skydiving

Milad H. Mobarhan

September 09, 2013

1 Project 1: Simulate parachuting

Summary. The aim of this project is to develop a general solver for the vertical motion of a body with quadratic air drag, verify the solver, apply the solver to a skydiver in free fall, and finally apply the solver to a complete parachute jump.

1.1 Mathematical problem

A body moving vertically is subject to three different types of forces: the gravity force, the drag force, and the buoyancy force. All the mentioned forces act in the vertical direction. Newton's second law of motion applied to the body says that the sum of these forces must equal the mass of the body times its acceleration a in the vertical direction. Taking downward as positive direction, Newton's second law gives us

$$ma = F_g + F_b + F_d,$$

The gravity froce is $F_g = mg$, where m is the mass of the body and g is the acceleration of gravity. The uplift or buoyancy force ("Archimedes force") is $F_b = -\varrho g V$, where ϱ is the density of the fluid, in our case air, and V is the volume of the body. The drag force is quadratic in the velocity:

$$F_d = -\frac{1}{2} C_D \varrho A |v| v,$$

where C_D is a dimensionless drag coefficient depending on the body's shape, and A is the cross-sectional area as produced by a cut plane, perpendicular to the motion, through the thickest part of the body.

By using $a = dv/dt$ and $m = \varrho_b V$, with ϱ_b as the density of the body, the equation of motion can be expressed as:

$$v'(t) = -\frac{1}{2} C_D \frac{\varrho A}{\varrho_b V} |v| v + g\left(\frac{\varrho}{\varrho_b} - 1\right) = -a|v|v + b$$

where $a = \frac{1}{2} C_D \frac{\varrho A}{\varrho_b V}$ and $b = g\left(\frac{\varrho}{\varrho_b} - 1\right)$.

1.2 Numerical solution method

We introduce a mesh in time with points $0 = t_0 < t_1 \cdots < t_{N_t} = T$. For simplicity, we assume constant spacing Δt between the mesh points: $\Delta t = t_n - t_{n-1}$, $n = 1, \ldots, N_t$. Let u^n be the numerical approximation to the exact solution at t_n.

The Crank-Nicolson method is used to solve the differential equation numerically. This method gives a nonlinear equation in v when applied to our system, but we can use a geometric average of $|v|v$ in time to linearize the equation of motion with quadratic drag:

$$u^{n+1} = \frac{u^n + \Delta t b^{n+\frac{1}{2}}}{1 + \Delta t a^{n+\frac{1}{2}} |u^n|}$$

for $n = 0, 1, \ldots, N_t - 1$.

1.3 Implementation

In this project classes have been used for implementation. A class `Problem` holds the definition of the physical problem, a class `ODESolver` holds the data and methods needed to numerically solve the problem, and a class `Visualizer` to make plots.

The `ODESolver` class is implemented as the superclass of all numerical methods for solving ODEs. This class provide all functionality that is commen to all numerical methods for ODEs. The `ODESolver`class has a `advance` method which is empty in the superclass, since the method is to be implemented by various subclasses for various numerical schemes. The Crank-Nicolson scheme is implemented in the `advance` method of `CNQuadratic`, where the later is a subclass of `ODESolver`.

In [1]:
```
class ODESolver():
    """
    Superclass for numerical methods solving scalar ODEs

      u'(t) = f(u, t)
    """
    def __init__(self,problem, u0, dt, T):
        self.problem, self.u0, self.dt, self.T = \
                                        problem, u0, dt ,T

    def advance(self):
        """Advance solution one time step."""
        raise NotImplementedError


    def solve(self):
       """
       Advance solution in time until t < T.
       """
       self.u = []
       self.t = []

       self.u.append(float(self.u0))
       self.t.append(0)
       self.dt = float(self.dt)            # avoid integer division
       Nt = int(round(self.T/self.dt))     # no of time intervals
       self.T = Nt*self.dt                 # adjust T to fit time step dt

       tNew = 0
       while tNew <= self.T:
           uNew = self.advance()
```

```
            self.u.append(uNew)
            tNew = self.t[-1] + self.dt
            self.t.append(tNew)

        return array(self.u), array(self.t)


class CNQuadratic(ODESolver):
    """
    Computes u(t_n+1) from u(t_n), by using a Crank-Nicolson scheme with
    geometeric average approximation,  for ODE of the type:

        u'(t) = -a(t)*|u(t)|u(t)+b(t).

    """

    def advance(self):
        u, dt  = self.u[-1], self.dt

        a = self.problem.a(self.t[-1]+dt*0.5)
        b = self.problem.b(self.t[-1]+dt*0.5)


        uNew = (u + dt*b)/(1+dt*a*abs(u))

        return uNew
```

1.4 Verfication

In order to verify the solver, the method of manufactured solutions is used, by fitting a source term. We choose a linear solution $u(t) = ct + d$. From the initial condition it follows that $d = u_0$. Inserting this u in the ODE results in:

$$\frac{u^{n+1} - u^n}{\Delta t} = -a^{n+1/2}|u^n|u^{n+1} + b^{n+1/2}$$

$$c = -a^{n+1/2}|ct_n + u_0|(ct_{n+1} + u_0) + b^{n+1/2}$$

Any function $u(t) = ct + u_0$ is then a correct solution if we choose

$$b^{n+1/2} = c + a^{n+1/2}|ct_n + u_0|(ct_{n+1} + u_0)$$

With this b there are no restrictions on a and c. Without the source term, the linear function do not satisfy the equation:

$$u^{n+1} = \frac{u^n}{1 + \Delta t a^{n+\frac{1}{2}}|u^n|},$$

so a linear function of t does not fulfill the discrete equations because of the geometric mean used for the quadratic drag term.

For further verification, the method of manufactured solutions, is used to compute the convergence rate in this problem. The expected error goes like Δt^2 because we use a centered finite difference approximation with error $O(\Delta t^2)$ and a geometric mean approximation with error $O(\Delta t^2)$. A nosetest is implemented for checking that the convergence rate is correct, by using a linear solution of type $u(t) = ct + u_0$. In this case the source term is set to

$$b^n = c + a^n|ct_n + u_0|(ct_n + u_0)$$

to satisfy

$$\frac{u^{n+1} - u^n}{\Delta t} = -a^n|u^n|u^n + b^n.$$

1.5 Running tests

We have implemented two nosetests, in a seperate file named `test_nose.py`. One of the test checks that a linear function of t is a solution of the discrete equations. The nose test checks that this solution is reproduced to machine precision. The other test checks that we have the correct convergence rate. Both tests can be executed by evaluating the cell below.

In [2]:
```
import subprocess
try:
    output = subprocess.check_output(["nosetests", "-s"], stderr=subproces
    print output
except subprocess.CalledProcessError as ex:
    print ex.output
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.178s

OK
test_linearSolution succeeded!
test_convergenceRate succeeded!
```

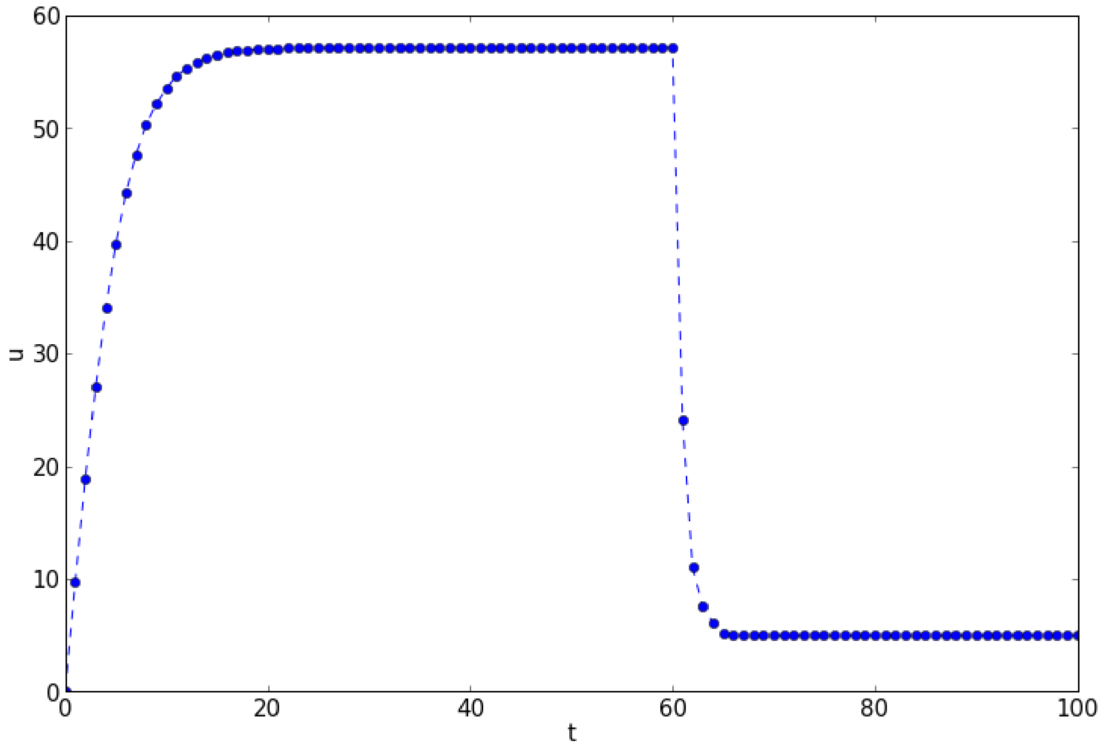
1.6 Complete parachute jump

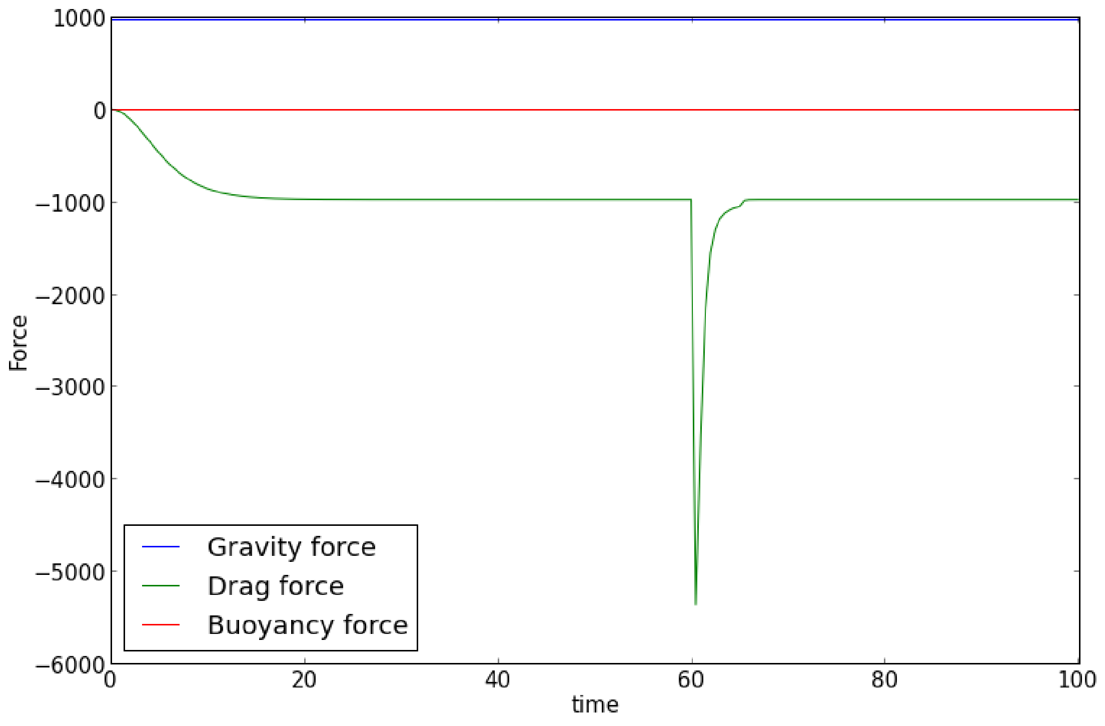
The mass of the human body and equipment can be set to 100kg. A skydiver in spread-eagle formation has a cross-section of $0.5m^2$ in the horizontal plane, while when the parachute is open the cross-section area perpendicular to the motion is $44m^2$. The density of air is set to be constant, $1kg/m^3$. The drag coefficient for a man in upright position can be set to 1.2, while for an open parachute it can be taken as 1.8. We sssume that it takes 5s to increase the area linearly from the original to the final value. We start with zero velocity and release the parachute after $t_p = 60$s.

In [3]:
```
%pylab inline
import sys
sys.path.append("../../src/")
import skydiving as sd

#Set up problem solver, problem and vizualizer
problem    = sd.skydiving(A = 0.5, Ap = 44.0, dt = 0.1, dtp = 5,
                       m = 100.0, rho = 1.0, Cd = 1.2, Cdp = 1.8, tp = 60.
solver     = sd.CNQuadratic(problem, u0 = 0.0, dt = 0.1, T = 100)
viz        = sd.Visualizer(solver,False)

# Solve
u,t = solver.solve()

# Make plot
rcParams["figure.figsize"] = (12,8)
rcParams["font.size"] = 15
plt = viz.plot()
plt.show()
```

```
pltforce = viz.plotForces()
pltforce.show()
```

```
Populating the interactive namespace from numpy and matplotlib
```





The first plot shows velocity as a function of time, while the second one shows the various forces as a function of time. The terminal velocity before the parachute opens is

```
In [4]: vt_freefall = u.max()
        print "Terminal velocity before the parachute opens: ",vt_freefall
```

```
Terminal velocity before the parachute opens:  57.1649254783
```

The terminal velocity after the parachute opens is

```
In [5]: vt_landing = u[-1]
        print "Terminal velocity after the parachute opens: ",vt_landing
```

```
Terminal velocity after the parachute opens:  4.97556812654
```