Modeliranje fizikalnih sustava: problem triju tijela

Kabaši, Anton

PRIRODOSLOVNO-MATEMATIČKI FAKULTET
SVEUČILIŠTA U SPLITU

Anton Kabaši

MODELIRANJE FIZIKALNIH SUSTAVA: PROBLEM TRIJU TIJELA

ZAVRŠNI RAD

Split, rujan 2015.

PRIRODOSLOVNO-MATEMATIČKI FAKULTET
SVEUČILIŠTA U SPLITU
ODJEL ZA FIZIKU

MODELIRANJE FIZIKALNIH SUSTAVA: PROBLEM TRIJU TIJELA

ZAVRŠNI RAD

Student:
Anton Kabaši

Mentor:
doc. dr. sc. Larisa Zoranić

Split, rujan 2015.

Sadržaj

Poglavlje 1

Uvod

1.1 Računarska fizika

Računarska fizika je polje fizike koje se bavi teoretskom analizom, numeričkim algoritmima i programiranjem u svrhu modeliranja složenih dinamičkih sustava. To je, uz eksperimente i teoriju, još jedan način istraživanja u fizici. Postoje mnoge jednadžbe koje opisuju ponašanje prirode ali nastaju problemi kada želimo naći točno rješenje osnovnih teorija. U većini slučajeva to je moguće samo za pojednostavljene modele.

Razvoj računala u 21. stoljeću nam je omogućio kvantitativno rješavanje problema u fizikalnim sustavima. Uz pomoć brzih računala možemo riješiti ne samo pojednostavljene modele već i realistične situacije. Računalne simulacije se koriste u mnogim znanstvenim disciplinama i često zamjenjuju eksperiment kao način potvrđivanja hipoteza.

Prednosti simulacija u znanstvenim istraživanjima su:

- rješavanje analitički nerješivih problema

- provjera ispravnosti aproksimacija
- usporedba teoretskih i eksperimentalnih rezultata
- vizualizacija složenih podataka

1.2 Programski jezici

Iako ne postoji savršeni programski jezik za numeričke simulacije i većina jezika je sposobna rješavati i najkompleksnije probleme, postoji nekoliko jezika koji se u tome izdvajaju. Objektno-orijentirani jezici kao što su Python i C++ su jednostavni za naučiti jer imaju visoku razinu apstrakcije.

1.2.1 Objektno-orijentirano programiranje

Proceduralno programiranje se koristi procedurama koje se izvršavaju u određenom poretku i na taj način dolazimo do željenog cilja. Objektno orijentirano programiranje, za razliku od proceduralnog, zasniva se na ideji da želimo programirati neku stvar(objekt) koja ima neka svojstva i način ponašanja.

Osnovni elementi objektno-orijentiranog programiranja su klasa, modularnost, nasljeđivanje, apstrakcija i polimorfizam.

Klasa je shema po kojoj stvaramo neki objekt. Unutar klase se definiraju polja i metode. Polja sadrže podatke o objektu(na primjer, liste i nizove podataka). Metode su funkcije koje mogu djelovati na klase i objekte i definiraju ponašanje objekata.

Instancu klase nazivamo objekt. Objekt sadrži sve elemente(polja i metode)

klase. Razlika između objekta i klase je da objekt postoji(instancira se), dok je klasa samo predložak po kojem se objekt stvara.

Modularizacija je razdjeljivanje programa na dijelove od kojih svaki ima svoj zadatak. Korisno je u većim projektima radi dijeljenja problema na manje, smislene cjeline.

Jedna klasa može proširiti definiciju klase koju nasljeđuje, to jest, sve elemente nadređene klase i uz to, implementirati nove, sebi svojstvene elemente. Omogućuje ponovnu iskoristivost, preglednost i dobru organizaciju koda.

Kad imamo više klasa koje pripadaju nekom općenitijem pojmu(psi i mačke su životinje) uvodimo apstraktnu klasu koja odgovara tom općenitijem pojmu. Apstraktna klasa sadrži svojstva i metode zajedničke svim klasama koje ju nasljeđuju.

Polimorfizam je sposobnost programskog jezika da obrađuje objekte različito, ovisno o njihovom tipu(klasi). Klasa može naslijediti drugu klasu i na novi način implementirati(premostiti) jednu od njenih metoda ako je ta metoda virtualna ili apstraktna.

Metode, procedure ili funkcije su dijelovi koda koji obavljaju određeni zadatak. Metoda se sastoji od imena, ulaznih parametara(argumenata), izlaznog tipa podataka, i bloka naredbi(tijela). Svaka klasa sadrži istoimenu metodu koju nazivamo konstruktor. On služi za instanciranje objekta koji je tipa te klase. Istoimene metode se mogu implementirati na više različitih načina, to jest, preopteretiti(overload) pri čemu im tijelo može, ali argumenti moraju

biti različiti.[2]

1.2.2 Open-source programske biblioteke

Prednost programskih jezika Python i C++ su velika zastupljenost u znanstvenoj zajednici. Postoji veliki broj besplatnih programskih biblioteka koje su dostupne svakome. Za Python neke od njih su:

- NumPy je paket koji definira višedimenzionalne nizove i brze matematičke funkcije koje na njih djeluju, također, sadrži jednostavne metode za linearnu algebru, Fourierove transformacije i sofisticirane generatore nasumičnih brojeva. Prednost ove biblioteke je to što je napisana u programskom jeziku C i zbog toga se naredbe koje ona sadrži izvršavaju puno brže od uobičajenih naredbi unutar Pythona. NumPy uz to sadrži i osnovne numeričke rutine, kao što su alati za nalaženje svojstvenih vektora.[7]

- SciPy je biblioteka koja sadrži mnoge znanstvene alate za Python. SciPy nadopunjava NumPy biblioteku i skuplja mnoge znanstvene i inženjerske module unutar jednog paketa. Sadrži module za linearnu algebru, optimizaciju, integraciju, specijalne funkcije, obradu slike i signala, statistiku, genetske algoritme i module za rješavanje običnih diferencijalnih jednadžbi.[4]

- matplotlib je biblioteka koja sadrži module za vizualizaciju podataka. Crta kvalitetne 2D i 3D grafove.[3]

1.3 Kompromis između performansi i potpunog programskog rješenja

Prednost programskog jezika Python nad C++ je mnogo jednostavnija sintaksa i veći broj dostupnih biblioteka. Iako je Python namijenjen učenju programiranja, ne ostaje na osnovama. Python je namijenjen pravljenju prototipova programa. Prva "skica" problema se riješi u Pythonu, jer je u njemu samo programiranje najbrže i najefikasnije. Program se zatim ponovno implementira u C-u ili C++ -u zbog toga što su ti jezici u prednosti po performansama. Zbog visoke razine apstrakcije, Python ima nedostatak, a to je njegova sporost. Numerički algoritmi mogu biti i do 100 puta sporiji u Pythonu.
Postoje mnogi razlozi za to. Python je interpretirani jezik, dok je C++ kompajlirani. Sve varijable u Pythonu su objekti i ne postoje primitivni tipovi podataka(int, float, double). Objekti u Pythonu mogu sadržavati veći broj tipova podataka i stoga zauzimaju više memorije.
Ako je naš numerički algoritam zahtjevan, pametnije ga je prvo implementirati u Pythonu, zatim nakon što smo se upoznali s brzinom rada algoritma, ponovno u C++. Tako dobivamo na performansama i podatke dobivene implementacijom u C++-u lako možemo vizualizirati pomoću biblioteka u Pythonu. Za jednostavnije probleme, kao rješavanje običnih diferencijalnih jednadžbi i manje komplicirane numeričke sustave, koristimo Python kao potpuno programsko rješenje.

Podatke dobivene programom u C++ -u spremimo unutar tekstualne datoteke i proslijedimo Python programu unutar kojeg ih lako možemo vizualizirati. To nam omogućava biblioteka matplotlib.

Poglavlje 2

Numeričko rješavanje Hamiltonovih jednadžbi

2.1 Rješavanje običnih diferencijalnih jednadžbi

Najjednostavnije jednadžbe koje možemo numerički integrirati su obične diferencijalne jednadžbe prvog reda[1]. To su jednadžbe oblika

$$\frac{dy}{dt} = f(y,t)$$

uz početni uvjet

$$y(t_0) = y_0$$

Početni uvjet y_0 i funkcija $f(y,t)$ su zadani.

Linearne obične diferencijalne jednadžbe i neke nelinearne mogu se vrlo lako riješiti analitički. Općenito, može biti teško naći analitičko rješenje običnih diferencijalnih jednadžbi.

Numerički, vrijednost funkcije $y(t)$ u kasnijem trenutku $t + \Delta t$ aproksimi-

ramo Taylorovim redom potencija

$$y(t_0 + \Delta t) = y(t_0) + \Delta t \frac{dy}{dt} = y(t_0) + \Delta t f(y_0, t_0) + O(\Delta t^2)$$

Ovu jednadžbu transformiramo uvođenjem notacije $t_n = t_0 + n\Delta t$ i $y_n = y(t_n)$ te dobijemo:

$$y_{n+1} = y_n + \Delta t f(y_n, t_n) + O(\Delta t^2)$$

Ovu metodu nazivamo Eulerovom metodom i ona proizvodi grešku reda veličine $O(\Delta t^2)$ u svakom vremenskom koraku.[5] Ako grešku želimo smanjiti, možemo odabrati manji Δt ili koristiti neku od metoda višeg reda. Metoda koja proizvodi grešku $O(\Delta t^n)$ po vremenskom koraku nazivamo lokalno n-tog reda. Ako želimo izračunati ukupnu akumuliranu grešku metode, iteriramo metodu koja je lokalno n-tog reda u nekom određenom vremenskom intervalu T. Pritom smo napravili $\frac{T}{\Delta t}$ koraka i u svakom koraku smo dodali grešku reda veličine $O(\Delta t^n)$. Ukupna greška nakon vremena T je tada:

$$\frac{T}{\Delta t} O(\Delta t^n) = O(\Delta t^{n-1})$$

i za tu metodu kažemo da je globalno reda (n-1). Tako je Eulerova metoda lokalno 2. reda, dok je globalno 1. reda.

2.2 Integracija jednadžbi gibanja

Drugi Newtonov zakon nam daje jednadžbe gibanja klasičnih čestica. Ako se radi o sustavu od N čestica, jednadžbe gibanja su obične diferencijalne jednadžbe 2. reda:

$$m_i \vec{a}_i = \vec{F}_i$$

pri čemu je

$$\vec{a}_i = \frac{d\vec{v}_i}{dt} = \frac{d^2\vec{x}_i}{dt^2}$$

i tako dobivamo dvije vezane jednadžbe:

$$m_i \frac{d\vec{v}_i}{dt} = \vec{F}_i(t, \vec{x}_1, \cdots, \vec{x}_N, \vec{v}_1, \cdots, \vec{v}_N)$$

$$\frac{d\vec{x}_i}{dt} = \vec{v}_i$$

gdje su m_i masa, $\vec{v}_i$ brzina i $\vec{x}_i$ položaj i-te čestice te $\vec{F}_i$ ukupna sila koja djeluje na tu česticu[6].

Koristeći notaciju $t_n = t_0 + n\Delta t$ za vremenski interval, $x_n = x(t_n)$ za položaj, $v_n = v(t_n)$ za brzinu čestice i $a_n = a(t_n, x_n, v_n) = F(t_n, x_n, v_n)/m$ za akceleraciju čestice u n-tom koraku, dobivamo za jednostavnu Eulerovu metodu:

$$v_{n+1} = v_n + a_n \Delta t$$

$$x_{n+1} = x_n + v_n \Delta t$$

Ova metoda je nestabilna čak i za najjednostavnije sustave kao što je linearni harmonički oscilator. Greške rastu eksponencijalno bez obzira na veličinu odabranog koraka Δt.

Jednostavan način za stabilizaciju Eulerove metode za sile koje nisu funkcija brzine je korištenje implicitne Eulerove sheme pri čemu je brzina sada

$$v_{n+1} \approx \frac{x_{n+1} - x_n}{\Delta t}$$

umjesto

$$v_n \approx \frac{x_{n+1} - x_n}{\Delta t}$$

i tako dobijemo stabilnu Eulerovu metodu:

$$v_{n+1} = v_n + a_n \Delta t$$

$$x_{n+1} = x_n + v_{n+1}\Delta t$$

Još jedan stabilni Eulerov algoritam je mid-point metoda koja uzima srednju vrijednost trenutne i prošle brzine:

$$v_{n+1} = v_n + a_n\Delta t$$
$$x_{n+1} = x_n + \frac{1}{2}(v_{n+1} + v_n)\Delta t$$

Za razliku od prošlih metoda koje su prvog reda, leapfrog metoda je drugog reda, no također je vrlo jednostavna:

$$v_{n+\frac{1}{2}} = v_{n-\frac{1}{2}} + a_n\Delta t$$
$$x_{n+1} = x_n + v_{n+\frac{1}{2}}\Delta t$$

ona računa pozicije i brzine u različitim trenucima. Potrebno je zadati prvi polu-korak običnom Eulerovom metodom:

$$v_{\frac{1}{2}} = v_0 + \frac{1}{2}a_0\Delta t$$

Za sile ovisne o brzini koristimo Euler-Richardson algoritam (koji je drugog reda):

$$a_{n+\frac{1}{2}} = a\left(x_n + \frac{1}{2}v_n\Delta t, v_n + \frac{1}{2}a_n\Delta t, t_n + \frac{1}{2}\Delta t\right)$$
$$v_{n+1} = v_n + a_{n+\frac{1}{2}}\Delta t$$
$$x_{n+1} = x_n + v_n\Delta t + \frac{1}{2}a_{n+\frac{1}{2}}\Delta t^2$$

Najčešće korištena metoda je Verlet metoda:

$$x_{n+1} = x_n + v_n\Delta t + \frac{1}{2}a_n(\Delta t)^2$$
$$v_{n+1} = v_n + \frac{1}{2}(a_n + a_{n+1})\Delta t$$

koja je trećeg reda u položajima i drugog reda u brzinama.

2.3 Runge-Kutta metoda

Runge-Kutta algoritam je algoritam za numeričko računanje diferencijalnih jednadžbi opće namjene. Jedini nedostatak je što nije simplektički algoritam, što znači da ne vrijedi zakon očuvanja energije. Jednostavni Eulerov algoritam koristi tangentu na krivulju rješenja y(t) u točki t:

$$y(t+dt) = y(t) + dt \left.\frac{dy}{dt}\right|_t = y(t) + dtf(y,t) \equiv y(t) + k_1$$

Runge-Kutta algoritam drugog reda uzima polovični Eulerov korak do točke

$$t + 0.5dty(t) + 0.5k_1$$

Postoji druga krivulja rješenja $y_1(t)$ koja prolazi kroz ovu točku. Tangenta u toj točki se koristi za izračun

$$k_2 \equiv dt \left.\frac{dy_1}{dt}\right|_{t=t+0.5dt} = dt(f(t) + 0.5k_1, t + 0.5dt)$$

Procjena Runge-Kutta drugog reda se tada računa

$$y(t+dt) = y(t) + k_2$$

Greška je u ovom slučaju reda veličine dt^3, što se pokaže razvojem u Taylorov red:

$$\begin{aligned}
y(t+dt) =& y(t) + dt\frac{dy}{dt} + \frac{dt^2}{2}\frac{d^2y}{dt^2} + O(dt^3) \\
=& y(t) + dtf(y,t) + \frac{dt^2}{2}\frac{df(y,t)}{dt} + O(dt^3) \\
=& y(t) + dtf(y,t) + \frac{dt^2}{2}\left(\frac{df}{dy}\frac{dy}{dt} + \frac{df}{dt}\right) + O(dt^3) \\
=& y(t) + dtf(y,t) + \frac{dt^2}{2}\left(\frac{df}{dy}f + \frac{df}{dt}\right) + O(dt^3)
\end{aligned}$$

Razvojem desne strane dobijemo

$$\begin{aligned}
y(t) + k_2 =& y(t) + dtf(y + 0.5k_1, t + 0.5dt) \\
=& y(t) + dtf(y,t) + dt\left(\frac{df}{dy}\frac{k_1}{2} + \frac{dt}{2}\frac{df}{dt}\right) + O(dt^3)
\end{aligned}$$

Vidimo da izjednačavanjem lijeve i desne strane $O(dt^3)$ iščezava.[5] Matematičari Runge i Kutta su pokazali da kombinacijom rezultata dviju dodatnih Eulerovih koraka možemo grešku smanjiti na $O(dt^5)$. Lako se vidi da se ovi algoritmi mogu proširiti na proizvoljno velike sustave običnih diferencijalnih jednadžbi prvog reda. Vektorski oblik Runge-Kutta algoritma 4. reda je

$$\begin{aligned}
\vec{k_1} =& dt\vec{f}(\vec{y}(t), t) \\
\vec{k_2} =& dt\vec{f}(\vec{y(t)} + 0.5\vec{k_1}, t + 0.5dt) \\
\vec{k_3} =& dt\vec{f}(\vec{y(t)} + 0.5\vec{k_2}, t + 0.5dt) \\
\vec{k_4} =& dt\vec{f}(\vec{y(t)} + \vec{k_3}, t + dt)
\end{aligned}$$

tako dobijemo vektorsku funkciju $y(t)$ u kasnijem trenutku $y(t + dt)$:

$$\vec{y}(t + dt) = \vec{y}(t) + \frac{\vec{k_1} + 2\vec{k_2} + 2\vec{k_3} + \vec{k_4}}{6} + O(dt^5)$$

Poglavlje 3

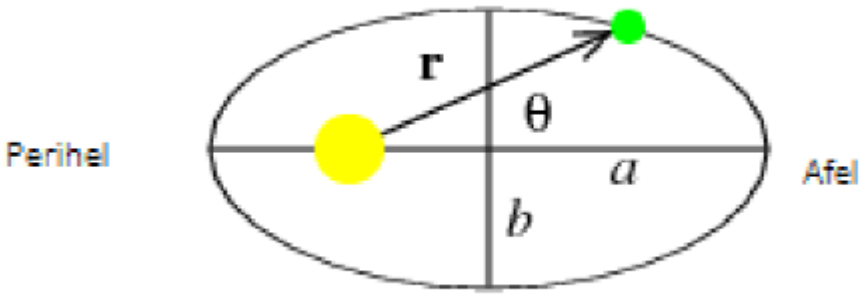
Keplerov 3-body problem

3.1 Keplerovi zakoni i dinamika tijela Sunčevog sustava

Orbite tijela Sunčevog sustava oko Sunca određene su Newtonovim zakonom gravitacije i Newtonovim zakonima gibanja. To su početni problemi ili sustavi od N običnih diferencijalnih jednadžbi drugog reda čija su rješenja određena uz početne uvjete u nekom trenutku.

Johannes Kepler je otkrio 3 zakona vezana uz gibanje planeta oko Sunca:

1. Planeti se gibaju po eliptičnim orbitama oko Sunca
2. Dužina koja spaja Sunce i planet „prebriše“ jednake površine u jednakim vremenskim intervalima.
3. Kvadrat perioda orbite planeta je proporcionalan kubu velike poluosi orbite.



Slika 1. Keplerove orbite

Na slici 1 vidimo skicu putanje planeta oko Sunca, pri čemu mu je a velika poluos, b mala poluos, a perihel i afel su točke koje leže na velikoj osi i najbliže su odnosno najdalje točke od Sunca u orbiti.

3.2 Newtonov zakon univerzalne gravitacije

Isaac Newton je dokazao da Keplerove zakone poštuje izolirani vezani sustav od bilo koja dva objekta koji međudjeluju centralnom silom obrnuto proporcionalnom kvadratu njihove udaljenosti. Gravitacija je takva sila:

$$\vec{F}_{12} = -Gm_1m_2\frac{\vec{r}_1 - \vec{r}_2}{|\vec{r}_1 - \vec{r}_2|^3}$$

$$\vec{F}_{21} = -Gm_1m_2\frac{\vec{r}_2 - \vec{r}_1}{|\vec{r}_2 - \vec{r}_1|^3}$$

Pri čemu je $\vec{F}_{21}$ sila koju čestica 2 vrši na česticu 1.

Ove sile poštuju 3. Newtonov zakon(zakon akcije i reakcije):

$$\vec{F}_{21} = -\vec{F}_{12}$$

3.3 Rješenje Keplerova problema dvaju tijela

S obzirom da je Sunce najveći objekt u Sunčevom sustavu, većinom koristimo Kopernikov koordinatni sustav sa Suncem centriranim u ishodištu. Ovaj sustav je samo aproksimativno inercijalan. Ova aproksimacija je dobra za komete i velike planete kao što je Jupiter. Ako dva tijela imaju sličnu masu, kao kod binarnih zvjezdanih sustava, tada je centar mase inercijalan. Rješenje se u ovom slučaju može prikazati u ovisnosti o reduciranoj masi i relativnoj koordinati:

$$\mu = \frac{m_1 m_2}{m_1 + m_2}$$

$$\vec{r} = \vec{r_1} - \vec{r_2}$$

$$m_1\vec{r_1} + m_2\vec{r_2} = 0$$

U Kopernikovom koordinatnom sustavu sa Suncem u ishodištu vrijedi da je $m_2 = M >> m_1 = m$ iz čega slijedi da su $\mu \approx m$, $r \approx r_1$ i $r_2 \approx 0$. Keplerova orbita se može napisati u parametarskom obliku kao:

$$r(\theta) = \frac{a(1-\epsilon^2)}{1 - \epsilon cos(\theta)}$$

$$b = a\sqrt{1-\epsilon^2}$$

Pri čemu je a velika, b mala poluos, a ϵ ekscentricitet orbite. Ekscentricitet određuje oblik putanje i za $\epsilon = 0$ putanja je kružna, za $0 < \epsilon < 1$ elipsa, za $\epsilon = 1$ parabola, a za $\epsilon > 1$ hiperbola. Orbitalna brzina je dana sa:

$$v = \sqrt{G(m_1 + m_2)\left(\frac{2}{r} - \frac{1}{a}\right)}$$

Zakon očuvanja energije daje:

$$E = \frac{1}{2}\mu v^2 - \frac{Gm_1m_2}{r} = -\frac{Gm_1m_2}{2a}$$

Angularni moment je također očuvan:

$$\vec{L} = m\vec{r} \times \dot{\vec{r}} = k\vec{on}st.$$

Period orbite dan je trećim Keplerovim zakonom:

$$T^2 = \frac{4\pi^2}{G(m_1 + m_2)} a^3$$

Naravno, čak niti sustav centra mase nije inercijalan kada se uzme u obzir gibanje sustava kroz galaksiju.

3.4 Gravitacijski N-body problem

Energija sustava od N klasičnih čestica masa m_i koje međudjeluju gravitacijskom silom je:

$$E = T + V = \frac{1}{2} \sum_{i=1}^{N} m_i \vec{v}_i^2 - G \sum_{i<j} \frac{m_i m_j}{|\vec{r}_i - \vec{r}_j|}$$

Masa, položaj i brzina čestice i su m_i, $\vec{r}_i$ i $\vec{v}_i$. Potencijalna energija je suma po svim parovima masa a broj parova je:

$$\frac{N(N-1)}{2}$$

N-body problem je naći putanje $\vec{r}_i$ svih masa m_i kao funkcije vremena t uz poznate početne uvjete položaja i brzina svih čestica $\{\vec{r}_i(t=0), \vec{v}_i(t=0)\}$.

3.5 Analitičko rješenje problema

Nakon što je Newton dokazao da je opće rješenje klasičnih jednadžbi gibanja dviju vezanih točkastih čestica Keplerova elipsa, krenula je potraga za općim rješenjem sustava od 3 gravitacijski međudjelujućih čestica.

Problem triju tijela je rješavanje sustava od 9 vezanih diferencijalnih jednadžbi drugog reda:

$$\frac{d^2\vec{r}_1}{dt^2} = -Gm_2\frac{\vec{r}_1 - \vec{r}_2}{|\vec{r}_1 - \vec{r}_2|^3} - Gm_3\frac{\vec{r}_1 - \vec{r}_3}{|\vec{r}_1 - \vec{r}_3|^3}$$

$$\frac{d^2\vec{r}_2}{dt^2} = -Gm_3\frac{\vec{r}_2 - \vec{r}_3}{|\vec{r}_2 - \vec{r}_3|^3} - Gm_1\frac{\vec{r}_2 - \vec{r}_1}{|\vec{r}_2 - \vec{r}_3|^1}$$

$$\frac{d^2\vec{r}_3}{dt^2} = -Gm_1\frac{\vec{r}_3 - \vec{r}_1}{|\vec{r}_3 - \vec{r}_1|^3} - Gm_2\frac{\vec{r}_3 - \vec{r}_2}{|\vec{r}_3 - \vec{r}_3|^2}$$

Uz poznavanje 19 početnih uvjeta vremena, položaja i brzina.

Postoji vrlo malen broj točnih analitičkih rješenja za problem triju tijela. Poincaré je pokazao da problem nije integrabilan. Većinom su orbite nestabilne i kaotične, što znači da proizvoljno bliski početni uvjeti rezultiraju orbitama koje se razdvajaju eksponencijalno brzo i ne mogu se numerički računati dugo vremena.

3.6 Stupnjevi slobode i integrabilnost

Broj stupnjeva slobode mehaničkog sustava je broj fizikalnih veličina koje moraju biti poznate da bi se opisala konfiguracija sustava u bilo kojem trenutku. Za sustav od N tijela koja se mogu slobodno kretati prostorom, postoji 3N komponenti vektora položaja.

U Newtonovoj mehanici, svaki stupanj slobode poštuje jednu diferencijalnu jednadžbu drugog reda. Broj početnih uvjeta koji moraju biti poznati da bismo dobili jedinstveno rješenje je dvostruki broj stupnjeva slobode, na primjer, za opis gibanja N točkastih masa trebamo 6N početnih vrijednosti

položaja i brzina.

Sustav od N tijela sa 3N stupnjeva slobode se može pokazati integrabilnim ako postoji 3N nezavisnih integrala gibanja, to jest, funkcija položaja i brzina koje su konstantne u vremenu. Tada se rješenje može reducirati na računanje određenih integrala.

Problem triju tijela ima 9 stupnjeva slobode: energija, ukupni moment i ukupni kutni moment čine 7 nezavisnih očuvanih veličina. Jednadžbe problema triju tijela nisu integrabilne bez dviju dodatnih očuvanih veličina.

Poglavlje 4

Rezultati

4.1 Uvod

U sklopu završnog rada zadatak je bio napraviti program u programskom jeziku Python koji računa orbite 3D sustava triju čestica. Program računa orbite pomoću Runge-Kutta algoritma 4. reda. Unutar programa je dio koda za vizualizaciju trodimenzionalnih orbita tijela i vremenskih promjena energije.

Programski jezik Python je vrlo jednostavan za korištenje i učenje. Postoje mnogi programski paketi koji sadrže sve potrebne biblioteke za pokretanje ovog koda. Mogu se koristiti programski paket Anaconda za Linux i Windows operativne sustave, koji se može preuzeti sa `http://www.continuum.io/`. Drugi programski paket je Python(x,y) za Windows operativni sustav i može se preuzeti sa `https://python-xy.github.io/`. Oba su open-source i besplatni za korištenje s time da paket Anaconda ima i naprednije funkcije koje se naplaćuju ali nisu potrebne osim za bavljenje paralelnim programiranjem. Unutar oba paketa, glavno integrirano razvojno okruženje je Spyder. Unutar

programskih paketa program koristi biblioteke Numpy i matplotlib. Prva služi za brze numeričke operacije nad matricama i kao generator nasumičnih brojeva. Druga služi za vizualizaciju rezultata grafovima.

Glavna prednost Pythona je da je to interpretirani programski jezik i kod napisan u bilo kojem operativnom sustavu se može pokrenuti u bilo kojem drugom. Tako se ovaj kod može koristiti za podučavanje numeričkih metoda rješavanja vezanih sustava diferencijalnih jednadžbi. Kod je prenosiv i razumljiv, pa možda i na račun performansi. Naglasak je na podučavanju ovih metoda i programski kod mora biti čitljiv i bez previše optimizacija, za koje uvijek ima vremena kasnije, a i obično se ne rade unutar jezika kao što je Python. Kad bi nam bile bitne performanse, koristili bismo C ili C++. S druge strane, to bi nam otežalo vizualizaciju podataka i morali bismo imati poseban program za to što oduzima vrijeme.

4.1.1 Objašnjenje dijelova koda

Programski kod je u prilogu i dijeli se na 3 dijela, pozivanje programskih biblioteka, funkcije i glavni dio programa.

Program sadrži 6 glavnih funkcija. Postoje funkcije za računanje udaljenosti između dvije točke, računanje energije stanja sustava u nekom trenutku, male perturbacije, jednadžbe gibanja koje predstavljaju derivacije položaja i brzina po vremenu, metodu za integriranje $RK4\_Step$ koja računa stanje sustava u kasnijem trenutku $t + dt$ i metodu Sustav koja gradi matricu sustava reda $19 \times N$ gdje je N broj koraka integracije i svaki redak matrice je stanje sustava u jednom trenutku t.

U glavnom dijelu programa zadajemo korak dt, vrijeme integracije tmax, i iznose triju masa. Početne uvjete sustava proslijedimo funkciji Sustav. Podatke vizualiziramo pomoću funkcija iz programske biblioteke matplotlib.

4.1.2 Odabir sustava mjernih jedinica

Gravitacijska konstanta

$$G = 6.67384(80) \times 10^{-11} m^3 kg^{-1} s^{-2} = 6.67384(80) \times 10^{-11} Nm^2 kg^{-2}$$

je vrlo malen broj i stvara probleme pri numeričkom rješavanju diferencijalnih jednadžbi. Dobra je ideja izbjegavati vrlo male vrijednosti radi numeričkog overflowa i underflowa. Zato koristimo sustav jedinica koji je prirodan problemu.

Zemljina velika poluos je dugačka $1AU = 1.496 \times 10^{11} m$. Po Keplerovom trećem zakonu imamo:

$$G(M_{Sunca} + m_{Zemlje}) = \frac{4\pi^2 a^3}{T^3}$$

Koristimo jedinice tako da je $T^2 = a^3$ to jest,

$$G(M_{Sunca} + m_{Zemlje}) = 4\pi^2$$

U našem slučaju radi se o tri tijela proizvoljnih masa:

$$G(m_1 + m_2 + m_3) = 4\pi^2$$

Radi toga dobivamo numerički stabilnije rješenje s manjim greškama i vidljivijim gravitacijskim utjecajima.

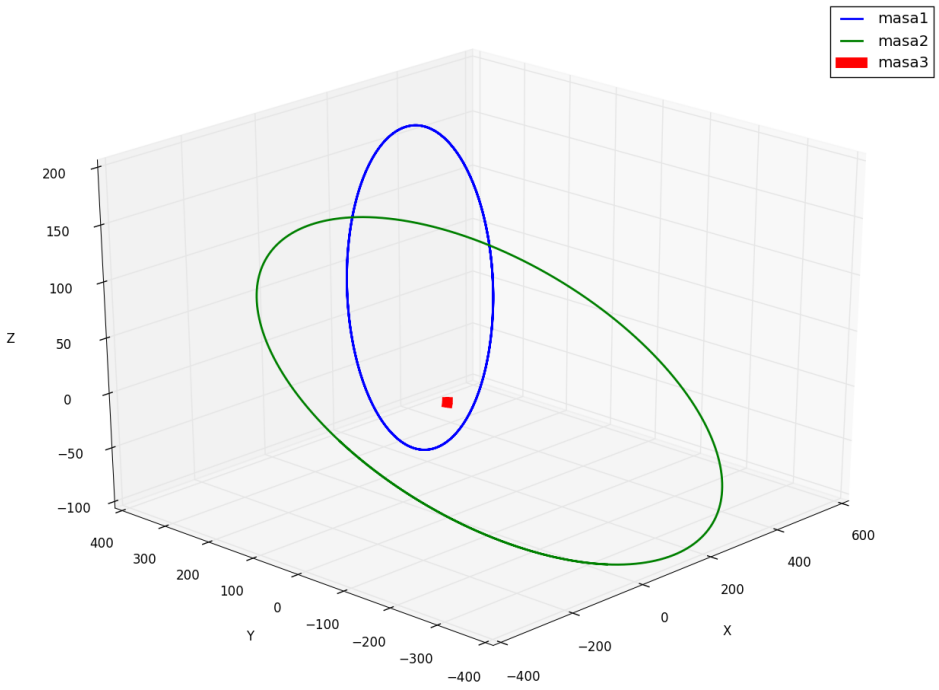
4.2 Orbite

Potencijal definiran unutar programa je najosnovniji oblik centralnog potencijala koji opada sa r^2, nismo koristili multipolni razvoj potencijala i zato orbite nemaju zakret perihela. Cilj je bio što jednostavniji program zbog lakog razumijevanje. Iz osnovnih zakona se mogu vidjeti prepoznatljive eliptične krivulje.

4.2.1 Eliptične orbite

Odabrali smo početne uvjete simulacije za eliptične putanje:

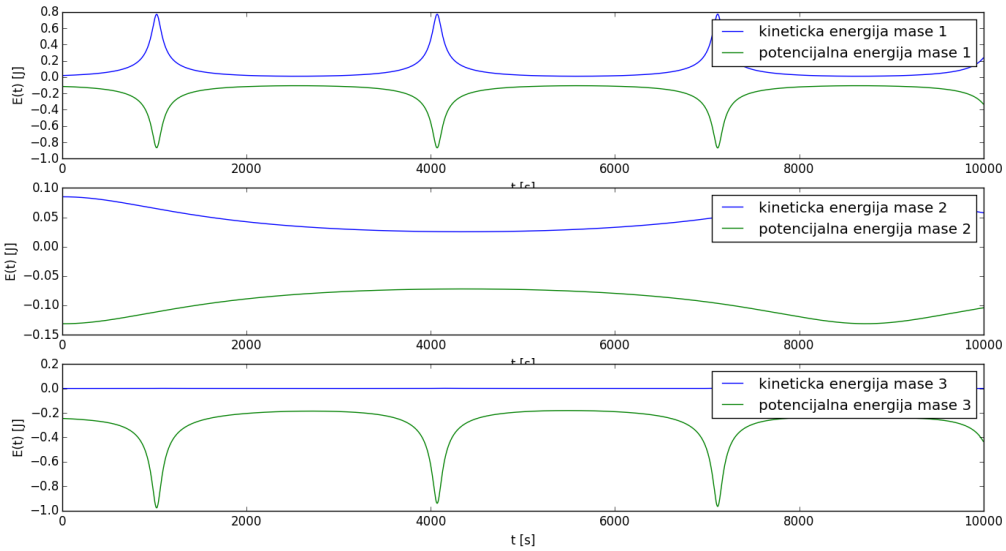
- mase: $m_1 = 1$ $m_2 = 1$ $m_3 = 1000$
- položaji: $\vec{r}_1 = (200, 200, 200)$ $\vec{r}_2 = (-300, 0, 0)$ $\vec{r}_3 = (0, 0, 0)$
- brzine: $\vec{v}_1 = (0, -0.2, 0)$ $\vec{v}_2 = (0, -0.4, -0.1)$ $\vec{v}_3 = (0, 0, 0)$
- vremenski korak $dt = 1$ i broj koraka $N = \frac{T_{max}}{dt} = \frac{10000}{1} = 10000$ rezultati su prikazani na slici 2.



Slika 2. Eliptične putanje

Za određene početne uvjete položaja i brzina triju masa dobijemo eliptične orbite. Kada su dvije od masa jednake i jedna 10^3 puta veća, dobijemo orbite nalik na one u Sunčevom sustavu. Masa m_3 ovdje prikazana crvenom bojom je analog Suncu u Sunčevom sustavu. Gravitacijski utjecaj ostale dvije mase na masu m_3 je zanemariv.

Potencijalne i kinetičke energije za tri mase su prikazane na slici 3 uz jednake početne uvjete i broj koraka $N = 10000$.

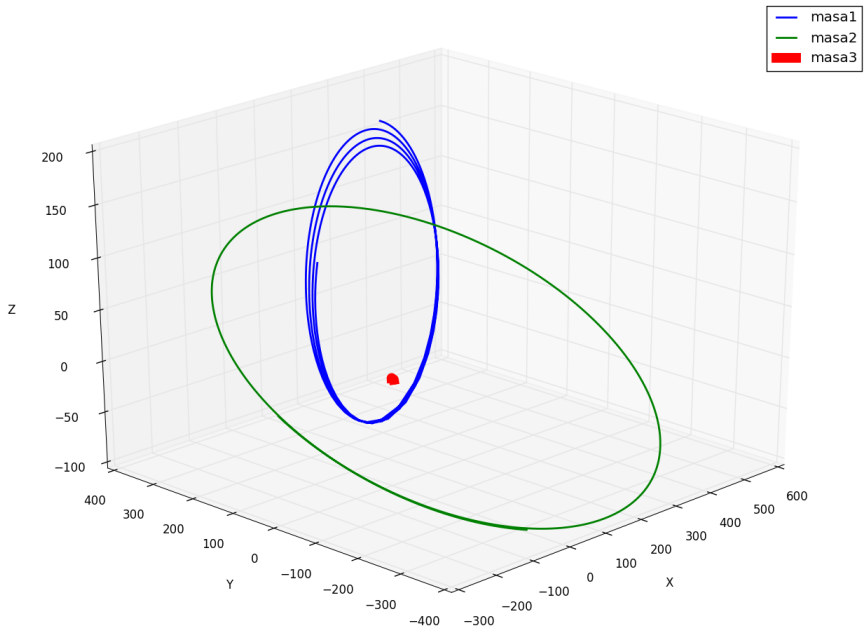


Slika 3. Odnos kinetičke i potencijalne energije

Na slici 3 možemo vidjeti da dolazi do prepoznatljivih oscilacija kinetičke i potencijalne energije. Ukupna energija je zbroj kinetičke i potencijalne i ima negativnu srednju vrijednost jer se radi o vezanom sustavu triju gravitacijski međudjelujućih čestica.

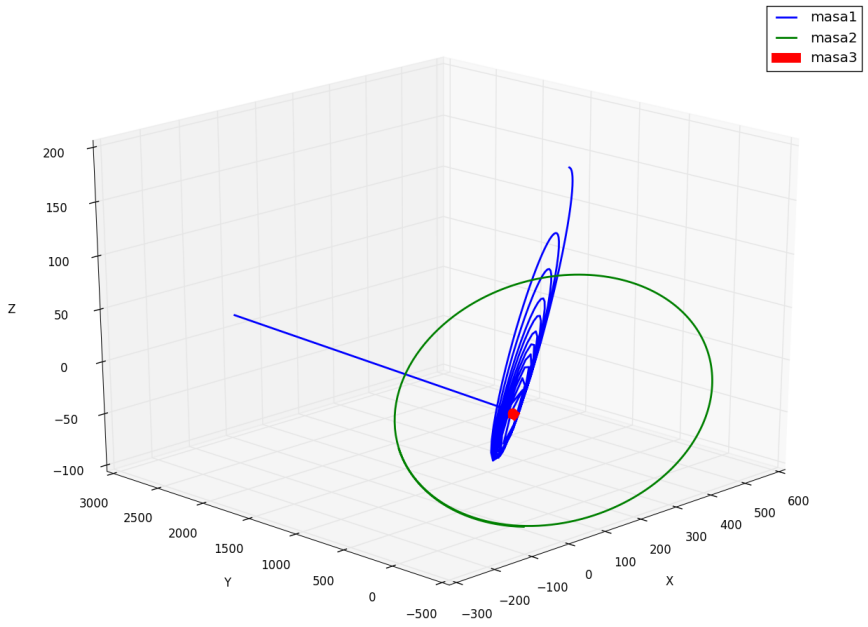
4.2.2 Utjecaj koraka dt na stabilnost rješenja

Smanjenje koraka pridonosi stabilnosti rješenja. U prethodnom primjeru vremenski korak je bio zadan kao $dt = 1$. Smanjenjem te vrijednosti, veća stabilnost rješenja neće biti vidljiva, ali povećanjem koraka, počinju se vidjeti nestabilnosti. Za vremenski korak $dt = 30$ već dolazi do rasipanja elipse. Na slici 4 su prikazani rezultati uz jednake početne uvjete kao u prvom primjeru.



Slika 4. Rasipanje elipse pri povećanju vremenskog koraka na $dt = 30$

Ako odaberemo korak $dt = 50$ uz jednake početne uvjete, putanja postaje vrlo drugačija od eliptične krivulje koje je bila za $dt = 1$ i prikazana slikom 2. Rezultati su prikazani na slici 5.



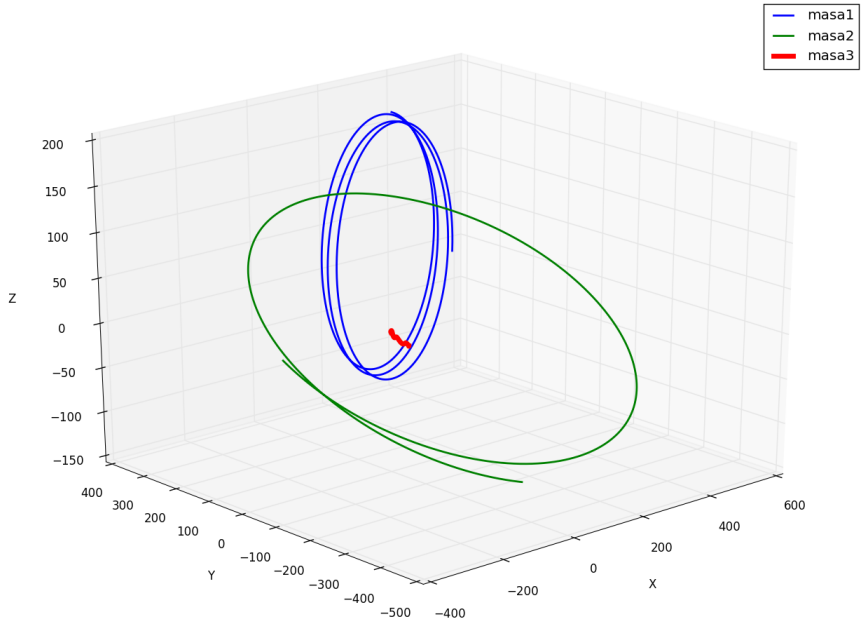
Slika 5. Vrlo nestabilna putanja za korak $dt = 50$

4.2.3 Promjena orbita s obzirom na masu

U prethodnim primjerima, masa m_3 je bila 10^3 puta veća od ostale dvije mase. Mijenjajući iznos m_3 može se uočiti prijelaz iz eliptičnih orbita u spiralne. Odaberemo li početne uvjete

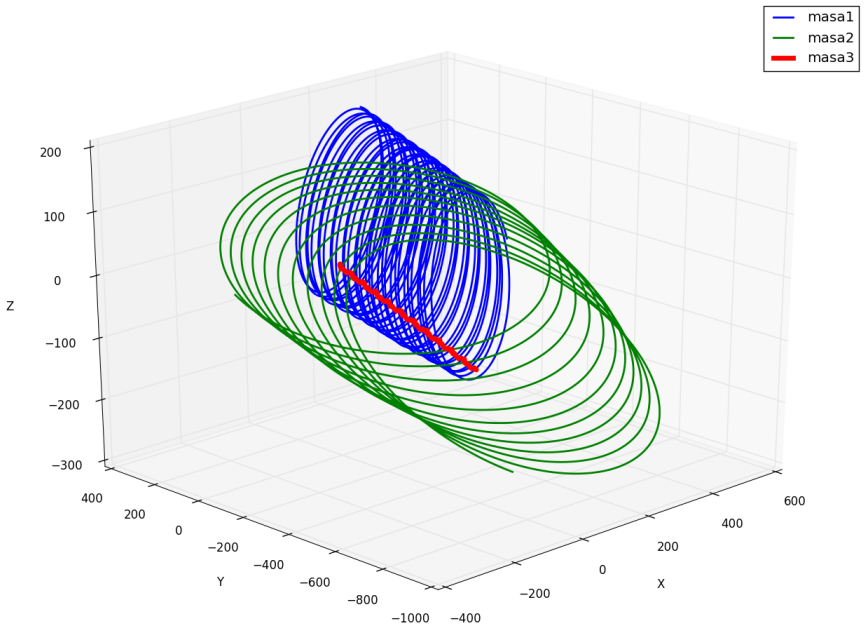
- mase: $m_1 = 1$ $m_2 = 1$ $m_3 = 100$
- položaji: $\vec{r}_1 = (200, 200, 200)$ $\vec{r}_2 = (-300, 0, 0)$ $\vec{r}_3 = (0, 0, 0)$
- brzine: $\vec{v}_1 = (0, -0.2, 0)$ $\vec{v}_2 = (0, -0.4, -0.1)$ $\vec{v}_3 = (0, 0, 0)$
- vremenski korak $dt = 1$ i broj koraka $N = \frac{T_{max}}{dt} = \frac{10000}{1} = 10000$

Rezultati su vidljivi na slici 6.



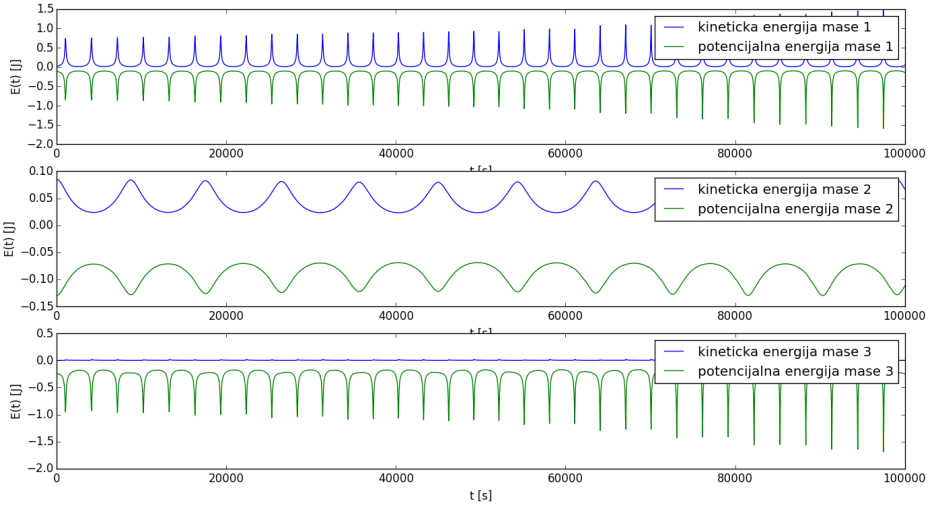
Slika 6. Početak spiralne orbite

Manje mase m_1 i m_2 sada imaju puno veći utjecaj na orbitu veće mase m_3 i ona se počinje gibati zajedno s njima. U tom primjeru možemo vidjeti gravitacijske utjecaje dvaju manjih tijela na veće. Za 10 puta veći broj koraka $N = 100000$ može se vidjeti spiralna orbita koja je prikazana na slici 7.



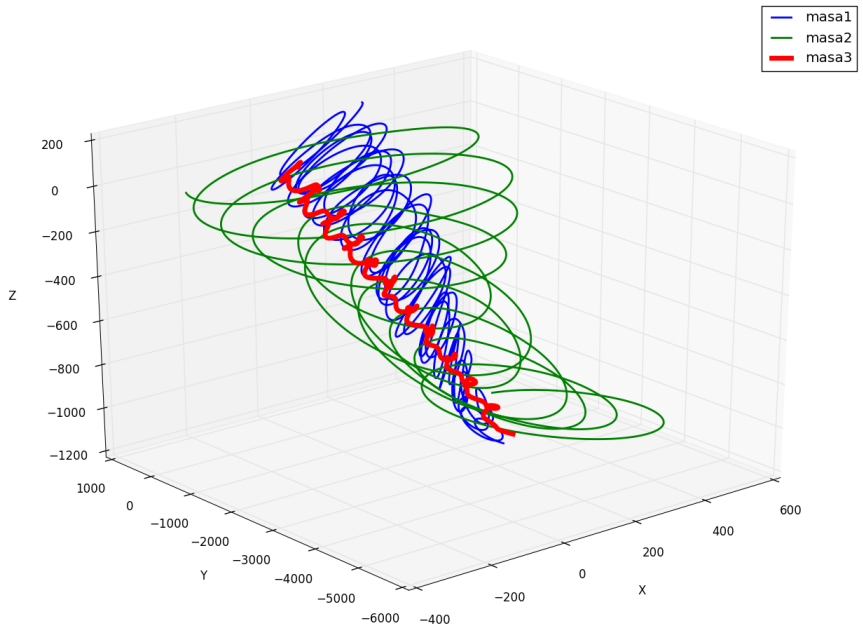
Slika 7. Spiralna orbita

Kinetičke i potencijalne energije triju masa su prikazane na slici 8.



Slika 8. Energije za spiralnu orbitu

Odaberemo li jednake početne uvjete i mase $m_1 = 1$, $m_2 = 1$ i $m_3 = 10$, na slici 9 vidimo povećan utjecaj manjih masa na veću.



Slika 9. Spiralna orbita najteže mase

Vidi se da i najteža masa ima spiralnu orbitu oko zajedničkog centra mase koji se giba konstantnom brzinom. Pri gibanju kroz Mliječnu stazu, gotovo svako tijelo Sunčevog sustava se giba spiralnom orbitom, čak i Sunce. Ako uzmemo u obzir da centar mase sustava dvaju tijela Jupiter-Sunce upada unutar Sunčevog radijusa, očito je spiralno gibanje Sunca slabo uočljivo.

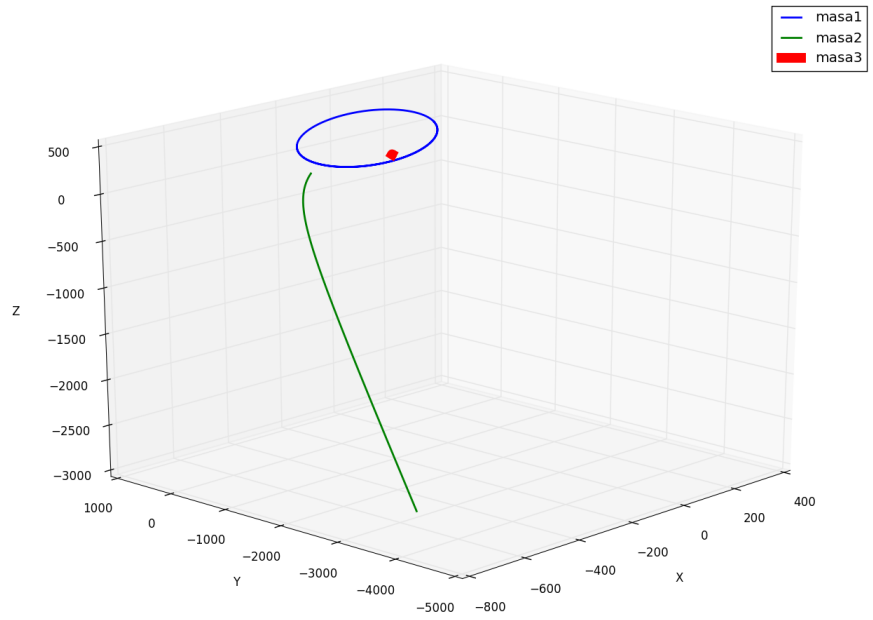
4.2.4 Male promjene početnih brzina

Ako odaberemo mase $m_1 = 1$ $m_2 = 1$ i $m_3 = 1000$, vremenski korak $dt = 1$, broj koraka $N = 10000$ i početne uvjete kao u prethodnim primjerima, uz malu promjenu početnih brzina za mase m_1 i m_2, dobijemo malu promjenu

u konačnom rezultatu jer najteža masa ima najveći utjecaj na gibanje ostale dvije mase. U programu, mala perturbacija se definira tako što se na iznose komponenti brzina masa m_1 i m_2 dodaje nasumičan broj u nekom rasponu. Funkcija koja definira perturbaciju unutar koda koristi programsku biblioteku Numpy i vraća realan broj u nekom rasponu:

```
def malaPerturbacija(raspon):
    return np.random.uniform(-raspon,raspon)
```

U ovom slučaju taj raspon je interval $[-0.3, 0.3]$. Rezultati promjene brzina se vide na slici 10.



Slika 10. Promjenjene putanje

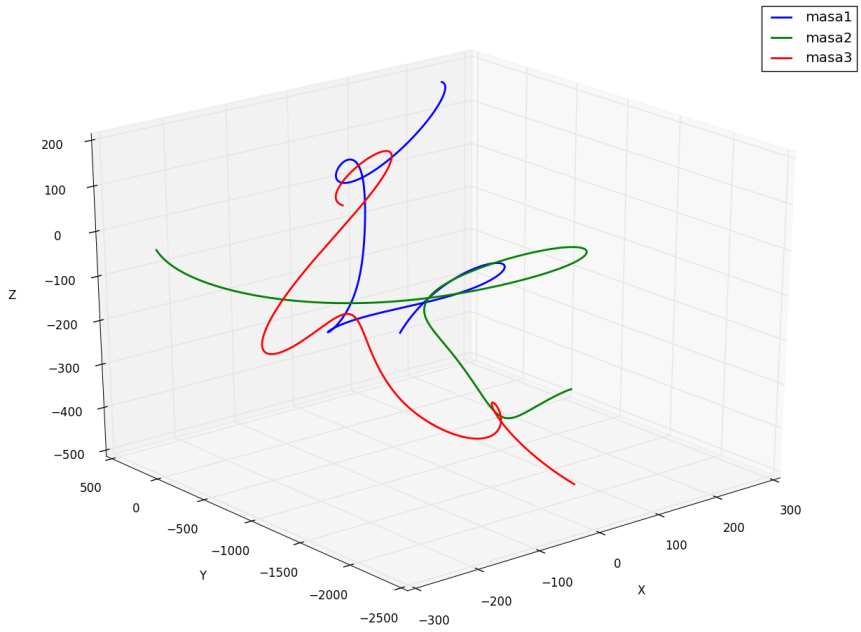
Vidimo da putanje mogu biti eliptičke i hiperboličke. Uz malu promjenu početnih uvjeta brzina, m_1 je ostala na eliptičnoj orbiti, dok je m_2 dosegla dovoljnu razinu kinetičke energije i postigla hiperboličku orbitu.

4.2.5 Jednaki iznosi masa

Kaotičnost sistema vidimo kada su sve tri mase istog reda veličina. Za početne parametre

- mase: $m_1 = 1$ $m_2 = 1$ $m_3 = 1$
- položaji: $\vec{r}_1 = (200, 200, 200)$ $\vec{r}_2 = (-300, 0, 0)$ $\vec{r}_3 = (0, 0, 0)$
- brzine: $\vec{v}_1 = (0, -0.2, 0)$ $\vec{v}_2 = (0, -0.4, -0.1)$ $\vec{v}_3 = (0, 0, 0)$
- vremenski korak $dt = 1$ i broj koraka $N = \frac{T_{max}}{dt} = \frac{10000}{1} = 10000$

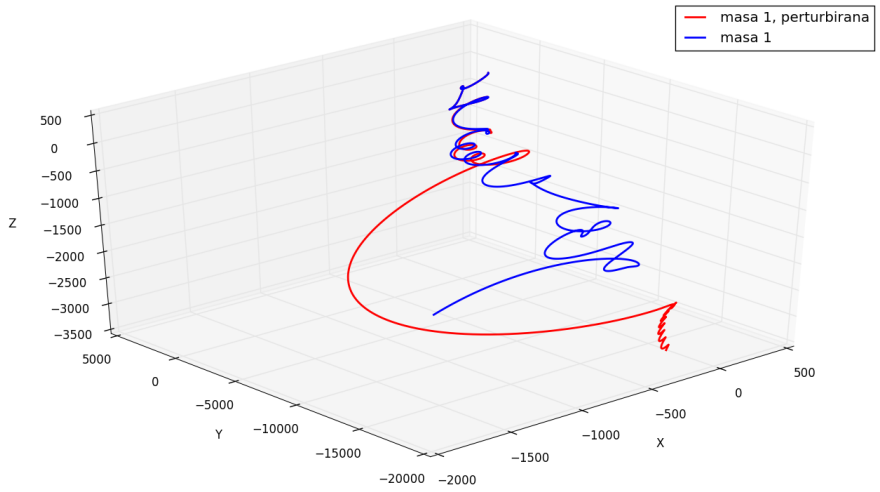
Rezultati su prikazani na slici 11.



Slika 11. Putanje za sustav od triju jednakih masa

Vidi se puno veća osjetljivost promjene gibanja čestica na međusobnu gravitacijsku interakciju. U ovom slučaju, kao i dosad, masa m_3 ima početni iznos brzine $|\vec{v_3}| = 0$ i njeno gibanje je u potpunosti uvjetovano gravitacijskim interakcijama s ostale dvije mase.

Vidjeli smo da su putanje jako ovisne zbog međusobno proporcionalnih gravitacijskih utjecaja masa. Sada radimo malu perturbaciju početnih brzina u rasponu $[-10^{-4}, 10^{-4}]$. Odaberemo li početne uvjete kao u prethodnom primjeru i jednake mase te uz malu perturbaciju početnih brzina dviju masa m_1 i m_2. Ne očekujemo veliku promjenu s obzirom na rezultate na slici 11, međutim na slici 12 vidimo kaotičnost ovog sustava.



Slika 12. Originalna i perturbirana putanja za masu 1

Originalna putanja čestice 1 je prikazana plavom crtom, dok je perturbirana putanja prikazana crvenom crtom na slici 11.

4.2.6 Očuvanje energije

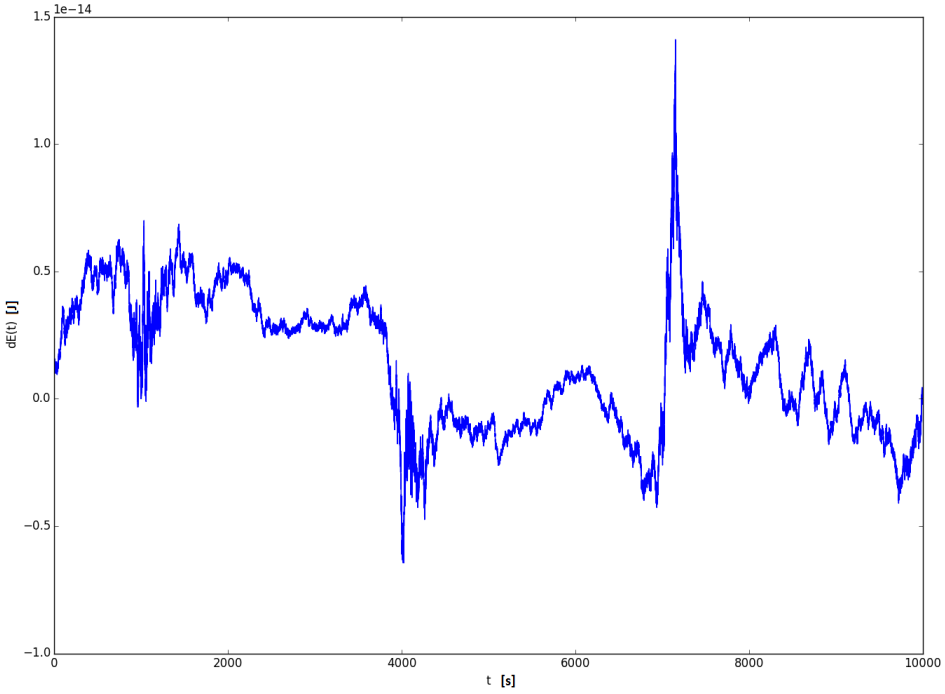
Prema zakonima mehanike, energija svih izoliranih sustava je očuvana veličina. U numeričkim simulacijama dolazi do odstupanja energije u vremenu. Odstupanja energije dolaze zbog akumulacije numeričkih grešaka.

Ako odaberemo početne uvjete:

- mase: $m_1 = 1\ m_2 = 1\ m_3 = 1000$
- položaji: $\vec{r}_1 = (200, 200, 200)\ \vec{r}_2 = (-300, 0, 0)\ \vec{r}_3 = (0, 0, 0)$
- brzine: $\vec{v}_1 = (0, -0.2, 0)\ \vec{v}_2 = (0, -0.4, -0.1)\ \vec{v}_3 = (0, 0, 0)$

- vremenski korak $dt = 0.1$ i broj koraka $N = 1000000$

Rezultati su prikazani na slici 13.



Slika 13. Vremenska ovisnost promjene ukupne energije

Veličina dE prikazuje razliku energije u nekom trenutku i energije u početnom trenutku. Može se vidjeti da je greška u energiji reda veličine 10^{-14} i oscilira oko nule. Ovakva preciznost je dovoljna za ovakve jednostavne simulacije.

Postoje algoritmi koji se baziraju na poštivanju zakona očuvanja energije kao što je velocity verlet. Oni se koriste u simulacijama molekularne dinamike.

Poglavlje 5

Zaključak

Svojstvo kaotičnih sistema je da mala promjena u početnim uvjetima uzrokuje veliku promjenu u konačnom rješenju. Problem triju tijela nema analitičko rješenje pa korištenjem numeričkih metoda, kao što je metoda konačnih razlika, možemo vrlo lako prikazati ponašanje takvih sistema na računalu. Za današnja računala, moguće je simulirati oko 50000 čestica pomoću programskih paketa za molekularnu dinamiku kao što su DL_POLY i gromacs. Dinamika sustava od 1 mola, $N = 6.022 \cdot 10^{23}$ čestica, za suvremena osobna računala je gotovo pa i nemoguća za prikazati. Preciznost takvih simulacija je upitna s obzirom da i za situacije kao što je problem triju tijela imamo probleme sa stabilnošću rješenja.

Završni rad je usmjeren prikazivanju numeričkog problema rješavanja jednadžbi gibanja sustava od 3 tijela. Primarna svrha je podučavanje jednostavnih numeričkih metoda. Mogućnosti za unaprjeđenje su poopćenje na sustav proizvoljnog broja čestica, uvođenje algoritama koji poštuju zakon očuvanja energije te prelazak na objektno orijentirane klase koje predstavljaju svaku od čestica i sadrže sve podatke o česticama i sva njihova svojstva.

Poglavlje 6

Prilozi

Python kod koji sam koristio unutar završnog rada:

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
import math
import time

########################
def RK4_Step(x, dt, derivacije):
    n = len(x)
    funkcija = [0 for s in range(19)]
    k1 = [0 for s in range(19)]
    k2 = [0 for s in range(19)]
    k3 = [0 for s in range(19)]
    k4 = [0 for s in range(19)]
    x_temp = [0 for s in range(19)]
```

```
    funkcija = derivacije(x)
    for i in range(n):
        k1[i] = dt * funkcija[i]
        x_temp[i] = x[i] + k1[i] / 2
    funkcija = derivacije(x_temp)
    for i in range(n):
        k2[i] = dt * funkcija[i]
        x_temp[i] = x[i] + k2[i] / 2
    funkcija = derivacije(x_temp)
    for i in range(n):
        k3[i] = dt * funkcija[i]
        x_temp[i] = x[i] + k3[i]
    funkcija = derivacije(x_temp)
    for i in range(n):
        k4[i] = dt * funkcija[i]
    for i in range(n):
        x[i] += (k1[i] + 2 * k2[i] + 2 * k3[i] + k4[i]) / 6.0
    return x
######################

def udaljenost(ri,rj):
    return ((ri[0] - rj[0]) **2 +
    (ri[1] - rj[1]) **2+
    (ri[2] - rj[2]) **2) ** 0.5

def malaPerturbacija(raspon):
    return np.random.uniform(-raspon,raspon)
```

```
def jednadzbeGibanja(stanje):
    x1 = stanje[1]; y1 = stanje[2]; z1 = stanje[3];
    vx1 = stanje[4]; vy1 = stanje[5]; vz1 = stanje[6];
    x2 = stanje[7]; y2 = stanje[8]; z2 = stanje[9];
    vx2 = stanje[10]; vy2 = stanje[11]; vz2 = stanje[12];
    x3 = stanje[13]; y3 = stanje[14]; z3 = stanje[15];
    vx3 = stanje[16]; vy3 = stanje[17]; vz3 = stanje[18];

    r12=udaljenost([x1,y1,z1],[x2,y2,z2])
    r13=udaljenost([x1,y1,z1],[x3,y3,z3])
    r23=udaljenost([x2,y2,z2],[x3,y3,z3])

    ax1 = -G * masa2 * (x1 - x2) / (r12 **3) \
    - G * masa3 * (x1 - x3) / (r13 **3)
    ay1 = -G * masa2 * (y1 - y2) / (r12 **3) \
    - G * masa3 * (y1 - y3) / (r13 **3)
    az1 = -G * masa2 * (z1 - z2) / (r12 **3) \
    - G * masa3 * (z1 - z3) / (r13 **3)

    ax2 = -G * masa1 * (x2 - x1) / (r12 **3) \
    - G * masa3 * (x2 - x3) / (r23 **3)
    ay2 = -G * masa1 * (y2 - y1) / (r12 **3) \
    - G * masa3 * (y2 - y3) / (r23 **3)
    az2 = -G * masa1 * (z2 - z1) / (r12 **3) \
    - G * masa3 * (z2 - z3) / (r23 **3)
```

```
    ax3 = -G * masa1 * (x3 - x1) / (r13 **3) \
    - G * masa2 * (x3 - x2) / (r23 **3)
    ay3 = -G * masa1 * (y3 - y1) / (r13 **3) \
    - G * masa2 * (y3 - y2) / (r23 **3)
    az3 = -G * masa1 * (z3 - z1) / (r13 **3) \
    - G * masa2 * (z3 - z2) / (r23 **3)

    return [1.0, vx1, vy1, vz1, ax1, ay1, az1,
            vx2, vy2, vz2, ax2, ay2, az2,
            vx3, vy3, vz3, ax3, ay3, az3]
######################
def Sustav(tmax, dt, m1, m2, m3, x1, y1, z1, vx1, vy1, vz1,
           x2, y2, z2, vx2, vy2, vz2,
           x3, y3, z3, vx3, vy3, vz3):
    N = int(tmax / dt)

    sustav = [[0 for x in range(19)] for x in range(N)]
    stanje = [0, x1, y1, z1, vx1, vy1, vz1,
              x2, y2, z2, vx2, vy2, vz2,
              x3, y3, z3, vx3, vy3, vz3]
    sustav[0] = stanje
    for i in range(1, N):
        for j in range(19):
            sustav[i][j] = stanje[j]
        stanje = RK4_Step(stanje, dt, jednadzbeGibanja)

    return sustav
```

```
def kineticka(stanje,m1,m2,m3,i):
    mase=[m1,m2,m3]
    if(i==0):
        brzina=[stanje[4],stanje[5],stanje[6]]
    elif(i==1):
        brzina=[stanje[10],stanje[11],stanje[12]]
    elif(i==2):
        brzina=[stanje[16],stanje[17],stanje[18]]
    kineticka = 0.5*(mase[i]*(brzina[0]**2+brzina[1]**2+brzina[2]**2))
    return kineticka

def potencijalna(stanje,m1,m2,m3,i):
    mase=[m1,m2,m3]
    r12=udaljenost([stanje[1],stanje[2],stanje[3]],
                   [stanje[7],stanje[8],stanje[9]])
    r13=udaljenost([stanje[1],stanje[2],stanje[3]],
                   [stanje[13],stanje[14],stanje[15]])
    r23=udaljenost([stanje[13],stanje[14],stanje[15]],
                   [stanje[7],stanje[8],stanje[9]])
    potencijalna=0.0
    if(i==0):
        potencijalna = G*((m1*m2/r12)+m1*m3/r13)
    elif(i==1):
        potencijalna = G*((m1*m2/r12)+m2*m3/r23)
    elif(i==2):
        potencijalna = G*((m1*m3/r13+m2*m3/r23))
```

```
    return -potencijalna

#######################

dt = 1
tmax = 10000
N = int(tmax / dt)
masa1,masa2,masa3=1.0,1.0,1000.0

G = 4.0 * (1.0 / (masa1 + masa2 + masa3)) * math.pi ** 2.0

start=time.time()

elipticneOrbite=[tmax, dt, masa1, masa2, masa3,
                 200, 200,200, 0, -0.2,0,
                 -300, 0,0, 0, -0.4,-0.1,
                 0, 0, 0, 0, 0, 0]

perturbacija=0.0001
'''sustav = Sustav(tmax, dt, masa1, masa2, masa3,
                200, 200,200,
                0+malaPerturbacija(perturbacija),
                -0.2+malaPerturbacija(perturbacija),
                0+malaPerturbacija(perturbacija),
                -300, 0,0,
                0+malaPerturbacija(perturbacija),
                -0.4+malaPerturbacija(perturbacija),
```

```
                -0.1+malaPerturbacija(perturbacija),
                0, 0, 0, 0, 0, 0)'''

sustav = Sustav(tmax, dt, masa1, masa2, masa3,
              200, 200,200, 0, -0.2,0,
              -300, 0,0, 0, -0.4,-0.1,
              0, 0, 0, 0, 0, 0)
t, x1, y1, z1, x2, y2, z2, x3, y3, z3 = [0],[],[],[],[],[],[],[],[],[]

kineticka1=[kineticka(sustav[0],masa1, masa2, masa3,1)]
potencijalna1=[potencijalna(sustav[0],masa1, masa2, masa3,1)]
kineticka2=[kineticka(sustav[0],masa1, masa2, masa3,1)]
potencijalna2=[potencijalna(sustav[0],masa1, masa2, masa3,1)]
kineticka3=[kineticka(sustav[0],masa1, masa2, masa3,1)]
potencijalna3=[potencijalna(sustav[0],masa1, masa2, masa3,1)]

for k in range(1,N):
    kineticka1.append(kineticka(sustav[k],masa1, masa2, masa3,0))
    potencijalna1.append(potencijalna(sustav[k],masa1, masa2, masa3,0))
    kineticka2.append(kineticka(sustav[k],masa1, masa2, masa3,1))
    potencijalna2.append(potencijalna(sustav[k],masa1, masa2, masa3,1))
    kineticka3.append(kineticka(sustav[k],masa1, masa2, masa3,2))
    potencijalna3.append(potencijalna(sustav[k],masa1, masa2, masa3,2))
    t.append(sustav[k][0])
    x1.append(sustav[k][1])
    y1.append(sustav[k][2])
    z1.append(sustav[k][3])
```

```
    x2.append(sustav[k][7])
    y2.append(sustav[k][8])
    z2.append(sustav[k][9])
    x3.append(sustav[k][13])
    y3.append(sustav[k][14])
    z3.append(sustav[k][15])

print(time.time()-start,"s")
#podaci=open('./podaci.txt', 'w+')
#print>>podaci,sustav

fig2 = plt.figure('Kineticka i potencijalna energija')
ax2 = fig2.add_subplot(311)
ax2.plot(t,kineticka1,label='kineticka energija mase 1')
ax2.plot(t,potencijalna1,label='potencijalna energija mase 1')
ax2.set_xlabel('t [s]')
ax2.set_ylabel('E(t) [J]')
plt.legend()
ax2 = fig2.add_subplot(312)
ax2.plot(t,kineticka2,label='kineticka energija mase 2')
ax2.plot(t,potencijalna2,label='potencijalna energija mase 2')
ax2.set_xlabel('t [s]')
ax2.set_ylabel('E(t) [J]')
plt.legend()
ax2 = fig2.add_subplot(313)
ax2.plot(t,kineticka3,label='kineticka energija mase 3')
ax2.plot(t,potencijalna3,label='potencijalna energija mase 3')
```

```
ax2.set_xlabel('t [s]')
ax2.set_ylabel('E(t) [J]')
plt.legend()

fig = plt.figure('Putanje')
ax = fig.gca(projection='3d')
ax.plot(x1,y1,z1, label='masa 1', color='b',linewidth=2,ls='-')
ax.plot(x2,y2,z2, label='masa 2', color='g',linewidth=2,ls='-')
ax.plot(x3,y3,z3, label='masa 3', color='r',linewidth=10,ls='-')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.text2D(0.05, 0.95, "Elipticne orbite", transform=ax.transAxes)
plt.legend()
plt.show()
```

Literatura

[1] W.E. Boyce and R.C. DiPrima. *Elementary Differential Equations and Boundary Value Problems*. Wiley, 2008.

[2] I. Griffiths, M. Adams, and J. Liberty. *Programming C# 4.0: Building Windows, Web, and RIA Applications for the .NET 4.0 Framework*. Animal Guide. O'Reilly Media, 2010.

[3] J.D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science Engineering*, 9(3):90–95, May 2007.

[4] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 2015-09-21].

[5] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C (2Nd Ed.): The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992.

[6] J.R. Taylor. *Classical Mechanics*. University Science Books, 2005.

[7] S. van der Walt, S.C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011.