LUND
UNIVERSITY

Module 1
How hard can I kick that lunch box?

Robin Emanuelsson

20 juli 2022

Course: FYTN03

Introduction

When doing things in real life thing all of a sudden get way more complicated then your average physics problem in a text book. Things like friction and air resistance become a factor to consider and just the fact that you must rely on measurements, with all it's flaws, makes things quite the task to solve with just pen and paper.
This is were a computer come in handy. With a computer you can numerically solve a problem, like ours, were we don't know the initial values, or how hard we kicked the lunch box, but only were it landed and how high it went.

Theory

Equation opf motions

The classical equation of motion is, of course, Newton's equation

$$\vec{F} = m\vec{a} \tag{1}$$

where F is the force, m is the mass of the object and a is the acceleration of the object. For the simple case where we have no air resistance we can easily solve this analytically. The Lagrangian of the problem is

$$\mathcal{L} = \frac{1}{2}m\left(\dot{x}^2 + \dot{y}^2\right) + mgy \tag{2}$$

where x is the horizontal coordinate, y is the vertical coordinate (both coordinates are of course dependent on time) and g is the gravitational acceleration.
Solving the Euler-Lagrange equation with the an unknown initial velocity v_0 and angle θ, and at height h we get

$$\begin{cases} x &= v_0\cos(\theta)t \\ y &= v_0\sin(\theta)t - \frac{gt^2}{2} + h \end{cases} \tag{3}$$

The range of the lunch box can easily be calculated. We know that the projectile lands at $y = 0$, meaning that we have

$$0 = v_0\sin(\theta)t - \frac{gt^2}{2} + h \tag{4}$$

Solving for t we get

$$t = \frac{v_0\sin(\theta)}{g} + \frac{\sqrt{v_o^2\sin^2(\theta) + 2gh}}{g} \tag{5}$$

The distance R will be this time t plugged in to the equation for the x-component. We get

$$R = v_0\cos(\theta)\left[\frac{v_0\sin(\theta)}{g} + \frac{\sqrt{v_o^2\sin^2(\theta) + 2gh}}{g}\right] \tag{6}$$

which can be simplified as

$$R = \frac{v_0^2\sin(\theta)}{2g}\left[1 + \sqrt{1 + \frac{2gh}{v_0^2\sin^2(\theta)}}\right] \tag{7}$$

Of course thing in life are never this easy. We must take air resistance into account. The drag force from the air resistance is approximately

$$F_{Drag} \approx -B_1 v - B_2 v^2 \tag{8}$$

The B_1 term will be negligible in our case and the B_2 term can be written as $B_2 \approx -\frac{1}{2} C \rho A$, where C is the drag coefficient, ρ is the air density and A is the area of the object.
The equation of motion, from Newton's equation, is

$$\begin{cases} \frac{d^2x}{dt^2} & = -\frac{B_2 v v_x}{m} \\ \frac{d^2x}{dt^2} & = -g - \frac{B_2 v v_y}{m} \end{cases} \tag{9}$$

here $v = \sqrt{v_x^2 + v_y^2}$.

Runge-Kutta method

A common method for solving 2nd second-order differential equation is to write it as two first-order differential equation. Using this equation 9can be written as

$$\begin{cases} \frac{dx}{dt} & = v_x \\ \frac{dv_x}{dt} & = \frac{-B_2 v v_x}{m} \\ \frac{dy}{dt} & = v_y \\ \frac{dv_y}{dt} & = \frac{-B_2 v v_y}{m} \end{cases} \tag{10}$$

or in a more convenient vector form

$$\frac{d\vec{Y}}{dt} = \vec{F} \tag{11}$$

where

$$\vec{Y} = \begin{bmatrix} x \\ v_x \\ y \\ v_y \end{bmatrix} \tag{12}$$

$$\tag{13}$$

and

$$\vec{F} = \begin{bmatrix} v_x \\ \frac{-B_2 v v_x}{m} \\ v_y \\ \frac{-B_2 v v_y}{m} \end{bmatrix} \tag{14}$$

To be able to numerically solve this, time must be discretized. A very popular method of solving ordinary differential equation in discretized time is the 4th order Runge-Kutta method, with its local truncation error of $\mathcal{O}(h^5)$. The Runge-Kutta methof is simply

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{15}$$

where

$$k_1 = hf(y_n) \tag{16}$$

$$k_2 = hf\left(y_n + \frac{k_1}{2}\right) \tag{17}$$

$$k_3 = hf\left(y_n + \frac{k_2}{2}\right) \tag{18}$$

$$k_4 = hf\left(y_n + k_3\right) \tag{19}$$

$$\tag{20}$$

where h is the lattice distance in the discretized time, y_{n+1} can be seen as $\frac{d\vec{V}}{dt}$ and the function f can be seen as $\vec{F}$.

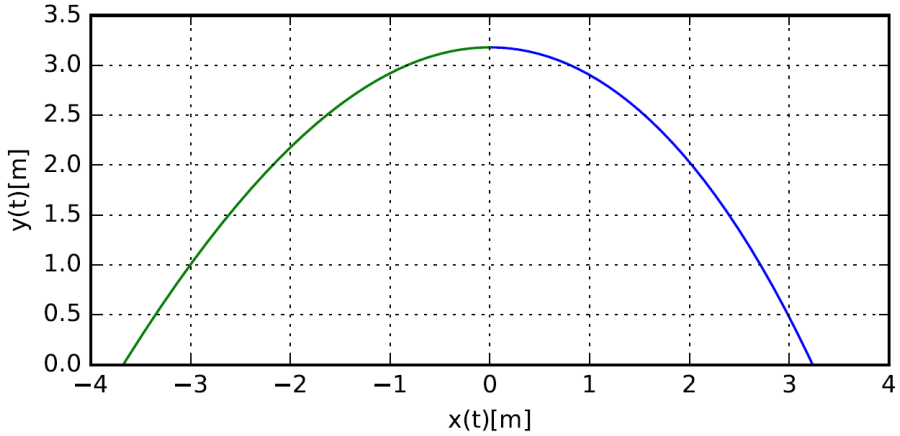
Method

The experimental part was done in the following way.
The lunch box was place kicked parallel to a wall which was used as a reference point. The trajectory was recorded with a mobile phone. The recording was used to determine the maximum height of the lunch box trajectory using the wall, which had clear markings on it. A regular measuring tape was used to determine the length of the box trajectory. The box did not fly perfectly parallel to the wall so the distance from the wall, at the point of impact, was also measured and the Pythagorean theorem could be used the get a good estimate on how far the box flew.
For the numerical part the fourth order Runge–Kutta method was used to approximate the differential equations. The only measured data available was the maximum height the lunch box attain and at this point it is also known that vertical velocity is zero, and the distance the lunch box flew. Using the point of maximum height as starting point, only the horizontal velocity needs to be found by trail and error to get the right horizontal distance. The simulation need to be run backwards in time to attain the real initial velocity. The simulation stopped when the numerical distance was within a threshold of the actual measured distance.

Results

Physical quantity	Value	Unit
Distance of the lunchbox	6.91	m
Height of the lunchbox	3.18	m
Drag of the lunch box	0.00339292006[1]	N/A
textMass of the lunch box	0.07	kg
Threshold to the measured distance	10^4	m
Time step	10^3	N/A

Tabell 1: Table of the physical quantities
[1] The drag is very hard to estimate, especially for a lunch box, so not much time went in to get a exact value.



Figur 1: Plot of the trajectory of the lunch box. The green is the simulation backwards in time and the blue is the simulation forwards in time.

The result for the horizontal velocity at the maximum height was $v_{x,maxH} = 4.42^{m}/_{s}$, with this value the inital velocities was $(v_{x,init}, v_{y,init}) = (3.95, 7.70)^{m}/_{s}$.
The difference between the measured distance and the simulated distance was $R_{sim} - R_{meas} = 6.51 \cdot 10^{-5}$m.
The truncation error is calculated by simulating with time step h and $2nh$,(n is an integer), and taking the diffrence between these two, called E, and multiplied with the term $A = \frac{1}{2^m - 1}$, where m is the theoretical magnitude of the error, which was 4.

n	$A \cdot E$
0	$8.6{\cdot}10^{-9}$
1	$2.6{\cdot}10^{-8}$
2	$1.4{\cdot}10^{-8}$

Tabell 2: Table of the truncation error

Conclusion

The results that we kick the lunch box about 8.65 $^{m}/_{s}$ seems very reasonable. I figure 1 we can see that the lunch box travels a smaller distance, relative to the point of maximum height, when we go forward in time. Which means that the extra friction term is going it's job.
The biggest source of error is of course the measuring error which we have completely neglect here. The drag coefficient is also a big source of error since the box doesn't fly perfectly still bit instead spins. Both of these has been either ignore of very simplified since the focus on this excises wasn't about this. The error was quite small which is not unreasonable since, first of all, the Runge-Kutta method is very good and, secondly, we had a very small time step and a small tolerance.

Appendix

Finding the right initial conditions

```
import matplotlib.pyplot as plt
import numpy as np
#Parameters
tol=1e-4 #How close we accept the algorithm to get to the real value
Rtrue=6.91#The length of the projectile motion
H=3.18#The height of the projectile motion
B2=4*np.pi*0.03**2*1.2/2*0.5#Drag
m=0.07#mass of the lunch box
deltaT=1e-3#timestep
g=9.82#gravity acc.


def fforw(array): #Function for the EoM forwards in time
    v=np.sqrt(array[1]**2+array[3]**2)
    return np.array([array[1], -B2*array[1]*v/m, +array[3], -(g+B2*array[3]*v/m)])



def fback(array): #Function for the EoM backwards in time
    v=np.sqrt(array[1]**2+array[3]**2)
    return np.array([-array[1], +B2*array[1]*v/m, -array[3], g+B2*array[3]*v/m])


v0x=4.2 #Starting trying at v0x=4.2 m/s since we know the answer is around 4.2(fro
Rtot=[0]
rbacklist=[]
rforwlist=[]
while True:
    rback=np.array([0,v0x,H,0])#r is our vector
    rforw=np.array([0,v0x,H,0])
    xforw=[]
    yforw=[]
    xback=[]
    yback=[]
    while rforw[2]>0: #Runge-Kutta forwards in time
        k1=deltaT*fforw(rforw)
        k2=deltaT*fforw(rforw+1/2*k1)
        k3=deltaT*fforw(rforw+1/2*k2)
        k4=deltaT*fforw(rforw+1*k3)
        rforw=rforw+k1/6+k2/3+k3/3+k4/6
        xforw.append(rforw[0])
        yforw.append(rforw[2])
    rforw[0]=xforw[-2]+(xforw[-1]-xforw[-2])*(yforw[-2]/(yforw[-2]-yforw[-1]))
    #Interpolation
    while rback[2]>0:#Runge-Kutta backwards in time
        k1=deltaT*fback(rback)
        k2=deltaT*fback(rback+1/2*k1)
        k3=deltaT*fback(rback+1/2*k2)
```

```
        k4=deltaT*fback(rback+1*k3)
        rback=rback+k1/6+k2/3+k3/3+k4/6
        xback.append(rback[0])
        yback.append(rback[2])
    rback[0]=xback[-2]+(xback[-1]-xback[-2])*(yback[-2]/(yback[-2]-yback[-1]))
    #Interpolation
    rbacklist.append(abs(rback[0]))
    rforwlist.append(rforw[0])
    Rtot.append(rforw[0]+abs(rback[0]))
    if abs(Rtot[-1]-Rtrue)>abs(Rtot[-2]-Rtrue): #stops when the best initail value
        v0x=v0x-tol
        break
    v0x+=tol
print(v0x)



def forwardtrajectory(v0x): #Get the initail values at t=0
    rforw=np.array([0,v0x,H,0])
    posx = [rforw[0]]
    posy = [rforw[2]]
    while rforw[2]>0:
        k1=deltaT*fforw(rforw)
        k2=deltaT*fforw(rforw+1/2*k1)
        k3=deltaT*fforw(rforw+1/2*k2)
        k4=deltaT*fforw(rforw+1*k3)
        rforw=rforw+k1/6+k2/3+k3/3+k4/6
        posx.append(rforw[0])
        posy.append(rforw[2])
    return posx,posy

def backtrajectory(v0x):
    rback=np.array([0,v0x,H,0])
    posx = [rback[0]]
    posy = [rback[2]]
    velox = [rback[1]]
    veloy= [rback[3]]
    while rback[2]>0:
        k1=deltaT*fback(rback)
        k2=deltaT*fback(rback+1/2*k1)
        k3=deltaT*fback(rback+1/2*k2)
        k4=deltaT*fback(rback+1*k3)
        rback=rback+k1/6+k2/3+k3/3+k4/6
        posx.append(rback[0])
        velox.append(rback[1])
        posy.append(rback[2])
        veloy.append(rback[3])
    velox[-1]=velox[-2]+(velox[-1]-velox[-2])*(posy[-2]/(posy[-2]-posy[-1]))
    veloy[-1]=veloy[-2]+(veloy[-1]-veloy[-2])*(posy[-2]/(posy[-2]-posy[-1]))
    return posx,velox[-1],posy, veloy[-1]

xforw, yforw = forwardtrajectory(v0x)
```

```
xvelo, yvelo = backtrajectory(v0x)[1], backtrajectory(v0x)[3]
xback, yback = backtrajectory(v0x)[0], backtrajectory(v0x)[2]

print(xvelo,yvelo)
plt.plot(xforw, yforw, xback,yback )
plt.gca().set_aspect('equal', adjustable='box')
plt.xlabel('x(t)[m]')
plt.ylabel('y(t)[m]')
plt.ylim(0, 3.5)
plt.grid()
plt.savefig('foo.pdf')
plt.show()
```

Finding the error

```
import matplotlib.pyplot as plt
import numpy as np
#Parameters
Rtrue=6.91 #Length of throw
H=3.18 #Maximum height
B2=4*np.pi*0.03**2*1.2*0.5/2 #A*density*C/2
m=0.07 #Mass of lunch box
g=9.82
v0x=5.4651850556 #Calculated by main program
v0y=8.76241539953 #Calculated from main program

def fforw(array): #function for EoM forwards in time
    v=np.sqrt(array[1]**2+array[3]**2)
    return np.array([array[1], -B2*array[1]*v/m, +array[3], -(g+B2*array[3]*v/m)])

def fback(array):#function for EoM backwards in time
    v=np.sqrt(array[1]**2+array[3]**2)
    return np.array([-array[1], +B2*array[1]*v/m, -array[3], g+B2*array[3]*v/m])

def backtrajectory(deltaT, v0x): #Trajectory backwards in time
    rback=np.array([0,v0x,H,0])
    posx = [rback[0]]
    posy = [rback[2]]
    velox = [rback[1]]
    veloy= [rback[3]]
    while rback[2]>0:#Runge-Kutta
        k1=deltaT*fback(rback)
        k2=deltaT*fback(rback+1/2*k1)
        k3=deltaT*fback(rback+1/2*k2)
        k4=deltaT*fback(rback+1*k3)
        rback=rback+k1/6+k2/3+k3/3+k4/6
        posx.append(rback[0])
        velox.append(rback[1])
        posy.append(rback[2])
        veloy.append(rback[3])#Interpolation
    velox[-1]=velox[-2]+(velox[-1]-velox[-2])*(posy[-2]/(posy[-2]-posy[-1]))
    veloy[-1]=veloy[-2]+(veloy[-1]-veloy[-2])*(posy[-2]/(posy[-2]-posy[-1]))
    return posx,velox[-1],posy, veloy[-1]
```

```
def forwtrajectory(deltaT): ##Trajectory forwards in time
    rforw=np.array([0,v0x,0,v0y])
    posx = [rforw[0]]
    posy = [rforw[2]]
    velox = [rforw[1]]
    veloy= [rforw[3]]
    while rforw[2]>=0:
        k1=deltaT*fforw(rforw)
        k2=deltaT*fforw(rforw+1/2*k1)
        k3=deltaT*fforw(rforw+1/2*k2)
        k4=deltaT*fforw(rforw+1*k3)
        rforw=rforw+k1/6+k2/3+k3/3+k4/6
        posx.append(rforw[0])
        velox.append(rforw[1])
        posy.append(rforw[2])
        veloy.append(rforw[3]) #Interpolation
    velox[-1]=velox[-2]+(velox[-1]-velox[-2])*(posy[-2]/(posy[-2]-posy[-1]))
    veloy[-1]=veloy[-2]+(veloy[-1]-veloy[-2])*(posy[-2]/(posy[-2]-posy[-1]))
    return posx,velox[-1],posy, veloy[-1]

h=0.001 #Changing this by a factor 10 changes the global error by approximetly 1E
deltaT=[h,2*h]

hxforw=[]
hyforw=[]
for i in deltaT: #Giving values of x,y for forwards and backwards trajectory for h
    hxforw.append(forwtrajectory(i)[0])
    hyforw.append(forwtrajectory(i)[2])

forwerrorx=[]
forwerrory=[]

for n in range(len(hxforw[1])-1): #Returns the difference between x and y position
    forwerrorx.append(hxforw[0][2*n]-hxforw[1][n])
    forwerrory.append(hyforw[0][2*n]-hyforw[1][n])

forwerrortot=[np.sqrt(x**2+y**2)/(2**5-1) for x,y in zip(forwerrorx,forwerrory)] #
print((forwerrortot[-1])) #Total truncation error
plt.plot(forwerrortot)
plt.show()
```