Arun Suresh

Computational Physics 1 - Homework 6

---

(3) We have the Ginzburg-Landau equation as follows

$$\frac{d}{dt}y(t) = y(t)[a - by(t)^2]$$

(a) For an explicit method, set time step Δt we have $y(t_{k+1}) = y(t_k) + \Delta t \frac{d}{dt}y(t)\Big|_{t=t_k}$.

So in essence, we have

$$y(t_{k+1}) = y(t_k) + \Delta t[y(t_k)(a - by(t_k)^2)]$$

(b) For an implicit method, set time step Δt we have $y(t_{k+1}) = y(t_k) + \Delta t \frac{d}{dt}y(t)\Big|_{t=t_{k+1}}$.

So in essence, we have

$$y(t_{k+1}) = y(t_k) + \Delta t[y(t_{k+1})(a - by(t_{k+1})^2)]$$

This yields a cubic polynomial in $y(t_{k+1})$, which can be solved using root finding algorithms like the Newton-Rhapson method, which would have the following set up.

For ease of typing let, $y(t_{k+1}) = x$, $y(t_k) = c$.
So, we have $f(x) = x - c - \Delta t[x(a - bx^2)] \implies f'(x) = 1 - \Delta t[a - 3bx^2]$. Now, we can set up Newton Rhapson to iteratively solve

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Upon reaching a satisfactory solution $\mathbf{x} = \lim_{n\to\infty} x_n$, we let $y_{k+1} = \mathbf{x}$. With the obtained y_{k+1}, we progress further in time to obtain y_{k+2} in a similar fashion.

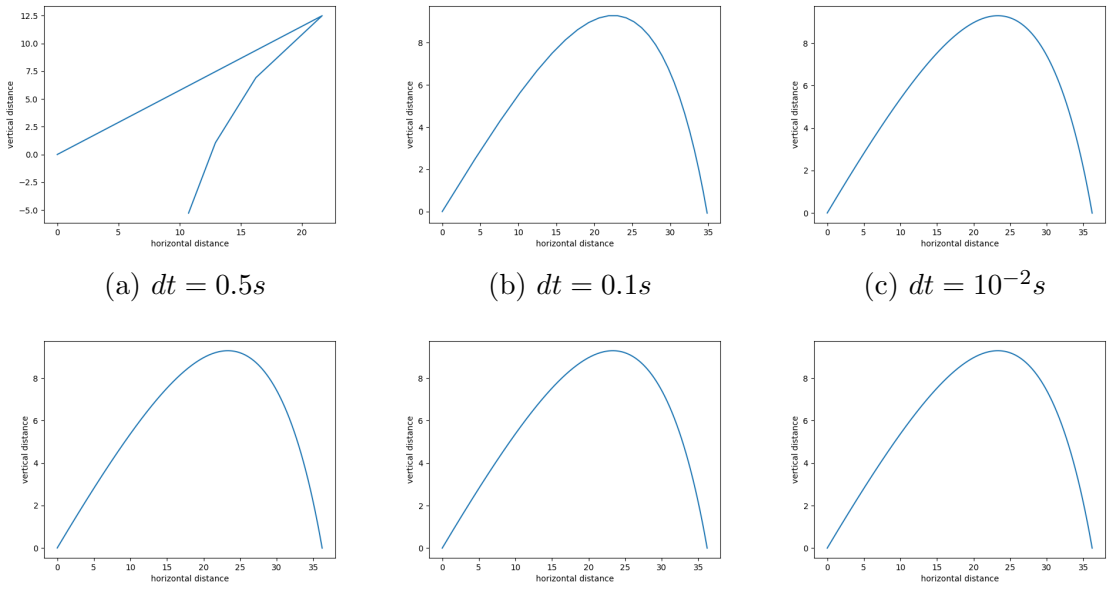
The difference between an explicit method and an implicit method lies in the iterative part, particularly in how we compute the value of the function in the next time step. In the explicit methods, we compute the function value at an unknown time, using the function values from the past (that is already known). In an implicit method however, we write the iterative step in terms of function values at the unknown time step, obtain an algebraic expression from which we solve for the function value at the unknown time-step.

(4) We aim to solve the following pair of coupled ordinary differential equations

$$\frac{d^2}{dt^2}x = -\frac{C_D}{m}\sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}\frac{dx}{dt}$$

$$\frac{d^2}{dt^2}y = -\frac{C_D}{m}\sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}\frac{dy}{dt} - g$$

using finite difference methods.
In order to impliment this, I chose the simplest finite difference method - forward Euler. The code for the forward Euler method is presented in the Appendix 1 as $4(a)$.
The method was computed using six different time steps, and the corresponding plots from each time steps are presented below.



(a) $dt = 0.5s$ (b) $dt = 0.1s$ (c) $dt = 10^{-2}s$

(d) $dt = 10^{-3}s$ (e) $dt = 10^{-4}s$ (f) $dt = 10^{-6}s$

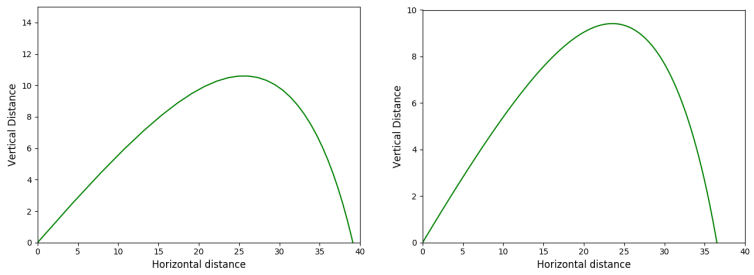
Figure 1: Forward Euler - Finite difference

It is clear that these plots fit the observation. The following results were obtained from each run as the "Time of flight" and the "Range" of the projectile.

dt	Range	Time of Flight
0.5	10.7329	2.5
0.1	34.8447	2.7
10^{-2}	36.1445	2.68
10^{-3}	36.2110	2.669
10^{-4}	36.2181	2.668
10^{-6}	36.2191	2.667939

It is easy to see that the method performs better with smaller time steps. Since it is an explicit method, it seems to be very unstable when dt is large (notice that the time of flight is $\sim$ 2.5s, and so, $0.5s$ is a pretty large time step). The most important take away is that - smaller time steps produce better results. However, depending upon the sensitivity of the problem, often times, we need not consider very small time steps. In this example, going from $dt = 10^{-4}$ to $dt = 10^{-6}$ is a hundred fold magnification, and yet still, the improvement of the output is only in the range of 10^{-4} - and more importantly, the average (over 10 runs) computation time when $dt = 10^{-4}$ was $0.002s$, however, the average (over 10 runs) computation time when $dt = 10^{-6}$ was $2.11s$; meaning computational cost of hundred fold magnification is about a thousand times large[1]. This difference will only get bigger as we consider more complex problems.

As sort of an extra personal project I also tried to impliment a slightly more stable explicit method for the same problem. For this I chose a 4^{th} order Runge Kutta method (because that was the one of the few stable methods in the notes that that related to coupled differential equations). With the help of a few online resources, I was able to impliment the method. I have attached my code for that in Appendix 1 as $4(b)$. The plots and results that I obtained from this method follow the same results produced by Forward Euler method. In what follows, I am assuming that the result obtained using Forward Euler is correct.



(a) $dt = 0.1s$ (b) $dt = 0.01s$

Figure 2: Fourth Order Runge Kutta

The following are the results obtained from the 4^{th} order RK method.

dt	Range	Time of Flight
0.1	38.8965	2.6
10^{-2}	36.4735	2.68

Surprisingly, the 4th order RK performs poorly by overestimating the result in both cases (in comparison to Forward Euler, that had a much faster convergence)[2]. I believe this is the tradeoff between stability and accuracy.

[1]Python's *timeit*() module seems to be performing poorly from run to run, sometimes even returning negative values. So, I averaged the first ten positive outputs to produce a rough estimate. Once I figure out the technical details behind using the module, I would like to analyse the time and computational costs involved with smaller timesteps. Assuming that a computer can perform 10^6 computations per second, I think that I would observe for $dt = 10^{-n}$, we would have computational time around 10^{n-6}

[2]If we can assume the true result to be 36.2191, then when $dt = 0.1$, we have $|36.2191-34.8447| < |36.2191-38.8965|$ and when $dt = 0.01$, we have $|36.2191-36.1445| < |36.2191-36.4735|$

APPENDIX 1

4(a)

```
import numpy as np
import matplotlib.pyplot as plt

''' For forward euler time integration, we have
x[k+1] = x[k] + dt*x'[k]
y[k+1] = y[k] + dt*y'[k]
x'[k+1] = x'[k] + dt*x''[k]
y'[k+1] = y'[k] + dt*y''[k]
The range of the projectile would be the x[k] for k such that y[k] = 0
The time of flight would be the total length(x[])*dt '''

# ---------- Initialization ------------
cD = 0.025
m = 0.5  # mass
k = cD/m
g = 9.80665
x = [0]  # x-values
y = [0]  # y-values
xpr = [25*np.sqrt(3)]  # x' - values
ypr = [25]  # y'-values
dt = 0.001  # timestep

# ------- Double prime functions -------


def xdpr(xpr, ypr):
    return -1*k*np.sqrt(xpr**2 + ypr**2)*xpr


def ydpr(xpr, ypr):
    return -1*k*np.sqrt(xpr**2 + ypr**2)*ypr - g


while True:
    x.append(x[-1] + dt*xpr[-1])  # updating x positiion
    y.append(y[-1] + dt*ypr[-1])  # updating y position
    xpr.append(xpr[-1] + dt*xdpr(xpr[-1], ypr[-1]))  # updating the dx/dt
    ypr.append(ypr[-1] + dt*ydpr(xpr[-2], ypr[-1]))  # updating dy/dt
    if(y[-1] < 0):  # exit condition
        break


print("Range", x[-1])
print("Time of flight", len(x)*dt)
fig, ax = plt.subplots()
plt.plot(x, y)
ax.set_xlabel("horizontal distance")
ax.set_ylabel("vertical distance")
plt.show()
```

4(b)

```python
import matplotlib.pyplot as plt
import numpy as np


def feval(funcName, *args):  # function Handle
    return eval(funcName)(*args)


def RK4thOrder(func, yinit, x_range, h):  # fourth order method
    m = len(yinit)
    n = int((x_range[-1] - x_range[0])/h)

    x = x_range[0]
    y = yinit

    # Containers for solutions
    xsol = np.empty(0)
    xsol = np.append(xsol, x)  # time axis

    ysol = np.empty(0)
    ysol = np.append(ysol, y)  # array to store both x'(t) and y'(t)

    for i in range(n):  # Runge Kutta implimentation
        k1 = h*feval(func, x, y)

        yp2 = y + k1/2

        k2 = h*feval(func, x+h/2, yp2)

        yp3 = y + k2/2

        k3 = h*feval(func, x+h/2, yp3)

        yp4 = y + k3

        k4 = h*feval(func, x+h, yp4)

        for j in range(m):  # y = [x'(t_k), y'(t_k)]
            y[j] = y[j] + (1/6)*(k1[j] + 2*k2[j] + 2*k3[j] + k4[j])

        x = x + h  # updating time
        xsol = np.append(xsol, x)

        for r in range(len(y)):
            ysol = np.append(ysol, y[r])  # x'(t_k) and y'(t_k) appended

    return [xsol, ysol]


def myFunc(x, y):
    cD = 0.025
    m = 0.5  # mass
```

```
    k = cD/m
    g = 9.80665
    dy = np.zeros((len(y)))
    dy[0] = -k*np.sqrt((y[0])**2+(y[1])**2)*y[0]  # x'(t)
    dy[1] = -k*np.sqrt((y[0])**2+(y[1])**2)*y[1] - g  # y'(t)
    return dy

# -----------------------


h = 0.001  # stepsize
x = np.array([0.0, 40.0])  # range of x
yinit = np.array([25*np.sqrt(3), 25])  # y' IC
xpos = [0]
ypos = [0]

[ts, ys] = RK4thOrder('myFunc', yinit, x, h)
node = len(yinit)
ys1 = ys[0::node]  # splitting only x'(t_k)
ys2 = ys[1::node]  # splitting only y'(t_k)

for t in range(len(ts)):
    xpos.append(xpos[-1] + h*ys1[t])  # obtaining x(t_k) from x'(t_{k-1})
    ypos.append(ypos[-1] + h*ys2[t])  # obtaining y(t_k) from y'(t_{k-1})

for i in range(len(ypos)):
    if(ypos[i] < 0):
        print("Range", xpos[i-1])  # finding range
        print("Time of Flight", h*i)  # finding time of flight
        break

# ------plotting-----------
plt.plot(xpos, ypos, 'g')
plt.xlim(x[0], x[1])
plt.ylim(0, 15)
plt.xlabel('Horizontal distance', fontsize=12)
plt.ylabel('Vertical Distance', fontsize=12)
plt.tight_layout()
plt.show()
```