Digital Compendium for TKT4140

Johan Kolstø Sønstabø[1,2]

Leif Rune Hellevik[2]

[1]Centre for Advanced Structural Analysis (CASA)
[2]Department of Structural Engineering, NTNU

Jan 4, 2015

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

Students are strongly encouraged to work through the following chapter.

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.

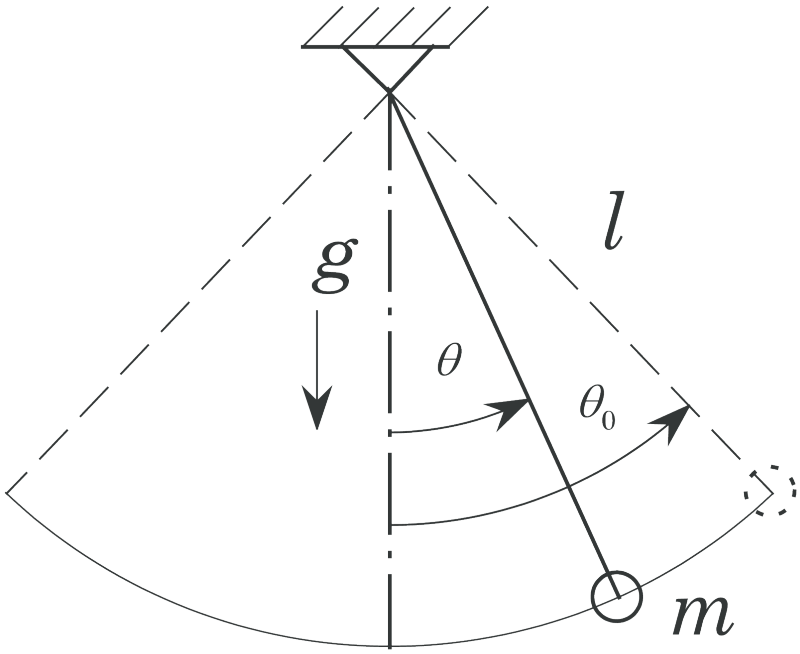
Both from an analytical and numerical viewpoint initial value problems are easier to solve than boundary value problems, and methods for solution of initial value problems are more developed than for boundary value problems.

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. A sufficient solution for this 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.

An example:

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

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 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.

1.1.1 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):

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

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

[1] `./NumeriskeBeregninger.pdf`

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 $\dddot{\theta}(t) = -\dot{\theta}(t) \to \dddot{\theta}(0) = -\theta(0) = -\theta_0$

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 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

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

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

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

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

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

Note the combination $\sqrt{2\pi e}$. See Hairer et al. [4] section 1.2 for more details on classical differential equations.

1.1.2 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:

$$y' = y_1' = y_2$$
$$y'' = y_1'' = y_2' = y_3$$

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

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

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

1.1.3 Example: Reduction of higher order system

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

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

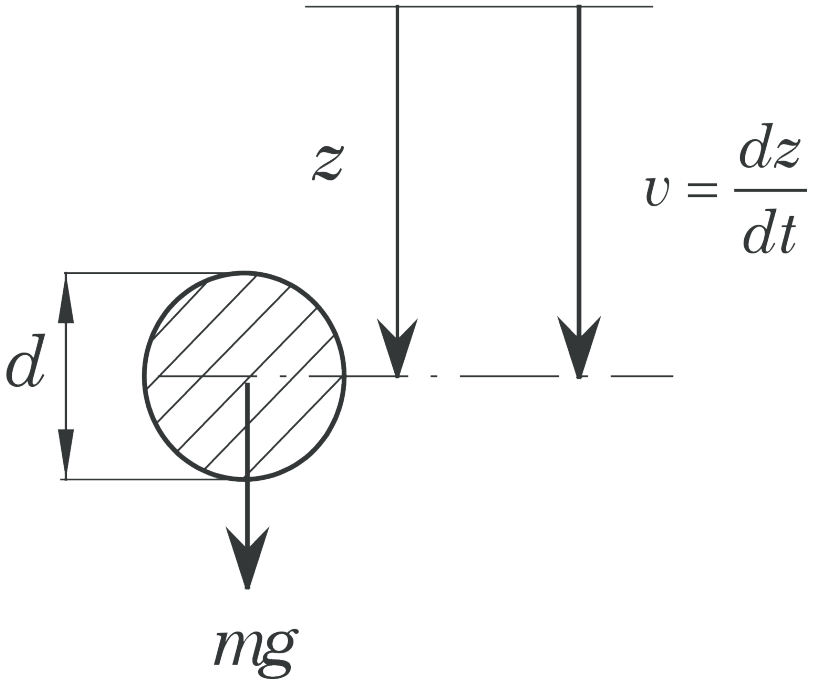
First we write $y''' = y'y'' + (y')^2 - 2y + x^3$.
By use of (**??**) we get

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

1.1.4 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[2] before proceeding. If you are familiar with **Matlab** the transfer to Python should not be a problem.

1.1.5 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

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

[2] `http://folk.ntnu.no/johankso/electronic_compendium/digital_compendium/python_intro/doc/src/bumpy.html`

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 hydrodynamical mass (added mass). The expression for the hydrodynamical mass is derived in White [6], 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 $R_e = \frac{vd}{\nu}$, 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 hydrodynamical mass, whereas this is not the case for a liquid. Introducing $v = \frac{dz}{dt}$ in Equation (1.14), we get a 2nd order ODE as follows

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

The Python program **CDsphere.py** produces the plot from a curve fit to the data of Evett and Liu [3]. The program uses a function `cd_sphere` which is shown at the end of this example. 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.

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

```
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 & Liu: "Fluid Mechanics & Hydraulics",
    # Schaum's Solved Problems McGraw - Hill 1989.

    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:
```

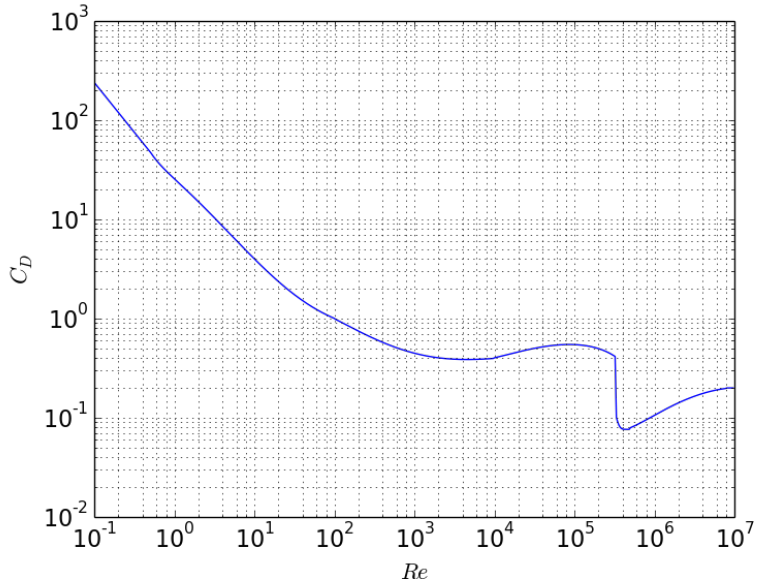


Figure 1.1: Drag coefficient C_D as function of the Reynold's number R_e.

```
        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])
```

Generating the plot in 1.1.5. The complete program **CDsphere.py** used to generate the plot in the example above will now be presented. The program may be downloaded here[3]. We will break up the program and explain the different parts. The complete program is as follows: @@@CODE ./chapter1/programs$_a nd_m odules/CDsphere.py$

In the first code line,

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

[3] https://raw.githubusercontent.com/lrhgit/tkt4140/master/allfiles/Utviklingsprosjekt/chapter1/programs_and_modules/CD

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 numbe
    # Curve fitted after fig. A-56 in Evett & Liu: "Fluid Mechanics & Hydraulics",
    # Schaum's Solved Problems McGraw - Hill 1989.

    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 funciton 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 oustide 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[4] for details about the different functions.

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

In the code block

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

the function `cd_sphere` is called. First, the number of data points to be calculated are stored in the integer variable `Npts`. Using the `logspace` function imported earlier, `Re` is assigned an array objet which has float elements with values ranging from 10^{-1} to 10^{7}. The values are uniformly distributed along a 10logarithmic 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

[4] `http://www.numpy.org`

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 similarily 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[5].

More efficient implementations. When solving numerical problems variables are often not single numbers, but arrays containing many numbers. The function `CDsphere` above takes a single number in and gives a single number out. 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. It is therefore of interest to implement functions that can take in whole arrays and output whole arrays in an efficient manner. This may be done in a variaty of ways. Some possibilities are presented in the following, and, as we shall see, some are more time consuming than others.

A simple extension of the single-valued function above 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, similarily as each `if` statement in the function `cd_sphere`. The function is shown here:

[5] `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 & Liu :% " Fluid Mechanics & Hydraulics ",
    # Schaum ' s Solved Problems McGraw - Hill 1989.

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

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

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

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

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

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

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

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

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

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

```
def cd_sphere_vector_bool(Re):
    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)
```

Using the function `clock` in the module `time`, the cpu-time for each of the array-type functions described above has been measured for a test example. An array with 500 elements was sent to the functions.

```
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
```

As seen, the function with the boolean variables was fastest. The simple vectorized version of the original function was much slower, and the built-in function `vectorize` was nearly as inefficient. The complete program **DragCoefficientGeneric.py** is listed below.

```
# chapter1/programs_and_modules/DragCoefficientGeneric.py

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

from numpy.core.multiarray import scalar

# single-valued function
def cd_sphere(Re):
    "Computes the drag coefficient of a sphere as a function of the Reynolds number Re."
    # Curve fitted after fig . A -56 in Evett & Liu :% " Fluid Mechanics & Hydraulics ",
    # Schaum ' s Solved Problems McGraw - Hill 1989.

    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 & Liu :% " Fluid Mechanics & Hydraulics ",
    # Schaum ' s Solved Problems McGraw - Hill 1989.

    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):
    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 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]


    # 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)

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

    fnames_sorted=sorted(exec_times,key=exec_times.__getitem__)
    exec_time_sorted=sorted(exec_times.values())

    for i in range(len(fnames_sorted)):
        print fnames_sorted[i], '\t execution time = ', '%6.6f'%(exec_time_sorted[i])

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

    # set line styles
```

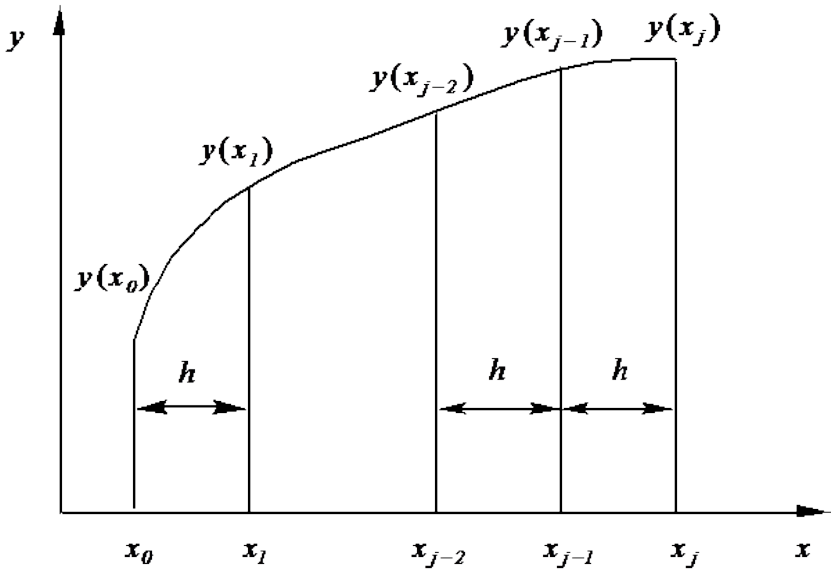
```
style = ['v-', '8-', '*-', 'o-']
mrkevry = [30, 35, 40, 45]

# 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

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)
show()
```

1.1.6 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:

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

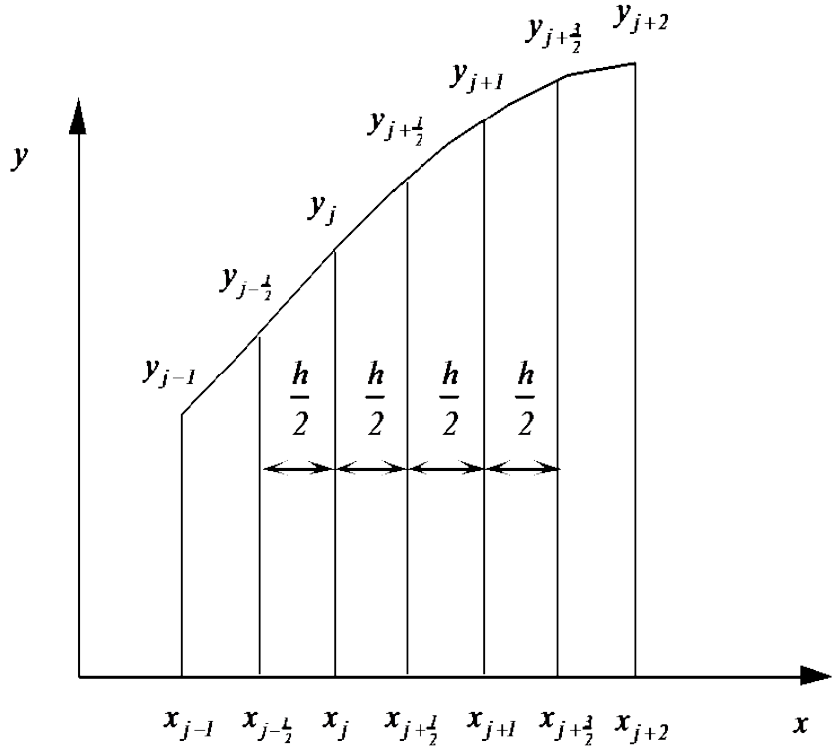


Figure 1.2: Illustration of how to obtain difference equations.

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 $\xi \in (x_0, x)$

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:

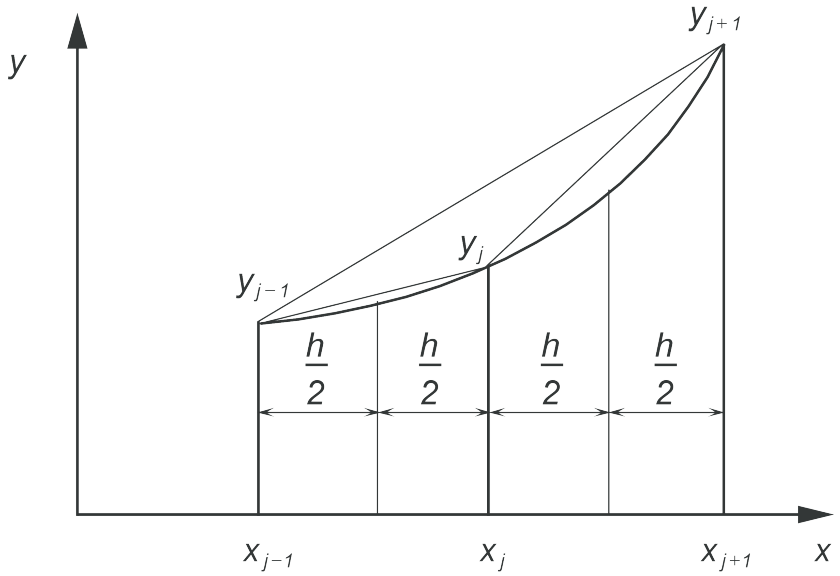
$$y'_j = \frac{y_{j+1} - y_j}{h}\ ;\ \text{truncation error } O(h) \tag{1.25}$$

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

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

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

(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:

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

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

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

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,

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

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

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

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:

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

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,

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

Using central differences again, we get

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

which inserted in the previous equation gives the final expression

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

where

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

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

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

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

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

Forward differences. We start with

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

which gives

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

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 $p(x) = 1 = \text{constant}$ 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

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

which gives

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

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

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

1.1.7 Euler's method

The ODE is given as

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

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

Inserting (1.21) 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.

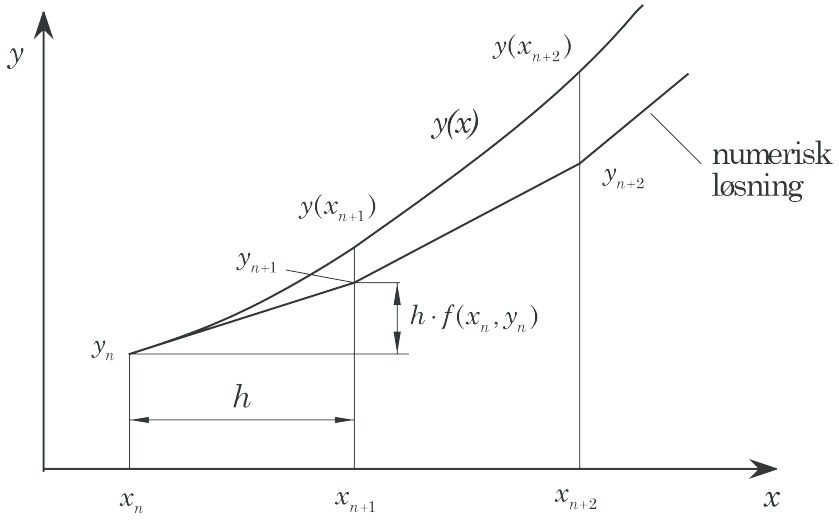


Figure 1.3: Graphical illustration of Euler's method.

1.1.8 Example: Falling sphere with constant and varying drag

We write (1.14) and (1.15) as a sytem 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

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

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

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

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

If we use Teylor's method from section 1.1.1 we get the following expression by using four terms in the series expansion:

$$z(t) = \frac{1}{2}gt^2 \cdot (1 - \frac{1}{6}\alpha g t^2) \tag{1.49}$$

$$v(t) = gt \cdot (1 - \frac{1}{3}\alpha g t^2) \tag{1.50}$$

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

$$v_{n+1} = v_n + \Delta t \cdot (g - \alpha \cdot v_n^2),\ n = 0, 1, \ldots \tag{1.51}$$

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] rm -rf *~
        z[i+1,:]=z[i,:] + np.asarray(func(z[i,:],time[i]))*dt

    return z
```

The program **FallingSphereEuler.py** (may be downloaded here[6]) 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 1.1.5. The complete program is as follows,

[6] `https://raw.githubusercontent.com/lrhgit/tkt4140/master/allfiles/Utviklingsprosjekt/chapter1/programs_a`

```
# chapter1/programs_and_modules/FallingSphereEuler.py;DragCoefficientGeneric.py @ git@lrhgit/tkt4140/allfile
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] rm -rf *~
        z[i+1,:]=z[i,:] + np.asarray(func(z[i,:],time[i]))*dt

    return z

# main program starts here

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

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

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

k1 = np.sqrt(g*4*rho_s*d/(3*rho_f*CD))
k2 = np.sqrt(3*rho_f*g*CD/(4*rho_s*d))
```

```
v_a = k1*np.tanh(k2*time)   # compute response with constant CD using analytical solution

# plotting

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

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

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

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

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

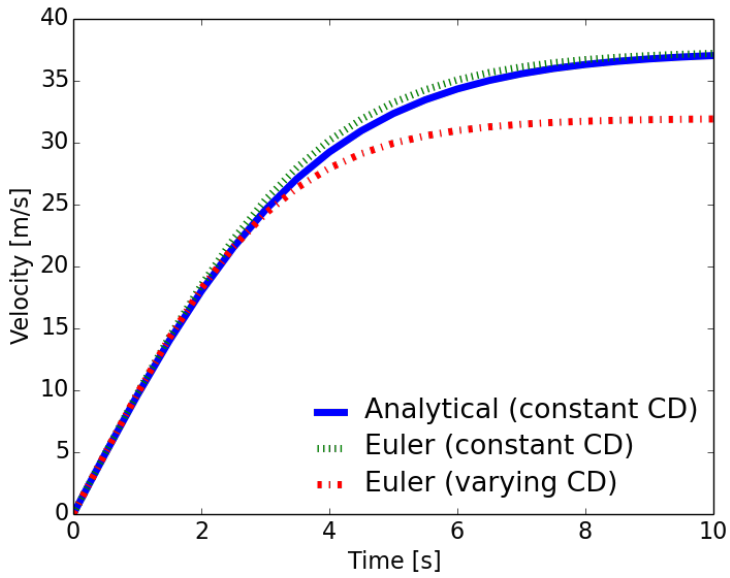


Figure 1.4: Euler's method with $\Delta t = 0.5$ s.

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

$$\begin{aligned} y_1' &= f_1(x, y_1, y_2, \ldots y_p) \\ y_2' &= f_2(x, y_1, y_2, \ldots y_p) \\ &. \\ &. \\ y_p' &= f_p(x, y_1, y_2, \ldots y_p) \end{aligned} \tag{1.52}$$

with initial values

$$y_1(x_0) = a_1,\ y_2(x_0) = a_2, \ldots,\ y_p(x_0) = a_p \tag{1.53}$$

Or, in vectorial format as follows,

$$\begin{aligned} \mathbf{y}' &= \mathbf{f}(x, \mathbf{y}) \\ \mathbf{y}(x_0) &= \mathbf{a} \end{aligned} \tag{1.54}$$

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

The Euler scheme (1.45) used on (1.54) gives

$$\mathbf{y_{n+1}} = \mathbf{y_n} + h \cdot \mathbf{f}(x_n, \mathbf{y_n}) \tag{1.55}$$

For a system of three equations we get

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= y_3 \\ y_3' &= -\, y_1 y_3 \end{aligned} \tag{1.56}$$

In this case (1.55) gives

$$\begin{aligned} (y_1)_{n+1} &= (y_1)_n + h \cdot (y_2)_n \\ (y_2)_{n+1} &= (y_2)_n + h \cdot (y_3)_n \\ (y_3)_{n+1} &= (y_3)_n - h \cdot (y_1)_n \cdot (y_3)_n \end{aligned} \tag{1.57}$$

$$\tag{1.58}$$

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

In section 1.1.2 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,

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

1.1.9 Heun's method

From (1.21) or (1.25) we have

$$y''(x_n, y_n) = f'\left(x_n, y(x_n, y_n)\right) \approx \frac{f(x_n + h) - f(x_n)}{h} \tag{1.59}$$

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 (1.59), gives

$$y_{n+1} = y_n + \frac{h}{2} \cdot [f(x_n, y_n) + f(x_{n+1}, y(x_{n+1}))] \tag{1.60}$$

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 (1.60). The resulting scheme is often denoted **Heun's method**.

The scheme for Heun's method becomes

$$y^p_{n+1} = y_n + h \cdot f(x_n, y_n) \tag{1.61}$$

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

Index p stands for "predicted". (1.61) is then the predictor and (1.62) is the corrector. This is a second order method. For more details, see [2]. Figure 1.5 is a graphical illustration of the method.

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

$$\mathbf{y^p_{n+1}} = \mathbf{y_n} + h \cdot \mathbf{f}(x_n, \mathbf{y_n}) \tag{1.63}$$

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

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

If we use (1.63) and (1.64) on the example in (1.56) we get

Predictor:

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

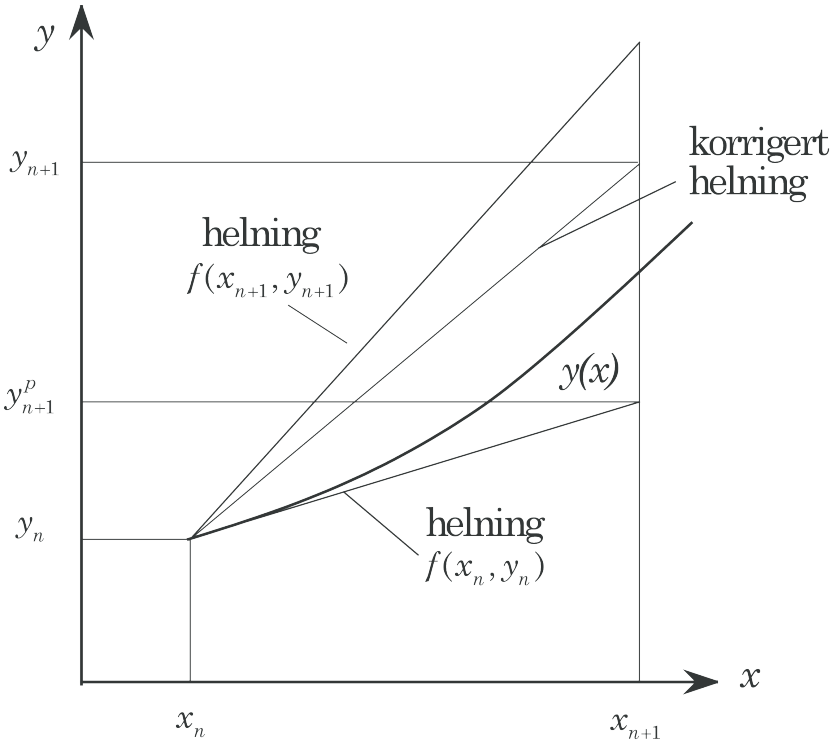


Figure 1.5: Illustration of Heun's method.

Corrector:

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

1.1.10 Example: Newton's equation

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

$$y'(x) = 1 - 3x + y + x^2 + xy, \; y(0) = 0 \tag{1.65}$$

with analytical solution

$$
\begin{aligned}
y(x) =& 3\sqrt{2\pi e} \cdot \exp\left(x\left(1 + \frac{x}{2}\right)\right) \cdot \left[\operatorname{erf}\left(\frac{\sqrt{2}}{2}(1+x)\right) - \operatorname{erf}\left(\frac{\sqrt{2}}{2}\right)\right] \\
&+ 4 \cdot \left[1 - \exp\left(x\left(1 + \frac{x}{2}\right)\right)\right] - x
\end{aligned} \tag{1.66}
$$

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 1.6.

```
# 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)

# 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()
```

1.1.11 Example: Falling sphere with Heun's method

Let's go back to 1.1.8, and implement a new function `heun()` in the program FallingSphereEuler.py[7].

We recall the system of equations as

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

[7] https://raw.githubusercontent.com/lrhgit/tkt4140/master/allfiles/Utviklingsprosjekt/chapter1/programs_a

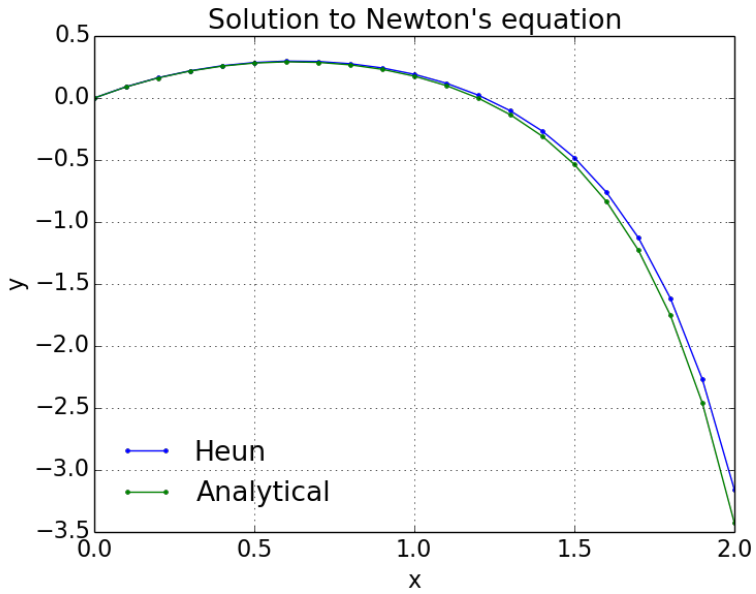


Figure 1.6: Velocity of falling sphere using Euler's and Heun's methods.

which by use of Heun's method in (1.63) and (1.64) becomes

Predictor:

$$\begin{aligned} z_{n+1}^p &= z_n + \Delta t v_n \\ v_{n+1}^p &= v_n + \Delta t \cdot (g - \alpha v_n^2) \end{aligned} \tag{1.67}$$

Corrector:

$$\begin{aligned} z_{n+1} &= z_n + 0.5\Delta t \cdot (v_n + v_{n+1}^p) \\ v_{n+1} &= v_n + 0.5\Delta t \cdot \left[2g - \alpha[v_n^2 + (v_{n+1}^p)^2\right] \end{aligned} \tag{1.68}$$

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

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

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

    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 1.1.8, we get the response plotted in Figure 1.7.

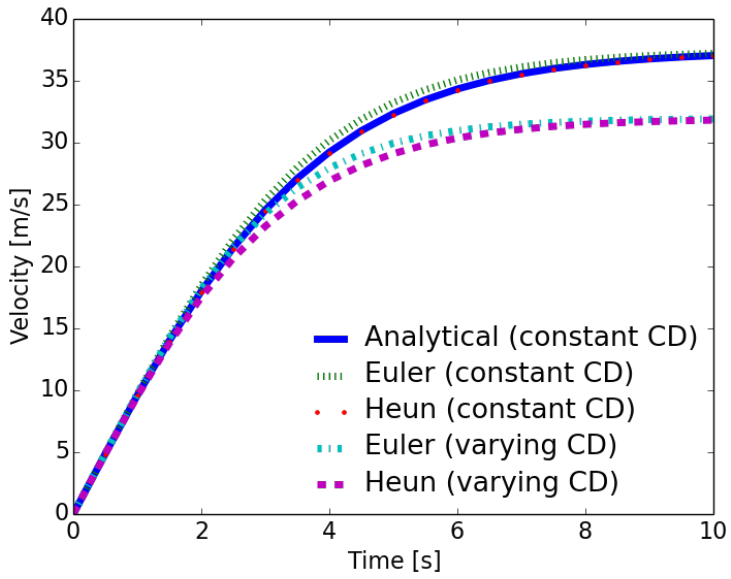


Figure 1.7: Velocity of falling sphere using Euler's and Heun's methods.

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

```
# 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 *
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()
```

1.1.12 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

$$\begin{aligned} k_1 &= f(x_n, y_n) \\ k_2 &= f(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \\ k_3 &= f(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_2) \\ k_4 &= f(x_n + h, y_n + hk_3) \\ y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \tag{1.69}$$

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 1.8 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 (1.69) on the equation system in (1.56) we get

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

$$(y_2)_{n+1} = (y_2)_n + \frac{h}{6}(l_1 + 2l_2 + 2l_3 + l_4) \tag{1.70}$$

$$(y_3)_{n+1} = (y_3)_n + \frac{h}{6}(m_1 + 2m_2 + 2m_3 + m_4) \tag{1.71}$$

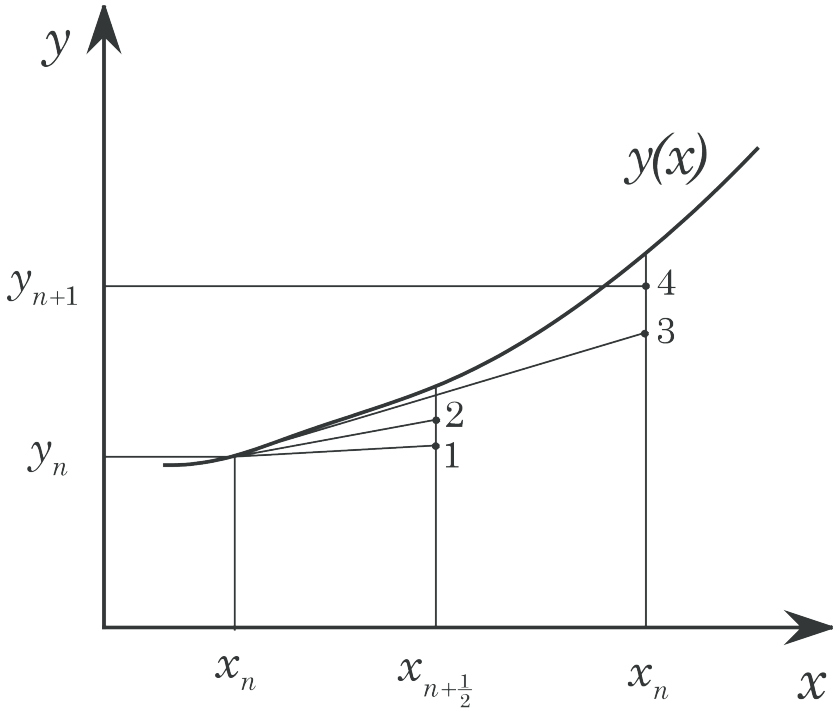


Figure 1.8: Illustration of the RK4 scheme.

where

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

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

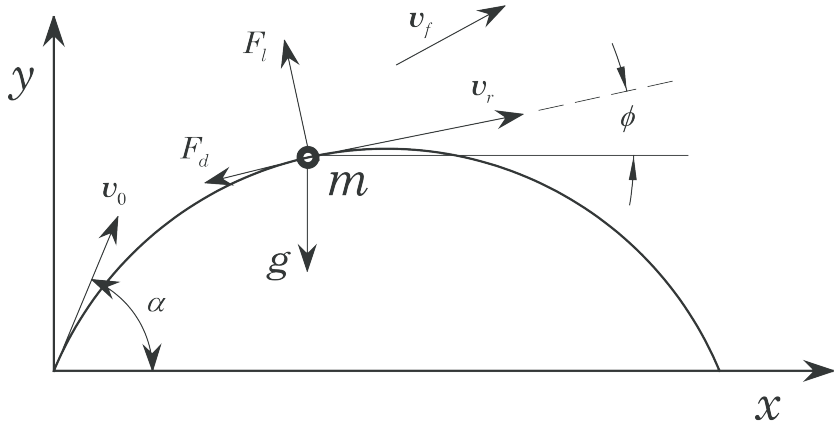
$$
\begin{aligned}
k_3 &= (y_2 + h l_2/2) \\
l_3 &= (y_3 + h m_2/2) \\
m_3 &= -[(y_1 + h k_2/2)(y_3 + h m_2/2)]
\end{aligned}
$$

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

1.1.13 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 1.1.5), 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 parallell to v_r. These forces are given by

$$\mathbf{F}_d = \frac{1}{2}\rho_f A C_D v_r^2 \tag{1.72}$$

$$\mathbf{F}_l = \frac{1}{2}\rho_f A C_L v_r^2 \tag{1.73}$$

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

$$\frac{dv_x}{dt} = -\rho_f \frac{A}{2m} v_r^2 (C_D \cdot \cos(\phi) + C_L \sin(\phi)) \tag{1.74}$$

$$\frac{dv_y}{dt} = \rho_f \frac{A}{2m} v_r^2 (C_L \cdot \cos(\phi) - C_D \sin(\phi)) - g \tag{1.75}$$

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 1.1.5. Here d is the diameter of the sphere and ρ_k the density of the sphere.

Now (1.74) and (1.75) become

$$\frac{dv_x}{dt} = -C \cdot v_r(C_D \cdot v_{rx} + C_L \cdot v_{ry}) \tag{1.76}$$
$$\frac{dv_y}{dt} = C \cdot v_r(C_L \cdot v_{rx} - C_D \cdot v_{ry}) - g \tag{1.77}$$

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

$$\begin{aligned}
\frac{dx}{dt} &= v_x \\
\frac{dy}{dt} &= v_y \\
\frac{dv_x}{dt} &= -C \cdot v_r(C_D \cdot v_{rx} + C_L \cdot v_{ry}) \\
\frac{dv_y}{dt} &= C \cdot v_r(C_L \cdot v_{rx} - C_D \cdot v_{ry}) - g
\end{aligned} \tag{1.78}$$

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

$$\begin{aligned}
\frac{dy_1}{dt} &= y_3 \\
\frac{dy_2}{dt} &= y_4 \\
\frac{dy_3}{dt} &= -C \cdot v_r(C_D \cdot v_{rx} + C_L \cdot v_{ry}) \\
\frac{dy_4}{dt} &= C \cdot v_r(C_L \cdot v_{rx} - C_D \cdot v_{ry}) - g
\end{aligned} \tag{1.79}$$

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 (1.79) 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 (1.79) 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 1.1.5 to calculate the drag coefficient of the smooth sphere.

The results are shown for some initial angles in Figure 1.9.

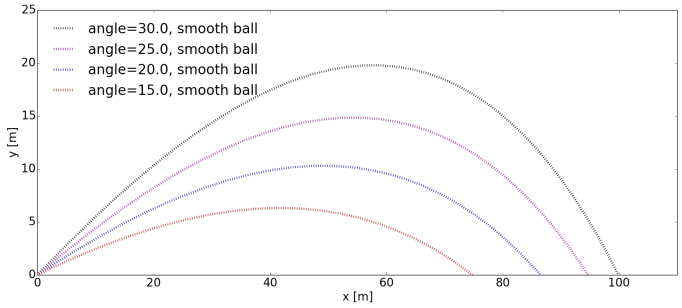


Figure 1.9: Motion of smooth ball with drag.

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

The right hand side in (1.79) 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 in Figure 1.10. The motion of a golf ball with drag but without lift is also included. We see that the golf ball goes much farther than the smooth sphere, due to less drag and the lift.

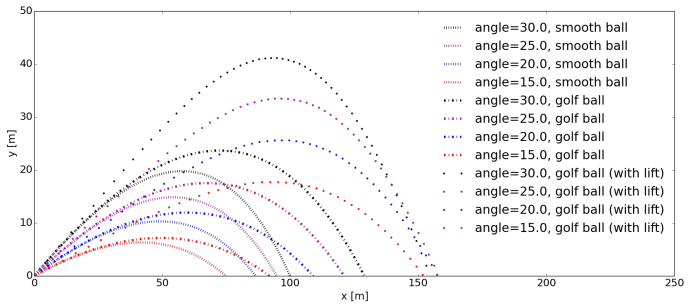


Figure 1.10: Motion of golf ball with drag and lift.

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

```
# chapter1/programs_and_modules/ParticleMotion2D.py;DragCoefficientGeneric.py @ git@lrhgit/tkt4140/allfiles/

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

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

nrpm = 3500   # no of rpm of golf ball

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

# golf ball without lift
```

[8] https://raw.githubusercontent.com/lrhgit/tkt4140/master/allfiles/Utviklingsprosjekt/chapter1/programs_and_modules/cd

```
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()
```

1.2 How to make a python-module

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. The module name is the file name without the suffix .py [5]. To make a module of the ODE-solvers A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the first time the module name is encountered in an import statement hello world. (They are also run if the file is executed as a script.)

```
    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)

        scheme_list  = [euler, euler2, euler3, euler4, heun, heun2, rk4]
```

```
        for scheme in scheme_list:
            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

    def plot_ODEschemes_solutions():
        """Plot the linear solutions for the test schemes in scheme_list"""
        from numpy import linspace
        T = 1.5  # end of simulation
        N = 5  # no of time steps
        time = linspace(0, T, N+1)

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

        scheme_list  = [euler, heun]
        legends = []


        for scheme in scheme_list:
            z = scheme(f_local,z0,time)
            plot(time,z[:,-1])
            legends.append(scheme.func_name)


        legend(legends)
        show()

    test_ODEschemes()
    #plot_ODEschemes_solutions()
```

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] Software Foundation Python. *The Python Tutorial*. Python Software Foundation, 2014.

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