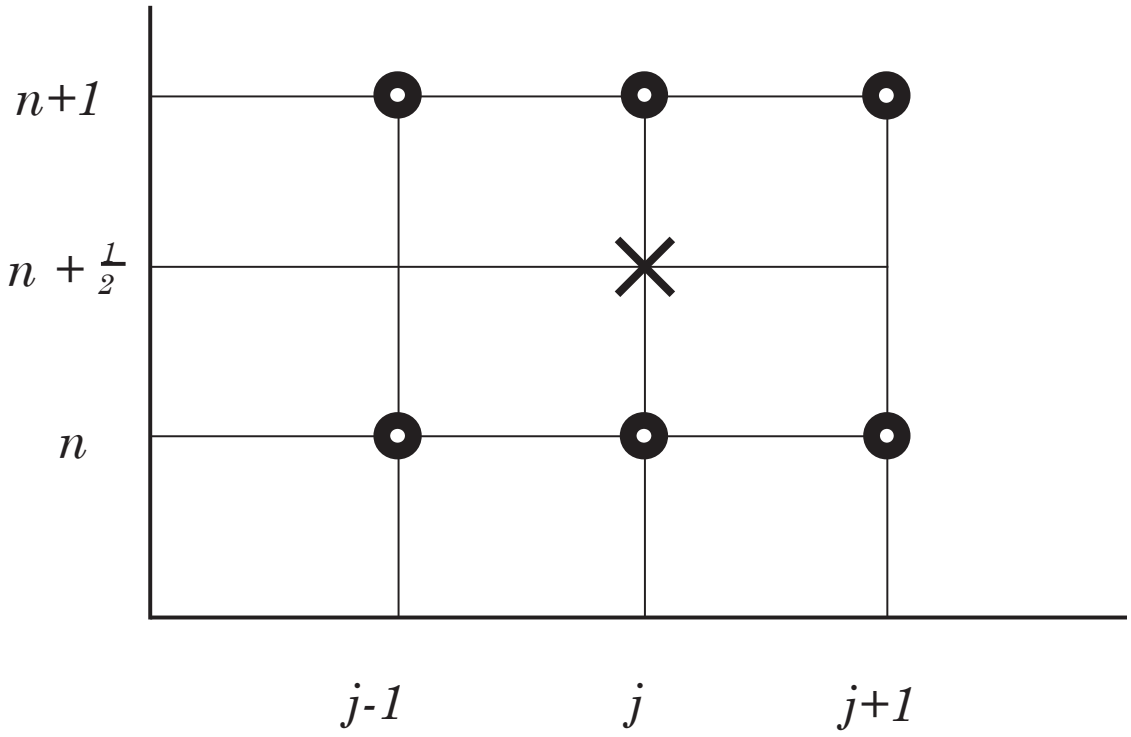


Figure 6.12:

Vi ser av (6.59) og Figure 6.12 at skjemaet er implisitt. Dette betyr at vi må løse et ligningsystem. I dette tilfellet er systemet tridiagonalt, slik at Thomasalgoritmen og programmet **tdma** kan brukes. **Marie 13**: Henvisning til Matlab

Vi skal nå undersøke stabiliteten av (6.59). For å slå flere fluer i et smekk, tar vi for oss følgende differanseligning som kalles θ-skjemaet:

$$u_j^{n+1} = u_j^n + r\left[\theta(u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}) + (1-\theta)(u_{j+1}^n - 2u_j^n + u_{j-1}^n)\right] \tag{6.60}$$

$$\text{der } 0 \le \theta \le 1 \tag{6.61}$$

For $\theta = 0$ får vi det eksplisitte FTCS-skjemaet. For $\theta = \frac{1}{2}$ får vi Crank-Nicolson skjemaet. For $\theta = 1$ fås et implisitt skjema som ofte kalles Laasonen-skjemaet (1949). I strømningsmekanikken brukes gjerne betegnelsen BTCS-skjemaet. (Backward Time Central Space) eller det totalt implisitte skjemaet.

(6.40) innsatt i (6.60) og divisjon med $G^n \cdot e^{i\cdot\beta x_j}$ gir:

$$\begin{aligned} G &= 1 + r[G\theta(e^{i\delta} + e^{-i\cdot\delta} - 2) + (1-\theta)(e^{i\delta} + e^{-i\delta} - 2)] \\ &= 1 + r(e^{i\delta} + e^{-i\delta} - 2)\cdot(G\theta + 1 - \theta) \end{aligned}$$

$$\text{der } \delta = \beta \cdot h$$

Med bruk av formlene i (6.47):

$$G = \frac{1 - 4r(1-\theta)\sin^2(\frac{\delta}{2})}{1 + 4r\theta\sin^2(\frac{\delta}{2})} \tag{6.62}$$

Betingelsen for stabilitet er $|G| \leq 1$ eller siden G er reell: $-1 \leq G \leq 1$

Da $0 \leq \theta \leq 1$, er den høyre betingelsen tilfredstilt med $r \geq 0$. For den venstre siden:

$$2r\sin^2\left(\frac{\delta}{2}\right)(1-2\theta) \leq 1 \text{ eller}$$

$$r(1-2\theta) \leq \frac{1}{2} \text{ da } \sin^2\left(\frac{\delta}{2}\right) \leq 1$$

For $\frac{1}{2} \leq \theta \leq 1$ er betingelsen oppfylt for alle $r \geq 0$, dvs.: Ubetinget stabil.

For $0 \leq \theta \leq \frac{1}{2}$ er skjemaet betinget stabilt.

Stabilitetsbetingelsen er da:

$$r(1-2\theta) \leq \frac{1}{2},\ 0 \leq \theta \leq 1 \tag{6.63}$$

Bruk av derivasjon

Vi skriver nå:

$$\begin{aligned} G &= 1 + r(e^{i\delta} + e^{-i\delta} - 2)\cdot(G\theta + 1 - \theta) = 1 + 2r\Big(\cos(\delta) - 1\Big)\cdot(G\theta + 1 - \theta) \\ \frac{dG}{d\delta} &= 2r\left[(G\theta + 1 - \theta)\frac{d}{d\delta}\Big(\cos(\delta) - 1\Big) + \Big(\cos(\delta) - 1\Big)\theta \cdot \frac{dG}{d\delta}\right] \end{aligned}$$

Med $\frac{dG}{d\delta} = 0$ får vi igjen max-min for $\delta = 0,\ \delta = \pm\pi$ ($\delta = 0$ gir $G = 1$ som ventet) $\delta = \pm\delta$ gir $G = \frac{1-4r(1-\theta)}{1+4r\theta}$ som er identisk med (6.62) innsatt for $\frac{\delta}{2} = 90°$. Betingelsen $|G| \leq 1$ blir $-1 \leq \frac{1-4r(1-\theta)}{1+4r\theta} \leq 1$ med samme resultat som tidligere.

Nøyaktighet

La oss se nærmere på nøyaktigheten av -skjemaet som gitt i (6.60).

Skriver et enkelt 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);
```

Vi har her brukt $h = \Delta x$ og $k = \Delta t$. Tnj er trunkeringsfeilen som er mer detaljert behandlet i avsnitt (6.5). Anta nå at $u(x,t)$ er den analytiske løsningen av diff.-ligningen $\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$ slik at vi kan sette $\frac{\partial}{\partial t}() = \frac{\partial^2}{\partial x^2}$ osv. Dersom vi bruker disse relasjonene i utskriften av Tnj, får vi følgende resultat:

$$T_j^n = \left[\left(\frac{1}{2} - \theta\right)\cdot\Delta t - \frac{1}{12}(\Delta x)^2\right]\cdot\frac{\partial^4 u}{\partial x^4} + \frac{1}{6}(\Delta t)^2(1-3\theta)\cdot\frac{\partial^6 u}{\partial x^6} + \ldots \tag{6.64}$$

Vi ser at $T_j^n = O(\Delta t) + O(\Delta x)^2$ for $\theta = 0$ og 1, altså for henholdsvis Euler-skjemaet og Laasonen-skjemaet, mens $T_j^n = O(\Delta t)^2 + O(\Delta x)^2$ for $\theta = \frac{1}{2}$ som er Crank-Nicolson skjemaet.

Dersom vi legger til en linje

```
Tnj:= simplify(subs(theta =(1-h^2/(6*k))/2,Tnj));
```

i programmet ovenfor, finner vi at $T_j^n = O(\Delta t)^2 + O(\Delta x)^4$ dersom vi velger $r = \frac{1}{6(1-2\theta)}$ når $r \leq \frac{1}{2(1-2\theta)}$.

Mer om stabiliteten.

Vi har funnet at skjemaet er ubetinget stabilt for $\frac{1}{2} \leq \theta \leq 1$. I praksis viser det seg at skjemaet kan gi oscillasjoner rundt diskontinuiteter for $\theta = \frac{1}{2}$. En θ-verdi > 0.5 vil dempe disse oscillasjonene og denne dempningen er sterkest for $\theta = 1$. La oss se på et eksempel der vi bruker varmeledningsligningen på dimensjonell form.

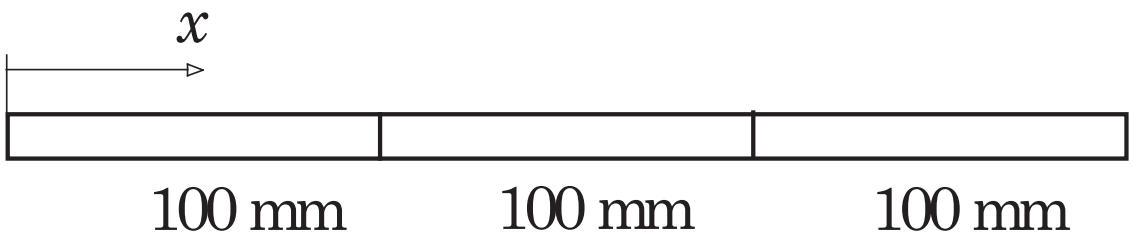


Figure 6.13:

Figuren viser en tynn aluminiumstang med lengde 300 mm.
Varmeledningsligningen er som vanlig gitt ved:

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2},\ T = T(x,t)$$

Randbetingelser:

$$T(0,t) = T(300,t) = 20°C$$

Startbetingelser:

$$T(x,0) = 270°C \text{ for } x \in (100,200)$$
$$T(x,0) = 20°C \text{ for } x \in (0,100) \text{ og } x \in (200,300)$$

Termisk diffusivitet:

$$\alpha = 100mm^2/s$$

Det numeriske Fourier-tallet:

$$r = \alpha \frac{\Delta t}{(\Delta x)^2}$$

Velger $\Delta t = 0.25$ s slik at $\Delta x = 5/\sqrt{r}$. Ved å velge $r = 4$, får vi $\Delta x = 2.5$ mm. Figurene på neste side viser en beregning med $\theta = \frac{1}{2}$ og en med $\theta = 1$; altså Crank-Nicolson skjemaet og Laasonen-skjemaet.

Vi ser at C-N-skjemaet gir kraftige oscillasjoner ved diskontinuitetene $x = 100$ of $x = 200$ for startprofilet. Etter $t = 4$ s er disse oscillajonene nesten borte. For Laasonen-skjemaet har vi ingen oscillasjoner selv rundt $t = 0$. For å finne årsaken til dette, må vi gå tilbake til lign. (6.62):

$$G = \frac{1 - 4r(1-\theta)\sin^2(\frac{\delta}{2})}{1 + 4r\sin^2(\frac{\delta}{2})}$$

For $\theta = \frac{1}{2}$ får vi:

$$G = \frac{1 - 2r\sin^2(\frac{\delta}{2})}{1 + 2r\sin^2(\frac{\delta}{2})}$$

For δ i nærheten av π, vil G ligge rundt -1 for store verdier av r. Dette ser vi tydeligst ved å velge $\delta = \pi = \beta \cdot h$ som gir $G = \frac{1-2r}{1+2r}$.
Fra (6.42) har vi:

$$E_j^n = G^n \cdot E_j^0,\ E_j^0 = e^{i \cdot \beta x_j}$$

Derfor ser vi at vi vil få oscillasjoner for δ i nærheten av π. Disse høye bølgetallene vil dø ut for økende t fordi startprofilet blir utglattet av dissipasjonen. Dersom vi setter $r = 4$ som brukt i eksemplet, vil vi etter 16 tidsteg med $\Delta t = 0.25$s få $G^{16} = \left(-\frac{7}{8}\right)^{16} \approx 0.12$, mens vi med $r = 1$ ville fått $G^{16} = 2.3 \cdot 10^{-8}$. Dette betyr at de høye bølgetallene avtar langsomt for store verdier av r. Mye likt Gibbs-fenomenet ved Fourierutvikling av diskontinuerlige funksjoner. (Se appendiks A.3, tilfelle 1 i kompendiet).

For Laasonen-skjemaet med $\theta = 1$ får vi derimot:

$$G = \frac{1}{1 + 4r\sin^2\left(\frac{\delta}{2}\right)}$$

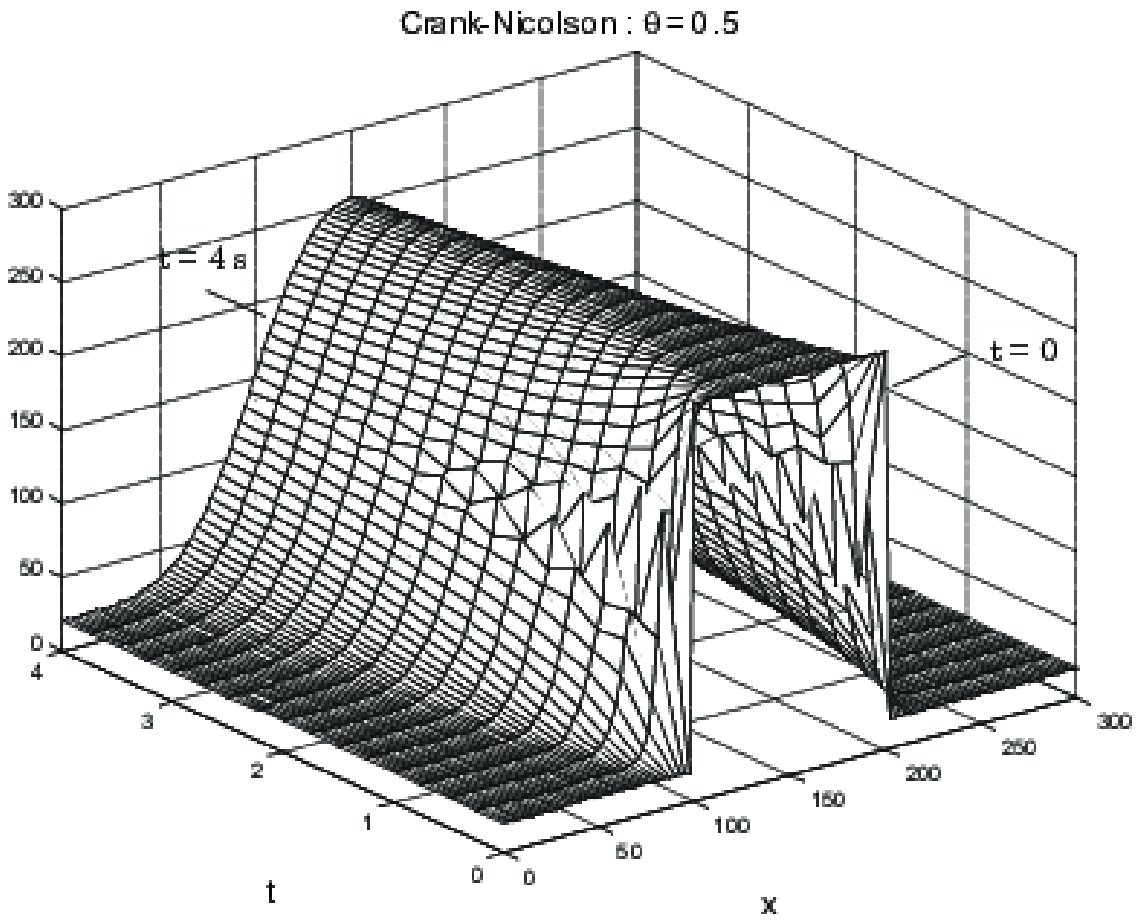


Figure 6.14:

Vi får her ingen oscillasjoner for noen av bølgetallene.

Dersom vi bruker uttrykk fra stabilitet av ordinære differential-ligninger, kan vi si at θ-skjemaet er absolutt stabilt (A-stabilt) for $\theta = \frac{1}{2}$ og strengt absolutt stabilt (eller L-stabilt) for $\theta = 1$. (Se avsnitt 1.6.1 i kompendiet)

Dersom vi bruker θ-skjemaet på et varmeledningsproblem av den typen som vist ovenfor med foreskrevet temperatur på begge rendene (Dirichlet-betingelser), vet vi at den maksimale temperaturen til enhver tid må ligge mellom den største som er gitt i startprofilet og den minste som er gitt på randen, eller $T_{min} \leq T_j^n \leq T_{max}$. For eksemplet ovenfor, er $T_{min} = 20°C$ og $T_{max} = 270°C$. Men da oppfyller vi kravene til PK-kriteriet .

Skriver (6.60) løst med hensyn på u_j^{n+1}:

$$u_j^{n+1} = \frac{1}{1+2\theta r}\left[\theta r(u_{j-1}^{n+1} + u_{j+1}^{n+1}) + (1-\theta)r(u_{j-1}^{n} + u_{j+1}^{n}) + \Big(1-(1-\theta)2r\Big)\right] \tag{6.65}$$

Ved å summere koeffisientene på høyre side, finner vi at summen er lik 1. Deretter må vi forlange at koeffisientene er positive. Dette betyr at $0 < \theta < 1$ og $1-(1-\theta)\cdot 2r > 0$. Den siste ulikheten er oppfylt for $r\cdot(1-\theta) < \frac{1}{2}$. Ved å sette $\theta = 0$ og $\theta = 1$, finner vi at summen er lik 1 også for disse verdiene, slik at vi får følgende betingelse for oppfyllelse av PK-kriteriet:

$$r\cdot(1-\theta) \leq \frac{1}{2} \tag{6.66}$$

Fra von Neumann-analysen fant vi følgende betingelse fra (6.63):

$$r\cdot(1-2\theta) \leq \frac{1}{2} \tag{6.67}$$

Vi ser at (6.66) er vesentlig strengere enn (6.67). Mens C-N-skjemaet med $\theta = \frac{1}{2}$ er ubetinget stabil i følge von Neumann-analysen, må vi ha $r \leq 1$ ifølge PK-kriteriet. PK-kriteriet gir her en sikker betingelse forat det fysiske max-min-kriteriet også oppfylles for differanseligningen. En test med eksemplet ovenfor, bekrefter dette kriteriet. Finnes det da et kriterium som er både

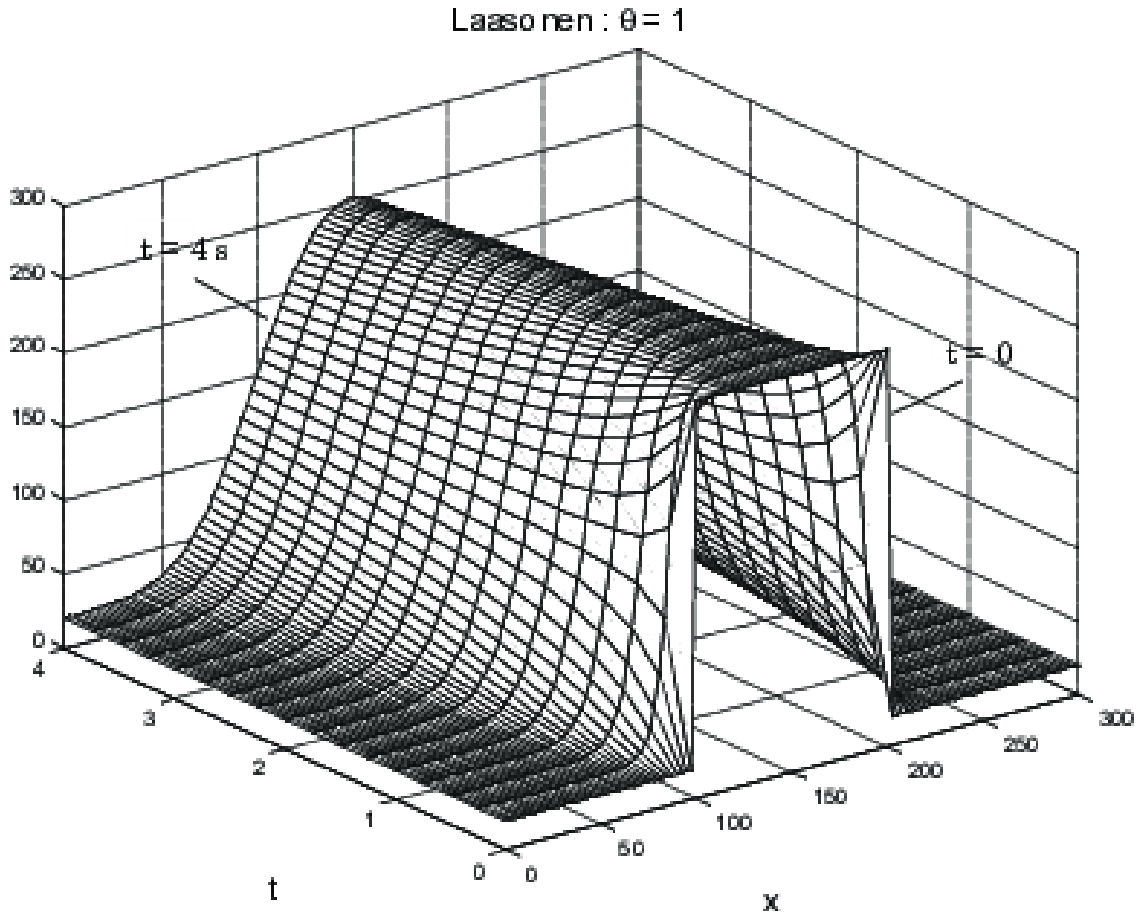


Figure 6.15:

nødvendig og tilstrekkelig for dennne enkle modell-ligningen? Kraaijevanger fant i 1992 følgende nødvendige og tilstrekkelige kriterium:

$$r \cdot (1-\theta) \leq \frac{2-\theta}{4(1-\theta)} \tag{6.68}$$

Vi ser at for $\theta = \frac{1}{2}$ gir dette kriteriet betingelsen $r \leq \frac{3}{2}$. For $\theta = \frac{3}{4}$ gir (6.68) $r \leq 5$ mens PK-kriteriet gir $r \leq 2$.

Hensikten med store r-verdier er for å kunne bruke forholdsvis store tidskritt når vi ønsker å følge hele tidsforløpet mot en stasjonær tilstand. Men vi må selvfølgelig tenke på nøyaktigheten også. Husk dessuten at dette er en enkel endimensjonal differanseligning med konstante koeffisienter som kan løses meget raskt med en moderne PC nærmest uavhengig at Δt og Δx så lenge vi holder oss innfor stabilitetsområdet. Ikke-stasjonære problem i tre dimensjoner krever fremdeles mye regnetid.

6.4.4 Von Neumanns generelle stabilitetsbetingelse

Vi har omtalt stabilitetsbetingelsen $|G| \leq 1$ som von Neumanns strenge stabilitetsbetingelse. Årsaken til denne betegnelsen, er at dersom $|G| \leq 1$ er oppfylt, kan ikke amplituden øke. I mange tilfeller har vi selvfølgelig fysikalske problemer der amplituden vokser med t. (t begrenset). Et enkelt eksempel er varmeledningsligningen med et kildeledd:

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2} + bT, \ b = \text{konstant}, \ t < t_{maks} \tag{6.69}$$

Med $b < 0$ har vi et varmesluk, mens for $b > 0$ har vi en varmekilde. I det første tilfellet kan vi bruke det strenge kriteriet, men for det andre tilfellet er det nødvendig å tillate $|G| > 1$.

En partikulær løsning av (6.69) er gitt ved:

$$T(x,t) = e^{bt} \cdot e^{-\alpha\beta^2 \cdot t} \cos(\beta x) = e^{(b-\alpha\beta^2)\cdot t} \cos(\beta x) \tag{6.70}$$

La oss bruke (6.70) til å bestemme en analytisk forsterkningsfaktor, se (6.37):

$$G_a = \frac{T(x_j,\ t_{n+1})}{T(x_j,\ t_n)} = \exp\left[(b-\alpha\beta^2)\cdot t_{n+1} - (b-\alpha\beta^2)\cdot t_n\right] = e^{b\Delta t}\cdot e^{-\alpha\beta^2\Delta t} \tag{6.71}$$

Vi ser at det er leddet $e^{b\cdot\Delta t}$ som får amplituden til å øke med positiv b.
En rekkeutvikling for små Δt:

$$e^{b\Delta t} = 1 + b\cdot\Delta t + \frac{b^2}{2}(\Delta t)^2 + \ldots \tag{6.72}$$

Dersom vi bruker FTCS-skjemaet, får vi:

$$T_j^{n+1} = r(T_{j+1}^n + T_{j-1}^n) + (1-2r)T_j^n + \Delta t b T_j^n \tag{6.73}$$

med

$$r = \alpha\frac{\Delta t}{(\Delta x)^2} \tag{6.74}$$

Dersom vi bruker PK-kriteriet, finner vi at summen av koeffisientene er lik $1 + b\cdot\Delta t$, slik at dette kriteriet bare kan brukes for $b < 0$.
Får ved å bruke von Neumanns metode på (6.73):

$$G = 1 - 4r\sin^2\left(\frac{\delta}{2}\right) + \Delta t\cdot b \tag{6.75}$$

Ved å sette $r = \frac{1}{2}$, får vi:

$$|G| \le \left|1 - 2\sin^2\left(\frac{\delta}{2}\right)\right| + |\Delta t\cdot b| \le 1 + \Delta t\cdot b,\ b > 0$$

Dersom vi sammenligner med (6.72), ser vi at vi får god overenstemmelse mellom den analytiske og den numeriske forsterkningsfaktoren i dette tilfellet.
Innfører von Neumanns generelle betingelsen ved:

$$|G| \le 1 + K\cdot\Delta t \tag{6.76}$$

der K er en positiv konstant.
Dette betyr at vi tillater amplituden å øke eksponentsielt for $t < t_{maks}$. I dette tilfellet kan vi bruke den strenge betingelsen dersom vi resonnerer på følgende måte:
Da kildeleddet i (6.69) ikke inneholder noen derivert størrelse, kan vi se bort fra dette leddet ved stabilitetsundersøkelsen. Vi har her samme type problemstilling som i avsnitt 1.6 i kompendiet, der vi diskuterer stive, ordinære differensialligninger. (Se f.eks. lign. (1.6.8) i avsnitt 1.6.1 i kompendiet).
For en voksende amplitude må vi minske skrittlengden for den uavhengige variable dersom vi skal oppnå en foreskrevet nøyaktighet. For en minskende amplitude må vi derimot holde oss under en maksimum skrittlengde for å få en stabil regneprosess.
La oss se på et annet eksempel med bruk av FTCS-skjemaet.

$$\frac{\partial u}{\partial t} = \alpha\frac{\partial^2 u}{\partial x^2} + a_0\frac{\partial u}{\partial x},\ \alpha > 0 \tag{6.77}$$

Denne ligningen som kalles adveksjon-diffusjonsligningen, er fremdeles en parabolsk ligning ifølge klassifiseringskjemaet i avsnitt 4.3 i kompendiet.

Ved bruk av FTCS-skjemaet på (6.77) med sentraldifferanser for $\dfrac{\partial u}{\partial x}$:

$$u_j^{n+1} = u_j^n + r\cdot(u_{j+1}^n - 2u_j^n + u_{j-1}^n) + a_0\frac{\Delta t}{2\Delta x}(u_{j+1}^n - u_{j-1}^n),\ r = \alpha\frac{\Delta t}{(\Delta x)^2}$$

Bruker von Neumanns metode:

$$G = 1 - 4r\sin^2\left(\frac{\delta}{2}\right) + i\cdot a_0\frac{\Delta t}{\Delta x}\sin(\delta)$$

som videre gir:

$$|G|^2 = \left(1 - 4r\sin^2\left(\frac{\delta}{2}\right)\right)^2 + \left(a_0\frac{\Delta t}{\Delta x}\sin(\delta)\right)^2 = \left(1 - 4r\sin^2\left(\frac{\delta}{2}\right)\right)^2 + \frac{a_0^2\cdot r}{\alpha}\cdot\Delta t\sin^2(\delta)$$

Velger igjen $r = \frac{1}{2}$:

$$|G| = \sqrt{\left(1 - 2\sin^2\left(\frac{\delta}{2}\right)\right)^2 + \frac{a_0^2}{2\alpha}\cdot\Delta t\cdot\sin^2(\delta)} \le 1 + \frac{a_0^2}{2\alpha}\cdot\Delta t$$

Vi har her brukt $\sqrt{x^2+y^2} \le |x| + |y|$

Med $K = \frac{a_0^2}{2\alpha}$, ser vi at den generelle betingelsen i (6.76) er oppfylt.
Adveksjon-diffusjonsligningen er behandlet mer detaljert i avsnitt 6.10 i kompendiet.

6.5 Trunkeringsfeil, konsistens og konvergens

La $U(x,t)$ være den eksakte løsningen av en PDL, skrevet $L(U)=0$, og u den eksakte løsningen av den tilhørende differanseligningen, skrevet $F(u)=0$. Den eksakte løsningen i $(x_i,\ t_n)$ er gitt ved:

$$U_i^n \equiv U(x_i,t_n) \text{ der } x_i = i\cdot\Delta x = i\cdot h,\ i=0,1,2,\ldots$$
$$t_n = n\cdot\Delta t = n\cdot k,\ n=0,1,2,\ldots$$

For den lokale trunkeringsfeilen T_i^n får vi da følgende uttrykk:

$$T_i^n = F(U_i^n) - L(U_i^n) = F(U_i^n) \tag{6.78}$$

T_i^n finnes ved Taylor-utvikling.
Noen rekkeutviklinger:

$$U_{i\pm1}^n \equiv U(x_{i\pm h},t_n) = U_i^n \pm h\cdot\left.\frac{\partial U}{\partial x}\right|_i^n + \frac{h^2}{2}\cdot\left.\frac{\partial^2 U}{\partial x^2}\right|_i^n \pm \frac{h^3}{6}\cdot\left.\frac{\partial^3 U}{\partial x^3}\right|_i^n + \ldots \tag{6.79}$$

$$U_i^{n\pm1} \equiv U(x_i,t_{n\pm k}) = U_i^n \pm k\cdot\left.\frac{\partial U}{\partial t}\right|_i^n + \frac{k^2}{2}\cdot\left.\frac{\partial^2 U}{\partial t^2}\right|_i^n \pm \frac{k^3}{6}\cdot\left.\frac{\partial^3 U}{\partial t^3}\right|_i^n + \ldots \tag{6.80}$$

La oss som eksempel finne den lokale trunkeringsfeilen T_i^n for FTCS-metoden anvendt på diffusjonsligningen $L(U)=0$ der

$$L(U) = \frac{\partial U}{\partial t} - \frac{\partial^2 U}{\partial x^2} = 0$$

$$T_i^n = F(U_i^n) = \frac{U_i^{n+1}-U_i^n}{k} - \frac{U_{i-1}^n - 2U_i^n + U_{i+1}^n}{h^2} \tag{6.81}$$

Innsatt fra (6.79) i (6.81):

$$T_i^n = \left(\frac{\partial U}{\partial t} - \frac{\partial^2 U}{\partial x^2}\right)_i^n + \left(\frac{k}{2}\frac{\partial^2 U}{\partial t^2} - \frac{h^2}{12}\frac{\partial^4 U}{\partial x^4}\right)_i^n + \frac{k^2}{6}\frac{\partial^3 U}{\partial t^3}\Big|_i^n + O(k^3,h^4)$$

$$\text{Da } \frac{\partial U}{\partial t} - \frac{\partial^2 U}{\partial x^2} = 0$$

$$T_i^n = \left(\frac{k}{2}\frac{\partial^2 U}{\partial t^2} - \frac{h^2}{12}\frac{\partial^4 U}{\partial x^4}\right)_i^n + \text{ høyere ordens ledd} \tag{6.82}$$

(6.82) viser at $T_i^n = O(k) + O(h^2)$ som ventet.
(6.82) kan også skrives:

$$T_i^n = \frac{h^2}{12}\cdot\left(6\frac{k}{h^2}\frac{\partial^2 U}{\partial t^2} - \frac{\partial^4 U}{\partial x^4}\right)_i^n + O(k^2) + O(h^4)$$

Ved å velge $r = \dfrac{k}{h^2} = \dfrac{1}{6}$, får vi:

$$T_i^n = O(k^2) + O(h^4) \tag{6.83}$$

Δt blir svært liten for $r = 1/6$, men med dagens PC-er er dette ikke noen problem, bortsett fra eventuell akkumulering av avrundingsfeil.

Konsistens

Vi sier at differanseligningen er konsistent med den gitte differensial-ligningen dersom den lokale trunkeringsfeilen $T_i^n \to 0$ når Δx og $\Delta t \to 0$ uavhengig av hverandre.

6.5.1 Example

Fra (6.82)

$$T_i^n = \left(\frac{k}{2}\frac{\partial^2 U}{\partial t^2} - \frac{h^2}{12}\frac{\partial^4 U}{\partial x^4}\right)_i^n \to 0 \text{ for } h \text{ og } k \to 0$$

Dette betyr at FTCS -skjemaet er konsistent med diffusjonsligningen .

La oss se nærmere på DuFort-Frankel skjemaet fra avsnitt (6.4.2):

$$T_i^n = \frac{U_i^{n+1}-U_i^{n-1}}{2k} - \frac{\left[U_{i-1}^n + U_{i+1}^n - (U_i^{n+1}+U_i^{n-1})\right]}{h^2}$$

Med bruk av rekkeutviklingene i (6.79) og (6.80):

$$T_i^n = \left[\frac{\partial U}{\partial t} - \frac{\partial^2 U}{\partial x^2} + \left(\frac{k}{h}\right)^2 \frac{\partial^2 U}{\partial t^2}\right]_i^n + \left[\frac{k^2}{6}\frac{\partial^3 U}{\partial t^3} - \frac{h^2}{12}\frac{\partial^4 U}{\partial x^4}\right]_i^n + O\left(\frac{k^4}{h^2}, k^4, h^4\right) \tag{6.84}$$

P.g.a faktoren $\left(\frac{k}{h}\right)^2$ er det viktig å spesifisere hvordan k og $h \to 0$. Skjemaet er ikke uten videre konsistent med den gitte ligningen. Et slikt skjema betegnes gjerne som betinget konsistent.

Tilfelle 1

Setter $r_0 = \frac{k}{h} \to k = r_0 \cdot h,\ r_0$ en konstant > 0.

Innsatt for k i (6.84):

$$T_i^n = \left(\frac{\partial U}{\partial t} - \frac{\partial^2 U}{\partial x^2} + r_0^2 \cdot \frac{\partial^2 U}{\partial t^2}\right)_i^n + O(h^2)$$

For $h \to 0$, ser vi at DuFort-Frankel skjemaet nå er konsistent med den hyperbolske ligningen $\frac{\partial U}{\partial t} + r_0^2 \frac{\partial^2 U}{\partial t^2} = \frac{\partial^2 U}{\partial x^2}$ og ikke den opprinnelige diffusjonsligningen.

Tilfelle 2

Setter $r_0 = \frac{k}{h^2} \to k = r_0 \cdot h^2$. Innsatt for k i (6.84):

$$\begin{aligned} T_i^n &= \left[\tfrac{\partial U}{\partial t} - \tfrac{\partial^2 U}{\partial x^2}\right]_i^n + \left[r_0^2 h^2 \tfrac{\partial^2 U}{\partial t^2} + \tfrac{k^2}{6}\tfrac{\partial^3 U}{\partial t^3} - \tfrac{h^2}{12}\tfrac{\partial^4 U}{\partial x^4}\right]_i^n + O\left(\tfrac{k^4}{h^2}, k^4, h^4\right) \\ &= \left[r_0^2 h^2 \tfrac{\partial^2 U}{\partial t^2} + \tfrac{k^2}{6}\tfrac{\partial^3 U}{\partial t^3} - \tfrac{h^2}{12}\tfrac{\partial^4 U}{\partial x^4}\right]_i^n + O\left(r_0^4 h^6,\ k^4,\ h^4\right) \end{aligned}$$

$$\text{da } \left[\frac{\partial U}{\partial T} - \frac{\partial^2 U}{\partial x^2}\right]_i^n = 0$$

Vi ser at $T_i^n \to 0$ for h og $t \to 0$ med $T_i^n = O(k^2) + O(h^2)$.

Skjemaet er nå konsistent med diffusjonsligningen. D-F-skjemaet kan derfor brukes med $k = r_0 \cdot h^2$. Men da har vi fått en begrensning på Δt, ikke som et stabilitetskrav, men ved kravet til konsistens. Ikke-konsistente skjema oppstår vanligvis når vi trikser med skjemaene etter at vi har Taylor-utviklet dem på vanlig måte.

Konvergens

Med U som den eksakte løsningen av differensialligningen og u som den eksakte løsningen av den tilhørende differanseligningen, sier vi at differanseligningen er konvergent dersom

$$\lim_{\Delta x, \Delta t \to 0} \to U \text{ for gitt } t_n \text{ og } x_i$$

Det er generelt vanskelig å bevise konvergensen av et differanseskjema. Derfor har det vært gjort mange forsøk på å erstatte definisjonen ovenfor med betingelser som er lettere å bevise hver for seg, men som tilsammen er tilstrekkelig for konvergens.

Det mest kjente av disse teoremene er *Lax's teorem*:

> Dersom det lineære initialverdiproblemet er velformulert og den tilhørende differanseligningen er konsistent, er stabilitet en nødvendig og tilstrekkelig betingelse for konvergens.

Se avsnitt 4.3 i kompendiet angående begrepet velformulert. Når vi ser alle betingelsene som må oppfylles forat Lax's teorem skal kunne anvendes, skjønner vi vanskelighetene med å bevise konvergens i mer generelle problemstillinger.

6.6 Eksempler med radiell symmetri

Dersom vi transformerer diffusjonsligningen $\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}$ til henholdsvis sylinder- og kule-koordinater og forlanger at u bare skal være funksjon av tiden t og radien r, får vi:

Sylinder:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial r^2} + \frac{1}{r}\frac{\partial u}{\partial r}$$

Kule:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial r^2} + \frac{2}{r}\frac{\partial u}{\partial r}$$

Ligningene kan da skrives:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial r^2} + \frac{\lambda}{r}\frac{\partial u}{\partial r},\ \lambda = 0, 1, 2 \tag{6.85}$$

$\lambda = 0$ med $r \to x$ gir det velkjente kartesiske tilfellet.

(6.85) er en partiell diff. ligning med variable koeffisienter. Vi vil nå forsøke en von Neumann-analyse av denne ligningen med θ-skjemaet fra avsnitt (6.4.3).

Stabilitetsanalyse med bruk av θ -skjemaet for radius $r > 0$

Setter $r_j = \Delta r \cdot j,\ j = 0, 1, \ldots$ og innfører $D = \frac{\Delta t}{(\Delta r)^2}$

For $r > 0$:

$$\begin{aligned} u_j^{n+1} = \quad & u_j^n + D\left[\theta(u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}) + (1-\theta)(u_{j+1}^n - 2u_j^n + u_{j-1}^n)\right] \\ & + \tfrac{\lambda D}{2j}\left[\theta(u_{j+1}^{n+1} - u_{j-1}^{n+1}) + (1-\theta)(u_{j+1}^n - u_{j-1}^n)\right] \end{aligned} \tag{6.86}$$

Utfører en von Neumann-analyse ved å sette inn $E_j^n = G^n \cdot e^{i\cdot\beta r_j} = G^n e^{i\cdot\delta\cdot j}$ med $\delta = \beta \cdot \Delta r$ og bruk av de vanlige formlene (6.47) og (6.48).

Vi får:

$$G \cdot \left(1 + 4\theta D \sin^2\left(\frac{\delta}{2}\right) - i \cdot \frac{\theta\lambda D}{j}\sin(\delta)\right) = 1 - 4(1-\theta)D \cdot \sin^2\left(\frac{\delta}{2}\right) + i\frac{(1-\theta)\lambda D}{j}\sin(\delta)$$

som ved bruk av formelen $\sin(\delta) = 2\sin\left(\frac{\delta}{2}\right)\cos\left(\frac{\delta}{2}\right)$ og betingelsen $|G| \le 1$ blir:

$$\left(1 - 4(1-\theta)D\sin^2\left(\frac{\delta}{2}\right)\right)^2 + \frac{(1-\theta)^2\lambda^2 D^2}{j^2}\sin^2(\delta) \le \left(1 + 4\theta D\sin^2\left(\frac{\delta}{2}\right)\right)^2 + \frac{\theta^2\lambda^2 D^2}{j^2}\sin^2(\delta)$$

og som videre gir:

$$D \cdot (1-2\theta) \cdot \left(\sin^2\left(\frac{\delta}{2}\right) \cdot \left(4 - \frac{\lambda^2}{j^2}\right) + \frac{\lambda^2}{j^2}\right) \le 2,\ j \ge 1 \tag{6.87}$$

Det er ikke vanskelig å se at leddet i parentesen har sin største verdi for $\sin^2\left(\frac{\delta}{2}\right) = 1$; dvs. for $\delta = \pi$. (Kan også finnes ved å derivere leddet m.h.p. δ som gir maksimum for $\delta = \pi$). Faktoren $\frac{\lambda^2}{j^2}$ faller da ut.

Vi får:

$$D \cdot (1-2\theta) \cdot 2 \le 1 \tag{6.88}$$

Som i avsnitt (6.4.3), må vi skille mellom to tilfeller.

1.

$$0 \le \theta \le \frac{1}{2}$$

$$D = \frac{\Delta t}{(\Delta r)^2} \le \frac{1}{2(1-2\theta)} \tag{6.89}$$

2.

$$\frac{1}{2} \le \theta \le 1$$

Skifter fortegn i (6.88):

$$D \cdot (2\theta - 1) \cdot 2 \ge -1$$

Denne betingelsen er alltid oppfylt, slik at differanseligningen er ubetinget stabil for disse θ-verdiene.

Vi har med andre ord fått de samme stabilitetsbetingelsene som for ligningen med konstante koeffisienter: $\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial r^2}$ der vi har FTCS-skjemaet for $\theta = 0$, Crank-Nicolson-skjemaet for $\theta = 1/2$, og Laasonen-skjemaet for $\theta = 1$.

Merk at denne analysen bare gjelder for $r > 0$.

Vi må da se på ligningen for $r = 0$.

Leddet $\frac{\lambda}{r}\frac{\partial u}{\partial r}$ må behandles spesielt for $r = 0$.

L'Hospitals regel:

$$\lim_{x\to 0}\frac{\lambda}{r}\frac{\partial u}{\partial r} = \lambda\frac{\partial^2 u}{\partial r^2} \to \frac{\partial u}{\partial t} = (1+\lambda)\frac{\partial^2 u}{\partial r^2} \text{ for } r = 0 \tag{6.90}$$

Vi har funnet at ved bruk av FTCS-skjemaet, har vi den vanlige begrensningen $D \le 1/2$. La oss derfor undersøke om randbetingelsene gir begrensninger når vi bruker FTCS-skjemaet.

Versjon 1

Diskretiserer (6.90) for $r = 0$ og utnytter symmetribetingelsen $\frac{\partial u}{\partial r}(0) = 0$:

$$u_0^{n+1} = [1 - 2(1+\lambda)D] \cdot u_0^n + 2(1+\lambda)D \cdot u_1^n \tag{6.91}$$

Dersom vi bruker PK-kriteriet på (6.91), får vi:

$$D \leq \frac{1}{2(1+\lambda)} \tag{6.92}$$

For $\lambda = 0$ får vi den velkjente betingelsen $D \leq 1/2$, mens vi for sylinderen med $\lambda = 1$ får $D \leq 1/4$ og for kule med $\lambda = 2$ får $D \leq 1/6$. Spørsmålet er om disse betingelsen for sylinder og kule er nødvendige. Vi vet at $D \leq 1/2$ er både tilstrekkelig og nødvendig for $\lambda = 0$.

Det er vanskelig å finne et nødvendig og tilstrekkelig kriterium i dette tilfellet. Ser derfor på et eksempel med oppstart av strømning i et rør, gitt som eksempel (6.6.1).

Versjon 2

For å unngå å bruke en separat ligning for $r = 0$, diskretiserer vi symmetribetingelsen $\frac{\partial u}{\partial r}(0) = 0$ med 2. ordens foroverdifferanser:

$$\frac{\partial u}{\partial r}(0) = 0 \rightarrow \frac{-3u_0^n + 4u_1^n - u_2^n}{2\Delta r} \rightarrow u_0^n = \frac{1}{3}(4u_1^n - u_2^n) \tag{6.93}$$

For kula finnes det en detaljert analyse av Dennis Eisen i tidskriftet Numerische Mathematik vol. 10, 1967, side 397-409. Han viser at en nødvendig og tilstrekkelig betingelse for løsning av (6.86) sammen med (6.91) (for $\lambda = 2$ og $\theta = 0$) er at $D < 1/3$. Dessuten viser han at ved å unngå å bruke (6.91), får vi stabilitet for FTCS-skjemaet når $D < 1/2$.

Vi beregner nå to tilfeller for å se hvilke stabilitetskrav vi får i praksis når vi bruker begge versjonene av randbetingelsene.

6.6.1 Example: Oppstart av rørstrømning

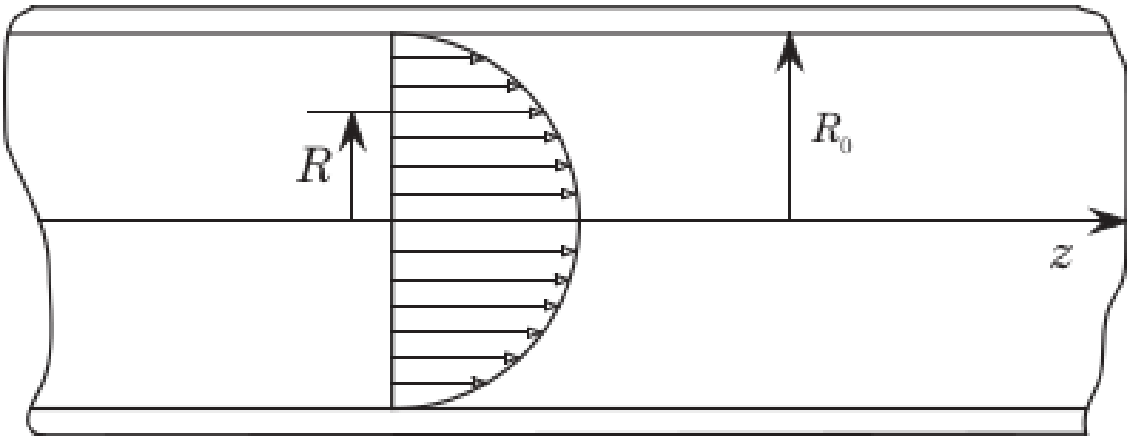


Figure 6.16:

Figuren viser hastighetsprofilet ved strømning i et rør av en inkompressibel fluid ved et gitt tidspunkt. Vi tenker oss at profilet har utviklet seg fra null ved å sette på en konstant trykkgradient $\frac{dp}{dz} < 0$, slik at hastighetsprofilet for det stasjonære tilfellet er det velkjente parabolske profilet for Poiseuille-strømning.

Bevegelsesligning:

$$\frac{\partial U}{\partial \tau} = -\frac{1}{\rho}\frac{dp}{dz} + \nu\left(\frac{\partial^2 U}{\partial R^2} + \frac{1}{R}\frac{\partial U}{\partial R}\right) \tag{6.94}$$

der hastighetsprofilet $U = U(R,\ \tau)$. $0 \leq R \leq R_0$, og τ er den fysikalske tiden.

Dimensjonsløse variable:

$$t = \nu\frac{\tau}{R_0^2},\ r = \frac{R}{R_0},\ u = \frac{U}{k},\ u_s = \frac{U_s}{k} \text{ der } k = -\frac{R_0^2}{4\mu}\frac{dp}{dz} \tag{6.95}$$

som innført i (6.94) gir:

$$\frac{\partial u}{\partial t} = 4 + \frac{\partial^2 u}{\partial r^2} + \frac{1}{r}\frac{\partial u}{\partial r} \tag{6.96}$$

Randbetingelser:

$$u(\pm 1, t) = 0, \ \frac{\partial u}{\partial r}(0, t) = 0 \tag{6.97}$$

Den siste er en symmetribetingelse. Finner stasjonær løsning u_s for $\frac{\partial u}{\partial t} = 0$:

$$\frac{d^2 u_s}{dr^2} + \frac{1}{r}\frac{du_s}{dr} = -4 \rightarrow \frac{1}{r}\frac{d}{dr}\left(r\frac{du_s}{dr}\right) = -4 \text{ som gir:}$$

$$\frac{du_s}{dr} = -2r + \frac{C_1}{r} \text{ med } C_1 = 0 \text{ da } \frac{du_s(0)}{dr} = 0$$

Etter en ny integrasjon og bruk av randbetingelsene, får vi den velkjente parabolske hastighetsfordelingen:

$$u_s = 1 - r^2 \tag{6.98}$$

Vi antar nå at vi har et tilfelle med fullt utviklet profil som gitt i (6.98). Plutselig fjerner vi trykkgradienten. Fra (6.94) ser vi at dette gir en enklere ligning. Hastigheten $\omega(r, t)$ for dette tilfellet er gitt ved:

$$\omega(r, t) = u_s - u(r, t) \text{ med } \omega = \frac{W}{k} \tag{6.99}$$

Vi skal nå løse følgende problem:

$$\frac{\partial \omega}{\partial t} = \frac{\partial^2 \omega}{\partial r^2} + \frac{1}{r}\frac{\partial \omega}{\partial r} \tag{6.100}$$

Randbetingelser:

$$\omega(\pm 1, t) = 0, \ \frac{\partial \omega}{\partial r}(0, t) = 0 \tag{6.101}$$

Startbetingelse:

$$\omega(r, 0) = u_s = 1 - r^2 \tag{6.102}$$

Det opprinnelige problemet er da:

$$u(r, t) = 1 - r^2 - \omega(r, t) \tag{6.103}$$

Den analytiske løsningen av (6.100), (6.103), først gitt av Szymanski i 1932, finnes i appendiks G.8 i kompendiet.
La oss se spesielt på FTCS-skjemaet.
Fra (6.86), med $\lambda = 1$, $\theta = 0$ og $j \geq 0$:

$$\omega_j^{n+1} = \omega_j^n + D \cdot (\omega_{j+1}^n - 2\omega_j^n + \omega_{j-1}^n) + \frac{D}{2j} \cdot (\omega_{j+1}^n - \omega_{j-1}^n) \tag{6.104}$$

For $j = 0$ får vi fra (6.91):

$$\omega_0^{n+1} = (1 - 4D) \cdot \omega_0^n + 4D \cdot \omega_1^n \tag{6.105}$$

Fra (6.89) får vi stabilitetsintervallet $0 < D \leq \frac{1}{2}$ for $r > 0$ og $0 < D \leq \frac{1}{4}$
fra (6.92) for $r = 0$ som er en tilstrekkelig betingelse.
Ved å løse (6.104), får vi følgende tabell for stabilitetsgrensa:

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 tilfredstillende nøyaktighet bør vi her ha $\Delta r \leq 0.1$. Av tabellen ovenfor ser vi at da er betingelsen $D < 0.4$ er tilstrekkelig. Med andre ord en slags midlere verdi av $D = \frac{1}{2}$ og $D = \frac{1}{4}$.
Det er ligningen i (6.105) som skaper problemer. Vi kan unngå denne ved isteden å bruke (6.93):

$$\omega_0^n = \frac{1}{3}(4\omega_1^n - \omega_2^n), \ n = 0, 1, \ldots \tag{6.106}$$

Beregningen viser da at grensa for hele systemet er gitt ved $0 < D \leq \frac{1}{2}$ for FTCS-skjemaet. Figuren nedenfor viser u-profilet for $D = 0.45$ med $\Delta r = 0.1$ etter 60 tidskritt med bruk av (6.105). Tydelig utvikling av instabilitet for $r = 0$.

Programmering av θ-skjemaet:

Randbetingelse gitt i (6.105)

Da 1 er det laveste matrise-indekset i Matlab, setter vi $r_j = \Delta r \cdot (j - 1)$, $\Delta r = \frac{1}{N}$, $j = 1, 2, \ldots, N + 1$ som vist på figuren ovenfor Ligningsystem fra (6.86) samt (6.90):

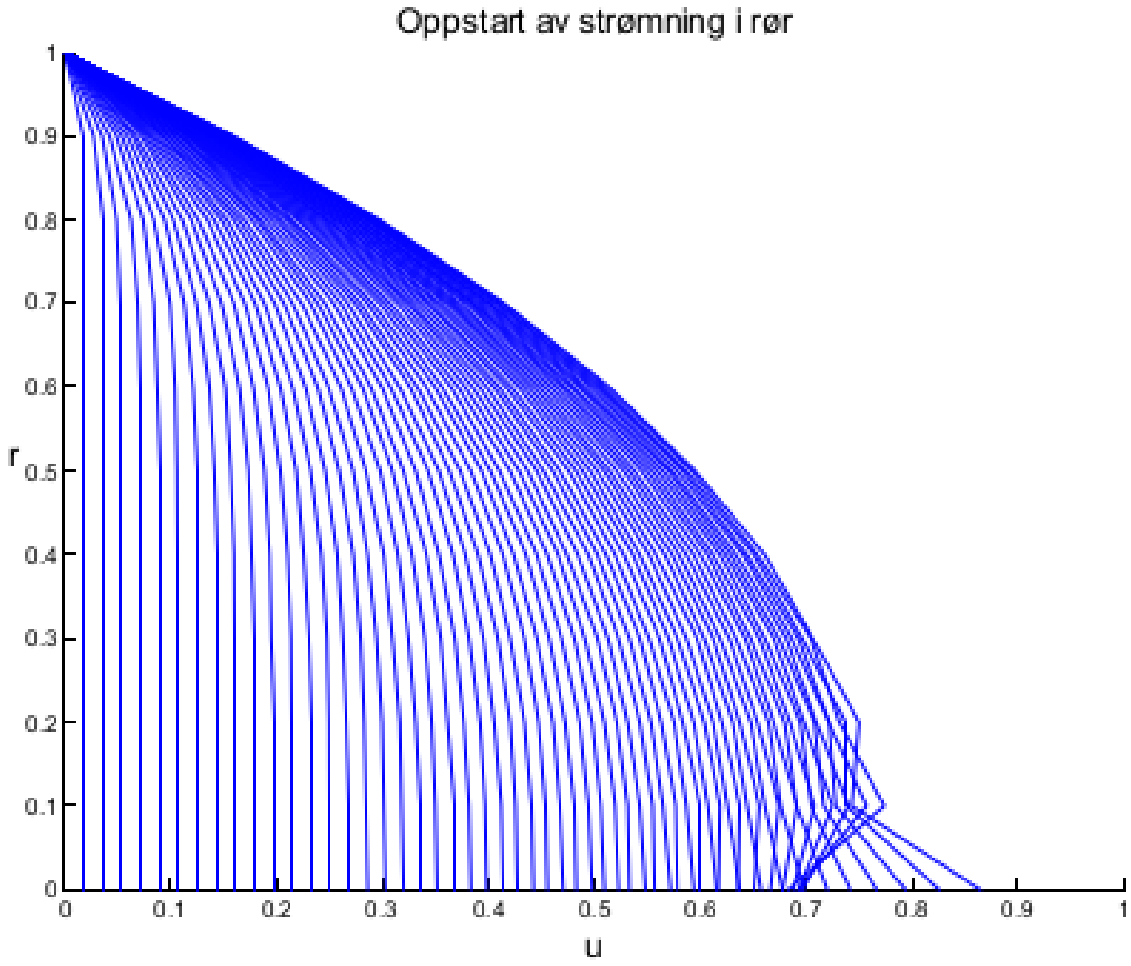


Figure 6.17:

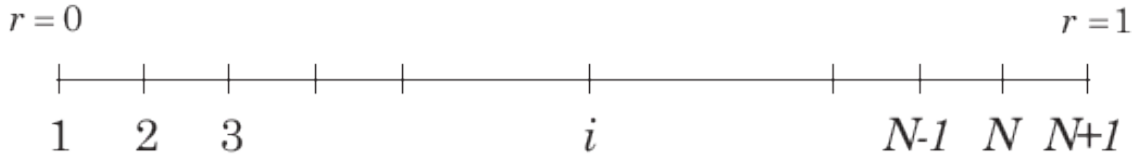


Figure 6.18:

For $j = 1$:

$$(1 + 4D \cdot \theta) \cdot \omega_1^{n+1} - 4D \cdot \theta \cdot \omega_2^{n+1} = \omega_1^n + 4D(1-\theta) \cdot (\omega_2^n - \omega_1^n) \tag{6.107}$$

For $j = 2, \ldots, N$:

$$\begin{aligned} &-D\theta \cdot \left(1 - \tfrac{1}{2(j-1)}\right) \cdot \omega_{j-1}^{n+1} + (1 + 2D\theta) \cdot \omega_j^{n+1} - D\theta \cdot \left(1 + \tfrac{1}{2(j-1)}\right) \cdot \omega_{j+1}^{n+1} \\ &= D(1-\theta) \cdot \left(1 - \tfrac{1}{2(j-1)}\right) \cdot \omega_{j-1}^n + [1 - 2D(1-\theta)] \cdot \omega_j^n \\ &+D(1-\theta) \cdot \left(1 + \tfrac{1}{2(j-1)}\right) \cdot \omega_{j+1}^n \end{aligned} \tag{6.108}$$

Får da følgende koeffisienter for bruk i **tdma**:

$$
\begin{gathered}
a_j = -D\theta\left(1-\tfrac{1}{2(j-1)}\right),\ j=2,\ldots,N \\
b_1 = 1+4D\theta \\
b_j = 1+2D\theta, j=2,\ldots,N \\
c_1 = -4D\theta \\
c_j = -D\theta\left(1+\tfrac{1}{2(j-1)}\right),\ j=2,\ldots,N-1 \\
d_1 = \omega_1^n + 4D(1-\theta)\cdot(\omega_2^n-\omega_1^n),\ n=1,\ldots \\
d_j = D(1-\theta)\Big\{\left(1-\tfrac{1}{2(j-1)}\right)\cdot\omega_{j-1}^n + \left(1+\tfrac{1}{2(j-1)}\right)\cdot\omega_{n+1}^n\Big\} \\
+[1-2D(1-\theta)]\cdot\omega_j^n,\ j=2,\ldots,N+1
\end{gathered}
\tag{6.109}
$$

Startverdier:

$$
\omega_j^0 = 1-r_j^2 = 1-[\Delta r\cdot(j-1)]^2,\ j=1,\ldots,N+1 \tag{6.110}
$$

Dessuten har vi at ω_{N+1} for alle n-verdier.

Merk at vi også kan bruke **tdma** for $\theta = 0$ (FTCS-skjemaet). I dette tilfellet forsvinner både a- og c-vektoren. Determinanten av matrisa er nå produktet av elementene i b-vektoren. Betingelsen for at matrisa er ikke-singulær er da at alle b-elementene $\neq 0$, noe som er oppfylt for alle verdier av D og θ.

La oss også undersøke den numeriske stabiliteten av **tdma** for dette systemet.

Fra lign. (3.1.4) i kompendiet har vi følgende tre betingelser:

$$
|b_1| > |c_1| > 0 \tag{6.111}
$$

$$
|b_j| \geq |a_j| + |c_j|,\ a_j\cdot c_j \neq 0,\ j=2,3,\ldots,N-1 \tag{6.112}
$$

$$
|b_N| > |a_N| > 0 \tag{6.113}
$$

Ulikhet 1:

$$
|1+4D\theta| > |4D\theta| \text{ som alltid er oppfylt.}
$$

Ulikhet 2:

$$
|1+2D\theta| \geq D\theta\left|1-\frac{1}{2(j-1)}\right| + D\theta\left|1-\frac{1}{2(j+1)}\right|
$$

Da alle leddene er positive:

$$
1+2D\theta \geq D\theta\left(1-\frac{1}{2(j-1)}\right) + D\theta\left(1-\frac{1}{2(j+1)}\right)
$$

$$
\rightarrow 1+2D\theta \geq 2D\theta \text{ som alltid er oppfylt}
$$

Ulikhet 3:

$$
|1+2D\theta| \geq D\theta\left|1-\frac{1}{2(j-1)}\right| \rightarrow 1+2D\theta \geq D\theta\left(1-\frac{1}{2(j-1)}\right)
$$

som alltid er oppfylt

Vi kan derfor bruke **tdma** som løsningsrutine.

Løsningen av dette systemet er gitt i programmet **startup** som løser systemet for $\theta = 0,\ \theta = 1/2$ og $\theta = 1$. Utskrift er gitt under. Vi har valgt $D = 0.4$ og $\Delta r = 0.02$ slik at vi er i det stabile området for FTCS-skjemaet. Utskriften gir u_s, ikke ω .

Legg merke til at det bare er høyresiden av systemet, dvs. d-vektoren, som er tidsavhengig. Vi kan derfor bruke en versjon av **tdma** der vi utfører elimineringen kun en gang, mens innsettingen utføres for hvert tidskritt.

En slik versjon er gitt i lign. (6.106), appendiks I i kompendiet og er programmert i **startupv3** som er 2-3 ganger raskere enn **startup**.

```
    ****************************************
    *    Impulsive start of pipeflow       *
    *    theta = 0:   FTCS-scheme          *
    *    theta = 1:   Laasonen             *
    *    theta = 1/2: Crank-Nicolson       *
    ****************************************

  No. of time-steps..... 500
  r-step length.....0.020
  Diffusion-number D..... 0.400
  Timestep.....   1.600e-004
  Elapsed time.....   8.000e-002
```

```
        r           u( )         u( )            u( )        analytisk

   0.000   3.1452e-001   3.1436e-001   3.1444e-001   3.1448e-001
 0.100   3.1368e-001   3.1351e-001   3.1360e-001   3.1363e-001
 0.200   3.1091e-001   3.1073e-001   3.1082e-001   3.1086e-001
 0.300   3.0548e-001   3.0528e-001   3.0538e-001   3.0542e-001
 0.400   2.9606e-001   2.9584e-001   2.9595e-001   2.9600e-001
 0.500   2.8069e-001   2.8047e-001   2.8058e-001   2.8063e-001
 0.600   2.5671e-001   2.5650e-001   2.5661e-001   2.5666e-001
 0.700   2.2078e-001   2.2060e-001   2.2069e-001   2.2073e-001
 0.800   1.6893e-001   1.6880e-001   1.6886e-001   1.6889e-001
 0.900   9.6816e-002   9.6748e-002   9.6782e-002   9.6798e-002
 1.000   0.0000e+000   0.0000e+000   0.0000e+000   0.0000e+000
```

Vi ser at det ikke er stor forskjell mellom resultatene for de ulike θ-verdiene da vi har en valgt en forholdsvis liten verdi for D.

Randbetingelse gitt i (6.93)

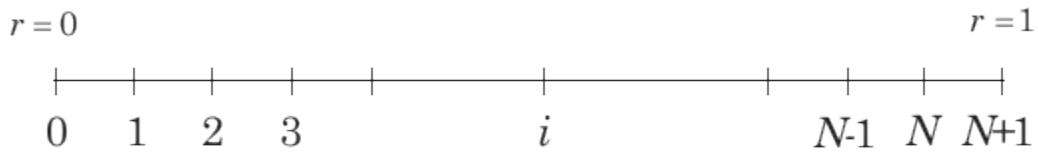


Figure 6.19:

Vi velger å renummere som vist på figuren:

$$r_j = \Delta r \cdot j,\ \Delta r = \frac{1}{N+1},\ j = 0, 1, \ldots, N+1$$

Ved å renummerere, kan vi også bruke **tdma** for tilfellet $\theta = 0$ (FTCS) slik som vi gjorde i den foregående beregningen. Uten renummerering, vil vi få $b_1 = 4D\theta$ slik at $b_1 = 0$ for $\theta = 0$. Dette vil føre til at determinanten av koeffisientmatrisa blir lik null.

Gjentar randbetingelsen gitt i (6.93):

$$\omega_0^n = \frac{1}{3}(4\omega_1^n - \omega_2^n),\ n = 0, 1, 2 \ldots \tag{6.114}$$

Ligningsystemet blir nå som gitt i (6.104) når vi lar $\frac{1}{2(j-1)} \to \frac{1}{2j}$ og $j = 1, 2, \ldots$ For $j = 1, \ldots, N$:

$$-D\theta \cdot \left(1 - \frac{1}{2j}\right) \cdot \omega_{j-1}^{n+1} + (1 + 2D\theta) \cdot \omega_j^{n+1} - D\theta \cdot \left(1 + \frac{1}{2j}\right) \cdot \omega_{j+1}^{n+1} \tag{6.115}$$

$$= D(1-\theta) \cdot \left(1 - \frac{1}{2j}\right) \cdot \omega_{j-1}^{n} + [1 - 2D(1-\theta)] \cdot \omega_j^n \tag{6.116}$$

$$+ D(1-\theta) \cdot \left(1 + \frac{1}{2j}\right) \cdot \omega_{j+1}^{n} \tag{6.117}$$

(6.116) utskrevet for $j = 1$, innsatt fra (6.114) og sammentrukket:

$$(1 + \frac{4}{3}D\theta) \cdot \omega_1^{n+1} - \frac{4}{3}D\theta\omega_2^{n+1} = [1 - \frac{4}{3}D(1-\theta)] \cdot \omega_1^n + \frac{4}{3}D(1-\theta)\omega_2^n \tag{6.118}$$

Etter at $\omega_1, \omega_2, \ldots$ er funnet, beregnes ω_0 fra (6.114), og lagres separat.
Startverdiene er som i (6.110), men med følgende indeksering:

$$\omega_j^0 = 1 - r_j^2 = 1 - (\Delta r \cdot j)^2,\ j = 0, \ldots, N+1 \tag{6.119}$$

Får da følgende koeffisienter for bruk i **tdma**:

$$
\begin{aligned}
a_j &= -D\theta\left(1-\tfrac{1}{2j}\right),\ j=2,\ldots,N \\
b_1 &= 1+\tfrac{4}{3}D\theta \\
b_j &= 1+2D\theta,\ j=2,\ldots,N \\
c_1 &= -\tfrac{4}{3}D\theta \\
c_j &= -D\theta\left(1+\tfrac{1}{2j}\right),\ j=2,\ldots,N-1 \\
d_1 &= [1-\tfrac{4}{3}D(1-\theta)]\cdot\omega_1^n+\tfrac{4}{3}D(1-\theta)\omega_2^n,\ n=1,\ldots \\
d_j &= D(1-\theta)\left\{\left(1-\tfrac{1}{2j}\right)\cdot\omega_{j-1}^n+\left(1+\tfrac{1}{2j}\right)\cdot\omega_{j+1}^n\right\} \\
&\quad +[1-2D(1-\theta)]\cdot\omega_j^n,\ j=2,\ldots,N,\ n=1,\ldots
\end{aligned}
\tag{6.120}
$$

Løsningen av dette systemet er gitt i programmet **startupv2**. Ulikhetene i (6.112) er oppfylt og **tdma** kan brukes. Med samme datasett, gir **startupv2** samme resultat som **startup**.

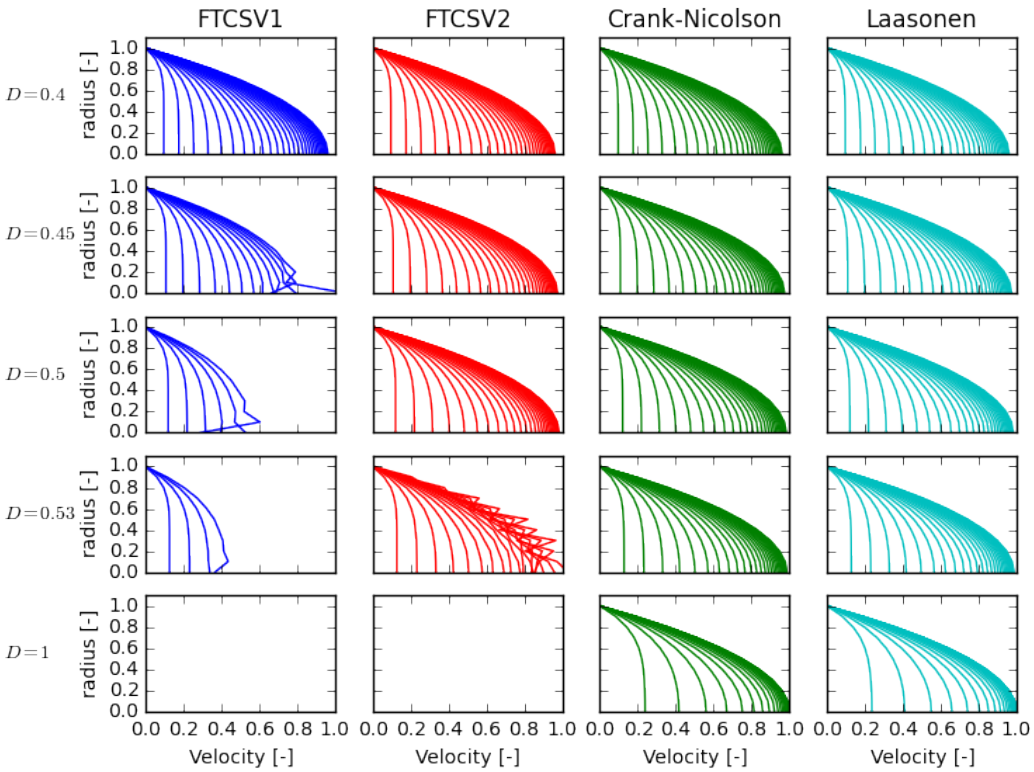
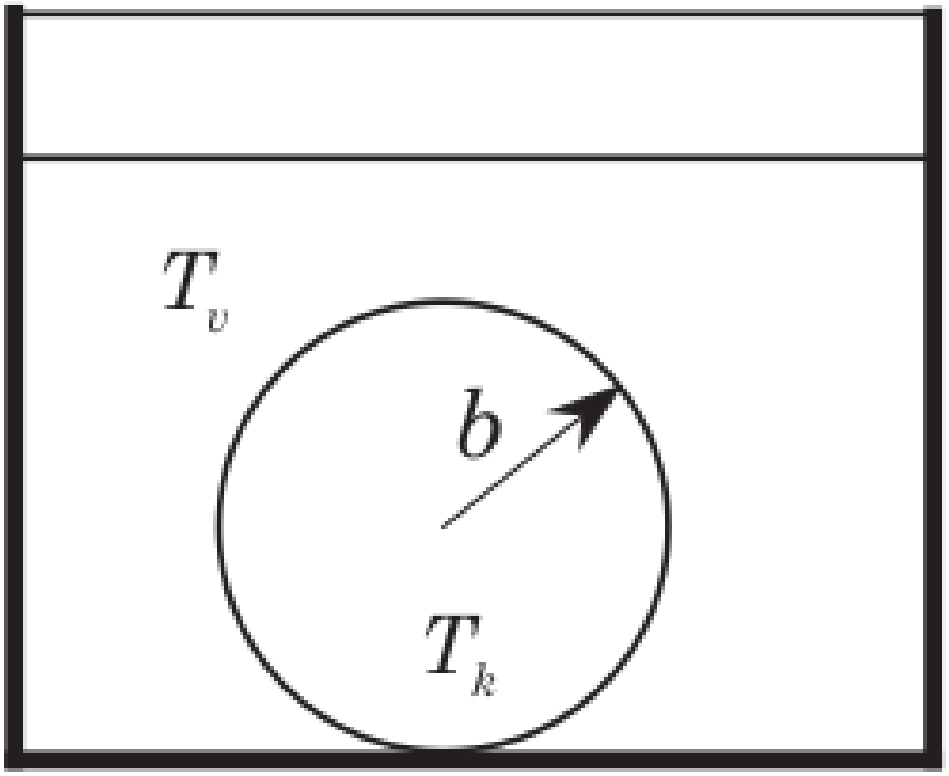


Figure 6.20: resultat fra test av stabilitet av de to forskjellige variantene av FTCS samt Crank og Laasonen. Ser at versjon 1 av FTCS utvikler innstabilitet før D=0.5, og at instabiliteten starter ved r=0. For versjon 2 oppstår innstabilitet ved D=0.5 og for alle r, som ventet.

`mov-ch5/startup.mp4`

Movie 1: Animasjon av eksepelet med bruk av tre θ-skjemaene samt analytisk øsning.

6.6.2 Example: Avkjøling av kule



Figuren viser en kule som blir avkjølt i vann. Kula har radius $b = 5$ og har temperatur T_k før den senkes i vannet. Vannet holder en konstant temperatur T_v under hele prosessen. Vi ser bort fra varmetap til omgivelsene.

Andre data:

$$Varmeledningstall : k = 0.1W/(cm \cdot^\circ C) \tag{6.121}$$

$$Varmeovergangstall : \bar{h} = 0.2W/(cm \cdot^\circ C) \tag{6.122}$$

$$Termiskdiffusitivitet : \alpha = 0.04cm^2/s \tag{6.123}$$

Vi har valgt verdiene slik at forholdet $\frac{\bar{h} \cdot b}{k} = 1$, noe som fører til en enklere analytisk løsning. Verdiene som er angitt i (6.121), (6.122) og (6.123), passer bra for nikkel-legeringer.

Vi skal nå løse følgende problem med $T = T(r, t)$:

$$\frac{\partial T}{\partial t} = \alpha \left(\frac{\partial^2 T}{\partial r^2} + \frac{2}{r} \frac{\partial T}{\partial r} \right) \tag{6.124}$$

Randbetingelser:

$$\frac{\partial T}{\partial r}(0, t) = 0, \text{ (symmetribetingelse)} \tag{6.125}$$

For $r = b$:

$$k \frac{\partial T}{\partial r} = \bar{h} \cdot (T_v - T_b) \tag{6.126}$$

Startbetingelse:

$$T(r,0) = T_k \tag{6.127}$$

I dette eksemplet nøyer vi oss med bruk av FTCS-skjemaet, men beregningen kan lett utvides til θ-skjemaet som vist i eksempel (6.6.1). Med $r_j = \Delta r \cdot j,\ \Delta r = \frac{1}{N+1},\ j = 0, 1, \ldots, N+1$ og $D = \alpha \frac{\Delta t}{(\Delta r)^2}$ får vi fra (6.86) når $u(r,t) \to T(r,t),\ \theta = 0$ og $\lambda = 2$:

$$T_j^{n+1} = (1-2D) \cdot T_j^n + D[(1-1/j) \cdot T_{j-1}^n + (1+1/j) \cdot T_{j+1}^n],\ j = 1, 2, \ldots \tag{6.128}$$

Som i eksempel (6.6.1), bruker vi to versjoner for symmetribetingelsen for $r = 0$. **1)**
Fra (6.91) med $\lambda = 2$:

$$T_0^{n+1} = (1-6D) \cdot T_0^n + 6D \cdot T_1^n \tag{6.129}$$

2) Fra (6.93):

$$T_0^n = \frac{1}{3}(4T_1^n - T_2^n),\ \text{alle } n \tag{6.130}$$

Randbetingelsen for $r = b$.
Diskretiserer (6.125) med bruk av 2. ordens bakoverdifferanser:

$$k \cdot \left(\frac{3T_{N+1}^n - 4T_N^n + T_{N-1}^n}{2 \cdot \Delta r} \right) = \bar{h} \cdot (T_v - T_b) \text{ som løst m.h.p } T_{N+1}^n \text{ gir:}$$

$$T_{N+1}^n = \frac{4T_N^n - T_{N-1}^n + 2\delta \cdot T_v}{3 + 2\delta} \tag{6.131}$$

$$\text{der } \delta = \frac{\Delta r \cdot \bar{h}}{k} \tag{6.132}$$

Vi har tidligere vist at vi må ha $D < 1/3$ når vi bruker randbetingelsen i (6.129)). I eksempel (6.6.1) for sylinderen, fant vi at stabilitetsgrensen for D økte når vi minsket Δr ved bruk av FTCS-skjemaet. For kula derimot viser det seg at betingelsen $D < 1/3$ er uavhengig av Δr. Årsaken til dette finner vi når vi skriver ut (6.128) for $j = 1$:

$$T_1^{n+1} = (1-2D) \cdot T_1^n + 2D \cdot T_2^n$$

Leddet $(1-1/j) \cdot T_{j-1}^n = (1-1) \cdot T_0^n$ forsvinner for $j = 1$, slik at temperaturen i kulas sentrum ikke influerer på noen av de andre verdiene. Dette forklarer hvorfor stabilitetsgrensa er uavhengig av Δr. Vi kan da faktisk løse (6.109) for $j = 1, 2, \ldots, N$ uten å bry oss om randbetingelsene i (6.129) og (6.130). Randbetingelsen i (6.130) trenger vi bare for å finne temperaturen T_0^n i sentrum av kula.

Neumann-analysen som ikke tar hensyn til rendene, viste at (6.128) er stabil for $D \leq 1/2$. Dessuten viste analysen at innflytelsen av den variable koeffisienten forsvant både for sylinderen og kula. (Se diskusjonen i forbindelse med (6.87)). Vi har ikke vist at $D \leq 1/2$ er en tilstrekkelig betingelse for hele systemet, siden vi da også må ta med randbetingelsen i (6.131).

Vi kan da oppsummerer for bruk av FTCS-skjemaet for kula:

Skjemaet i (6.131) er stabilt for $D < 1/2$ for $j = 1, 2, \ldots$

Dersom sentrumstemperaturen også beregnes, må vi ha $D < 1/3$ når (6.129) brukes.

Med bruk av (6.130), kan sentrumstemperaturen beregnes for $D < 1/2$.

Dette passer godt med Eisens analyse, omtalt i forbindelse med (6.93).

Nedenfor er vist utskrift fra programmet **kule** av en beregning med $D = 0.4$ og $\Delta r = 0.1$cm. $T_k = 300°C$ og $T_v = 20°C$ og beregningen varer i 10 min. med tidskritt 1 sekund. De analytiske verdiene er beregnet av funksjonen **kanalyt**. Vi ser at det er god overenstemmelse mellom de numeriske og de analytiske verdiene. (Analytisk løsning i appendiks G.9)

r(cm)	T ($^\circ C$)	T_a	r(cm)	T ($^\circ C$)	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			

`mov-ch5/sphere.mp4`

Movie 2: Animasjon av eksepelet med bruk av tre θ-skjemaene samt analytisk øsning.

Chapter 7

Convection problems and hyperbolic PDEs

7.1 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:

$$\frac{\partial u}{\partial t} + a_0 \frac{\partial u}{\partial x} = 0 \tag{7.1}$$

Another convenient feature of the model equation (7.1) is that is has an analytical solution:

$$u = u_0\, f(x - a_0\, t) \tag{7.2}$$

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 (7.1) 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 (7.1) on the following form:

$$\frac{\partial u}{\partial t} + \frac{\partial F}{\partial x} = 0 \tag{7.3}$$

where for the linear advection equation $F(u) = a_0\, u$.

7.1.1 Forward in time central in space discretization

We may discretize (7.1) 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 (7.1.1) into the advection equation (7.1) to yield:

$$u_j^{n+1} = u_j^n - \frac{C}{2}(u_{j+1}^n - u_{j-1}^n) \tag{7.4}$$

For convenience we have introduced the non-dimensional Courant-Friedrich-Lewy number (or CFL-number for short):

$$C = a_0 \frac{\Delta t}{\Delta x} \tag{7.5}$$

The scheme in (7.4 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 7.1 and (7.4.

We will try to solve model equation (7.1) with the scheme (7.4) and initial conditions illustrated in Fig (7.2) 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 7.3. 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 have indications of an

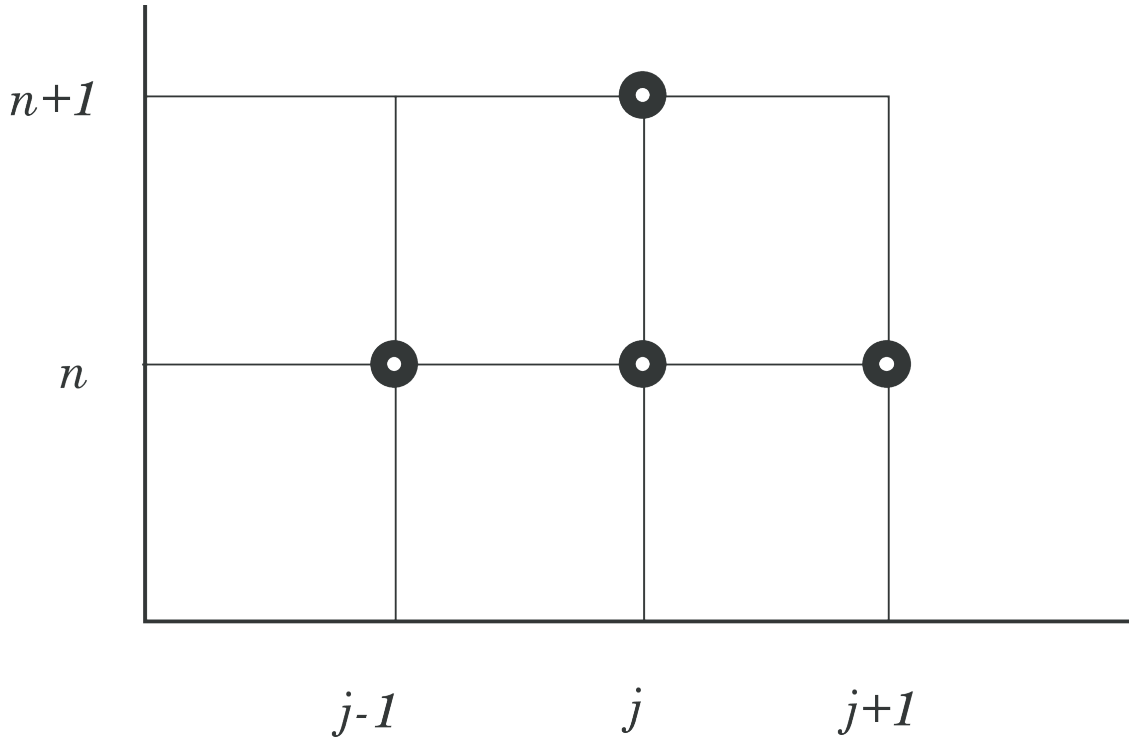


Figure 7.1: Illustration of the first order in time central in space scheme.

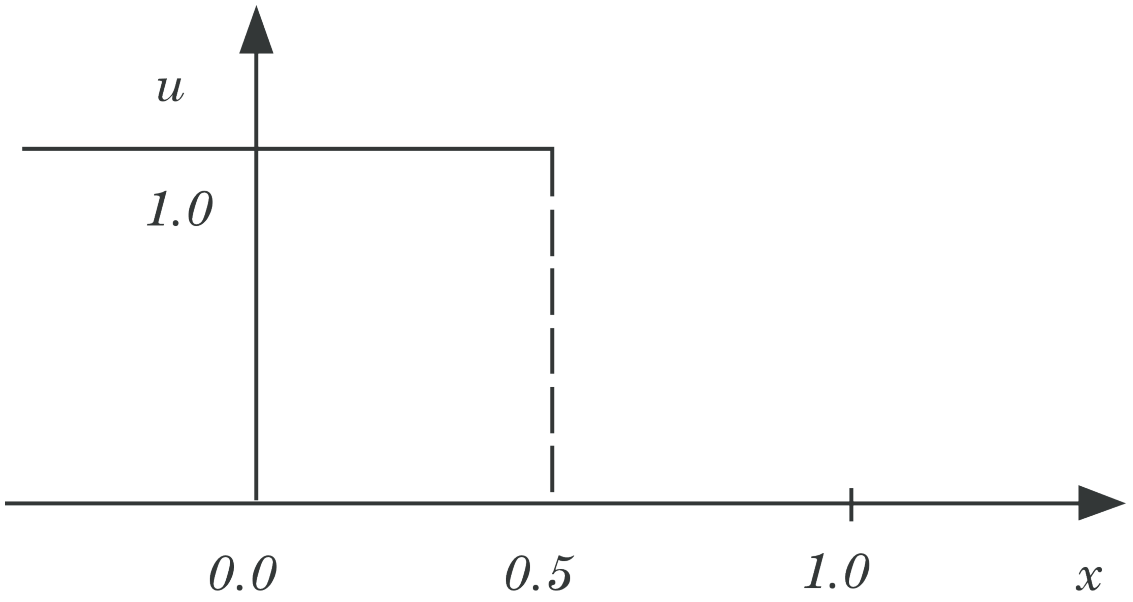


Figure 7.2: Initial values for the advection equaution (7.1).

unstable scheme. As a first approach observe that the coefficient for u_{j+1}^n in (7.4) always will be negative, and thus the criterion of positive coefficients (PC-criterion) may not be satisfied for any value of C.

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

$$u_j^n \rightarrow E_j^n = G^n \cdot e^{i\cdot\beta x_j} \tag{7.6}$$

Substitution of (7.6) into (7.4) 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:

$$2\cos(x) = e^{ix} + e^{-ix} \tag{7.7}$$

$$i\cdot 2\sin(x) = e^{ix} - e^{-ix} \tag{7.8}$$

$$\cos(x) = 1 - 2\sin^2(\frac{x}{2}) \tag{7.9}$$

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.

7.1.2 ftbs/upwind

Fredrik 14: add ftbs/upwind. check if ftbs is the same as upwind.

7.1.3 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_i^n over the neighbour cells in the in the temporal approximation:

$$\frac{u_i^{n+1} - \frac{1}{2}(u_{i+1}^n + u_{i-1}^n)}{\Delta t} = -\frac{F_{i+1}^n - F_{i-1}^n}{2\Delta x} \tag{7.10}$$

The Lax-Friedrich scheme is the obtained by isolation u_i^{n+1} at the right hand side:

$$u_i^{n+1} = \frac{1}{2}(u_{i+1}^n + u_{i-1}^n) - \frac{\Delta t}{2\Delta x}(F_{i+1}^n - F_{i-1}^n) \tag{7.11}$$

By assuming a linear flux $F = a_0\, u$ it may be shown that the Lax-Friedrich scheme takes the form:

$$u_i^{n+1} = \frac{1}{2}(u_{i+1}^n + u_{i-1}^n) - \frac{C}{2}(u_{i+1}^n - u_{i-1}^n) \tag{7.12}$$

where we have introduced the CFL-number as given by (7.5) 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]
```

7.1.4 Lax Wendroff Schemes

These schemes were proposed in 1960 by P.D. Lax and B. Wendroff [15] for solving, approximately, systems of hyperbolic conservation laws on the generic form given in (7.3).

A large class of numerical methods for solving (7.3) are the so-called conservative methods:

$$u_j^{n+1} = u_j^n + \frac{\Delta t}{\Delta x}\left(F_{i-1/2} - F_{i+1/2}\right) \tag{7.13}$$

Fredrik 15: mix of i and j indices in the above equation?

Linear advection. The Lax–Wendroff method belongs to the class of conservative schemes (7.3) and can be derived in various ways. For simplicity, we will derive the method by using a simple model equation for (7.3), namely the linear advection equation with $F(u) = a\,u$ as in (7.1), where a is a constant propagation velocity. The Lax-Wendroff outset is a Taylor approximation of u_j^{n+1}:

$$u_j^{n+1} = u_j^n + \Delta t \left.\frac{\partial u}{\partial t}\right|_j^n + \frac{(\Delta t)}{2} \left.\frac{\partial^2 u}{\partial t^2}\right|_j^n + \cdots \tag{7.14}$$

From the differential equation (7.3) we get by differentiation

$$\left.\frac{\partial u}{\partial t}\right|_j^n = -a_0 \left.\frac{\partial u}{\partial x}\right|_j^n \qquad \text{and} \qquad \left.\frac{\partial^2 u}{\partial t^2}\right|_j^n = a_0^2 \left.\frac{\partial^2 u}{\partial x^2}\right|_j^n \tag{7.15}$$

Before substitution of (7.61) in the Taylor expansion (7.14) we approximate the spatial derivatives by central differences:

$$\left.\frac{\partial u}{\partial x}\right|_j^n \approx \frac{u_{j+1}^n - u_{j-1}^n}{(2\Delta x)} \qquad \text{and} \qquad \left.\frac{\partial^2 u}{\partial x^2}\right|_j^n \approx \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2} \tag{7.16}$$

and then the Lax-Wendroff scheme follows by substitution:

$$u_j^{n+1} = u_j^n - \frac{C}{2}\left(u_{j+1}^n - u_{j-1}^n\right) + \frac{C^2}{2}\left(u_{j+1}^n - 2u_j^n + u_{j-1}^n\right) \tag{7.17}$$

with the local truncation error T_j^n:

$$T_j^n = \frac{1}{6} \cdot \left[(\Delta t)^2 \frac{\partial^3 u}{\partial t^3} + a_0 (\Delta x)^2 \frac{\partial^3 u}{\partial x^3}\right]_j^n = O[(\Delta t)^2, (\Delta x)^2] \tag{7.18}$$

The resulting difference equation in (7.17) may also be formulated as:

$$u_j^{n+1} = \frac{C}{2}(1+C)u_{j-1}^n + (1-C^2)u_j^n - \frac{C}{2}(1-C)u_{j+1}^n \tag{7.19}$$

The explicit Lax Wendroff stenticl is illustrated in Figure 7.4

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]
```

7.1.5 Lax-Wendroff for non-linear systems of hyperbolic PDEs

For non-linear equations (7.3) 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 (7.61)) is not straightforward and unique.

Richthmyer Scheme. One of the earliest extensions of the scheme is the Richtmyer two-step Lax–Wendroff method, which is on the conservative form (7.13) with the numerical fluxes computed as follows:

$$u_{j+1/2}^{n+1/2} = \frac{1}{2}\left(u_j^n + u_{j+1}^n\right) + \frac{1}{2}\frac{\Delta t}{\Delta x}\left(F_j^n - F_{j+1}^n\right) \tag{7.20}$$

$$F_{j+1/2} = F(u_{j+1/2}^{n+1/2}) \tag{7.21}$$

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:

$$u_{j+1/2}^{n+1/2} = \frac{1}{2}\left(u_{j+1}^n + u_j^n\right) - \frac{\Delta t}{2\Delta x}\left(F(u_{j+1}^n) - F(u_j^n)\right) \tag{7.22}$$

$$u_{j-1/2}^{n+1/2} = \frac{1}{2}\left(u_j^n + u_{j-1}^n\right) - \frac{\Delta t}{2\Delta x}\left(F(u_j^n) - F(u_{j-1}^n)\right) \tag{7.23}$$

Second step:

$$u_j^{n+1} = u_j^n - \frac{\Delta t}{\Delta x}\left(F(u_{j+1/2}^{n+1/2}) - F(u_{j-1/2}^{n+1/2})\right) \tag{7.24}$$

Notice that for a linear flux $F = a_0\,u$, the two-step Lax-Wendroff method ((7.23) and (7.24)) may be shown to reduce to the one-step Lax-Wendroff method outlined in (7.17) or (7.19).

MacCormack Scheme. A simpler and popular extension/variant of Lax-Wendroff schemes like in the previous section, is the MacCormack scheme [16]:

$$u_j^p = u_j^n + \frac{\Delta t}{\Delta x}\left(F_j^n - F_{j+1}^n\right) \tag{7.25}$$

$$u_j^{n+1} = \frac{1}{2}\left(u_j^n + u_j^p\right) + \frac{1}{2}\frac{\Delta t}{\Delta x}\left(F_{j-1}^p - F_j^p\right) \tag{7.26}$$

$$\tag{7.27}$$

where we have introduced the convention $F_j^p = F(u_j^p)$.

Note that in the predictor step we employ the conservative formula (7.13) 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 (7.13) 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 (7.27) is not written in conservative form (7.13). However, it easy to express the scheme in conservative form by expressing the flux in (7.13) as:

$$F_{j+1}^m = \frac{1}{2}\left(F_j^p + F_{j+1}^n\right) \tag{7.28}$$

For a linear flux $F(u) = a_0\, u$, one may show that the MacCormack scheme in (7.27) reduces to a two-step scheme:

$$u_j^p = u_j^n + C\left(u_j^n - u_{j+1}^n\right) \tag{7.29}$$

$$u_j^{n+1} = \frac{1}{2}\left(u_j^n + u_j^p\right) + \frac{C}{2}\left(u_{j-1}^p - u_j^p\right) \tag{7.30}$$

and substitution of (7.29) into (7.30) shows that the MacCormack scheme is identical to the Lax-Wendroff scheme (7.19) 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]
```

7.1.6 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:

```
# ../Kap6/advection_schemes.py

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import animation
from scipy import interpolate
from numpy import where
from math import sin

LNWDT=2; FNT=15
plt.rcParams['lines.linewidth'] = LNWDT; plt.rcParams['font.size'] = FNT

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

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]

# 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 = [lax_wendroff]


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()
```

mov-ch6/step.mp4

Movie 3: Result from code example above using a step function as the initial value

mov-ch6/sine.mp4

Movie 4: Result from code example above using a sine squared function as the initial value

7.1.7 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**: $\mathcal{O}\left(\Delta x^2, \Delta t\right)$, **Lax-Wendroff**: $\mathcal{O}\left(\Delta x^2, \Delta t^2\right)$, **MacCormack**: $\mathcal{O}\left(\Delta x^2, \Delta t^2\right)$. 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 [XXXX] in chapter 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:

$$\epsilon \approx Ch^p \tag{7.31}$$

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 7.5, 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
    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 empty list (e.g:
        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 boundary value

            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, Ntds, dx, dt, t)
    # 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, Ntds)]

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')
```

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 (7.31) is a simplification of the more general expression

$$\epsilon \approx C_x\, h_x^p + C_t\, h_t^q \tag{7.32}$$

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 (7.32) 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 (7.32). 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 7.5 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:

$$ftbs \rightarrow \epsilon = 1.3\, h_x^{0.98} + 6.5\, h_t^{0.98} \tag{7.33}$$

$$laxFriedrich \rightarrow \epsilon = -1484\, h_x^{1.9} + 26\, h_t^{1.0} \tag{7.34}$$

$$laxWendroff \rightarrow \epsilon = -148\, h_x^{2.0} + 364.\, h_t^{2.0} \tag{7.35}$$

```
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 optimized.
        """
        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 optimized.
        """
        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_friedrich, E_lax

            # 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 asymptotic r

    Cx0, Ct0 = optimize_error_cxct(E[-5:], hxList[-5:], htList[-5:],
                                   p=p_theoretical[k], q=q_theoretical[k]) # Find appropriate initial guesses for

    Cx, p, Ct, q = optimize_error(E[-5:], hxList[-5:], htList[-5:],
                                  Cx0, p_theoretical[k], Ct0, q_theoretical[k]) # Optimize for all parameters Cx,

    # 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] + ':')
```

7.1.8 Flux limiters

Looking at the previous examples and especially Figure 7.7 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 (4). (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 16**: 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 **??** 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.

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 (7.13) repeated here for convenience

$$u_j^{n+1} = u_j^n + \frac{\Delta t}{\Delta x}\left(F_{j-1/2} - F_{j+1/2}\right) \tag{7.36}$$

Fredrik 17: do we need clarifying more about conservative methods and Fluxes calculated at midpoints and halftime?

The general flux limiter method solves (7.36) with the following definitions of the fluxes $F_{i-1/2}$ and $F_{i+1/2}$

$$F_{j-1/2} = F_l(j-1/2) + \phi(r_{j-1/2})\Big[F_h(j-1/2) - F_l(j-1/2)\Big] \tag{7.37}$$

$$F_{j+1/2} = F_l(j+1/2) + \phi(r_{j+1/2})\Big[F_h(j+1/2) - F_l(j+1/2)\Big] \tag{7.38}$$

where $\phi(r)$ is the limiter function, and r is a measure of the smoothness of the solution. 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

$$r_{j-1/2} = \frac{u_{j-1} - u_{j-2}}{u_j - u_{j-1}} r_{j+1/2} = \frac{u_j - u_{j-1}}{u_{j+1} - u_j} \tag{7.39}$$

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 (7.36) by defining the Lax Wendroff Flux F_{LW} as:

$$F_{LW}\,(j-1/2) = F_{LW}\,(u_{j-1}, u_j) = a\,u_{j-1} + \frac{1}{2}a\left(1 - \frac{\Delta t}{\Delta x}a\right)\Big[u_j - u_{j-1}\Big] \tag{7.40}$$

$$F_{LW}\,(j+1/2) = F_{LW}\,(u_j, u_{j+1}) = a\,u_j + \frac{1}{2}a\left(1 - \frac{\Delta t}{\Delta x}a\right)\Big[u_{j+1} - u_j\Big] \tag{7.41}$$

which may be showed to be the Lax-Wendroff two step method condensed to a one step method as outlined in (7.1.5). 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 ($\frac{1}{2}a\,(1-c)\Big[u_j - u_{j-1}\Big]$ for $F_{LW}\,(j-1/2)$). A flux limited version of the Lax-Wendroff scheme could thus be obtained by adding a limiter function ϕ to the second term

$$F\,(j-1/2) = a\,u_{j-1} + \phi\left(r_{j-1/2}\right)\frac{1}{2}a\,(1-c)\Big[u_j - u_{j-1}\Big] \tag{7.42}$$

$$F\,(j+1/2) = a\,u_j + \phi\left(r_{j+1/2}\right)\frac{1}{2}a\,(1-c)\Big[u_{j+1} - u_j\Big] \tag{7.43}$$

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?

$$0 \le \left(\frac{\phi(r)}{r}, \phi(r)\right) \le 2 \tag{7.44}$$

Where we require

$$\phi(r) = 0, \quad r \le 0 \tag{7.45}$$

Hence for the scheme to be TVD the limiter must lie in the shaded region of Figure 7.9, where the limiter function for the two second order schemes, Lax-Wendroff and Warming and Beam are also plotted.

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 7.10

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:

$$Superbee: \quad \phi(r) = max\left(0, min(1, 2r), min(2, r)\right) \tag{7.46}$$

$$Van - Leer: \quad \phi(r) = \frac{r + |r|}{1 + |r|} \tag{7.47}$$

$$minmod: \quad \phi(r) = minmod(1, r) \tag{7.48}$$

where minmod of two arguments is defined as

$$minmod(a, b) = \begin{cases} a & if \quad a \cdot b > 0 \quad and \quad |a| > |b| \\ b & if \quad a \cdot b > 0 \quad and \quad |b| > |a| > 0 \\ 0 & if \quad a \cdot b < 0 \end{cases} \tag{7.49}$$

The limiters given in (7.46 -(7.48 are showed in Figure 7.11. 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:

$$upwind: \quad \phi(r) = 0 \tag{7.50}$$

$$Lax - Wendroff: \quad \phi(r) = 1 \tag{7.51}$$

in which upwind, is TVD, but first order, and Lax-Wendroff is second order, but not TVD.

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 (5)

`mov-ch6/flux_limiter.mp4`

Movie 5: The effect of Flux limiting schemes on solutions containing continuos and discontinuous sections

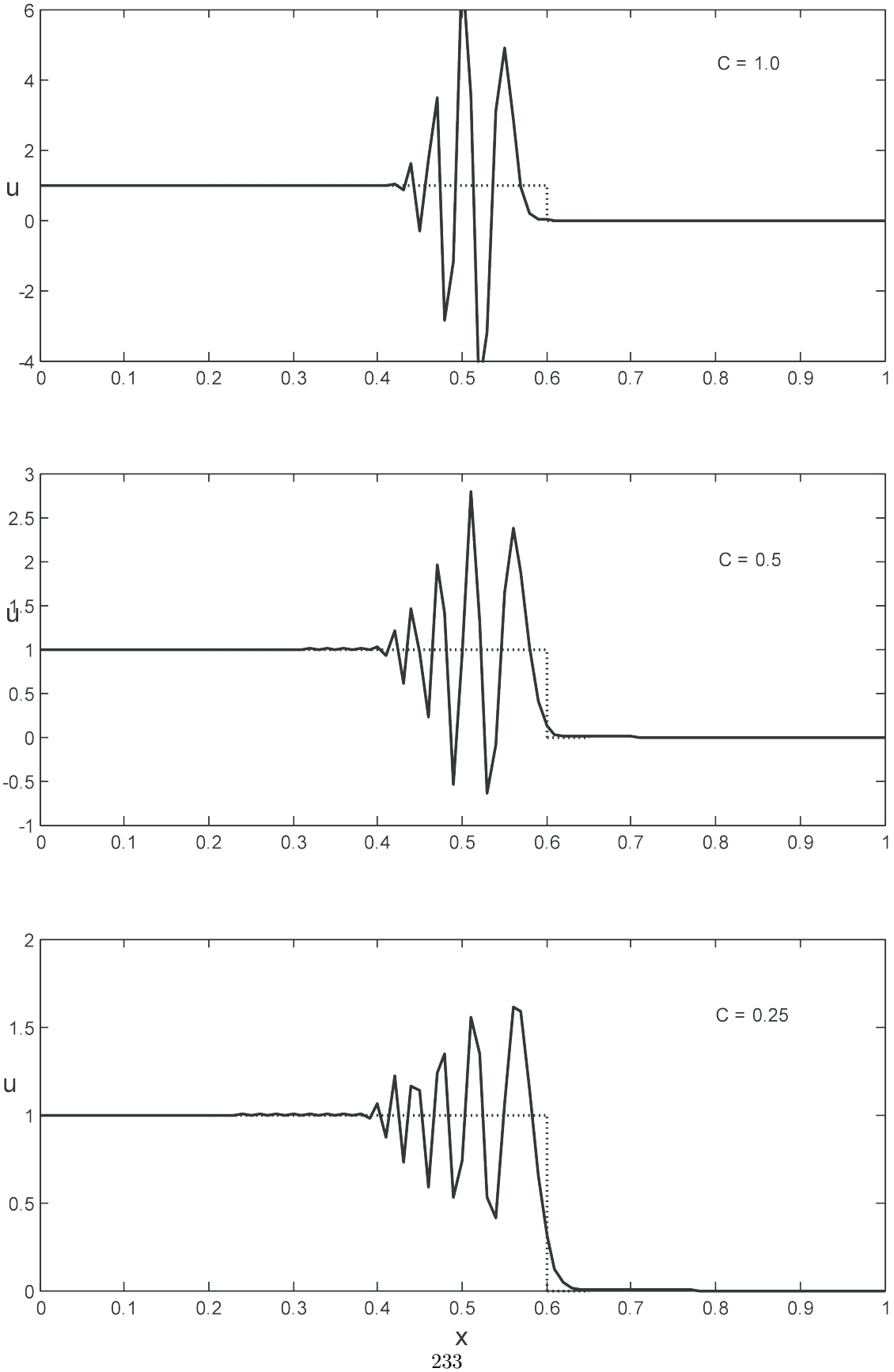


Figure 7.3: Computed solutions with the (7.4). Dotted line: analytical solution, solid line: computed soultion.

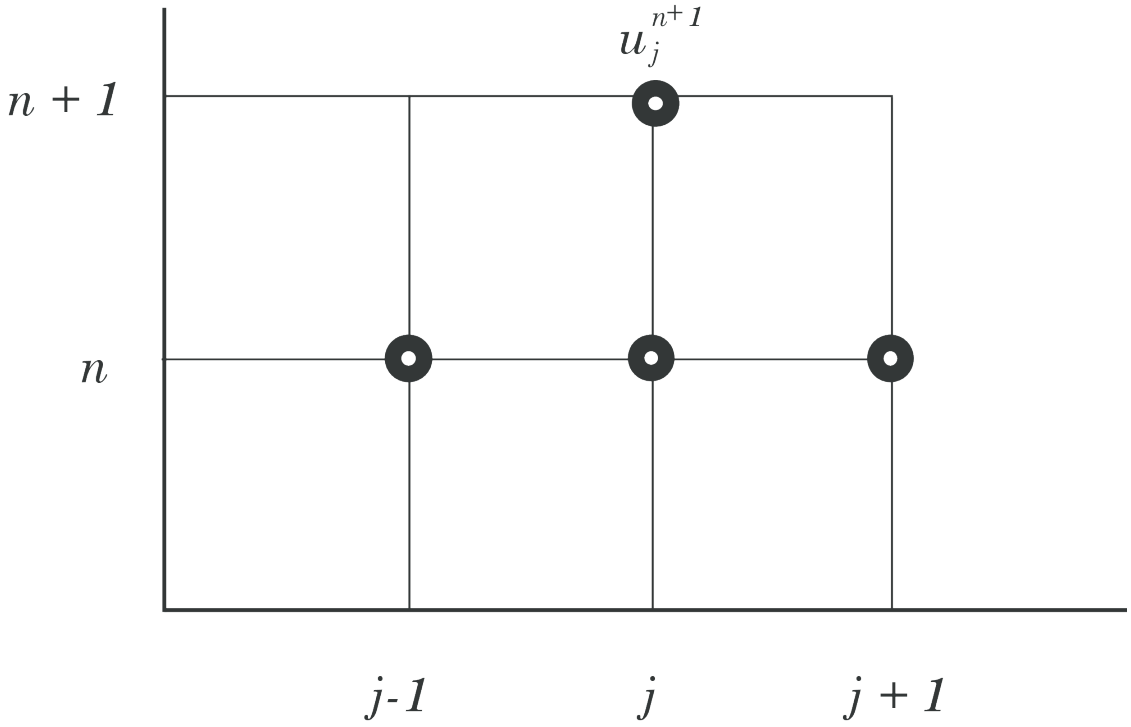


Figure 7.4: Schematic of the Lax-Wendroff scheme.

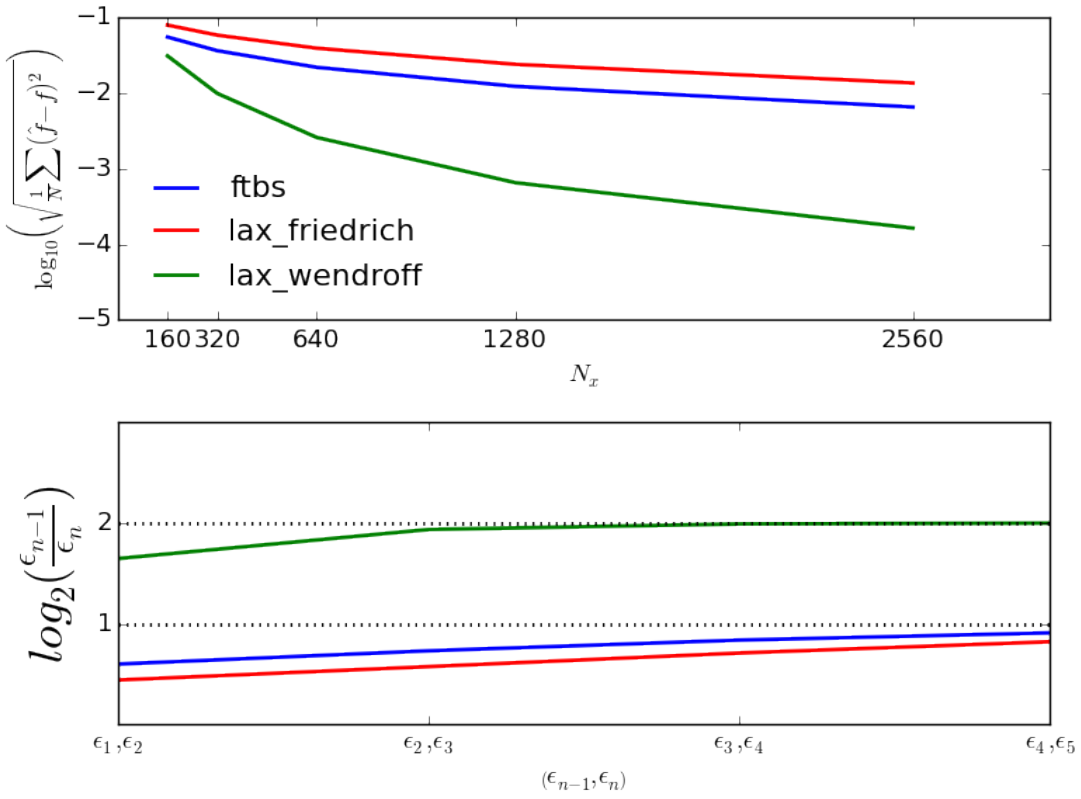


Figure 7.5: The root mean square error E for the various advection schemes as a function of the number of spatial nodes (top), and corresponding observed convergence rates (bottom).

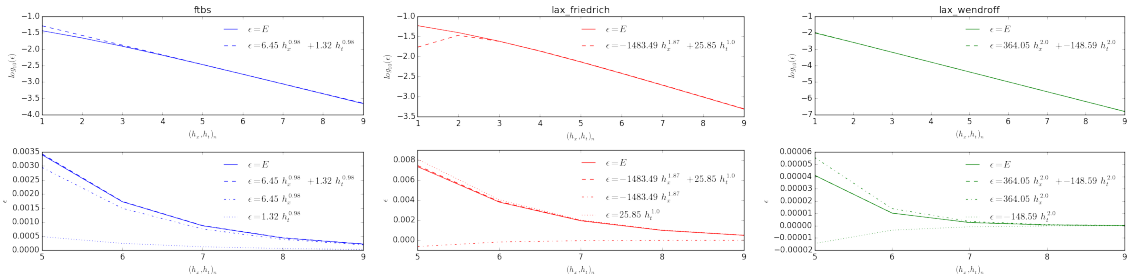


Figure 7.6: Optimized error expressions for all schemes. Test using doconce combine images

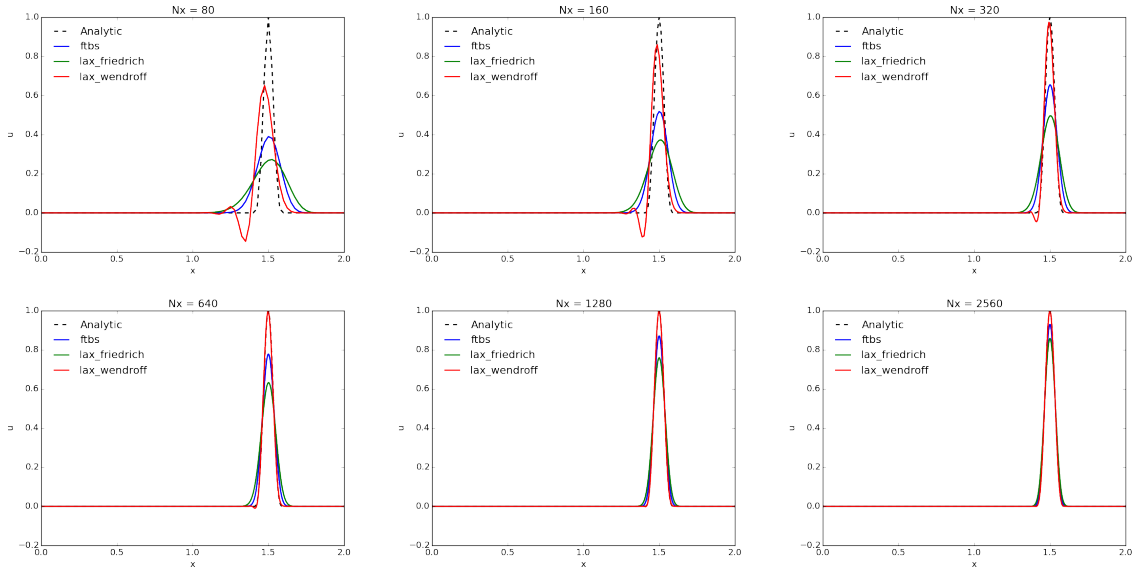


Figure 7.7: The solutions at time t=T=1 for different grids corresponding to the code-example above. A sin^4 wave with period $T = 0.4$ travelling with wavespeed $a = 1$. The solutions were calculated with a cfl-number of 0.8

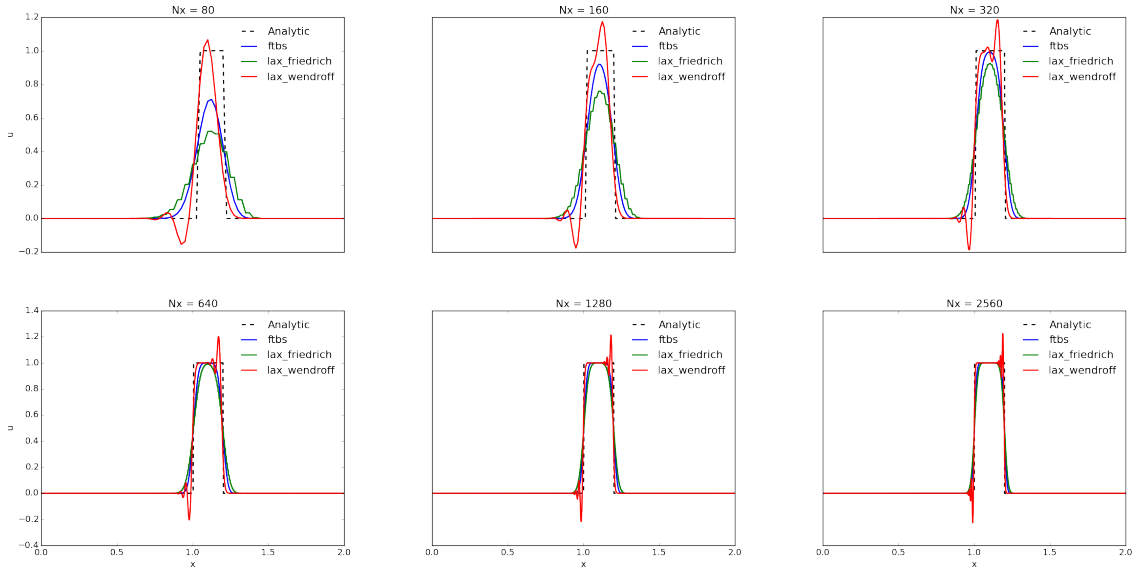


Figure 7.8: Effect of refining grid on a solution containing discontinuities; a square box moving with wave speed a=1. Solutions are showed at t=1, using a cfl-number of 0.8.

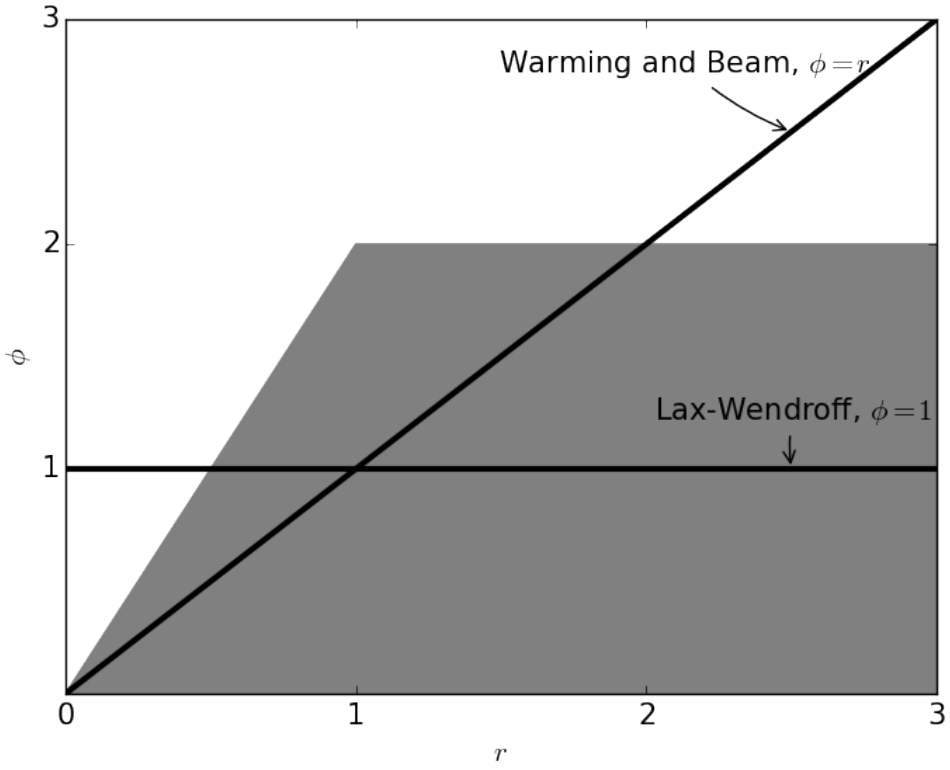


Figure 7.9: TVD region for flux limiters (shaded), and the limiters for the second order schemes; Lax Wendroff, and Warming and Beam

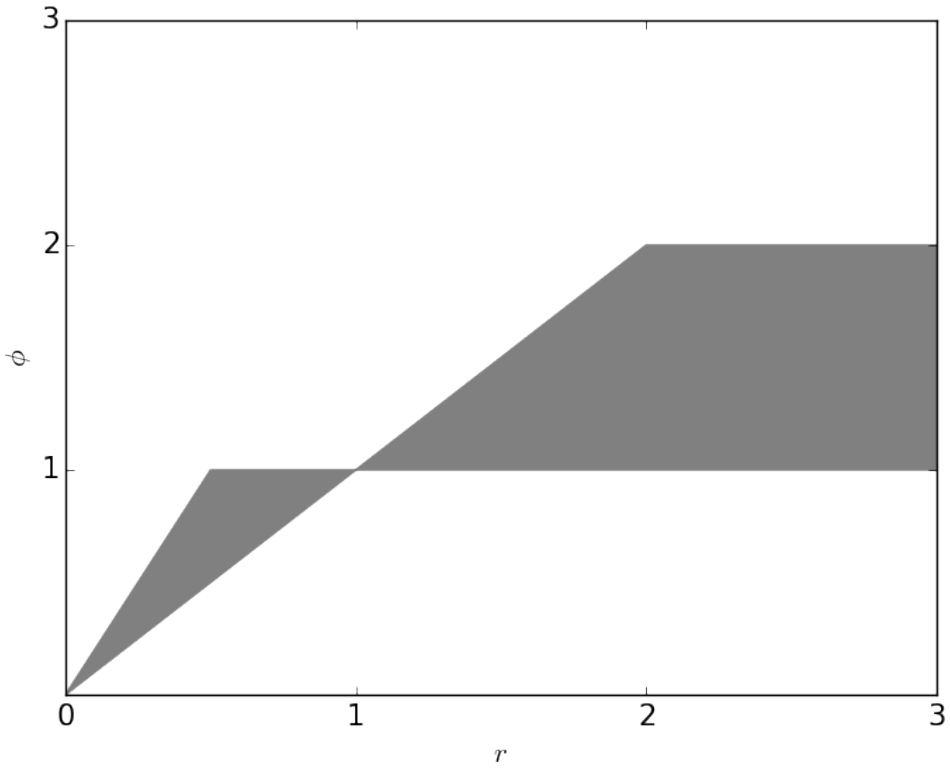


Figure 7.10: Second orderTVD region for flux limiters; sweby diagram **Fredrik 22**: cite sweby?

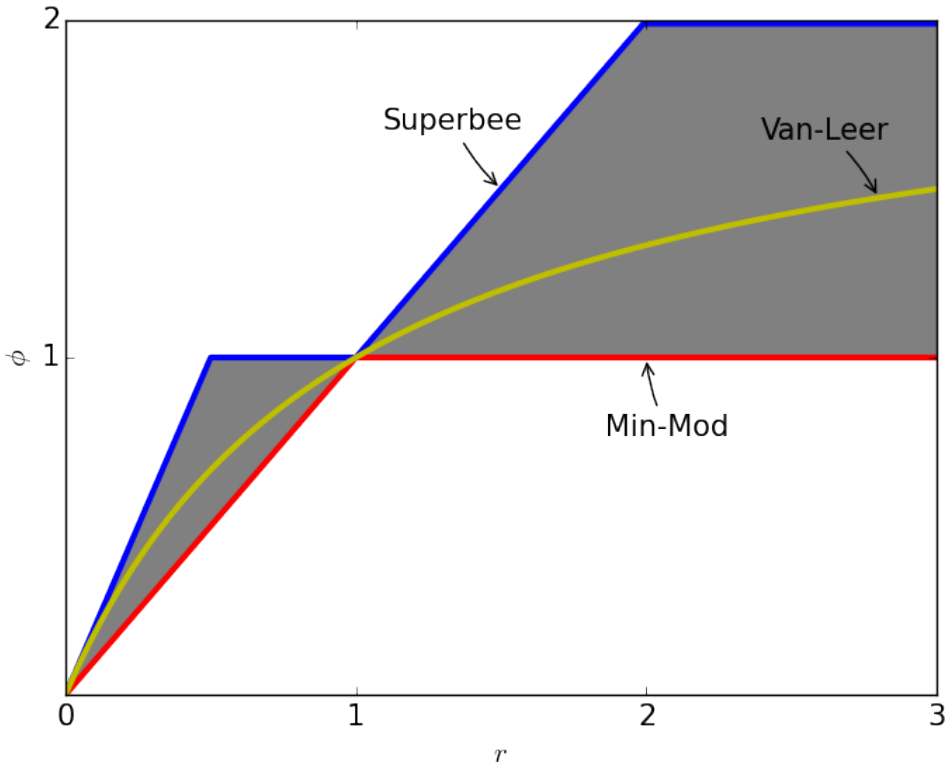


Figure 7.11: Second order TVD limiters

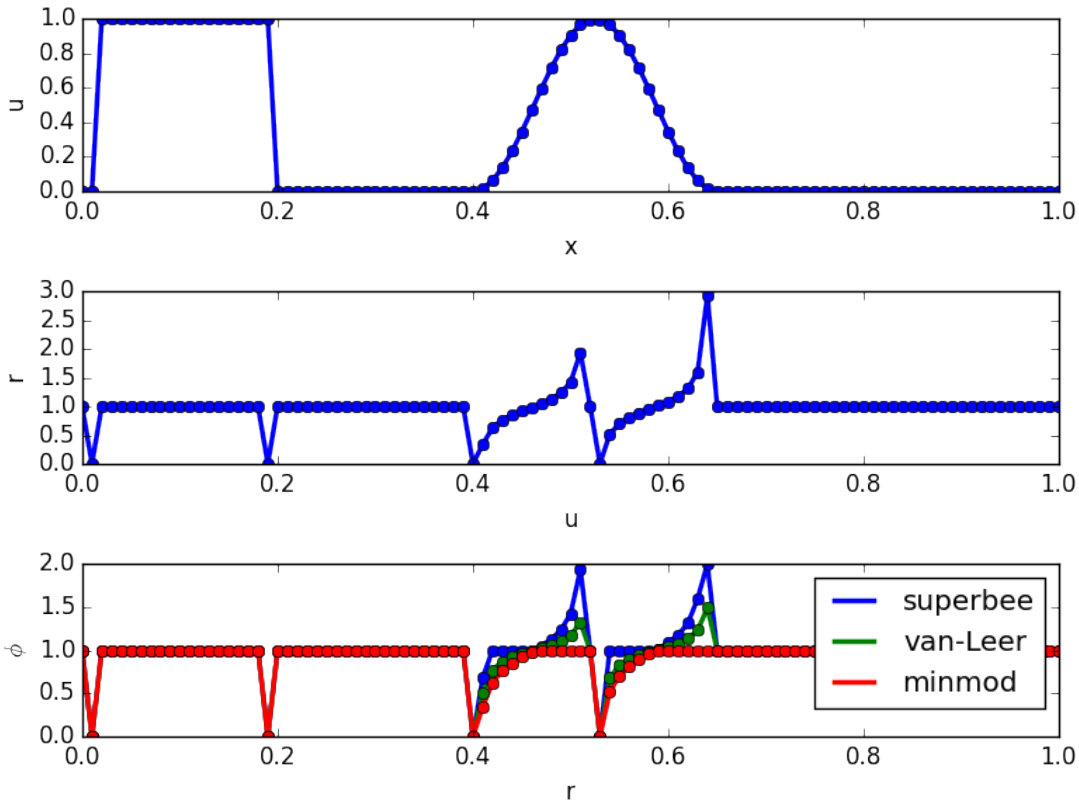


Figure 7.12: Relationship between u and smoothness measure r, and different limiters ϕ, on continuos and discontinuous solutions

7.2 Example: Burgers equation

The 1D Burgers 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:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0 \tag{7.52}$$

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:

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\left(\frac{u^2}{2}\right) = 0 \tag{7.53}$$

and by introducing a flux function

$$F(u) = \frac{u^2}{2} \tag{7.54}$$

the conservative formulation of the Burgers equation may be represented by a generic transport equation:

$$\frac{\partial u}{\partial t} + \frac{\partial F(u)}{\partial x} = 0 \tag{7.55}$$

7.2.1 upwind

The upwind scheme for the general conservation equation take the form:

$$u_j^{n+1} = u_j^n - \frac{\Delta t}{\Delta x}\left(F(u_{j+1}^n) - F(u_j^n)\right) \tag{7.56}$$

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 $\frac{1}{\Delta x}\left(F(u_j^n) - F(u_{j-1}^n)\right)$

7.2.2 lax-Friedrich

The lax-Friedrich conservation equation take the form as given in (7.10), repeated here for convinience:

$$u_j^{n+1} = \frac{\Delta t}{2}(u_{j+1}^n + u_{j-1}^n) - \frac{F_{j+1}^n - F_{j-1}^n}{2\Delta x} \tag{7.57}$$

7.2.3 Lax-Wendroff

As outlined in ((7.1.5)), the general Lax-Wendroff two step method takes the form as given in (7.23) and (7.24) repeated here for convinience:

First step:

$$u_{j+1/2}^{n+1/2} = \frac{1}{2}\left(u_{j+1}^n + u_j^n\right) - \frac{\Delta t}{2\Delta x}\left(F(u_{j+1}^n) - F(u_j^n)\right) \tag{7.58}$$

$$u_{j-1/2}^{n+1/2} = \frac{1}{2}\left(u_j^n + u_{j-1}^n\right) - \frac{\Delta t}{2\Delta x}\left(F(u_j^n) - F(u_{j-1}^n)\right) \tag{7.59}$$

$$u_j^{n+1} = u_j^n - \frac{\Delta t}{\Delta x}\left(F(u_{j+1/2}^{n+1/2}) - F(u_{j-1/2}^{n+1/2})\right) \tag{7.60}$$

In the previous section ((7.1.5)) 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 (7.55). However for the nonlinear case (7.61) no longer holds. This may be overcome be rewriting (7.61):

$$\begin{aligned}\left.\frac{\partial u}{\partial t}\right|_j^n &= -\left.\frac{\partial F}{\partial x}\right|_j^n \qquad \text{and} \qquad \left.\frac{\partial^2 u}{\partial t^2}\right|_j^n = -\left.\frac{\partial^2 F(u)}{\partial t \partial x}\right|_j^n = -\left.\frac{\partial\left(\frac{\partial F(u)}{\partial t}\right)}{\partial x}\right|_j^n \\ &= -\left.\frac{\partial\left(\frac{\partial F(u)}{\partial u}\frac{\partial u}{\partial t}\right)}{\partial x}\right|_j^n = \left.\frac{\partial\left(a(u)\frac{\partial F}{\partial x}\right)}{\partial x}\right|_j^n\end{aligned} \tag{7.61}$$

Now inserting into the Taylor series we get

$$u_j^{n+1} = u_j^n - \Delta t\frac{\partial F(u)}{\partial x} + \frac{\Delta t^2}{2}\frac{\partial\left(a(u)\frac{\partial F}{\partial x}\right)}{\partial x} \tag{7.62}$$

and further we may obtain the general Lax-Wendroff one-step method for a generic transport equation

$$u_j^{n+1} = u_j^n - \frac{\Delta t}{2\Delta x}\left(F_{j+1} - F_{j-1}\right) + \frac{\Delta t^2}{2\Delta x^2}\left[a_{j+1/2}\left(F_{j+1} - F_j\right) - a_{j-1/2}\left(F_j - F_{j-1}\right)\right] \tag{7.63}$$

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:

$$a_{j+1/2} = \frac{1}{2}\left(u_j^n + u_{j+1}^n\right) \tag{7.64}$$

for the burger equation. Another method that assure conservation is to use the following approximation

$$a_{j+1/2} = \begin{cases} \frac{F_{j+1}^n - F_j^n}{u_{j+1}^n - u_j^n} & if \quad u_{j+1} \neq u_j \\ u_j & otherwise \end{cases} \tag{7.65}$$

7.2.4 MacCormack

The MacCormack scheme was discussed in ((7.1.5)) and is given by (7.68) repeated her for convinience

$$u_j^p = u_j^n + \frac{\Delta t}{\Delta x}\left(F_j^n - F_{j+1}^n\right) \tag{7.66}$$

$$u_j^{n+1} = \frac{1}{2}\left(u_j^n + u_j^p\right) + \frac{1}{2}\frac{\Delta t}{\Delta x}\left(F_{j-1}^p - F_j^p\right) \tag{7.67}$$

$$\tag{7.68}$$

7.2.5 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

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = Q \tag{7.69}$$

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 (7.2.5).

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	$u_{j+1/2}^{n+1/2} = \frac{1}{2}\left(u_j^n + u_{j+1}^n\right) - \frac{\Delta t}{2\Delta x}\left(F(u_{j+1}^n) - F(u_j^n)\right) + \frac{\Delta t}{2}Q_{j+1/2}^n$ $u_{j-1/2}^{n+1/2} = \frac{1}{2}\left(u_j^n + u_{j-1}^n\right) - \frac{\Delta t}{2\Delta x}\left(F(u_j^n) - F(u_{j-1}^n)\right) + \frac{\Delta t}{2}Q_{j-1/2}^n$ $u_j^{n+1} = u_j^n - \frac{\Delta t}{\Delta x}\left(F(u_{j+1/2}^{n+1/2}) - F(u_{j-1/2}^{n+1/2})\right) + \Delta t Q_j^n$	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[2:]))/2.0
    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+0.5dt,j-0.5dx
    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+0.5dt,j+0.5dx,

    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]
```

Chapter 8

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. [12].

8.1 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.

8.2 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()
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:

```
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
```

Consider also the parabolic function which may describe the velocity profile for fully developed flow in a cylinder.

```
v0, r =sympy.symbols('v0 r')
v = v0*(1-r**2)
Q=sympy.integrate(v,r)
```

```
Q=sympy.factor(Q)
```

```
A=sympy.Function('A')(x)
```

```
mu,d =sympy.symbols('mu d')
sympy.integrate(A**2*A.diff(x),x)
```

```
sympy.integrate(8*sympy.pi*mu*d**2*Q,x)
```

Bibliography

[1] D.A. Anderson. *Computational Fluid Mechanics and Heat Transfer*. Taylor & Francis, 1997.

[2] P.W. Bearman and J.K. Harvey. Golf ball aerodynamics. *Aeronaut Q*, 27(pt 2):112–122, 1976. cited By 119.

[3] William L. Briggs. *A Multigrid Tutorial*. SIAM, 2. edition, 2000.

[4] E. Cheney and David Kincaid. *Numerical Mathematics and Computing*. Cengage Learning, 4th edition, 1999.

[5] E. Cheney and David Kincaid. *Numerical Mathematics and Computing 7th*. Cengage Learning, 7th edition, 2012.

[6] H. Evans. *Laminar Boundary-Layer Theory*. Addison-Wesley Publishing Company, 1968.

[7] J. Evett and C. Liu. *2,500 Solved Problems in Fluid Mechanics and Hydraulics*. Schaum's Solved Problems Series. McGraw-Hill Education, 1989.

[8] G.E. Forsythe. *Computer Methods for Mathematical Computations*. Prentice-Hall, 1977.

[9] Louis A. Hageman and David M. Young. *Applied Iterative Methods*. Academic Press, 1981.

[10] Ernst Hairer, Syvert Paul Norsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*, volume 1. Springer Science & Business, 2008.

[11] C. Hirsch. *Numerical Computation of Internal and External Flows*. Elsevier, 2007.

[12] Robert Johansson. *Numerical Python. a Practical Techniques Approach for Industry*. Springer, 2015.

[13] C. T. Kelley. *Iterative Methods for Linear and Nonlinear Equations*. SIAM, 1995.

[14] Hans Petter Langtangen. *A Primer on Scientific Programming With Python*. Springer, Berlin; Heidelberg; New York, fourth edition, 2011.

[15] 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.

[16] R. W. MacCormack. The effect of viscosity in hypervelocity impact cratering. *Astronautics, AIAA*, 69:354, 1969.

[17] W. H. Press. *Numerical Recipes in Fortran. the Art of Scientific Computing*, volume 1 & 2. Cambridge University Press, 2. edition, 1992.

[18] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2. edition, 2003.

[19] H. Schlichting. *Boundary Layer Theory*. McGraw-Hill, 7th edition, 1978.

[20] G. D. Smith. *Numerical Solution of Partial Diff. Equations : Finite Difference Methods*. Oxford, 3. edition, 1985.

[21] F.M. White. *Viscous Fluid Flow*. WCB/McGraw-Hill, 1991.

[22] F.M. White. *Fluid Mechanics*. McGraw-Hill series in mechanical engineering. WCB/McGraw-Hill, 1999.

Index