AST1100 1B.8

Andreas Helland

14. november 2016

1 Introduksjon

I denne oppgaven skal vi Lande en satellitt på en av planetene i solsystemet vi har fått oppgitt. Som i forrige oppgave (1B.7) bruker vi Euler-Cromer for å regne ut satellittens nye posisjon i hvert tidsintervall ettersom den beveger seg inn mot planeten, treffer atmosfæren og til slutt lander på overflaten.

2 Metode & Fremgangsmåte

Som sist begynner vi med å sette inn de oppgitte start-verdiene. I denne oppgaven brukes SI-enheter for å gjøre utregningene.

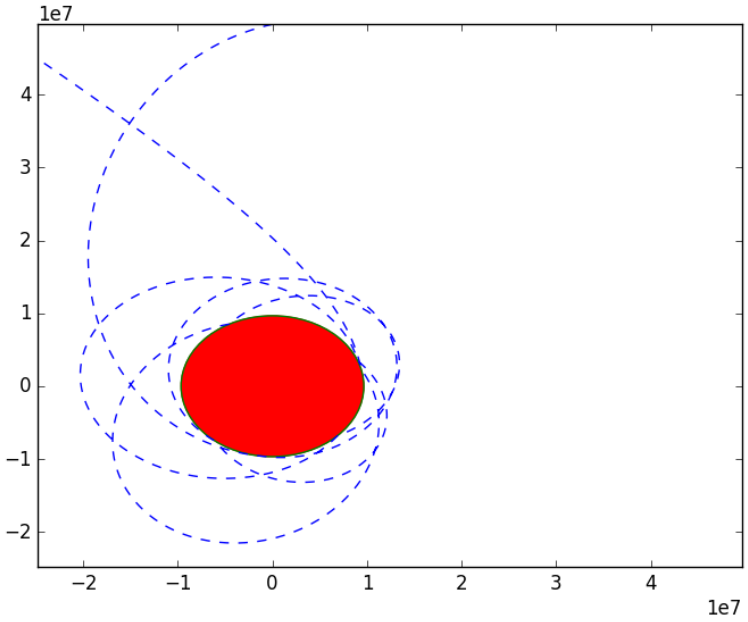
Det som skal regnes ut her er som nevnt i oppgaven, nokså likt det vi gjorde i forrige oppgave. Vi finner posisjonen til satellitten over tid. Vi opererer i et to-dimensjonalt plan, og deler utregningene av nye posisjoner inn i x og y retning. Vi finner også et tidsintervall mellom hver utregning. Fordi utregningene ikke er over før satellitten har landet på planeten, fortsetter vi å regne ut posisjonen frem til den er ved overflaten $r_{satellitt} <= r_{planetRadius}$.

Den største forskjellen i utregninger fra forrige oppgave var atmosfærens luftmotstandskraft. Når satellitten er innenfor atmosfæren $r_{sat} <= r_{radius} + self.AtmosphereHeight$ må vi ta hensyn til både gravitasjon fra planeten og luftmotstanden, som virker i motsatt retning av hastigheten til satellitten.

$$\begin{aligned} \sum F &= F_D + F_g \\ \sum F &= (\tfrac{1}{2}\rho A v^2 - \tfrac{GmM}{r^2})\vec{e}_r \end{aligned} \tag{1}$$

Denne hastigheten er nokså høy, som dermed fører til en enorm luftmotstand i det den treffer atmosfæren. På grunn av dette må tidsintervallet *dt* være veldig lavt. Skal bevegelsen være realistisk, vil hastigheten reduseres hele tiden av luftmotstanden, og dermed redusere videre luftmotstand. Hvis tidsintervallet blir for stort vil satellitten får en kraftig akselerasjon i motsatt retning over lang tid (dårlig dt), og luftmotstanden får en uproporsjonalt høy effekt. Dette kan føre til at satellitten rett og slett bare spretter vekk fra planeten i det den treffer atmosfæren, som vist i figure 1.

Den hovedsakelige utfordringen i oppgaven er å lande satellitten med lav nok hastighet til at den overlever fallet ($3\frac{m}{s}$). For å oppnå dette trenger vi en høy luftmotstand som reduserer gravitasjonskraftens effekt. Dette kan vi oppnå



Figur 1: Den røde sirkelen er planeten, den tynne grønne streken er atmosfæren. Den blåe linjen viser satellitbanen inn mot planeten med lange tidsintervall $dt = 0.15s$. $v_{0,y} = -2837\frac{m}{s}$, $v_{0,x} = -810\frac{m}{s}$ $A = 2.5m^2$.

ved å øke størrelsen på satellittens fallskjerm A, men som nevnt tidligere, krever høy luftmotstand at tidsintervallene blir lavere for at bevegelsen skal bli realistisk. Dette kan føre til at utregningene blir veldig mange og kjøretiden til programmet blir høy.

Som i forrige oppgave er utregningene strukterert i et objekt. Man finner dette objektet i figur 24. Når objektet kalles med gitt tidsintervall som parameter, gjøres posisjonsutregningene internt og lagres i objektet. Vi henter dermed de utregnede verdiene fra objektes liste og formaterer dem sammen med tidsintervaller slik at det kan leses av den ferdiglagede metoden som printer verdiene til XML (som igjen kan leses av grafikkprogrammet vårt). [1]

I satellite objektet regner vi først ut hastigheten i det nye tidsintervallet ut ifra akselerasjonen satellitten får fra gravitasjonskraften fra planeten (grav()). Når satellitten er innenfor atmosfæren, inkluderer vi luftmotstandskraften i utregning av ny akselerasjon for tidsintervallet (a()). Som i forrige oppgave

[1] plot utskrift og diverse testing gjøres i separat modul test.py

(1B.7) bruker vi Euler-Cromers metode. Det gir oss de oppgitte (dekomponerte) ligningene for akselerasjon i oppgaven.

Akkurat som sist, kan vi dermed finne den nye posisjonen (newPosition) etter vi har funnet v_{n+1} i newVelocity. Etter disse verdiene er funnet, lagres x_{n+1}, y_{n+1}, $v_{x,n+1}$ og $v_{y,n+1}$ for hvert eneste tidssteg i orbitHistory. Dette er arrayen vi henter verdiene fra som nevnt over. For å sjekke hastigheten ved landing av satellitten, henter vi rett og slett bare verdiene i self.vX og self.vY.[2]

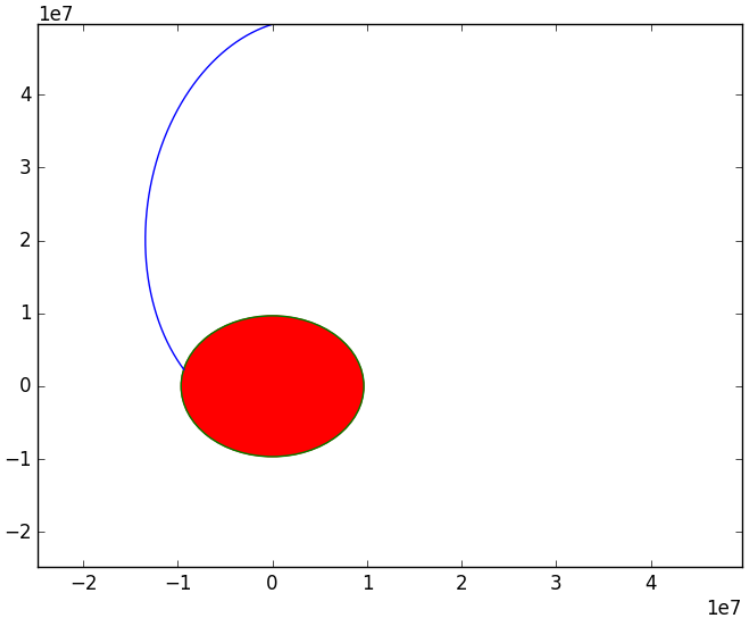
I håp om å løse det største problemet med denne oppgaven (feil verdi for luftmotstandskraft på grunn av høye tidsintervaller dt), har atmosfæren blitt skalert slik at tettheten er lavere høyt oppe, og ikke uniformt fordelt slik oppgaven mener vi skal regne med. Dette hjelper satellitten å bremse opp sakte men sikkert istedenfor å bli truffet med en enorm luftmotstand akkurat i det den er innenfor atmosfæren. En mer korrekt måte å løse dette problemet på ville vært å redusere tidsintervallet, men på grunn av kodens dårlige kjøretid, er det dessverre ikke en mulighet uten betydelig omskriving.

3 Resultater

Etter å ha prøvd mange forskjellige initialhastigheter, endte verdiene på $v_{0,x} = -800\frac{m}{s}$ og $v_{0,y} = -2000\frac{m}{s}$. Dette fører til at satellitten treffer planeten med en høy vinkel, men det var dessverre nødvendig for å kunne redusere tidsintervallet dt lavt nok ($0.03s$) til at luftmotstanden ble regnet ut nogenlunde realistisk (uten at kjøretiden ble for høy). Dette førte til en landingshastighet på $23\frac{m}{s}$, som er betydelig høyere enn det vi ville komme frem til.

Dersom en bedre innfallsvinkel blir valgt, ville luftmotstanden føre til at satellitten 'spratt' ut igjen som vist i figur 1. Med en veldig liten fallskjerm ville ikke kraften sprette satellitten helt vekk, men da får vi igjen en landingshastighet som er for høy. Et mer oversiktelig eksempel ser man i figur 3.

[2]Orginalt var planen å lagre hele hastighetshistorikken til satellitten, men kjøretiden til programmet ble et stort nok problem til at unødvendige operasjoner måtte fjernes.



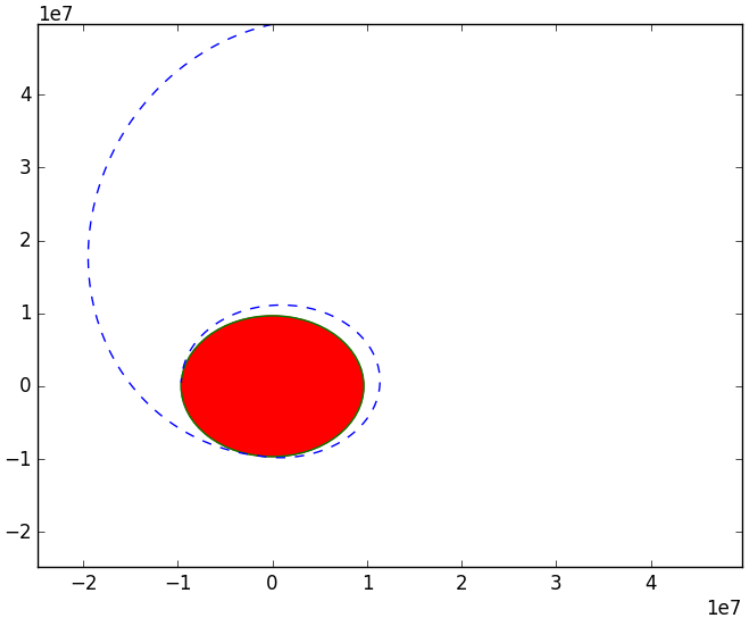
Figur 2: Den røde sirkelen er planeten, den tynne grønne streken er atmosfæren. Den blåe linjen viser satellitbanen inn mot planeten med dårlig infallsvinkel, men lavere dt og noe større fallskjerm. $dt = 0.03s$. $v_{0,y} = -2000\frac{m}{s}$, $v_{0,x} = -800\frac{m}{s}$ $A = 20m^2$. Her fikk vi en landingshastighet på $23\frac{m}{s}$

4 Diskusjon og konklusjon

Dette programmet fikk dessverre ikke landet satellitten trygt på bakken i en hastighet under $3\frac{m}{s}$. Programmet fikk derimot utregnet en landing med lavere og lavere hastighet ettersom tidsintervallet dt ble redusert og fallskjermen A ble større. Dersom kjøretiden ikke hadde vært et problem virker det som om man kunne funnet initialverdier som fører til en myk landing.

Et annet problem med koden er at atmosfærens tetthet er betydelig redusert i toppen for å gjøre det enklere for satellitten å bremse opp sakte før den fulle luftmotstanden tar tak. Igjen ville dette problemet muligens kunne fikses dersom dt var stor nok og kjøretiden ikke var et problem. En uniformt fordelt atmosfære er ikke den mest realistiske situasjonen å gjøre utregninger i, men fordelingen brukt i denne koden er også litt overdreven.[3]

[3]Nå rett før innleveringen leste jeg også at Tanken er at du skal ta med med luft-



Figur 3: Den røde sirkelen er planeten, den tynne grønne streken er atmosfæren. Den blåe linjen viser satellitbanen inn mot planeten med god infallsvinkel, men høy dt og liten fallskjerm. $dt = 0.25s$. $v_{0,y} = -2837\frac{m}{s}$, $v_{0,x} = -810\frac{m}{s}$ $A = 2m^2$. Her fikk vi en landingshastighet på $80\frac{m}{s}$

En konklusjon som kan trekkes her er at koden skulle egentlig blitt helt omskrevet med effektiv kjøretid i fokus istedenfor strukturert kode.

A Vedlegg

motstanden i beregningene helt fra starten av 'the instructor' på piazza. Dette er også en potensiell løsning på satellittspretteproblemet mitt, fordi dersom luftmotstanden virker på satellitten fra starten av, vil ikke satellitten bygge opp en såpass stor hastighet før den treffer luften. Det ser derimot ut til å stride med oppgaveteksten som sier at satellitten skal starte 40000 km over planetens overflate, som er betydelig lengre oppe enn C/g.

```
class satellite():

    def __init__(self, m, M, x, y, v0x, v0y, r, density, A): #, atmosphereHeightRatio
        # Initial values
        self.x = x  #x-position of satellite
        self.y = y  #y-position of satellite

        self.vX = v0x #x-velocity of satellite
        self.vY = v0y #y-velocity of satellite

        self.A = A          #satellite parachute area
        self.p = density    #atmospheric density
        self.r = r                  #radius of planet
        self.planetMass = M         #mass of planet it orbits
        self.m = m                  #mass of satellite
        self.G = 6.67408*10**(-11)  #gravitational constant
        self.AtmosphereHight = ((322000)/(self.G*self.planetMass/self.r**2))
        # initiate array for position and velocity history, for each N steps
        self.orbitHistory = np.array([[self.x, self.y]])

    # Execute movement of planet over time T in N intervals
    def __call__(self, dt):
        y0 = self.y
        topOfTheAtmosphere = (self.r+self.AtmosphereHight)

        while (sqrt(self.x**2+self.y**2) > self.r) and (sqrt(self.x**2+self.y**2)<= y0*1.5):
            if sqrt(self.x**2+self.y**2) > topOfTheAtmosphere: #if above atmosphere
                self.vX, self.vY = self.newVelocity(self.grav(), dt)  #set next velocity n+1 (with only g
                self.x, self.y = self.newPosition(self.grav(), dt)    #set next position n+1 (with only g
            else:   # if the satellite has reached the atmosphere
                self.vX, self.vY = self.newVelocity(self.a(), dt)  #set next velocity n+1 (with gravity +
                self.x, self.y = self.newPosition(self.a(), dt)    #set next position n+1 (with gravity +
            self.orbitHistory = np.vstack([self.orbitHistory,[self.x, self.y]])    #store current positio

    # returns position in the next step
    def newPosition(self, a, dt):
        x = (self.x + self.vX*dt )#+ 0.5*a[0]*dt**2) # next x_(n+1) = x_n + Vx_(n+1)dt
        y = (self.y + self.vY*dt )#+ 0.5*a[1]*dt**2) # next y_(n+1) = y_n + Vy_(n+1)dt
        return x, y

    #returns velocity in the next step
    def newVelocity(self, a, dt):
        return self.vX + a[0]*dt, self.vY + a[1]*dt #v_(n+1) = v_n +a*dt

    #returns acceleration from gravitational force
    def grav(self):
        return [-self.x*self.G*self.planetMass/(sqrt(self.x**2+self.y**2)**3),  # x direction
                -self.y*self.G*self.planetMass/(sqrt(self.x**2+self.y**2)**3)]  # y direction
    # combining gravity + drag
    def a(self):
        rr = sqrt(self.x**2+self.y**2)
        p = self.p*exp(-(self.r+(rr-self.r)*1000)/self.r) #make atmosphere thinner at the top to improve
        return [-(self.x*self.G*self.planetMass/rr**3)+((0.5*p*self.A*self.vX**2)/self.m)*(self.x/rr),
                -(self.y*self.G*self.planetMass/rr**3)+((0.5*p*self.A*self.vY**2)/self.m)*(self.y/rr)]
```

Figur 4: satellite objekt kode

```
from AST1100SolarSystemViewer import AST1100SolarSystemViewer
import numpy as np
from Satellite import satellite

seed = 88850
system = AST1100SolarSystemViewer(seed)

planetsRadius = system.radius     # Radiuses of planets, [km].
planetsMass = system.mass         # Mass of the planets, [solar masses].
rho0 = system.rho0                # Atmospheric density at surface

# initialize values to be used
m = 100                             #mass of satellite
M = 1.989 * 10**30 * planetsMass[0] #mass of planet converted to [kg]
r = planetsRadius[0]*1000    #radius of planet [m]
x0 = 0                      #initial x position of satellite
y0 = 40000000+r             #initial y position of satellite
vx0 = -2000                 #initial x velocity of satellite
vy0 = -800                  #initial y velocity of satellite
p = rho0[0]                 #atmospheric density
A = 20                      #size of parachute
dt = 0.03                   #time interval in seconds

#run satellite module
lander = satellite(m, M, x0, y0, vx0, vy0, r, p, A)
lander(dt)

#get position values from lander object
satPos = np.array([lander.orbitHistory[:,0], lander.orbitHistory[:,1]])
satPos = satPos.T               #Transpose instead of initializing it properly (sorry)
times = np.zeros(len(satPos))   #create array of time intervals

for i in xrange(len(satPos)):
    satPos[i][0] = satPos[i][0]*6.68459*10**-12 #convert from meters to AU
    satPos[i][1] = satPos[i][1]*6.68459*10**-12 #convert from meters to AU
    times[i] = (3.17098*10**-8*dt*i) #converted from seconds to years

system.landingSat(satPos, times, 0) #generate the xml file
```

Figur 5: Main koden som kjører programmet og oppretter xml filen.