Inf5620 Project 1

Andreas Nygård Osnes

7. september 2013

Sammendrag

This project develops a solver for the differential equation governing the velocity of a skydiver with a quadratic air drag. The differential equation is derived, then implemented as a python program. The program is verified using nosetests, and then finally used to simulate a full parachute jump.

Deriving the differential equation

The differential equation is obtained by using Newtons second law:

$$\sum_i F_i = ma. \tag{1}$$

where $\sum F_i$ are the forces acting on the body, m is the mass of the body and a is the acceleration. The forces acting on the body are the gravity force and a quadratic drag force. A source force is also added for testing. We then have:

$$\sum_i F_i = F_g + F_d + F_s.$$

where the subscripts indicate the force (g = gravity, d=drag, s=source). The differential equation for the velocity of the motion is then:

$$m\frac{dv}{dt} = F_g + F_d + F_s.$$

Where v is the velocity. Inserting expressions for the forces we get:

$$m\frac{dv}{dt} = -mg - \frac{1}{2}C_D\rho A\,|v|\,v + F_s.$$

where g is the gravity acceleration, C_D is the drag coefficient, ρ is the mass density of the medium the body is moving in and A is the cross-sectional area of the body. In short, this equation can be written:

$$\frac{dv}{dt} = -g - a\,|v|\,v + \frac{F_s}{m} \tag{2}$$

where $a = \frac{1}{2m}C_D\rho A$. The input to the program should then be the mass of the body, the initial velocity, the final time, the timestep and the values of the constants C_D, ρ and A.

The discrete differential equation:

For the discretization of equation 2 we will apply a Crank-Nicolson scheme. This scheme reads:

$$\frac{v\,(t+\Delta t) - v\,(t)}{\Delta t} = \frac{1}{2}\left.\frac{dv}{dt}\right|_t + \frac{1}{2}\left.\frac{dv}{dt}\right|_{t+\Delta t} \tag{3}$$

Inserting equation 2 on the right-hand side we get:

$$\frac{v\,(t+\Delta t) - v\,(t)}{\Delta t} = \frac{1}{2}\left[-g - a\,|v\,(t)|\,v\,(t) + \frac{F_s\,(v\,(t)\,,t)}{m}\right] + \frac{1}{2}\left[-g - a\,|v\,(t+\Delta t)|\,v\,(t+\Delta t) + \frac{F_s\,(v\,(t+\Delta t)\,,t+\Delta t)}{m}\right]$$

$$\frac{v(t+\Delta t)-v(t)}{\Delta t} = -g + \frac{1}{2}\frac{F_s(v(t),t)}{m} + \frac{1}{2}\frac{F_s(v(t+\Delta t),t+\Delta t)}{m} - \frac{1}{2}a\left[|v(t)|\,v(t) + |v(t+\Delta t)|\,v(t+\Delta t)\right] \tag{4}$$

We now use a geometric average to approximate the term $[|v(t)|\,v(t) + |v(t+\Delta t)|\,v(t+\Delta t)]$ as:

$$\left[|v(t)|\,v(t) + |v(t+\Delta t)|\,v(t+\Delta t)\right] \approx |v(t)|\,v(t+\Delta t) \tag{5}$$

Inserting this approximation into equation 4 we get:

$$\frac{v(t+\Delta t)-v(t)}{\Delta t} = -g + \frac{1}{2}\frac{F_s(v(t),t)}{m} + \frac{1}{2}\frac{F_s(v(t+\Delta t),t+\Delta t)}{m} - a\,|v(t)|\,v(t+\Delta t)$$

which can be further manipulated to give:

$$v(t+\Delta t)\left(1+\Delta t a\,|v(t)|\right) = -\Delta t g + \Delta t\frac{1}{2}\frac{F_s(v(t),t)}{m} + \Delta t\frac{1}{2}\frac{F_s(v(t+\Delta t),t+\Delta t)}{m} + v(t)$$

$$v(t+\Delta t) = \frac{\Delta t\left(-g + \frac{1}{2}\frac{F_s(t)}{m} + \frac{1}{2}\frac{F_s(t+\Delta t)}{m}\right) + v(t)}{\left(1+\Delta t a\,|v(t)|\right)} \tag{6}$$

or in short:

$$v(t+\Delta t) = \frac{v(t) + \Delta t\cdot X\left(t+\frac{\Delta t}{2}\right)}{1+\Delta t\cdot Y\left(t+\frac{\Delta t}{2}\right)|v(t)|} \tag{7}$$

where

$$X\left(t+\frac{\Delta t}{2}\right) = -g + \frac{1}{2m}\left(F_s(t) + F_s(t+\Delta t)\right), \qquad Y(t) = a$$

Showing that a linear equation is not a solution

Because of the geometric average used in deriving equation 6 a general linear equation will not be a solution of our discrete scheme. This section will show this fact by inserting a general solution into the scheme. Without the source term, equation 6 is equal to:

$$\frac{v(t+\Delta t)-v(t)}{\Delta t} = -g - a\,|v(t)|\,v(t+\Delta t)$$

If we assume that v should be a linear function of t, e.g. $v(t) = At + B$ we get:

$$\frac{At + A\Delta t + B - At - B}{\Delta t} = -g - a\,|(At+B)|\,(At + A\Delta t + B)$$

We now make an assumption to get rid of the absolute value. If we take $At + B < 0$ we get:

$$A = -g + a\left(t^2A^2 + tA^2\Delta t + t2AB + AB\Delta t + B^2\right)$$

$$0 = \left(A^2a\right)t^2 + \left(A^2a\Delta t + 2ABa\right)t + \left(-g + ABa\Delta t + B^2a - A\right)$$

For this to hold for all t, the coefficients in front of each term must be separatly zero, so we get three equations:

$$A^2a = 0$$

$$\left(A^2a\Delta t + 2ABa\right) = 0$$

$$\left(-g + ABa\Delta t + B^2a\right) = 0$$

These equations imply that A must allways be zero to satisfy the discrete equation. Therefore a general linear function does not satisfy the discrete equation.

Fitting the source term so that a linear function is a solution

In general, we want our solver to be able to reproduce a linear function. This is very useful for verifying that the solver works properly. The differential equation including the source term is:

$$\frac{dv}{dt} = -g - a\left|v\right|v + \frac{F_s}{m}$$

solving for the source term gives:

$$F_s = m\frac{dv}{dt} + mg + ma\left|v\right|v$$

So if we want a linear solution, for example $v_e = At + B$ we get:

$$F_s\left(t\right) = mA + mg + ma\left|At + B\right|At + B \tag{8}$$

This is the general expression for the source term. In the discrete scheme however, the function must be slightly different as will be shown below.

Fitting the source term to a linear solution in the discrete scheme: In the discrete Crank-Nicolson scheme, we have:

$$\frac{v\left(t+\Delta t\right) - v\left(t\right)}{\Delta t} = -g + \frac{1}{2}\frac{F_s\left(v\left(t\right),t\right)}{m} + \frac{1}{2}\frac{F_s\left(v\left(t+\Delta t\right),t+\Delta t\right)}{m} - a\left|v\left(t\right)\right|v\left(t+\Delta t\right)$$

So any form of function will be a solution to the discrete equations if we require that:

$$\frac{1}{2}\frac{F_s\left(t\right)}{m} + \frac{1}{2}\frac{F_s\left(t+\Delta t\right)}{m} = g + \frac{v_e\left(t+\Delta t\right) - v_e\left(t\right)}{\Delta t} + a\left|v\left(t\right)\right|v\left(t+\Delta t\right)$$

Inserting a linear solution gives:

$$F_s\left(t + \frac{\Delta t}{2}\right) = mg + mA + ma\left|At + B\right|\left(At + A\Delta t + B\right)$$

now, to get the function $F_s\left(t\right)$ we shift the equation $\frac{\Delta t}{2}$ back in time and get:

$$F_s\left(t\right) = mg + mA + ma\left|A\left(t - \Delta t/2\right)\right|\left(At + A\Delta t/2 + B\right) \tag{9}$$

which is as can easily be seen, is not the same equation as obtained when fitting the source term from the differential equation. Equation 9 is the expression that will be implemented to perform a nosetest for the linear solution.

Nosetest for the linear solution When we apply the Crank-Nicolson scheme to a linear function, we expect it to be exact. As shown above, a linear solution is not in general a solution to the differential equation with the geometric average approximation. When inserting the source term derived above (Equation 9), a general linear solution (given by A and B) **will** be a solution, and our solver should be able to calculate it to machine precision. This can be tested in a nosetest.

Testing the convergence rate of the discrete scheme

Another useful test of our program will be to examine the convergence rate of our numerical scheme as Δt approaches zero. The error in the Crank-Nicolson scheme goes as $\mathscr{O}\left(\Delta t^2\right)$ and the error in the geometric average also goes as $\mathscr{O}\left(\Delta t^2\right)$. The total error in our numerical scheme should therefore also go as $\mathscr{O}\left(\Delta t^2\right)$. This can be tested by assuming that the error in the numerical solution goes as

$$E = C\Delta t^r$$

Where C is a constant, and r is a number reflecting how fast the numerical approximation converges to the exact solution. If we now fit a source term to a function which will not be represented exactly by the Crank-Nicolson scheme, as in the general form in Equation 8, we can compute the error in our numerical approximation. A representative number for the error in the numerical approximation is:

$$E = \sqrt{\Delta t \sum_{i}^{N} \left(v_e\left(t_i\right) - v\left(t_i\right)\right)^2} \tag{10}$$

This is the expression that will be used for the error in the numerical approximation. Using equation 10, we can compute r from:

$$r_i = \frac{\ln\left(E_{i-1}/E_i\right)}{\ln\left(\Delta t_{i-1}/\Delta t_i\right)}$$

where the values for E_i are taken from simulations with corresponding Δt_i. If our solver works correctly, r should now be close to 2. This can be tested in a nosetest.

Plots

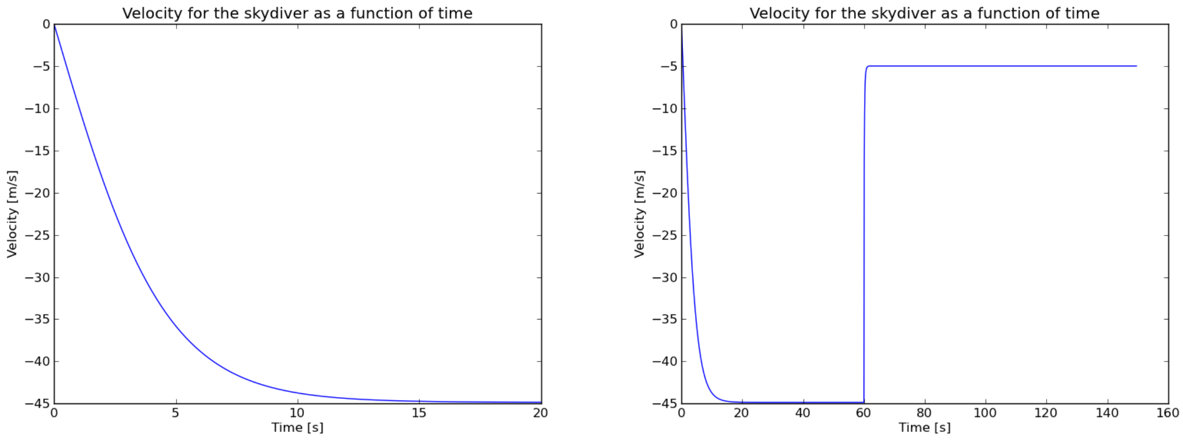
The visualization of the forces as functions of time is implemented as two separate functions. One for visualizing the forces without opening the parachute, and one with opening of the parachute. These functions are compute the forces and then show and save the plots. There are also separate functions for plotting the velocity of the skydiver and the position. The values for the different variables used for obtaining these plots are:

$$C_D = 1.5, \qquad A = 0.65, \qquad \rho = 1.0, \qquad m = 100.0, \qquad v_0 = 0, \qquad t_p = 60$$

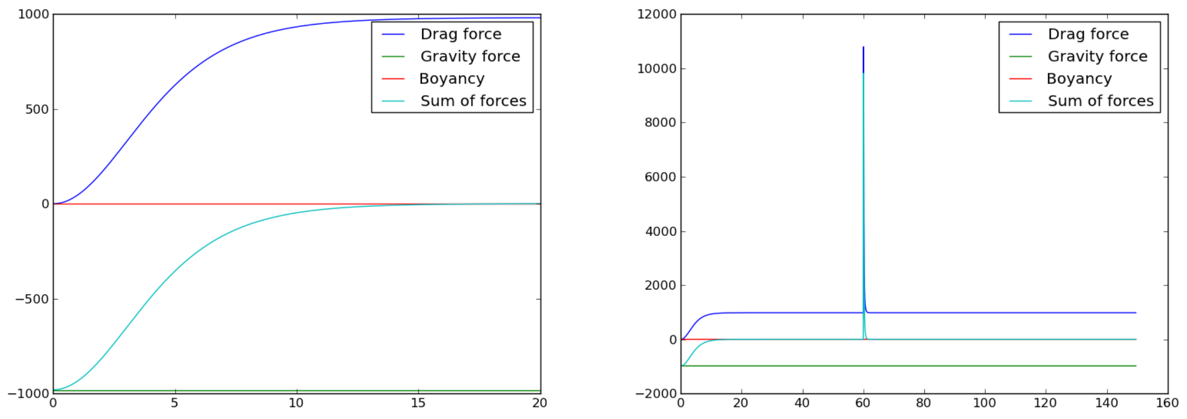
and after opening the parachute at time $t_p = 60s$, C_D and A were changed to

$$C_D = 1.8, \qquad A = 44.0$$

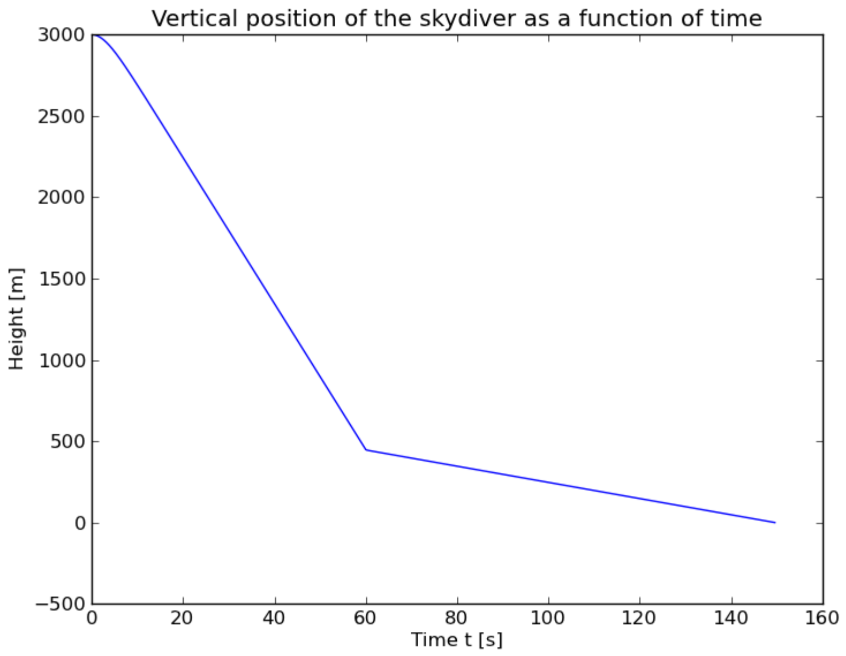
The plots are all shown in figure 1-3.



Figur 1: Skydiver velocity without opening the parachute (left) and with opening the parachute at $t_p = 60s$ (right)



Figur 2: Forces without opening the parachute (left) and with opening the parachute at $t_p = 60s$ (right)



Figur 3: Position of the skydiver as a function of time

Source code

Listing 1: skydiving.py

```
import sys
from scitools.std import zeros, linspace, sqrt, log, concatenate
import matplotlib.pyplot as plt

def define_command_line_options():
        import argparse
        parser = argparse.ArgumentParser()
        parser.add_argument('--T', '--stop_time', type=float,
                        default=20.0, help='end␣time␣of␣simulation',
                                                  metavar='t')
        parser.add_argument('--dt', type=float,
                                          default=0.1, help='timestep for the discrete 
                                             apporoximation',
                                          metavar='dt')
        parser.add_argument('--v0', '--initial_condition', type=float,
                                          default=-0.0, help='initial condition v(0)',
                                          metavar='v0')
        parser.add_argument('--makeplot', action='store_true',
                                          help='display plot or not')
        parser.add_argument('--rho', type=float,
                                default=1.0, help='air mass density',
                                metavar='rho')
        parser.add_argument('--Cd', type=float,
                                default=1.2, help='drag coefficient',
                                metavar='Cd')
        parser.add_argument('--m', '--body_mass', type=float,
                                          default=100., help='body mass', metavar='m')
        parser.add_argument('--A', type=float,
                                default=0.5, help='body cross sectional area',
                                metavar='A')
        parser.add_argument('--tp', type=float, default=-1, help='time of parachute 
            release',
                                metavar='tp')
        return parser


def plot_v(t, v):
    p1 = plt.plot(t,v)
    plt.xlabel('Time [s]')
    plt.ylabel('Velocity [m/s]')
    plt.title('Velocity for the skydiver as a function of time')
    plt.show()
    plt.savefig('Parachute_velocity.png')

def plot_forces(t, v, m, a):
    plt.figure()
    drag = -m*a*abs(v)*v
    grav = [-m*9.81]*len(v)
    Boyancy = [1. * 9.81 * 0.1]*len(v) # rho * g * V
    Fsum = drag+grav+Boyancy
    plt.plot(t, drag, t, grav, t, Boyancy,  t, Fsum)
    plt.legend(["Drag force", "Gravity force", "Boyancy", "Sum of forces"])
    plt.savefig('Forces.png')

def plot_forces_parachute(t, v, dt, tp, m, a_first, a_last):
    """
    Function for plotting the forces when running
    the program with parachute deployment
    """
    plt.figure()
    drag = zeros(len(v))
    for i in range(len(v)):
        if(i*dt <= tp):
            drag[i] = -m*a_first*abs(v[i])*v[i]
        else:
            drag[i] = -m*a_last*abs(v[i])*v[i]
    grav = [-m*9.81]*len(v)
    Boyancy = [1. * 9.81 * 0.1]*len(v) # rho * g * V
    Fsum = drag+grav+Boyancy
    plt.plot(t, drag, t, grav, t, Boyancy,  t, Fsum)
    plt.legend(["Drag force", "Gravity force", "Boyancy", "Sum of forces"])
    plt.savefig('Parachute_forces.png')

def plot_x(t, x):
    """
    Function for plotting the vertical position of the skydiver
    as a function of time
    """
    plt.figure()
    plt.plot(t, x)
    plt.title("Vertical position of the skydiver as a function of time")
    plt.xlabel("Time t [s]")
    plt.ylabel("Height [m]")
    plt.savefig('Parachute_position.png')


def skydiving_iterate(v, t, dt, X, Y):
    """
    Problem specific function. Implements the needed computation for
    doing one time-step iteration.

    """
    return (v + dt*X(t))/(1 + dt*Y(t)*abs(v))

def solver(T, dt, v0, Cd, rho, A, m, Source=None):
    a = Cd*rho*A/(2*m)
    v = zeros(int(T/dt) + 1)
    v[0] = v0; g = 9.81;
    def X(t):
        if(Source == None):
            return -g
        else:
            return -g + Source(t+dt/2.)/m
            #return -g + 0.5*(Source(t) + Source(t+dt))/m

    def Y(t):
        return a

    for i in range(1,len(v)):
        v[i] = skydiving_iterate(v[i-1], dt*(i-1), dt, X, Y)
    return v, linspace(0, T, T/dt +1)

def solver_parachute(T, dt, v0, tp, Cd, rho, A, m):
    v = []; v.append(v0); t=[]; t.append(0);
    x = []; x.append(3000.)
    def X(t):
        return -9.81
    def Y(t):
        return Cd*rho*A/(2.*m)
    while t[-1]<=tp:
        v.append(skydiving_iterate(v[-1], t[-1], dt, X, Y))
        x.append(x[-1] + dt*v[-1])
        t.append(t[-1] + dt)
    def Z(t):
        return 1.8*1.0*44./(2.*m)
    while x[-1] > 0:
        v.append(skydiving_iterate(v[-1], t[-1], dt, X, Z))
        x.append(x[-1] + dt*v[-1])
        t.append(t[-1] + dt)
    return v, t, x


def main():
    parser = define_command_line_options()
    args = parser.parse_args()
    if(args.tp == -1):
        v, t = solver(args.T, args.dt, args.v0, args.Cd, args.rho, args.A, args.m)
        if(args.makeplot):
            plot_v(t, v)
            plot_forces(t, v, args.m, args.Cd*args.rho*args.A/(2.*args.m))
            raw_input()
    else:
        v, t, x = solver_parachute(args.T, args.dt, args.v0, args.tp, args.Cd, args.rho,
                                   args.A, args.m)
        if(args.makeplot):
            plot_v(t, v)
            plot_forces_parachute(t, v, args.dt, args.tp, args.m, args.Cd*args.rho*args.
                A/(2.*args.m),
                                  1.8*args.rho*44/(2.*args.m))
            plot_x(t, x)
            raw_input()

if(__name__ == "__main__"):
        main()

def test_constant():
    """
    Test for the solvers ability to reproduce a constant solution.
    """
    import nose.tools as nt

    C = -0.11; g = 9.81; m = 50.; T = 10.; dt = 0.01;
    Cd = 1.2; rho = 1.0; A = 0.5;
    a = Cd*rho*A/(2.*m)
    def src(t):
        return m*g + m*a*abs(C)*C
    v, t = solver(T, dt, C, Cd, rho, A, m, Source=src)
    for i in range(len(v)):
        nt.assert_almost_equal(C, v[i], delta=1e-12)

def test_linear():
    """
    Test for the solvers ability to reproduce a linear solution
    by fitting the source term so that a linear solution should indeed
    be a solution to the differential equation
    """
    import nose.tools as nt
    A = -0.11; B = -0.13; g = 9.81; m = 50.; T = 10.; dt = 0.01;
    Cd = 1.2; rho = 1.0; A = 0.5;
    a = Cd*rho*A/(2.*m)
    def exact(t):
        return A*t+B

    def src(t):
        return m*g + m*a*abs(exact(t-dt/2.))*exact(t+dt/2.) + m*A
    v, t = solver(T, dt, B, Cd, rho, A, m, Source=src)
    ve = exact(t)
    diff = abs(ve - v)
    nt.assert_almost_equal(diff.max(), 0, delta=1e-12)

def test_terminalVelocity():
    """
    Test for the terminal velocity with no source term.
    """
    import nose.tools as nt
    T = 30.; dt = 0.1; g = 9.81; m = 50.;
    Cd = 1.2; rho = 1.0; A = 0.5;
    a = Cd*rho*A/(2.*m)
    v, t = solver(T, dt, -0.1, Cd, rho, A, m)
    nt.assert_almost_equal(v[-1], -sqrt(g/a), delta=1e-4)

def test_convergenceRates():
    dt_start = 1.0; num_dt = 10
    E_values = zeros(num_dt)
    T = 10.; g = 9.81; m = 50.;
    Cd = 1.2; rho = 1.0; A = 0.5;
    a = Cd*rho*A/(2.*m)
    dt = zeros(num_dt); dt[0] = dt_start
    for i in range(1,len(dt)):
        dt[i] = dt[i-1]/2.
    print "dt=", dt

    D = -0.39; B = 0.76; C = -0.145
    def exact(t):
        return D*t**3  + B*t + C
    def src(t):
        return m*g + m*a*abs(exact(t))*exact(t) + m*(3*D*t**2 + B)

    for i in range(num_dt):
        v, t = solver(T, dt[i], exact(0), Cd, rho, A, m, src)
        ve = exact(t)
        diff = v - ve
        E_values[i] = sqrt(dt[i]*sum(diff**2))

    r=zeros(len(E_values)-1)
    for i in range(1, len(r)):
        r[i] = (log(E_values[i-1]/E_values[i]))/(log(dt[i-1]/dt[i]))
    print("E=", E_values)
    print("r=", r)
    import nose.tools as nt
    nt.assert_almost_equal(r[-1], 2, delta=0.1)
```