Numerical Methods for Engineers

A digital compendium

Johan Kolstø Sønstabø[1]

Leif Rune Hellevik[1]

[1]Department of Structural Engineering, NTNU

May 28, 2015

Contents

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[1] at UiO (hpl@simula.no), who has developed Doconce[2] for flexible typesetting, and associate professor Hallvard Trætteberg[3] 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.[4]

0.1 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[5] before proceeding. If you are familiar with **Matlab** the transfer to Python should not be a problem.

[1] `http://folk.uio.no/hpl/`
[2] `https://github.com/hplgit/doconce`
[3] `http://www.ntnu.edu/employees/hal`
[4] `https://www.ntnu.no/wiki/pages/viewpage.action?pageId=72646826`
[5] `http://lrhgit.github.io/tkt4140/allfiles/digital_compendium/python_intro/doc/src/bumpy.html`

Chapter 1

Chapter 1: Initial value problems for Ordinary Differential Equations

1.1 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.

$$
\begin{aligned}
y'(x) &= f(x, y) \\
y(x_0) &= a
\end{aligned}
\tag{1.1}
$$

while for a second order ODE we get

$$
\begin{aligned}
y''(x) &= f(x, y, y') \\
y(x_0) &= a,\ y'(x_0) = b
\end{aligned}
\tag{1.2}
$$

A first order ODE, as shown in Equation (1.1), will always be an *initial value problem.* For Equation (1.2), 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 (1.2) 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 (1.1), 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 (1.1) is that both $f(x,y)$ and $\frac{\partial f}{\partial y}$ are continuous in and around x_0.

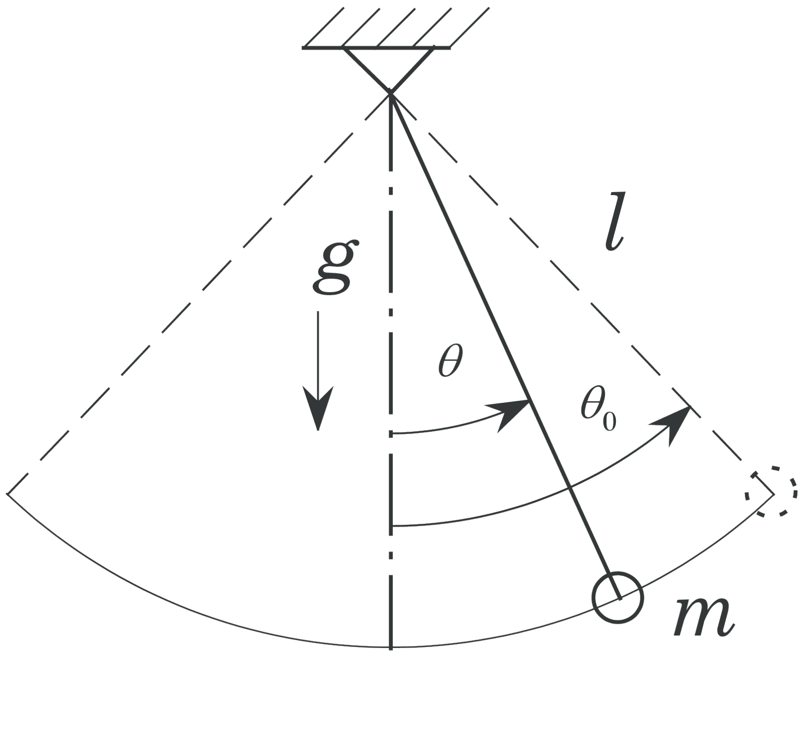
For (1.2) this conditions becomes that $f(x,y)$, $\frac{\partial f}{\partial y}$ and $\frac{\partial f}{\partial y'}$ are continuous in and around x_0. Similarly for higher order equations.

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:

$$\frac{\partial^2\theta}{\partial\tau^2} + \frac{g}{l}\sin(\theta) = 0 \tag{1.3}$$

$$\theta(0) = \theta_0,\ \frac{d\theta}{d\tau}(0) = 0 \tag{1.4}$$

We introduce a dimensionless time t given by $t = \sqrt{\frac{g}{l}} \cdot \tau$ such that (1.3) and (1.4) may be written as

$$\ddot{\theta}(t) + \sin(\theta(t)) = 0 \tag{1.5}$$

$$\theta(0) = \theta_0,\ \dot{\theta}(0) = 0 \tag{1.6}$$

The dot denotes derivation with respect to the dimensionless time t. For small displacements we can set $\sin(\theta) \approx \theta$, such that (1.5) and (1.6) becomes

$$\ddot{\theta}(t) + \theta(t) = 0 \tag{1.7}$$

$$\theta(0) = \theta_0,\ \dot{\theta}(0) = 0 \tag{1.8}$$

The difference between (1.5) and (1.7) is that the latter is linear, while the first is non-linear. The analytical solution of Equations (1.5) and (1.6) is given in Appendix G.2. in the compendium[1]. An n'th order linear ODE may be written on the form

$$a_n(x)y^{(n)}(x) + a_{n-1}(x)y^{(n-1)}(x) + \cdots + a_1(x)y'(x) + a_0(x)y(x) = b(x) \tag{1.9}$$

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 (1.9) it follows that (1.5) is non-linear and (1.7) 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.

[1] `./NumeriskeBeregninger.pdf`

1.2 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 (1.7):

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

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 (1.5)

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

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

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

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

1.3 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

$$y^{(m)} = f(x, y, y', y'', \ldots, y^{(m-1)}) \tag{1.10}$$

$$y(x_0) = a_1,\ y'(x_0) = a_2,\ \ldots, y^{(m-1)}(x_0) = a_m \tag{1.11}$$

where

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

with $y = y_1$, we set

$$\begin{aligned}
&y_1' = y_2\\
&y_2' = y_3\\
&.\\
&.\\
&y_{m-1}' = y_m\\
\\
&y_1(x_0) = a_1, y_2(x_0) = a_2, \dots, y_m(x_0) = a_m
\end{aligned}$$

1.3.1 Example: Reduction of higher order systems

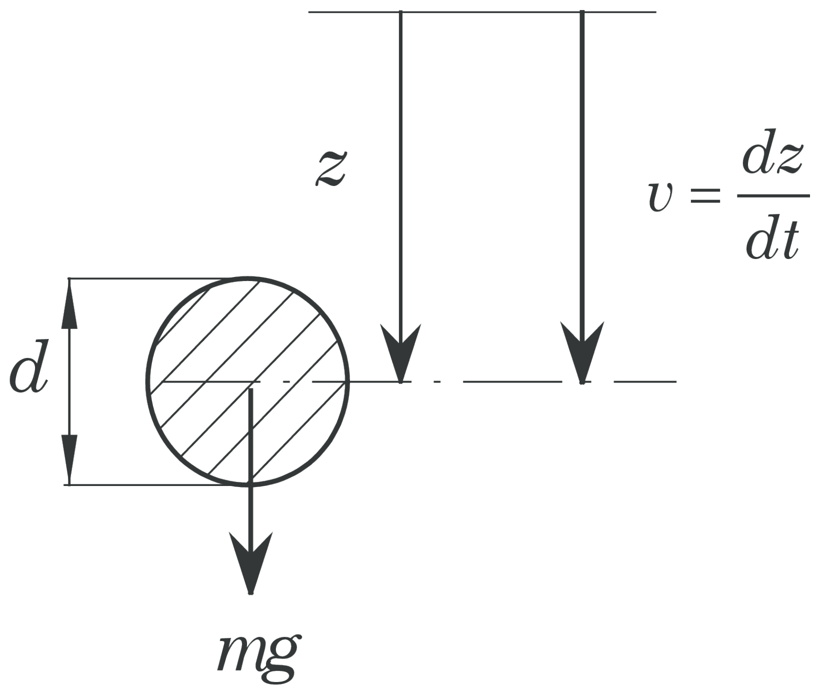
Write the following ODE as a system of first order ODEs:

$$y''' - y'y'' - (y')^2 + 2y = x^3$$
$$y(0) = a,\ y'(0) = b,\ y''(0) = c$$

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

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

1.3.2 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

$$m\frac{dv}{dt} = mg - m_f g - \frac{1}{2}m_f\frac{dv}{dt} - \frac{1}{2}\rho_f v\,|v|\,A_k C_D, \tag{1.12}$$

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 [8], page 539-540. To write Equation (1.12) on a more convenient form we introduce the following abbreviations:

$$\rho = \frac{\rho_f}{\rho_k},\ A = 1 + \frac{\rho}{2},\ B = (1-\rho)g,\ C = \frac{3\rho}{4d}. \tag{1.13}$$

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 (1.12) may then be written as

$$\frac{dv}{dt} = \frac{1}{A}(B - C\cdot v\,|v|\,C_d). \tag{1.14}$$

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 (1.14),

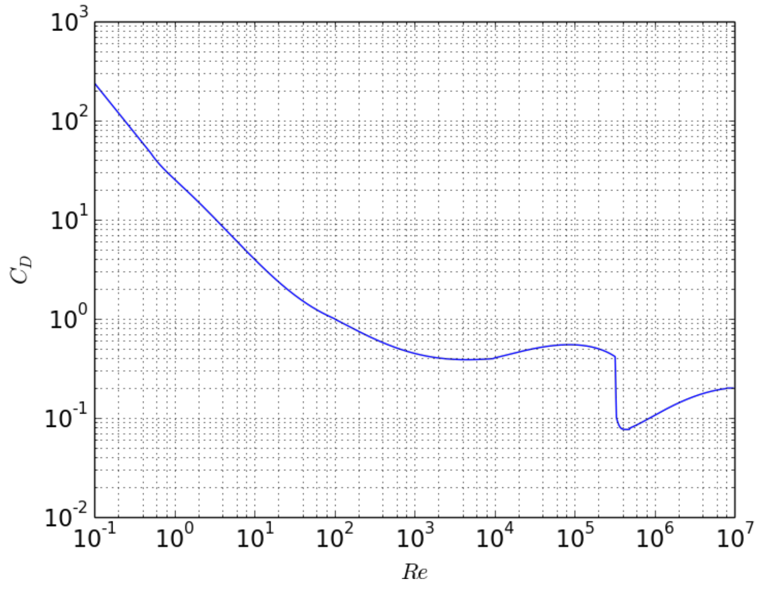


Figure 1.1: Drag coefficient C_D as function of the Reynold's number R_e.

we get a 2nd order ODE as follows

$$\frac{d^2z}{dt^2} = \frac{1}{A}\left(B - C \cdot \frac{dz}{dt}\left|\frac{dz}{dt}\right|C_d\right) \tag{1.15}$$

For Equation (1.15) two initial conditions must be specified, e.g. $v = v_0$ and $z = z_0$ for $t = 0$.

Figure 1.1 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 (1.14). We will use the function $C_D(Re)$ as an example of how functions may be implemented in Python.

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 [3]. 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 (1.14) and (1.15).

```
# chapter1/programs_and_modules/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 numbe
    # 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[2] for details about the different functions.

[2] http://www.numpy.org

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[3].

1.4 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:

[3] http://matplotlib.org/api/pyplot_summary.html

```
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) #conditio

    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 1.5.

1.5 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 (1.3.2). 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 [5], 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` [5], 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[4] blocks:

[4]`https://wiki.python.org/moin/HandlingExceptions`

```
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[5], 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[6] 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/programs_and_modules/DragCoefficientGeneric.py

from numpy import linspace,array,append,logspace,zeros_like,where,vectorize,\
```

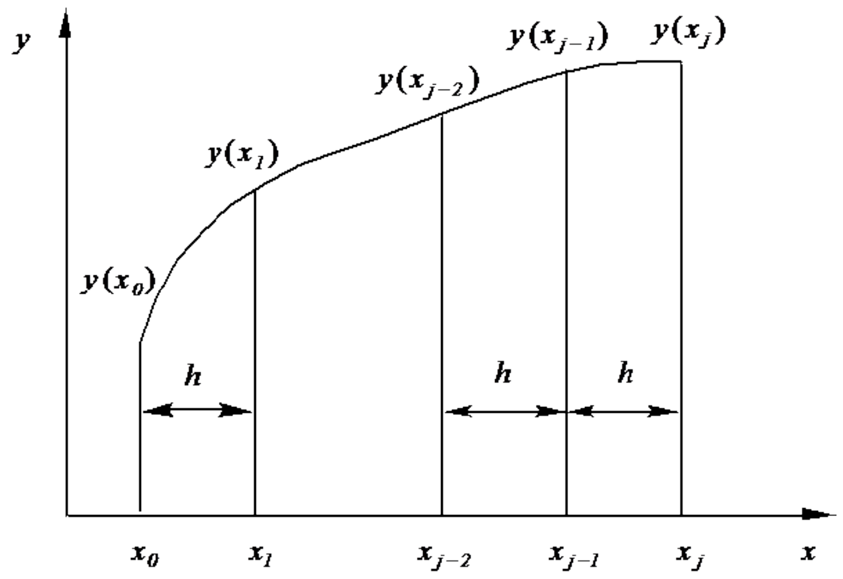
[5]https://docs.python.org/2/library/time.html
[6]http://stackoverflow.com/questions/575819/sorting-dictionary-keys-in-python

```
    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()
```

1.6 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:

$$\nabla y_j = y_j - y_{j-1} \tag{1.16}$$

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**:

$$\begin{aligned} y(x) =& y(x_0) + y'(x_0) \cdot (x - x_0) + \frac{1}{2} y''(x_0) \cdot (x - x_0)^2 + \\ & \cdots + \frac{1}{n!} y^{(n)}(x_0) \cdot (x - x_0)^n + R_n \end{aligned} \tag{1.17}$$

The remainder R_n is given by

$$R_n = \frac{1}{(n+1)!} y^{(n+1)}(\xi) \cdot (x - x_0)^{n+1} \tag{1.18}$$

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

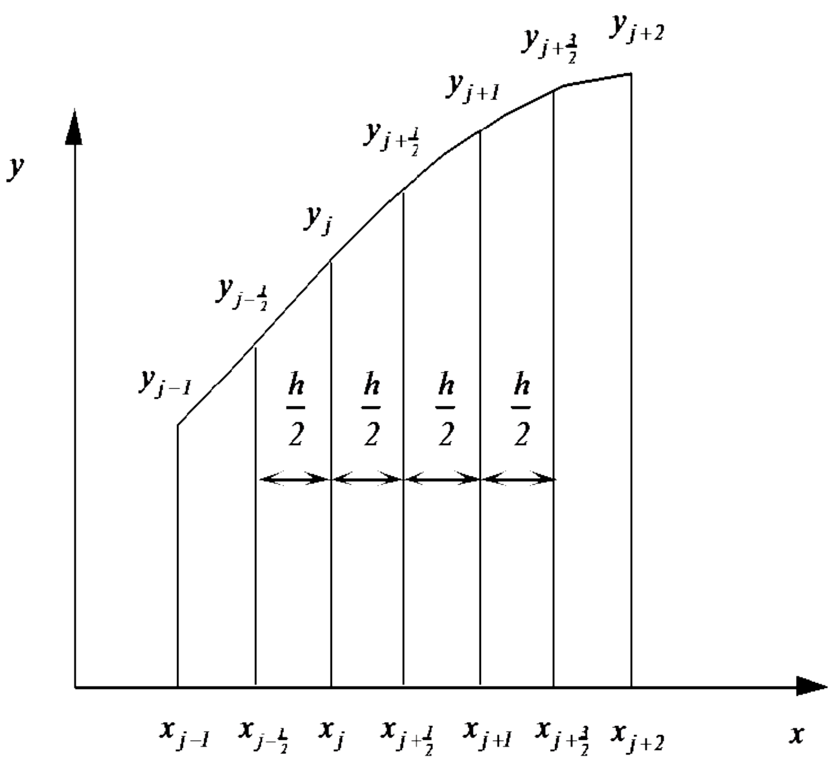


Figure 1.2: Illustration of how to obtain difference equations.

By use of (1.17) we get

$$y(x_{j+1}) \equiv y(x_j + h) = y(x_j) + hy'(x_j) + \frac{h^2}{2}y''(x_j) + \cdots + \frac{h^n y^{(n)}(x_j)}{n!} + R_n \tag{1.19}$$

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

From (1.19) we also get

$$y(x_{j-1}) \equiv y(x_j - h) = y(x_j) - hy'(x_j) + \frac{h^2}{2}y''(x_j) + \cdots + \frac{h^k(-1)^k y^{(k)}(x_j)}{k!} + \ldots \tag{1.20}$$

We will here and subsequently assume that h is positive.

We solve (1.19) with respect to y':

$$y'(x_j) = \frac{y(x_{j+1}) - y(x_j)}{h} + O(h) \tag{1.21}$$

We solve (1.20) with respect to y':

$$y'(x_j) = \frac{y(x_j) - y(x_{j-1})}{h} + O(h) \tag{1.22}$$

By addition of (1.20) and (1.19) we get

$$y''(x_j) = \frac{y(x_{j+1}) - 2y(x_j) + y(x_{j-1})}{h^2} + O(h^2) \tag{1.23}$$

By subtraction of (1.20) from (1.19) we get

$$y'(x_j) = \frac{y(x_{j+1}) - y(x_{j-1})}{2h} + O(h^2) \tag{1.24}$$

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 (1.21), (1.22), (1.23) and (1.24) then gives the following difference expressions:

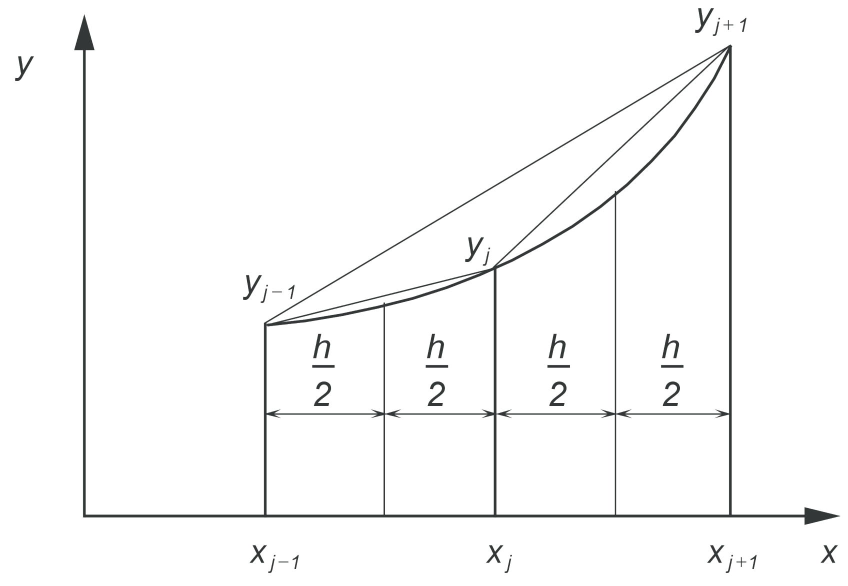
$$y'_j = \frac{y_{j+1} - y_j}{h} \ ; \text{ truncation error } O(h) \tag{1.25}$$

$$y'_j = \frac{y_j - y_{j-1}}{h} \ ; \text{ truncation error } O(h) \tag{1.26}$$

$$y''_j = \frac{y_{j+1} - 2y_j + y_{j-1}}{h^2} \ ; \text{ truncation error } O(h^2) \tag{1.27}$$

$$y'_j = \frac{y_{j+1} - y_{j-1}}{2h} \ ; \text{ truncation error } O(h^2) \tag{1.28}$$

(1.25) is a forward difference, (1.26) is a backward difference while (1.27) and (1.28) are central differences.



The expressions in (1.25), (1.26), (1.27) and (1.28) are easily established from the figure.

(1.25) follows directly.

(1.27):

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

(1.28):

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

To find the truncation error we must use the Taylor series expansion.

The derivation above may be done more systematically. We set

$$y'(x_j) = a \cdot y(x_{j-1}) + b \cdot y(x_j) + c \cdot y(x_{j+1}) + O(h^m) \tag{1.29}$$

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 (1.19) and (1.20) 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:

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

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

$$\begin{aligned}
a + b + c &= 0 \\
(c - a) \cdot h &= 1 \\
a + c &= 1
\end{aligned} \tag{1.30}$$

The solution to (1.30) is

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

which when inserted in (1.29) gives

$$y_j' = \frac{y_{j+1} - y_{j-1}}{2h} - \frac{h^2}{6} y'''(\xi) \tag{1.31}$$

Comparing (1.31) with (1.29) we see that the error term is $O(h^m) = -\frac{h^2}{6}y'''(\xi)$, which means that $m = 2$. As expected, (1.31) is identical to (1.24).

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,

$$y'(x_j) = a \cdot y_j + b \cdot y_{j+1} + c \cdot y_{j+2} + O(h^m) \tag{1.32}$$

The procedure goes as in the previous example as follows,

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

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

$$\begin{aligned}
a + b + c &= 0 \\
(b + 2c) \cdot h &= 1 \\
\frac{b}{2} + 2c &= 0
\end{aligned} \tag{1.33}$$

The solution of (1.33) is

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

which inserted in (1.32) gives

$$y'_j = \frac{-3y_j + 4y_{j+1} - y_{j+2}}{2h} + \frac{h^2}{3}y'''(\xi) \tag{1.34}$$

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 1.2) as follows,

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

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

$$\frac{d}{dx}\left[p(x)\cdot\frac{d}{dx}u(x)\right]\Bigg|_i \approx \frac{p_{i-\frac{1}{2}}\cdot u_{i-1} - (p_{i+\frac{1}{2}}+p_{i-\frac{1}{2}})\cdot u_i + p_{i+\frac{1}{2}}\cdot u_{i+1}}{h^2} + \text{error term} \tag{1.35}$$

where

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

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

$$p(x_{1+\frac{1}{2}}) \approx \frac{1}{2}(p_{i+1}+p_i),\ p(x_{1-\frac{1}{2}}) \approx \frac{1}{2}(p_i + p_{i-1}) \tag{1.36}$$

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

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

which gives

$$\frac{d}{dx}\left[p(x)\cdot\frac{du}{dx}\right]\Bigg|_i = \frac{2\cdot[p(x_{i+\frac{1}{2}})\cdot(u_{i+1}-u_i) - h\cdot p(x_i)\cdot u'(x_i)]}{h^2} + \text{error term} \tag{1.37}$$

where

$$\text{error term} = -\frac{h}{12}[3p''(x_i)\cdot u'(x_i) + 6p'(x_i)\cdot u''(x_i) + 4p(x_i)\cdot u'''(x_i)] + O(h^2) \tag{1.38}$$

We have kept the term $u'(x_i)$ since (1.37) usually is used at the boundary, and $u'(x_i)$ may be prescribed there. For $p(x) = 1 = $ constant we get the expression

$$u_i'' = \frac{2\cdot[u_{i+1} - u_i - h\cdot u'(x_i)]}{h^2} - \frac{h}{3}u'''(x_i) + O(h^2) \tag{1.39}$$

Backward Differences: We start with

$$\frac{d}{dx}\left[p(x)\frac{du}{dx}\right]\bigg|_i \approx \frac{p(x_i)\cdot u'(x_i) - p(x_i)\cdot u'(x_{i-\frac{1}{2}})\cdot u'(x_{i-\frac{1}{2}})}{\frac{h}{2}}$$

$$\approx \frac{p(x_i)\cdot u'(x_i) - p(x_{i-\frac{1}{2}})\left(\frac{u_i-u_{i-1}}{h}\right)}{\frac{h}{2}}$$

which gives

$$\frac{d}{dx}\left[p(x)\frac{du}{dx}\right]\bigg|_i = \frac{2\cdot[h\cdot p(x_i)u'(x_i) - p(x_{i-\frac{1}{2}})\cdot(u_i-u_{i-1})]}{h^2} + \text{error term} \tag{1.40}$$

where

$$\text{error term} = \frac{h}{12}[3p''(x_i)\cdot u'(x_i)+6p'(x_i)\cdot u''(x_i)+4p(x_i)\cdot u'''(x_i)]+O(h^2) \tag{1.41}$$

This is the same error term as in (1.38) except from the sign. Also here we have kept the term $u'(x_i)$ since (1.41) usually is used at the boundary where $u'(x_i)$ may be prescribed. For $p(x) = 1 = \text{constant}$ we get the expression

$$u_i'' = \frac{2\cdot[h\cdot u'(x_i) - (u_i-u_{i-1})]}{h^2} + \frac{h}{3}u'''(x_i) + O(h^2) \tag{1.42}$$

1.7 Euler's method

The ODE is given as

$$\frac{dy}{dx} = y'(x) = f(x,y) \tag{1.43}$$

$$y(x_0) = y_0 \tag{1.44}$$

By using a first order forward approximation (1.21) of the derivative in (1.43) we obtain:

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

or

$$y_{n+1} = y_n + h\cdot f(x_n, y_n) \tag{1.45}$$

(1.45) is a difference equation and the scheme is called **Euler's method** (1768). The scheme is illustrated graphically in Figure 1.3. 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.

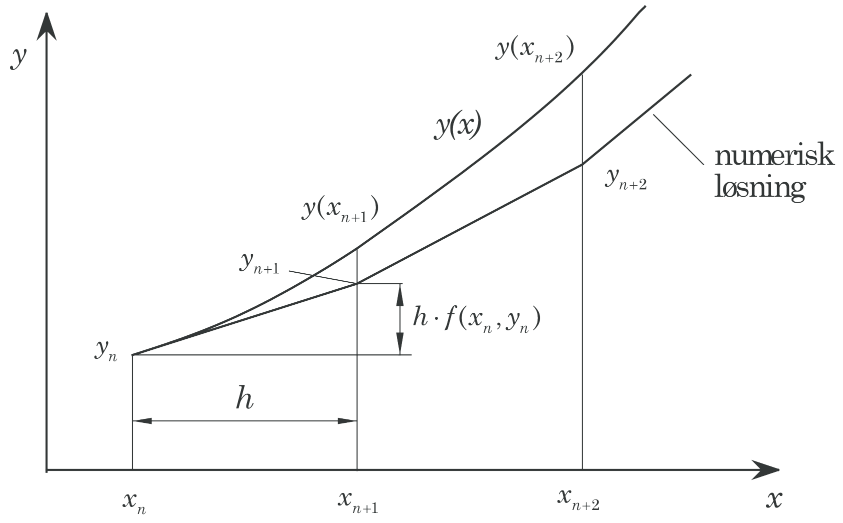


Figure 1.3: Graphical illustration of Euler's method.

1.7.1 Example: Falling sphere with constant and varying drag

We write (1.14) and (1.15) as a system as follows,

$$\frac{dz}{dt} = v \tag{1.46}$$

$$\frac{dv}{dt} = g - \alpha v^2 \tag{1.47}$$

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

$$z(t) = \frac{\ln(\cosh(\sqrt{\alpha g} \cdot t)}{\alpha} \tag{1.48}$$

$$v(t) = \sqrt{\frac{g}{\alpha}} \cdot \tanh(\sqrt{\alpha g} \cdot t) \tag{1.49}$$

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

$$z(t) =\frac{1}{2}gt^2 \cdot (1 - \frac{1}{6}\alpha g t^2) \tag{1.50}$$

$$v(t) =gt \cdot (1 - \frac{1}{3}\alpha g t^2) \tag{1.51}$$

The Euler scheme (1.45) used on (1.47) gives

$$v_{n+1} = v_n + \Delta t \cdot (g - \alpha \cdot v_n^2),\ n = 0, 1, \ldots \tag{1.52}$$

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 1.4. 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 (1.3.2). The complete program is as follows,

```
# chapter1/programs_and_modules/FallingSphereEuler.py;DragCoefficientGeneric.py @ git@lrhgit/tkt
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)')
```

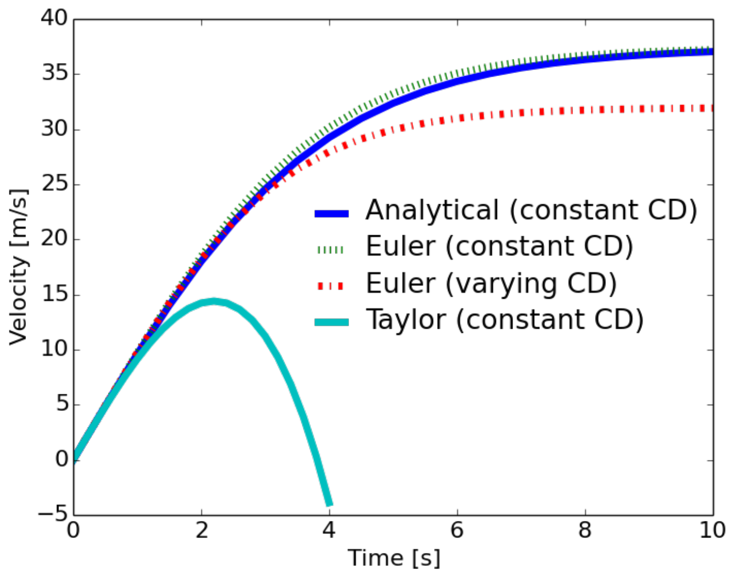


Figure 1.4: Euler's method with $\Delta t = 0.5$ s.

```
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()
```

1.7.2 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/programs_and_modules/Euler_timestep_ctrl.py;DragCoefficientGeneric.py @ git@lrhgit/tkt4140/allfi
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 (1.5). 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 (1.5) yields $d_{i+1} \approx 0.3$, from which we may deduce:

$$\log 10\frac{\epsilon_i}{\epsilon_{i+1}} \approx 0.3 \Rightarrow \epsilon_{i+1} \approx 10^{-0.3}\,\epsilon_i \approx 0.501\,\epsilon_i \tag{1.53}$$

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

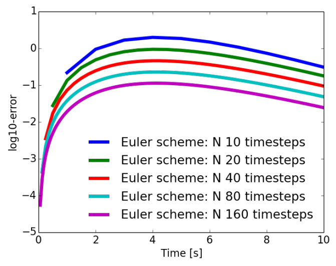


Figure 1.5: Plots for the logarithmic errors for a falling sphere with constant drag. The timestep Δt is reduced by a factor two from one curve to the one immediately below.

even slightly more than the theoretically expected value for a first order scheme, i.e. $\Delta t_{i+1} = \Delta t_i/2$ yields $\epsilon_{i+1} \approx \epsilon_i/2$.

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

$$\begin{aligned} y_1' &= f_1(x, y_1, y_2, \dots y_p) \\ y_2' &= f_2(x, y_1, y_2, \dots y_p) \\ &. \\ &. \\ y_p' &= f_p(x, y_1, y_2, \dots y_p) \end{aligned} \tag{1.54}$$

with initial values

$$y_1(x_0) = a_1,\ y_2(x_0) = a_2, \dots,\ y_p(x_0) = a_p \tag{1.55}$$

Or, in vectorial format as follows,

$$\begin{aligned} \mathbf{y}' &= \mathbf{f}(x, \mathbf{y}) \\ \mathbf{y}(x_0) &= \mathbf{a} \end{aligned} \tag{1.56}$$

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

The Euler scheme (1.45) used on (1.56) gives

$$\mathbf{y_{n+1}} = \mathbf{y_n} + h \cdot \mathbf{f}(x_n, \mathbf{y_n}) \tag{1.57}$$

For a system of three equations we get

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= y_3 \\ y_3' &= -y_1 y_3 \end{aligned} \tag{1.58}$$

In this case (1.57) gives

$$\begin{aligned}(y_1)_{n+1} &= (y_1)_n + h\cdot(y_2)_n \\ (y_2)_{n+1} &= (y_2)_n + h\cdot(y_3)_n \\ (y_3)_{n+1} &= (y_3)_n - h\cdot(y_1)_n\cdot(y_3)_n\end{aligned} \tag{1.59}$$

$$\tag{1.60}$$

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

In section 1.3 we have seen how we can reduce a higher order ODE to a set of first order ODEs. In (1.46) and (1.47) 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,

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

1.8 Heun's method

From (1.21) or (1.25) we have

$$y''(x_n, y_n) = f'\left(x_n, y(x_n, y_n)\right) \approx \frac{f(x_n+h) - f(x_n)}{h} \tag{1.61}$$

The Taylor series expansion (1.19) 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 (1.61), gives

$$y_{n+1} = y_n + \frac{h}{2}\cdot[f(x_n, y_n) + f(x_{n+1}, y(x_{n+1}))] \tag{1.62}$$

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 (1.45) to calculate $y(x_{n+1})$ on the right side of (1.62). The resulting scheme is often denoted **Heun's method**.

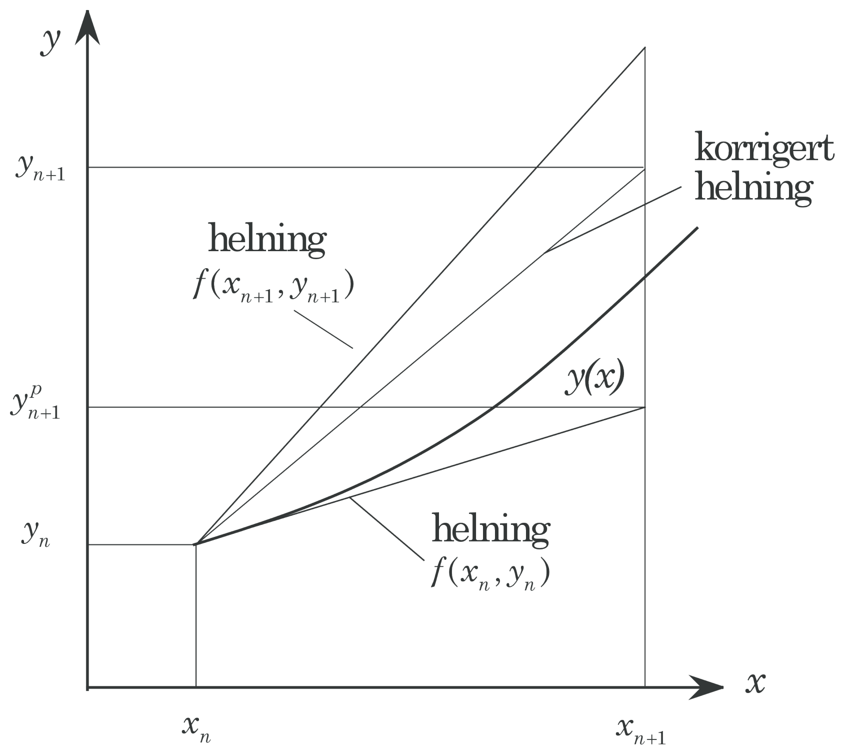


Figure 1.6: Illustration of Heun's method.

The scheme for Heun's method becomes

$$y_{n+1}^p = y_n + h \cdot f(x_n, y_n) \tag{1.63}$$

$$y_{n+1} = y_n + \frac{h}{2} \cdot [f(x_n, y_n) + f(x_{n+1}, y_{n+1}^p)] \tag{1.64}$$

Index p stands for "predicted". (1.63) is then the predictor and (1.64) is the corrector. This is a second order method. For more details, see [2]. Figure 1.6 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

$$\mathbf{y_{n+1}^p} = \mathbf{y_n} + h \cdot \mathbf{f}(x_n, \mathbf{y_n}) \tag{1.65}$$

$$\mathbf{y_{n+1}} = \mathbf{y_n} + \frac{h}{2} \cdot [\mathbf{f}(x_n, \mathbf{y_n}) + \mathbf{f}(x_{n+1}, \mathbf{y_{n+1}^p})] \tag{1.66}$$

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

If we use (1.65) and (1.66) on the example in (1.58) 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:

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

1.8.1 Example: Newton's equation

Let's use Heun's method to solve Newton's equation from section 1.1,

$$y'(x) = 1 - 3x + y + x^2 + xy,\ y(0) = 0 \tag{1.67}$$

with analytical solution

$$
\begin{aligned}
y(x) =& 3\sqrt{2\pi e} \cdot \exp\left(x\left(1+\frac{x}{2}\right)\right) \cdot \left[\operatorname{erf}\left(\frac{\sqrt{2}}{2}(1+x)\right) - \operatorname{erf}\left(\frac{\sqrt{2}}{2}\right)\right] \\
&+4 \cdot \left[1 - \exp\left(x\left(1+\frac{x}{2}\right)\right)\right] - x
\end{aligned} \tag{1.68}
$$

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 1.7.

```
# chapter1/programs_and_modules/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)
```

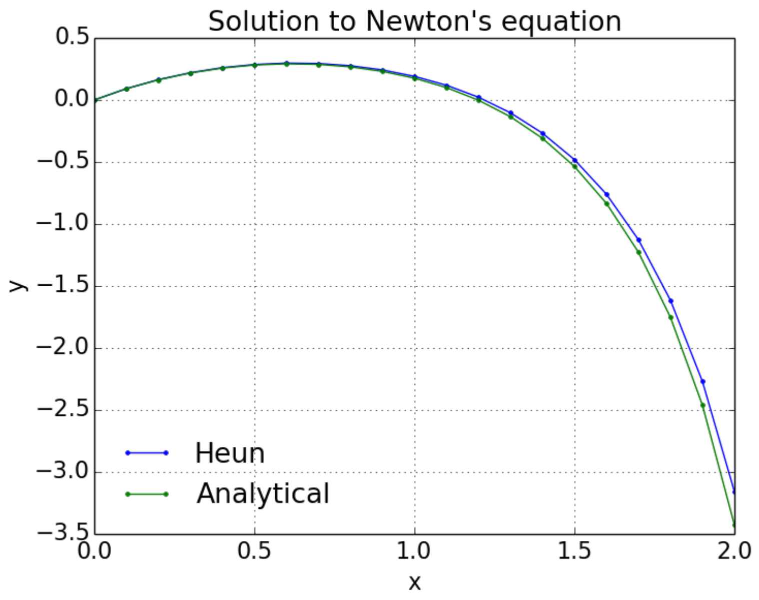


Figure 1.7: Velocity of falling sphere using Euler's and Heun's methods.

```
# 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()
```

1.8.2 Example: Falling sphere with Heun's method

Let's go back to (1.7.1), and implement a new function `heun()` in the program FallingSphereEuler.py[7].

[7]https://lrhgit.github.io/tkt4140/allfiles/digital_compendium/chapter1/programs_and_modules/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 (1.65) and (1.66) becomes
Predictor:

$$\begin{aligned} z^p_{n+1} &= z_n + \Delta t v_n \\ v^p_{n+1} &= v_n + \Delta t \cdot (g - \alpha v_n^2) \end{aligned} \tag{1.69}$$

Corrector:

$$\begin{aligned} z_{n+1} &= z_n + 0.5\Delta t \cdot (v_n + v^p_{n+1}) \\ v_{n+1} &= v_n + 0.5\Delta t \cdot \left[2g - \alpha[v_n^2 + (v^p_{n+1})^2\right] \end{aligned} \tag{1.70}$$

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 (1.7.1), we get the response plotted in Figure 1.8.

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

```
# chapter1/programs_and_modules/FallingSphereEulerHeun.py;ODEschemes.py @ git@lrhgit/tkt4140/all
from DragCoefficientGeneric import cd_sphere
from ODEschemes import euler, heun
from matplotlib.pyplot import *
```

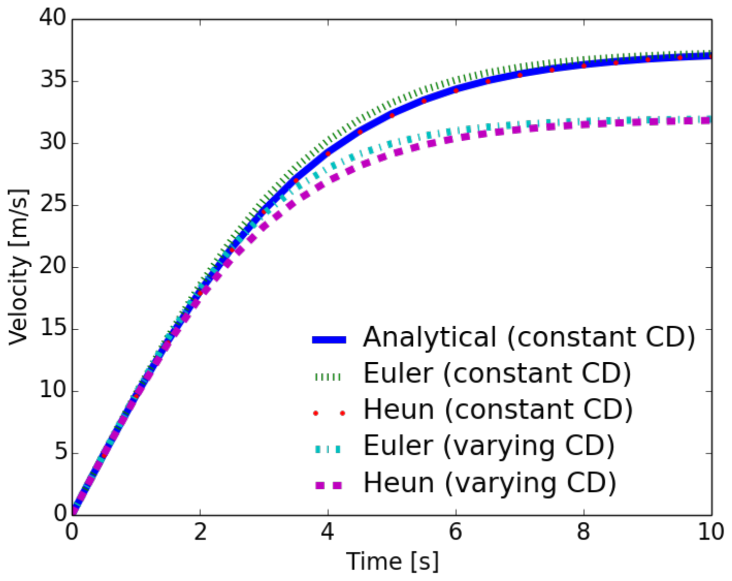


Figure 1.8: Velocity of falling sphere using Euler's and Heun's methods.

```
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()
```

1.9 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

$$\begin{aligned}
k_1 &= f(x_n, y_n)\\
k_2 &= f(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1)\\
k_3 &= f(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_2)\\
k_4 &= f(x_n + h, y_n + hk_3)\\
y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)
\end{aligned} \tag{1.71}$$

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 [2].

Figure 1.9 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 (1.71) on the equation system in (1.58) we get

$$\begin{aligned}
(y_1)_{n+1} &= (y_1)_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)\\
(y_2)_{n+1} &= (y_2)_n + \frac{h}{6}(l_1 + 2l_2 + 2l_3 + l_4) \qquad (1.72)\\
(y_3)_{n+1} &= (y_3)_n + \frac{h}{6}(m_1 + 2m_2 + 2m_3 + m_4)
\end{aligned}$$

(1.73)

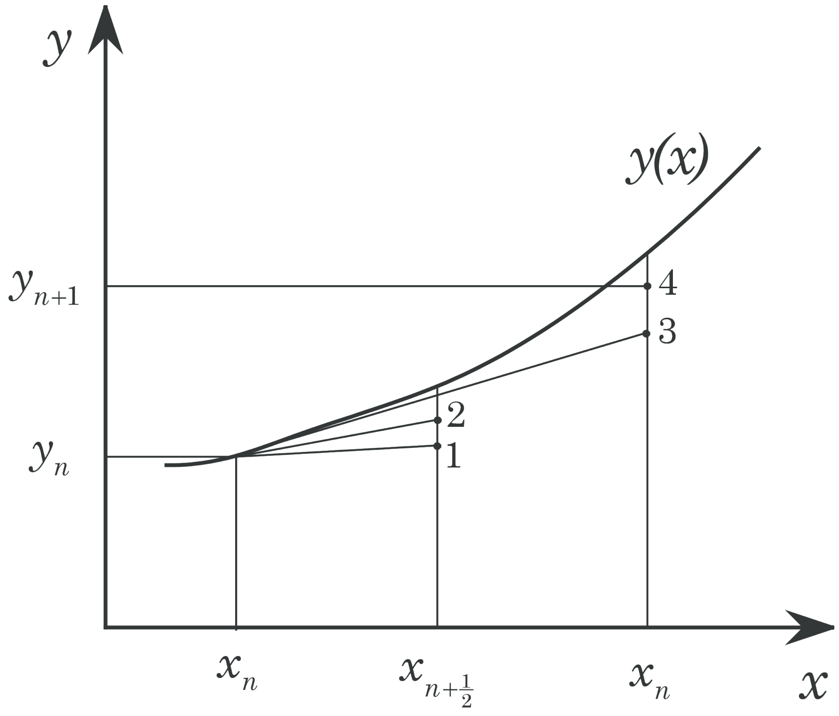


Figure 1.9: Illustration of the RK4 scheme.

where

$$
\begin{aligned}
k_1 &= y_2 \\
l_1 &= y_3 \\
m_1 &= -y_1 y_3 \\
\\
k_2 &= (y_2 + h l_l/2) \\
l_2 &= (y_3 + h m_1/2) \\
m_2 &= -[(y_1 + h k_1/2)(y_3 + h m_1/2)] \\
\\
k_3 &= (y_2 + h l_2/2) \\
l_3 &= (y_3 + h m_2/2) \\
m_3 &= -[(y_1 + h k_2/2)(y_3 + h m_2/2)] \\
\\
k_4 &= (y_2 + h l_3) \\
l_4 &= (y_3 + h m_3) \\
m_4 &= -[(y_1 + h k_3)(y_3 + h m_3)
\end{aligned}
$$

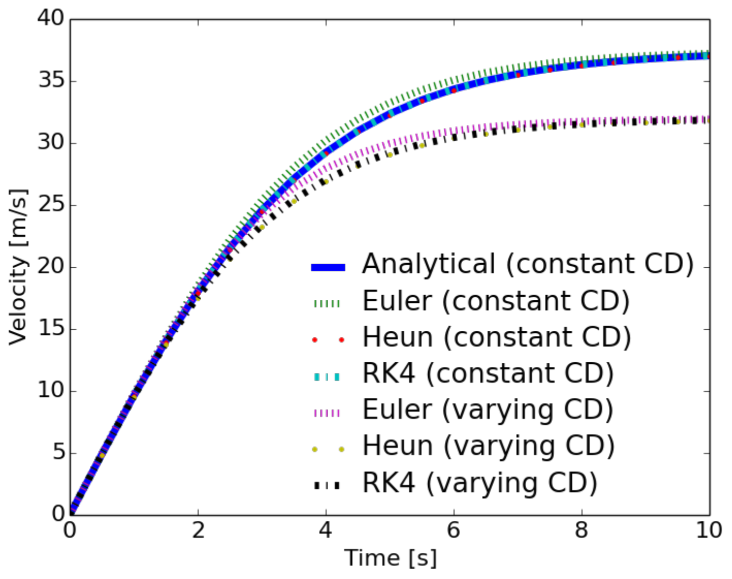


Figure 1.10: Velocity of falling sphere using Euler, Heun and RK4.

1.9.1 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 1.10 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**.

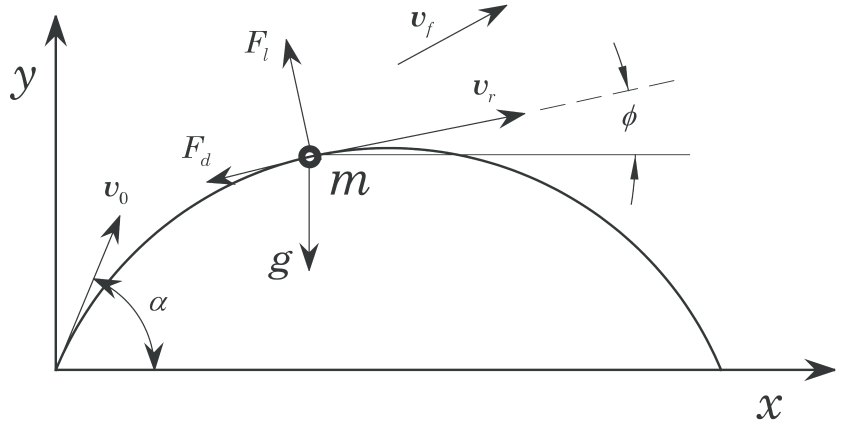
```
# chapter1/programs_and_modules/FallingSphereEulerHeunRK4.py;ODEschemes.py @ git@lrhgit/tkt4140/

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()
```

1.9.2 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 1.3.2), 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

$$\mathbf{F}_d = \frac{1}{2}\rho_f A C_D v_r^2 \tag{1.74}$$

$$\mathbf{F}_l = \frac{1}{2}\rho_f A C_L v_r^2 \tag{1.75}$$

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

$$\frac{dv_x}{dt} = -\rho_f \frac{A}{2m} v_r^2 (C_D \cdot \cos(\phi) + C_L \sin(\phi)) \tag{1.76}$$

$$\frac{dv_y}{dt} = \rho_f \frac{A}{2m} v_r^2 (C_L \cdot \cos(\phi) - C_D \sin(\phi)) - g \tag{1.77}$$

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

Now (1.76) and (1.77) become

$$\frac{dv_x}{dt} = -C \cdot v_r (C_D \cdot v_{rx} + C_L \cdot v_{ry}) \tag{1.78}$$

$$\frac{dv_y}{dt} = C \cdot v_r (C_L \cdot v_{rx} - C_D \cdot v_{ry}) - g \tag{1.79}$$

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

$$\begin{aligned}
\frac{dx}{dt} &= v_x \\
\frac{dy}{dt} &= v_y \\
\frac{dv_x}{dt} &= -C \cdot v_r (C_D \cdot v_{rx} + C_L \cdot v_{ry}) \\
\frac{dv_y}{dt} &= C \cdot v_r (C_L \cdot v_{rx} - C_D \cdot v_{ry}) - g
\end{aligned} \tag{1.80}$$

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

$$\begin{aligned}
\frac{dy_1}{dt} &= y_3 \\
\frac{dy_2}{dt} &= y_4 \\
\frac{dy_3}{dt} &= -C \cdot v_r(C_D \cdot v_{rx} + C_L \cdot v_{ry}) \\
\frac{dy_4}{dt} &= C \cdot v_r(C_L \cdot v_{rx} - C_D \cdot v_{ry}) - g
\end{aligned} \tag{1.81}$$

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

Initial conditions for $t = 0$ are

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

—–

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

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

We use the initial velocity $v_0 = 50$ m/s and solve (1.81) 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 (1.81) 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 (1.3.2) to calculate the drag coefficient of the smooth sphere.

The results are shown for some initial angles in Figure 1.11.

—–

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

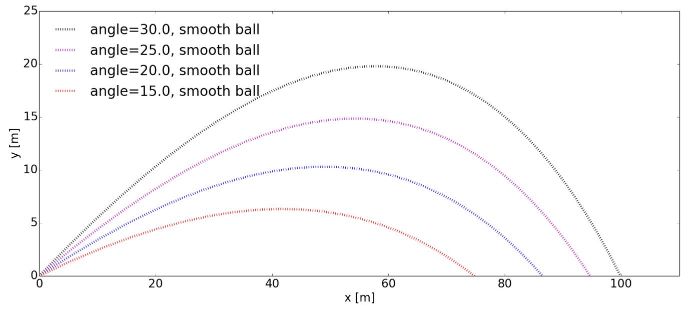


Figure 1.11: Motion of smooth ball with drag.

different from the smooth sphere. We use the data from Bearman and Harvey [1] who measured the drag and lift of a golf ball for different spin velocities in a vindtunnel. We choose as an example 3500 rpm, and an initial velocity of $v_0 = 50$ m/s.

The right hand side in (1.81) 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[8]) gives the drag and lift data for a given velocity and spin.

The results are shown in Figure 1.12. 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.

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

```
# chapter1/programs_and_modules/ParticleMotion2D.py;DragCoefficientGeneric.py @ git@lrhgit/tkt41

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]
```

[8]https://lrhgit.github.io/tkt4140/allfiles/digital_compendium/chapter1/programs_and_modules/cdclgolfball

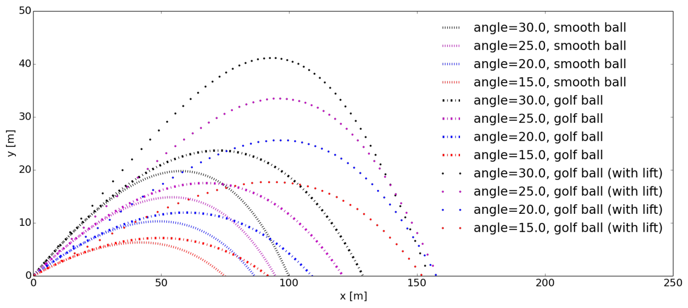


Figure 1.12: Motion of golf ball with drag and lift.

```
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()
```

1.9.3 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 1.7.2. 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:

$$\begin{aligned} u'(t) &= a\,u + b \\ u(t_0) &= u_0 \end{aligned} \tag{1.82}$$

which has the analytical solutions:

$$u = \begin{cases} \left(u_0 + \frac{b}{a}\right)\ e^{a\,t} - \frac{b}{a}, & a \neq 0 \\ u_0 + b\,t, & a = 0 \end{cases} \tag{1.83}$$

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 (1.45) is $O(h)$, whereas the Heun scheme (1.63) is $O(h^2)$, and Runge-Kutta (1.71) is $O(h^4)$, where the h denote a generic step size which for the current example is the timestep Δt. The order of a particular scheme is given exponent n in the error term $O(h^n)$. Consequently, the Euler scheme is a first oder scheme, Heun is second order, whereas Runge-Kutta is fourth order.

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

$$\epsilon_{i+1} = \frac{1}{2^n}\epsilon_i \tag{1.84}$$

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:

$$n \approx \log_2 \frac{\epsilon_i}{\epsilon_{i+1}} \tag{1.85}$$

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 (1.85) with the clause `order_approx.append(previous_max_log_err - max_log_err)`. Note that we can not compute this for the first iteration (`i=0`), and that we use a an initial empty list `order_approx` to store the approximation of the order `n` for each grid refinement. For each grid we plot $\log_2(\epsilon)$ as a function of time with: `plot(time[1:], log_error, linestyles[i]+colors[iclr], markevery=N/5)` and for each plot we construct the corresponding legend by appending a new element to the legends-list `legends.append(scheme.func_name +': N = ' + str(N))`. This construct produces a string with both the scheme name and the number of elements N. The plot is not reproduced below, but you may see the result by downloading and running the module yourself.

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

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

and the resulting plot is shown in Figure 1.13, 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={}
```

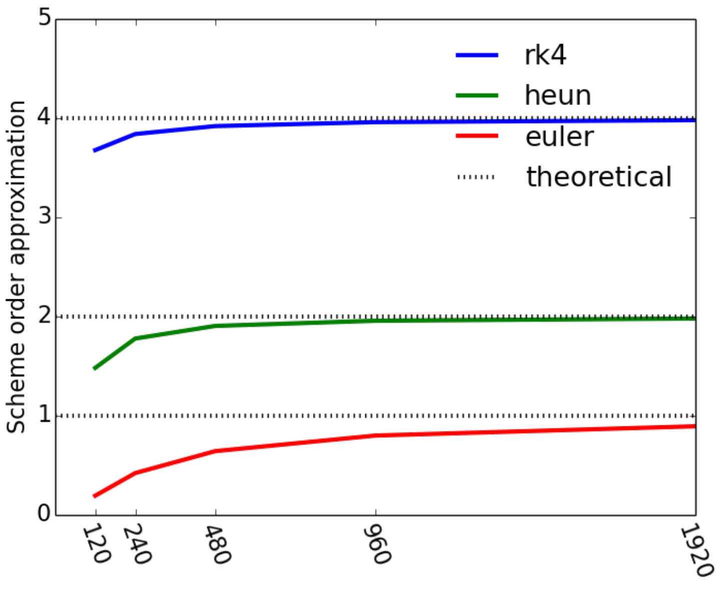


Figure 1.13: The convergence rate for the various ODE-solvers a function of the number of timesteps.

```
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)
```

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

```
# chapter1/programs_and_modules/ODEschemes.py

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

# 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)


# 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 i
                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 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)


    test_ODEschemes()
    convergence_test()
    plot_ODEschemes_solutions()
    show()
```

Chapter 2

Chapter 6: Convection problems and hyperbolic PDEs

2.1 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:

$$\frac{\partial u}{\partial t} + a_0 \frac{\partial u}{\partial x} = 0 \tag{2.1}$$

Another convenient feature of the model equation (2.1) is that is has an analytical solution:

$$u = u_0\, f(x - a_0\, t) \tag{2.2}$$

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 (2.1) 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 (2.1) on the following form:

$$\frac{\partial u}{\partial t} + \frac{\partial F}{\partial x} = 0 \tag{2.3}$$

where for the linear advection equation $F(u) = a_0\, u$.

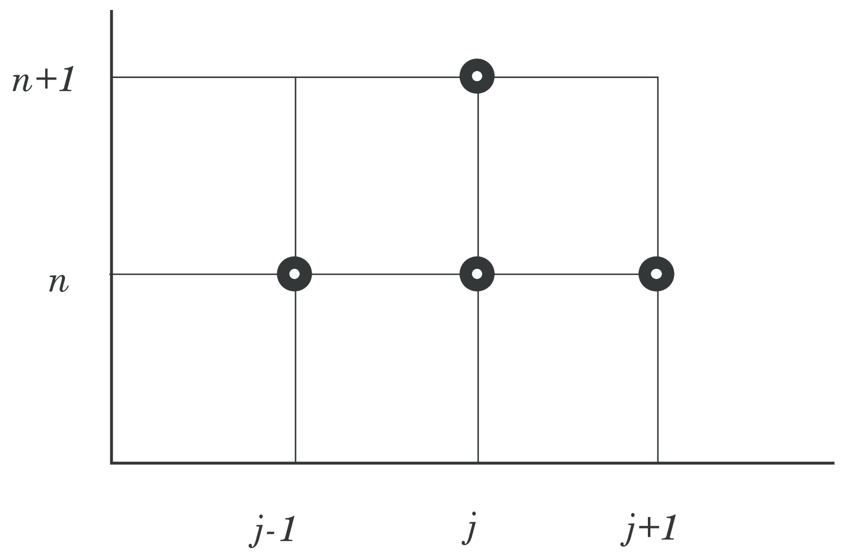


Figure 2.1: Illustration of the first order in time central in space scheme.

2.1.1 Forward in time central in space discretization

We may discretize (2.1) 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 (2.1.1) into the advection equation (2.1) to yield:

$$u_j^{n+1} = u_j^n - \frac{C}{2}(u_{j+1}^n - u_{j-1}^n) \tag{2.4}$$

For convenience we have introduced the non-dimensional Courant-Friedrich-Lewy[1] number (or CFL-number for short):

$$C = a_0 \frac{\Delta t}{\Delta x} \tag{2.5}$$

The scheme in (2.4) 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 Fig. (2.1) and (2.4).

We will try to solve model equation (2.1) with the scheme (2.4) and initial conditions illustrated in Fig 2.2 with the mathematical representation:

$$u(x,0) = 1 \text{ for } x < 0.5$$
$$u(x,0) = 0 \text{ for } x > 0.5$$

[1] `http://en.wikipedia.org/wiki/Courant%E2%80%93Friedrichs%E2%80%93Lewy_condition`

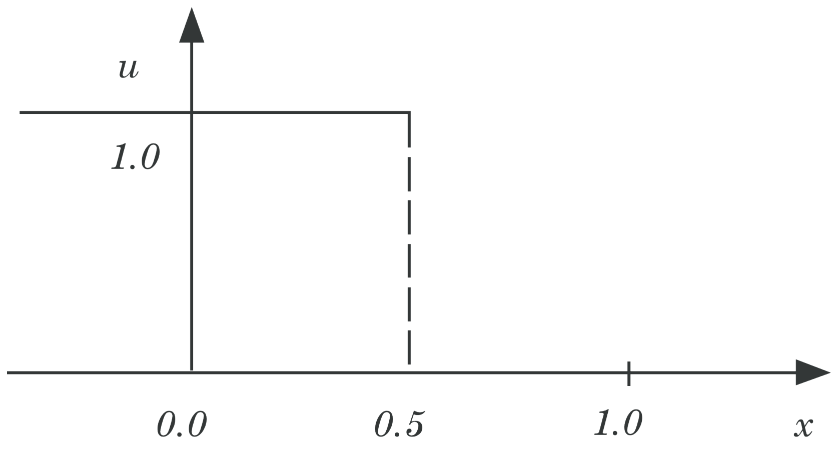


Figure 2.2: Initial values for the advection equaution (2.1).

Solutions for three CFL-numbers: C=0.25, 0.5 and 1.0 are illustrated in Fig. (2.3). 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 unstable scheme. As a first approach observe that the coefficient for u^n_{j+1} in (2.4) 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^n_j in the numerical scheme to be analyzed

$$u^n_j \to E^n_j = G^n \cdot e^{i\cdot\beta x_j} \tag{2.6}$$

Substitution of (2.6) into (2.4) 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:

$$2\cos(x) = e^{ix} + e^{-ix} \tag{2.7}$$

$$i\cdot 2\sin(x) = e^{ix} - e^{-ix} \tag{2.8}$$

$$\cos(x) = 1 - 2\sin^2(\frac{x}{2}) \tag{2.9}$$

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.

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:

$$\frac{u_i^{n+1} - \frac{1}{2}(u_{i+1}^n + u_{i-1}^n)}{\Delta t} = -\frac{F_{i+1}^n - F_{i-1}^n}{2\Delta x} \tag{2.10}$$

The Lax-Friedrich scheme is the obtained by isolation u_i^{n+1} at the right hand side:

$$u_i^{n+1} = \frac{1}{2}(u_{i+1}^n + u_{i-1}^n) - \frac{\Delta t}{2\Delta x}(F_{i+1}^n - F_{i-1}^n) \tag{2.11}$$

By assuming a linear flux $F = a_0\, u$ it may be shown that the Lax-Friedrich scheme takes the form:

$$u_i^{n+1} = \frac{1}{2}(u_{i+1}^n + u_{i-1}^n) - \frac{C}{2}(u_{i+1}^n - u_{i-1}^n) \tag{2.12}$$

where we have introduced the CFL-number as given by (2.5) 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]
```

Lax Wendroff Schemes. These schemes were proposed in 1960 by P.D. Lax and B. Wendroff [6] for solving, approximately, systems of hyperbolic conservation laws on the generic form given in (2.3).

A large class of numerical methods for solving (2.3) are the so-called conservative methods:

$$u_j^{n+1} = u_j^n + \frac{\Delta t}{\Delta x}\left(F_{i-1/2} - F_{i+1/2}\right) \tag{2.13}$$

Linear advection The Lax–Wendroff method belongs to the class of conservative schemes (2.3) and can be derived in various ways. For simplicity, we will derive the method by using a simple model equation for (2.3), namely the linear advection equation with $F(u) = a\,u$ as in (2.1), where a is a constant propagation velocity. The Lax-Wendroff outset is a Taylor approximation of u_j^{n+1}:

$$u_j^{n+1} = u_j^n + \Delta t \frac{\partial u}{\partial t}\bigg|_j^n + \frac{(\Delta t)}{2}\frac{\partial^2 u}{\partial t^2}\bigg|_j^n + \cdots \tag{2.14}$$

From the differential equation (2.3) we get by differentiation

$$\frac{\partial u}{\partial t}\bigg|_j^n = -a_0 \frac{\partial u}{\partial x}\bigg|_j^n \qquad \text{and} \qquad \frac{\partial^2 u}{\partial t^2}\bigg|_j^n = a_0^2 \frac{\partial^2 u}{\partial x^2}\bigg|_j^n \tag{2.15}$$

Before substitution of (2.15) in the Taylor expansion (2.14) we approximate the spatial derivatives by central differences:

$$\frac{\partial u}{\partial x}\bigg|_j^n \approx \frac{u_{j+1}^n - u_{j-1}^n}{(2\Delta x)} \qquad \text{and} \qquad \frac{\partial^2 u}{\partial x^2}\bigg|_j^n \approx \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2} \tag{2.16}$$

and then the Lax-Wendroff scheme follows by substitution:

$$u_j^{n+1} = u_j^n - \frac{C}{2}\left(u_{j+1}^n - u_{j-1}^n\right) + \frac{C^2}{2}\left(u_{j+1}^n - 2u_j^n + u_{j-1}^n\right) \tag{2.17}$$

with the local truncation error T_j^n:

$$T_j^n = \frac{1}{6}\cdot\left[(\Delta t)^2 \frac{\partial^3 u}{\partial t^3} + a_0 (\Delta x)^2 \frac{\partial^3 u}{\partial x^3}\right]_j^n = O[(\Delta t)^2, (\Delta x)^2] \tag{2.18}$$

The resulting difference equation in (2.17) may also be formulated as:

$$u_j^{n+1} = \frac{C}{2}(1+C)u_{j-1}^n + (1-C^2)u_j^n - \frac{C}{2}(1-C)u_{j+1}^n \tag{2.19}$$

The explicit Lax Wendroff stenticl is illustrated in Fig. (2.4)

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]
```

Lax-Wendroff for non-linear systems of hyperbolic PDEs. For non-linear equations (2.3) 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 (2.15)) is not straightforward and unique.

Ricthmyer Scheme One of the earliest extensions of the scheme is the Richtmyer two-step Lax–Wendroff method[2], which is on the conservative form (2.13) with the numerical fluxes computed as follows:

$$u_{j+1/2}^{n+1/2} = \frac{1}{2}\left(u_j^n + u_{j+1}^n\right) + \frac{1}{2}\frac{\Delta t}{\Delta x}\left(F_j^n - F_{j+1}^n\right) \tag{2.20}$$

$$F_{j+1/2} = F(u_{j+1/2}^{n+1/2}) \tag{2.21}$$

Lax-Wendroff two step A Lax-Wendroff two step method[3] 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:

$$u_{j+1/2}^{n+1/2} = \frac{1}{2}\left(u_{j+1}^n + u_j^n\right) - \frac{\Delta t}{2\Delta x}\left(F(u_{j+1}^n) - F(u_j^n)\right) \tag{2.22}$$

$$u_{j-1/2}^{n+1/2} = \frac{1}{2}\left(u_j^n + u_{j-1}^n\right) - \frac{\Delta t}{2\Delta x}\left(F(u_j^n) - F(u_{j-1}^n)\right) \tag{2.23}$$

Second step:

$$u_j^{n+1} = u_j^n - \frac{\Delta t}{\Delta x}\left(F(u_{j+1/2}^{n+1/2}) - F(u_{j-1/2}^{n+1/2})\right) \tag{2.24}$$

Notice that for a linear flux $F = a_0\, u$, the two-step Lax-Wendroff method ((2.23) and (2.24)) may be shown to reduce to the one-step Lax-Wendroff method outlined in (2.17) or (2.19).

MacCormack Scheme A simpler and popular extension/variant of Lax-Wendroff schemes like in the previous section, is the MacCormack scheme [7]:

$$u_j^p = u_j^n + \frac{\Delta t}{\Delta x}\left(F_j^n - F_{j+1}^n\right) \tag{2.25}$$

$$u_j^{n+1} = \frac{1}{2}\left(u_j^n + u_j^p\right) + \frac{1}{2}\frac{\Delta t}{\Delta x}\left(F_{j-1}^p - F_j^p\right) \tag{2.26}$$

$$\tag{2.27}$$

where we have introduced the convention $F_j^p = F(u_j^p)$.

Note that in the predictor step we employ the conservative formula (2.13) 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 (2.13) 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 (2.27) is not written in

[2]`http://www.encyclopediaofmath.org/index.php/Lax-Wendroff_method`
[3]`http://en.wikipedia.org/wiki/Lax%E2%80%93Wendroff_method`

conservative form (2.13). However, it easy to express the scheme in conservative form by expressing the flux in (2.13) as:

$$F^m_{j+1} = \frac{1}{2}\left(F^p_j + F^n_{j+1}\right) \tag{2.28}$$

For a linear flux $F(u) = a_0\, u$, one may show that the MacCormack scheme in (2.27) reduces to a two-step scheme:

$$u^p_j = u^n_j + C\left(u^n_j - u^n_{j+1}\right) \tag{2.29}$$

$$u^{n+1}_j = \frac{1}{2}\left(u^n_j + u^p_j\right) + \frac{C}{2}\left(u^p_{j-1} - u^p_j\right) \tag{2.30}$$

and substitution of (2.29) into (2.30) shows that the MacCormack scheme is identical to the Lax-Wendroff scheme (2.19) 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]
```

Code example for various schemes for the advenction 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, 1.0 # start and end of spatial domain
Nx = 80 # number of spatial points
c = 0.9 # courant number, need c<=1 for stability

init_func=0   # 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 = [ftbs,lax_friedrich]


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()
```

2.1.2 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:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0 \tag{2.31}$$

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:

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\left(\frac{u^2}{2}\right) = 0 \tag{2.32}$$

and by introducing a flux function

$$F(u) = \frac{u^2}{2} \tag{2.33}$$

the conservative formulation of the Burgers equation may be represented by a generic transport equation:

$$\frac{\partial u}{\partial t} + \frac{\partial F(u)}{\partial x} = 0 \tag{2.34}$$

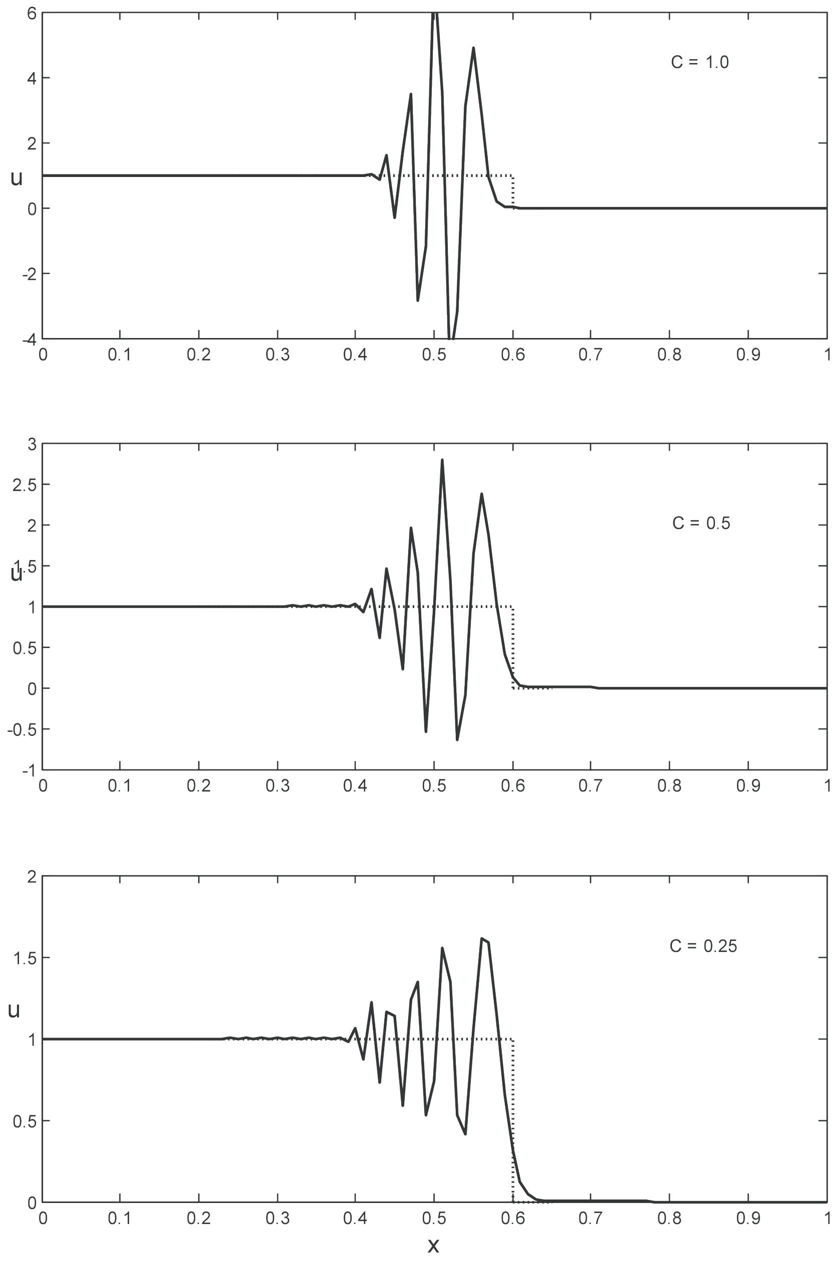


Figure 2.3: Computed solutions with the (2.4). Dotted line: analytical solution, solid line: computed soultion.

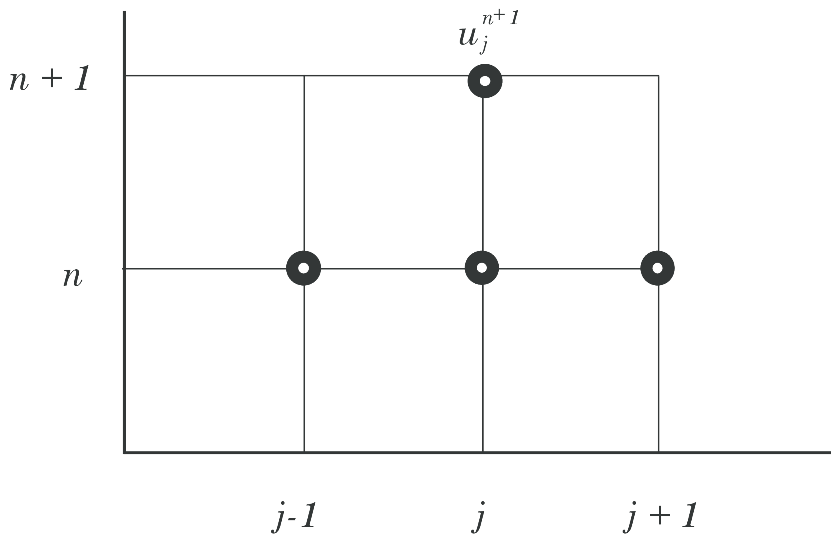


Figure 2.4: Schematic of the Lax-Wendroff scheme.

Bibliography

[1] P.W. Bearman and J.K. Harvey. Golf ball aerodynamics. *Aeronaut Q*, 27(pt 2):112–122, 1976. cited By 119.

[2] E. Cheney and David Kincaid. *Numerical Mathematics and Computing.* Cengage Learning, 2012.

[3] J. Evett and C. Liu. *2,500 Solved Problems in Fluid Mechanics and Hydraulics.* Schaum's Solved Problems Series. McGraw-Hill Education, 1989.

[4] Ernst Hairer, Syvert Paul Norsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*, volume 1. Springer Science & Business, 2008.

[5] Hans Petter Langtangen. *A Primer on Scientific Programming With Python.* Springer, Berlin; Heidelberg; New York, fourth edition, 2011.

[6] Peter D. Lax. *Hyperbolic Systems of Conservation Laws and the Mathematical Theory of Shock Waves.* Regional conference series in applied mathematics. Society for Industrial and Applied Mathematics, Philadelphia, 1973.

[7] Robert William MacCormack. The effect of viscosity in hypervelocity impact cratering. Technical Report AIAA-69-354, Astronautics, 1969.

[8] F.M. White. *Fluid Mechanics.* McGraw-Hill series in mechanical engineering. WCB/McGraw-Hill, 1999.