AST1100 1B.7

Andreas Helland

14. november 2016

1 Introduksjon

I denne oppgaven skal vi løse to-legeme problemet og tre-legeme problemet numerisk ved hjelp av Euler-Cromer metoden. Mer spesifikt skal vi finne bevegelsen til en rekke planeter i bane rundt en stjerne over tid med oppgitte start-posisjoner, hastigheter og masser.

2 Metode & Fremgangsmåte

Det første som måtte gjøres for å løse det oppgitte to-legeme problemet var å få tak i start-verdiene og de ferdiglagde metodene som gjør at verdiene vi finner kan leses av Solar system view (SSView). Avstander er oversatt til Astronomiske enheter (AU) og tidsenheter som blir brukt her er år.

Det som skal regnes ut her er i bunn og grunn posisjonen til planetene over tid. Vi opererer i et to-dimensjonalt plan, og kan dermed dele utregningene av nye posisjoner inn i x og y retning, men først må vi også velge et tidsintervall mellom hver utregning og ikke minst, hvor mange utregninger som skal gjøres.

I dette programmet blir $T = 500000$ tidsintervaller kjørt gjennom og dekker 50 år. Det vil si tidsintervallet $dt = 0.0001y = 3153.6s$. Dette er en nokså lang tidsperiode. Den er såpass høy fordi de ytre planetene har nokså lang omløpstid (og vi vil helst la en av de ytre planetene ta en hel runde rundt stjernen). Den kjørte koden finner man i figur 3.[1]

Utregningene er strukterert i et Planet objekt. Man finner dette objektet i figur 4. Når objektet kalles med gitte tidsparameter, gjøres posisjonsutregningene internt og lagres lokalt. 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).[2]

I planet objektet regner vi først ut hastigheten i det nye tidsintervallet ut ifra akselerasjonen planeten får fra gravitasjonskraften fra stjernen (som oppgitt ignorerer vi her gravitasjon fra andre himmellegemer i systemet og planetenes

[1] Utskrift av plot og testing er utført i en egen modul (test.py)

[2] Jeg oppdaget nå rett før innleveringsfristen at jeg har gjort oppgaven som forklart i en eldre versjon av Lecture note 1b som ikke bruker dumpToXml, men OrbitXml. Jeg må bare beklage og håpe det går fint, fordi det er dessverre ikke nok tid igjen til at jeg kan se over forskjellene og gjøre det på nytt. XML filen fungerte i SSView i det minste.

påvirkning på stjernen). Fra seksjon 1.1 i pensum vet vi at ved Euler-Cromer metoden finner vi

$$f_{n+1} = f_n + f'(x_{n+1})\Delta x \tag{1}$$

Som gir oss de oppgitte (dekomponerte) ligningene for akselerasjon.

$$\begin{aligned} \frac{dv_x}{dt} &= \frac{F_x}{m} \\ \frac{dv_y}{dt} &= \frac{F_y}{m} \end{aligned} \tag{2}$$

Etter vi har funnet v_{n+1} i newVelocity, kan vi dermed finne den nye posisjonen (newPosition). 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 planetHistory. Dette er arrayen vi henter verdiene fra som nevnt over.[3]

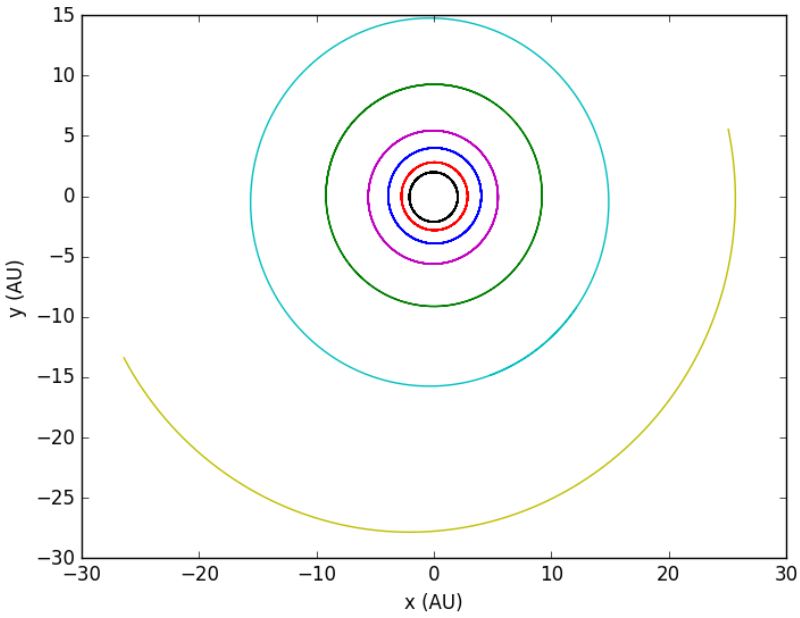
For å løse tre-legeme problemet bruker vi en lignende fremgangsmåte, men istedenfor å kun regne ut planetens nye posisjon ut ifra gravitasjonskraften, regner vi ut den nye posisjonen til hvert av himmellegmnene ut ifra akselerasjonen gravitasjonskraften fra begge de andre objektene gir. Utregningen er gjort i koden fra figur 5. Opprettelsen og utskriften er gjort separat, men på samme måte som det forrige problemet. Her brukes dualStarXml metoden til å skrive ut xml filen som kan kjøres i SSView.

3 Resultater

Posisjonsverdiene for planetene i 2-legeme problemet finner man i figur 4. Her ser vi at planetene går i stabile baner som er nokså tilnærmet sirkulære.

[3]Strengt talt er det helt unødvendig i denne oppgaven å lagre $v_{x,n+1}$ og $v_{y,n+1}$. Da jeg først begynte å strukturere programmet mitt på denne måten tenkte jeg at det skulle være et fint, generelt og strukturert program som kan brukes til mer enn hva som var nødvendig, men endte opp med å gjøre det på en unødvendig måte som sannsynligvis har mye dårligere kjøretid enn det ellers kunne hatt. Forhåpentligvis er koden litt mer oversiktelig/lettlest, men dessverre kan ikke det samme sies om de neste oppgavene hvor koden baserer seg på hva som ble gjort her (1B7.2 og 1B8). Skulle de også vært oversiktelige og hatt pen kode, kunne jeg ikke gjenbrukt så mye av denne første oppgaven som jeg gjorde. Ideelt sett skulle alt blitt omskrevet med en annen objektstruktur (SolarSystem med interaksjonsutregning, CelestialBody med verdilagring osv.), evt. droppet det helt og bare programmert det rett frem.

Tidsperioden på 50 år er lang nok til at alle planetene utenom den ytterste rekker å fullføre et helt omløp. Den kjørbare XML filen bekrefter dette resultatet.

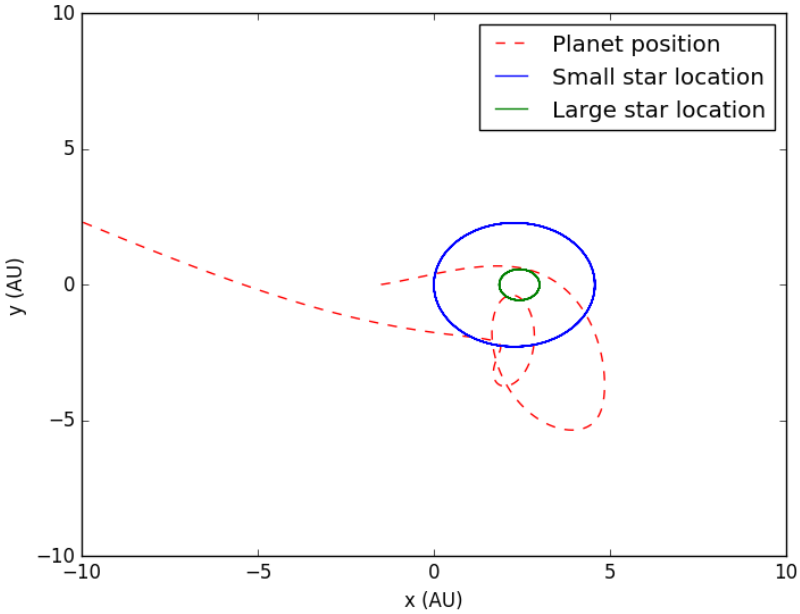


Figur 1: Plot som viser planetbanene i xy-planet. Enhetene i begge retninger er astronomiske enheter (AU).

I tre-legeme problemet går stjernene i bane om hverandre nokså stabilt, men planeten ender opp med å bli kastet ut av systemet etter den kom for nærme en av stjernene og ble akselerert forbi unnslipningshastigheten. Se figur 5. Det samme skjer dersom man kjører xml filen i SSView.

4 Diskusjon og konklusjon

Selv om tidsintervall på nesten en time, som kan virke nokså lenge, gir det ikke stor nok feilmargin til å endre resultatet bemerkelsesverdig i to-legeme problemoppgaven. Hvis man tenker over avstandene vi opererer med i et slikt system er det kanskje ikke så rart. På en times tid vil ikke planetene endre posisjonen sin så langt vekk fra den realistiske banen (hvor dt går mot 0) at feilen vil være synlig. Hvis vi kjørte systemet lenge nok ($T >> 50$ år), ville feilene sansynligvis komme frem bedre.



Figur 2: Plot som viser planet og stjernebanene i xy-planet for 3-legeme problemoppgaven. Enhetene i begge retninger er astronomiske enheter (AU).

3-legeme problemoppgaven derimot, akkumulere feil i utregningene betydelig raskere fordi for hvert tidsintervall gjøres flere posisjonsendringer som alle avhenger av hverandre. Selv om tidsintervallene er betydelig lavere her, vil et realistisk bilde av bevegelsen bli mye vanskeligere å opprette.

Skulle planeten inneholdt kjeminen som kreves for liv, ville det nok alikevel hatt store problemer med å oppstå i et system med to stjerner, med tanke på den ustabile banen [4]. Selv hvis den ikke blir kastet ut av systemet, vil distansen fra stjernene variere betydelig, som vil skape radikale klimaendringer over korte tidsperioder. Liv som har adaptert seg til en viss tilværelse vil ha store problemer med å overleve slike store endringer.

A Vedlegg

[4]med midre den var langt nok vekke til at begge stjernene kan behandles som en gravitasjonskilde, men alikevel nærme nok til at energien utstrålt kan opprettholde livet.

```
from AST1100SolarSystemViewer import AST1100SolarSystemViewer
import numpy as np
from Planet import planet

seed = 88850
system = AST1100SolarSystemViewer(seed)

planetsRadius = system.radius     # Radiuses of planets, [km].
planetsMass = system.mass         # Mass of the planets, [solar masses].
planets_initial_x0 = system.x0    # Initial x-position of planets, [AU].
planets_initial_y0 = system.y0    # Initial y-position of planets, [AU].
planets_initial_vx0 = system.vx0  # Initial x-velocity of planets, [AU].
planets_initial_vy0 = system.vy0  # Initial y-velocity of planets, [AU].

# Data about the system.
numberOfPlanets = system.numberOfPlanets # Number of planets in the system.
starRadius = system.starRadius      # Radius of star, [km]
starMass = system.starMass          # Mass of the star, [solar masses].

planets = np.empty((numberOfPlanets,1), dtype=object) #planet object array
T = 50.0        #number of years
N = 500000      #total number of time steps
dt = T/N        #time interval
times = np.zeros(N)
pos_computed = np.zeros((2, system.numberOfPlanets, N))

#initiate planets
planets = [planet(planetsMass[i], starMass, planets_initial_x0[i],
                  planets_initial_y0[i], planets_initial_vx0[i],
                  planets_initial_vy0[i]) for i in range(numberOfPlanets)]

# fill out time array
for t_i in xrange(N):
    times[t_i] = dt*t_i

#calculate planet orbits
for i in xrange(numberOfPlanets):
    planets[i](T,N)

#create the array required to create the xml file
for p_no in xrange(numberOfPlanets):
    for t_i in xrange(N):
        pos_computed[0, p_no, t_i] = planets[p_no].planetHistory[t_i][2]
        pos_computed[1, p_no, t_i] = planets[p_no].planetHistory[t_i][3]

system.orbitXml(pos_computed, times) #Will generate the xml file
```

Figur 3: 1.B7 - Main kode

```
class planet():

    def __init__(self, m, M, x, y, v0x, v0y):
        # Initial values
        self.x = x  #x-position of planet
        self.y = y  #y-position of planet

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

        self.starMass = M   #mass of star planet orbits
        self.m = m          #mass of planet
        self.G = 4*pi*pi    #6.67408*10**(-11) #gravitational constant

        # initiate array for position and velocity history, for each N steps
        self.planetHistory = np.zeros((1, 4))

    # execute movement of planet over time T in N intervals
    def __call__(self, T, N):
        dt = T/N                                #seconds per given time interva
        self.planetHistory = np.zeros((N, 4))   #define size of planet history

        for i in xrange(N):
            self.vX, self.vY = self.newVelocity(self.A(), dt)  #set next veloci
            self.x, self.y = self.newPosition(self.A(), dt)    #set next positi
            self.planetHistory[i][0] = self.vX     #store current x velocity n+
            self.planetHistory[i][1] = self.vY     #store current y velocity n+
            self.planetHistory[i][2] = self.x      #store current x position n+
            self.planetHistory[i][3] = self.y      #store current y position n+

    # 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_
        y = (self.y + self.vY*dt )#+ 0.5*a[1]*dt**2) # next y_(n+1) = y_n + Vy_
        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 A(self):
        return [-self.x*self.G*self.starMass/(sqrt(self.x**2+self.y**2)**3),
                -self.y*self.G*self.starMass/(sqrt(self.x**2+self.y**2)**3)]
```

Figur 4: Planet objekt kode

```
def __init__(self, m, x, y, v0x, v0y):
    # Initial values
    self.x, self.SSx, self.BSx = x, 0, 3 #x-position of objects (x = -1.5 )
    self.y, self.SSy, self.BSy = y, 0, 0 #y-position of objects  (y = 0
    self.vX, self.SSvX, self.BSvX = v0x, 0,0                  # velocity of objects
    self.vY, self.SSvY, self.BSvY = v0y, 6.32415, -1.5810375  # converted from km/s to au/year (yy =
    self.BSm = 4.0      # mass of the largest star
    self.SSm = 1.0      # mass of the small star
    self.m = m          #mass of the planet
    self.G = 4.0*pi*pi  #6.67408*10**(-11) #gravitational constant
    # initiate arrays for position and velocity history of the three objects
    self.planetHistory = np.zeros((1, 4))
    self.smallStarHistory = np.zeros((1, 4))
    self.bigStarHistory = np.zeros((1, 4))

# execute movement of planet over time T in N intervals
def __call__(self, T, N):
    dt = T/N    # seconds per given time interval
    self.planetHistory = np.zeros((N, 4))   #array of stored position & velocity values for planet
    self.smallStarHistory = np.zeros((N, 4))#array of stored position & velocity values for the smal
    self.bigStarHistory = np.zeros((N, 4))  #array of stored position & velocity values for the big

    for i in xrange(N):
        #set next iteration of planet position and velocity
        a = self.A(self.x, self.SSx, self.BSx, self.y, self.SSy, self.BSy, self.SSm, self.BSm) #acce
        self.vX, self.vY = self.newVelocity(a, dt, self.vX, self.vY)               #set next veloci
        self.x, self.y = self.newPosition(a, dt, self.x, self.y, self.vX, self.vY)  #set next positi
        self.planetHistory[i][0] = self.vX     #store current x velocity n+1 of the planet
        self.planetHistory[i][1] = self.vY     #store current y velocity n+1 of the planet
        self.planetHistory[i][2] = self.x      #store current x position n+1 of the planet
        self.planetHistory[i][3] = self.y      #store current y position n+1 of the planet
        #set next iteration of small star position and velocity
        a = self.A(self.SSx, self.x, self.BSx, self.SSy, self.y, self.BSy, self.m, self.BSm) #accele
        self.SSvX, self.SSvY = self.newVelocity(a, dt, self.SSvX, self.SSvY)                  #set
        self.SSx, self.SSy = self.newPosition(a, dt, self.SSx, self.SSy, self.SSvX, self.SSvY)  #set
        self.smallStarHistory[i][0] = self.SSvX     #store current x velocity n+1 of the small star
        self.smallStarHistory[i][1] = self.SSvY     #store current y velocity n+1 of the small star
        self.smallStarHistory[i][2] = self.SSx      #store current x position n+1 of the small star
        self.smallStarHistory[i][3] = self.SSy      #store current y position n+1 of the small star
        #set next iteration of big star position and velocity
        a = self.A(self.BSx, self.SSx, self.x, self.BSy, self.SSy, self.y, self.SSm, self.m) #accele
        self.BSvX, self.BSvY = self.newVelocity(a, dt, self.BSvX, self.BSvY)                  #set
        self.BSx, self.BSy = self.newPosition(a, dt, self.BSx, self.BSy, self.BSvX, self.BSvY)  #set
        self.bigStarHistory[i][0] = self.BSvX     #store current x velocity n+1 of the big star
        self.bigStarHistory[i][1] = self.BSvY     #store current y velocity n+1 of the big star
        self.bigStarHistory[i][2] = self.BSx      #store current x position n+1 of the big star
        self.bigStarHistory[i][3] = self.BSy      #store current y position n+1 of the big star

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

#returns acceleration from gravitational force
def A(self, x, x2, x3, y, y2, y3, m2, m3): #where x & y are the current object, while 2 & 3 are the
    r2 = sqrt((x-x2)**2+(y-y2)**2)# r vector between self & object 2
    r3 = sqrt((x-x3)**2+(y-y3)**2)# r vector between self & object 3
    #returns acceleration due to gravitational force from both m2 & m3
    return [(-(x-x2)*self.G*m2/r2**3)+(-(x-x3)*self.G*m3/r3**3),    #x direction
            (-(y-y2)*self.G*m2/r2**3)+(-(y-y3)*self.G*m3/r3**3)]    #y direction
```

Figur 5: koden til 3-legeme problem objektet