TFY4235/FY8904 - Computational Physics - Exam

Erik Liodden, eriklio@stud.ntnu.no

July 17, 2018

Innledning

Jeg har valgt å skrive koden i Python 3, fordi det språket jeg har mest erfaring med. Jeg har skrevet Python-koden i en Jupyter Notebook slik at de ulike segmentene/ulike oppgave kommer tydelig frem i koden. 'Problem 1' i koden hører til 'Oppgave 1' her osv. Alle (eller de fleste) av oppgavene benytter seg av funksjoner jeg har definert helt i starten av koden. Metoder og funksjoner som tilhører en spesifikk oppgave er definert i starten av den oppgaven. All kode tilhørende eksamensbesvarelsen er i `eksamen_erik_liodden.ipynb`. Bibliotekene jeg har benyttet meg av er: `NumPy` for vektorer og matriser, `heapq` for prioritetskø, `matplotlib` and `seaborn` for plotting and `h5py` for lagre og laste datasett. Programmet lagrer figurer i mappen 'figs/' hvis `saveFig == True`, og programmet vil også lagre posisjonen til partiklene i 'veggen' i Oppgave 5 så dette kan brukes igjen senere.

Oppgaven tar for seg et system med partikler i en 2D-boks definert av domenet $[0, 1]^2$, altså avgrenset av veggene $x = 0$, $x = 1$, $y = 0$, $y = 1$. Partiklene er antatt å være harde 2-dimesjonale kuler. Oppgave 1 – 5 nedenfor ser på forskjellige egenskaper til dette systemet. I alle oppgavene er partiklene lagret som en matrise der rad i tilsvarer partikkel i. Hver rad er bygd opp på formen $[x, y, v_x, v_y, r, m, c, t_{\text{last}}]$ der x, y, v_x, v_y er posisjon og fart til partikkelen, r er radien, m er massen, c er kollisjonstelleren, og t_{last} er tidspunktet for partikkelens forrige kollisjon. Simuleringen er en 'event'-drevet type som beskrevet i eksamensoppgaven. Alternativt ville det vært en tidsdrevet simulering.

Oppgave 1

Koden til denne oppgaven er under 'Problem 1' i python-koden. Systemet ser på spredning etter en kollisjon mellom en stor og tung kule med radius $R = 0.1$ og masse $M = 10^6$ plassert i sentrum av boksen, og en liten lett kule med radius $r = 0.001$ og masse $m = 1$ som treffer den store med en hastighet i x-retning. Jeg satte 'impact parameteret' b (den lille kulas startavstand fra en linje gjennom sentrum til den store kula parallelt med x-aksen) til å variere fra $0 \to R + r - (r/100)$ over 1000 steg. Jeg trakk fra $r/100$ for å være sikker på at den lilla kula alltid ville treffe den store. En spredt vinkel $\theta = 0$ tilsvarer ingen spredning og $\theta = \pi$ tilsvarer refleksjon. Figur 1 viser spredt vinkel θ som funksjon av 'impact parameter' b.

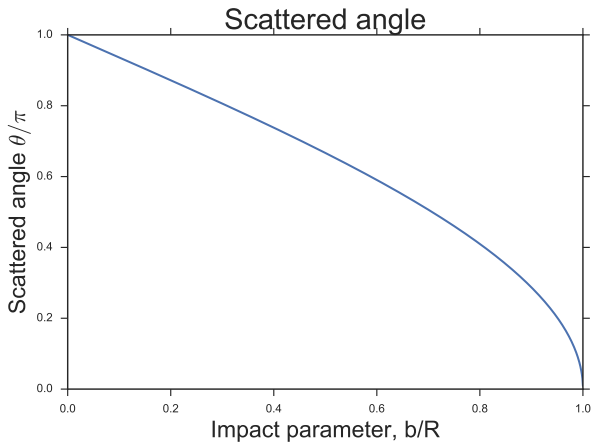


Figure 1: Spredning som funksjon av 'impact parameter' b/R.

Oppgave 2

Koden til denne oppgaven er under 'Problem 2' i python-koden. Systemet bedtår av 1000 partikler med masse $m = 1$ og radius $r = 0.01$. Jeg valgte denne størrelsen på kulene for å få pakketettheten ≈ 0.25, for dermed å prøve å gjøre andelen partikkel–partikkel kollisjoner stor. Partiklene ble uniformt tilfeldig plassert i boksen, og fikk startfarten $\mathbf{v} = [v_0 \cos\theta, v_0 \sin\theta]$ der $v_0 = 0.1$ og θ er en vinkel valgt uniformt tilfeldig mellom 0 og 2π. Jeg brukte `NumPy.random` for å velge ut tilfeldige tall. Startposisjonen til partiklene er vist i figur 2. Kollisjonene antas å være elastiske, altså med støtfaktor ('restitution coefficient') $\xi = 1$.

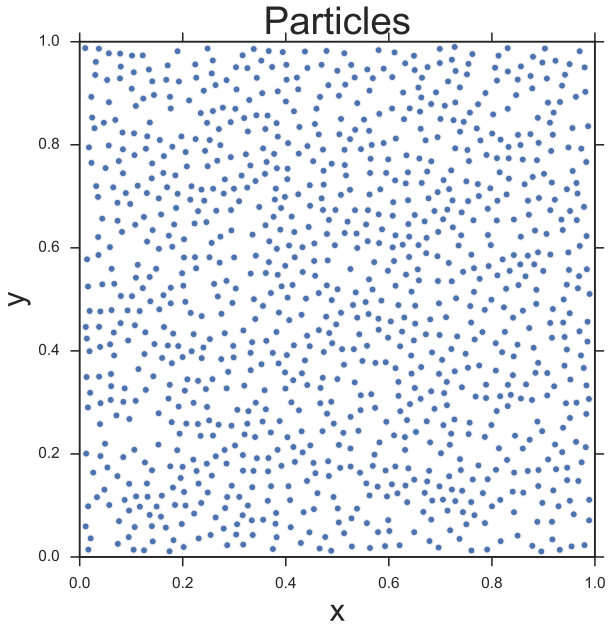


Figure 2: Startposisjon til partiklene. Antall = 1000

Jeg lot systemet utvikle seg til gjennomsnittlig antall kollisjoner per partikkel var ≈ 10. Jeg gjorde dette ved å

la simuleringen kjøre til totalt antall kollisjoner var $10\times$`antallPartikler`. For å forbedre statistikken forberedte jeg 10 individuelle systemer (en 'ensamble' bestående av 10 systemer), lot de utvikle seg hver for seg, for dermed å slå sammen resultatene til slutt. Initial og endelig fartfordeling til partiklene i boksen er vist i figur 3.

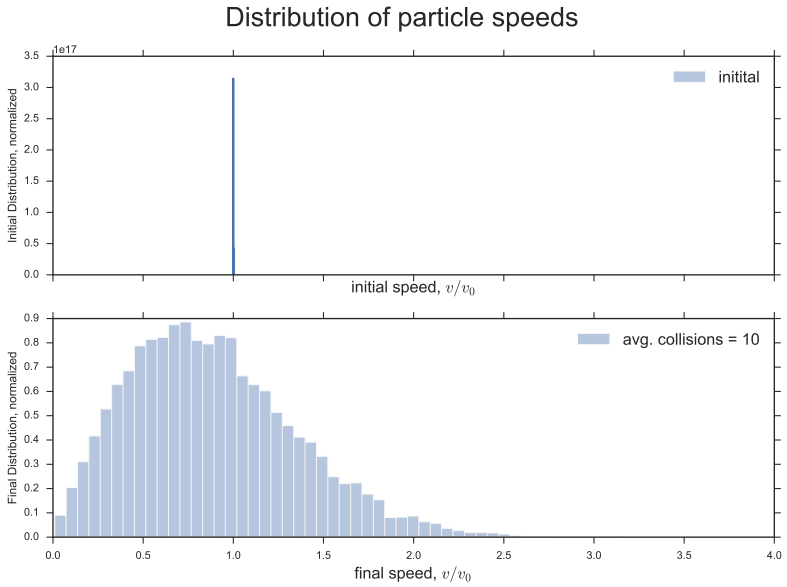


Figure 3: Distribusjon av fart på partiklene. Den øverste grafen viser distribusjonen i startøyeblikket, og den nederste viser distribusjonen etter ca. 10 kollisjoner per partikkel.

Fartsfordelingen til partiklene etter $\approx$ 10 kollisjoner per partikkel ser ut til å følge en Maxwell–Boltzmann distribusjon som man vil forvente.

Oppgave 3

Koden til denne oppgaven er under 'Problem 3' i python-koden. Systemet har et tilsvarende oppsett som i oppgave 2, men med den forskjellen at den ene halvparten av partiklene har masse $m = m_0$ og den andre halvparten har masse $m = 4m_0$. $m_0 = 1$ og alle partiklene har radius $r = 0.01$. Startfarten til partiklene ble satt på samme måte som i oppgave 2. Totalt er det 1000 partikler i systemet, og kollisjonene er elastiske, $\xi = 1$. Startposisjonen til partiklene er vist i figur 4.

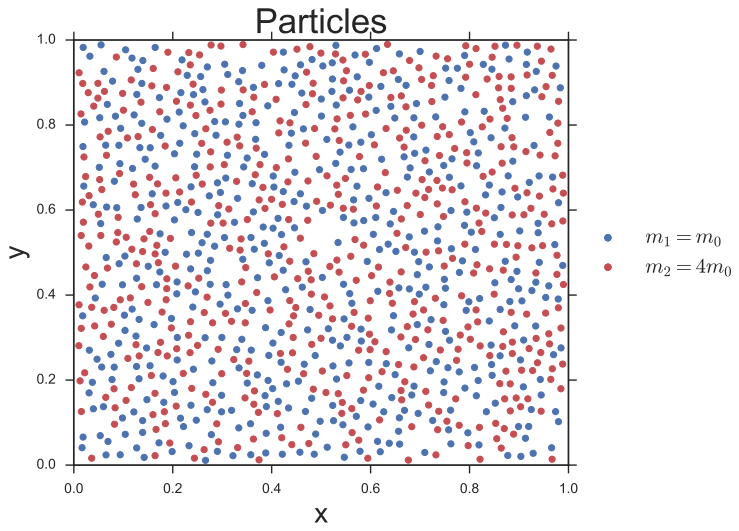


Figure 4: Startposisjon til partiklene. Antall = 1000. Blå partikler har masse $m = m_0$, rød partikler har masse $m = 4m_0$.

For å enkelt kunne skille mellom hvilke partikler som tilhørte hvilken halvdel, satte jeg $m = m_0$ på partiklene i den første (øvre) halvdelen av 'partikkel-matrisen' beskrevet i innledningen, og $m = 4m_0$ på partiklene i den andre (nedre) halvdelen. Siden partiklene er tilfeldig plassert i boksen, blir ikke dette et mindre generelt system. Som i oppgave 2 lot jeg systemet utvikle seg til gjennomsnittlig antall kollisjoner per partikkel var 10. Jeg gjorde dette ved å la simuleringen kjøre til totalt antall kollisjoner var $10 \times$ `antallPartikler`. Også her forbedret jeg statistikken ved å simulere 10 uavhengige systemer separat (en 'ensamble' av 10 systemer), for så å flette sammen resultatet til slutt. Figur 5 viser distribusjonen av fart for de to gassene til å begynne med og etter ≈ 10 kollisjoner. Gjennomsnittlig fart og kinetisk energi er printet i `eksamen_erik_liodden.ipynb` under oppgave 3.

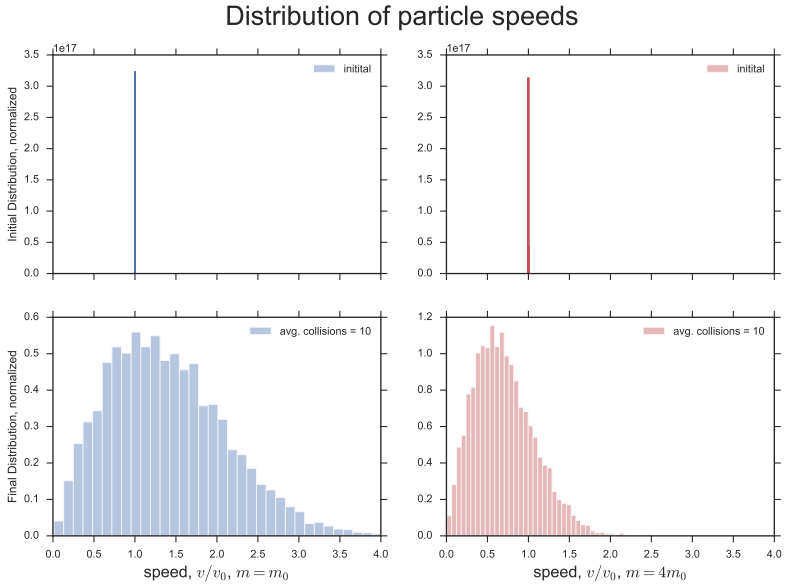


Figure 5: Distribusjon av fart på partiklene. Den øverste grafen viser distribusjonen i startøyeblikket, og den nederste viser distribusjonen etter ca. 10 kollisjoner per partikkel. Venstre kolonne er for $m = m_0$, høyre kolonne er for $m = 4m_0$.

Jeg ser av figur 5 at partiklene med lavere masse oppnår høyere gjennomsnittlig hastighet enn de med tyngre masse, noe som er å forvente ved elastiske støt. Som i oppgave 2 følger hastighetsfordelingen til de ulike partiklene en Maxwell–Boltzmann distribusjon.

Oppgave 4

Koden til denne oppgaven er under 'Problem 4' i python-koden. Oppsettet her er det samme som i oppgave 3. Totalt 1000 partikler, halvparten med masse $m = m_0 = 1$ og andre halvparten med masse $m = 4m_0$. Alle partiklene har startfart $\mathbf{v} = [v_0 \cos\theta, v_0 \sin\theta]$ der θ er en vinkel valgt uniformt tilfeldig mellom 0 og 2π og $v_0 = 0.1$. Ved ethvert støt ble gjennomsnittlig kinetisk energi over alle partikler med masse m_0 og $4m_0$ regnet ut hver for seg. Simuleringen gikk til det hadde vært gjennomsnittlig 15 kollisjoner per partikkel ved at jeg lot systemet utvikle seg til det totalt var utført $15\times$`antallPartikler` kollisjoner. Jeg gjorde simuleringen med tre ulike verdier for støtfaktoren ('restitution coefficient'), $\xi \in \{1, 0.9, 0.8\}$. Med $\xi < 1$ er det mulig at systemet opplever 'uelastisk kollaps' der det er uendelig mange kollisjoner som skjer i et lite tidsintervall. For å unngå dette problemet benyttet jeg meg av TC modellen som går ut på å anta et en kollisjon skal behandles elastisk dersom tiden siden forrige kollisjon er mindre enn en bestemt t_c. Siden 'uelastisk kollaps' ser ut til å gi tider mellom en kollisjon og den neste på størrelsesordenen tilsvarende presisjonen til variablene (10^{-15}) kan jeg sette t_c ganske lav. Jeg satte $t_c = 10^{-12}$.

Figur 6, 7 og 8 viser gjennomsnittlig energi for partikler med masse m_0 og $4m_0$, samt total energi relativt initial total energi per partikkel.

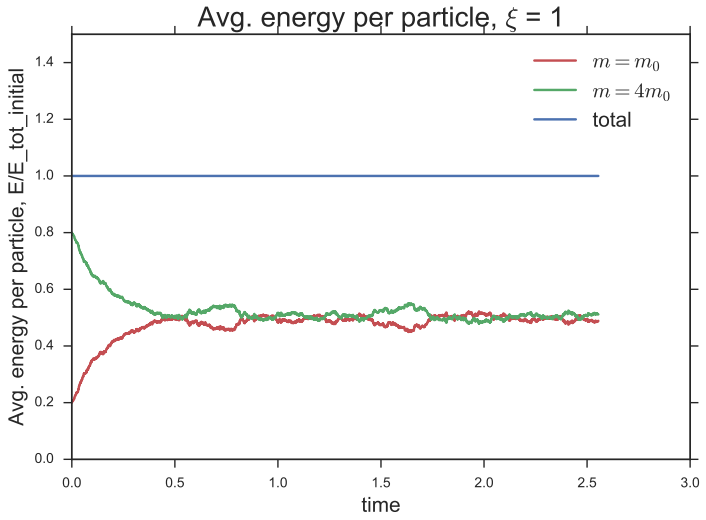


Figure 6: 1000 partikler, halvparten med masse $m = m_0$ og andre halvparten med masse $m = 4m_0$. $\xi = 1$, altså elastisk støt. Total energi bevart.

Man kan se av figur 6 at ved elastiske støt blir den gjennomsnittlige kinetiske energien per partikkel i de to gassene den samme som betyr at det er likevekt og de har dermed samme temperatur.

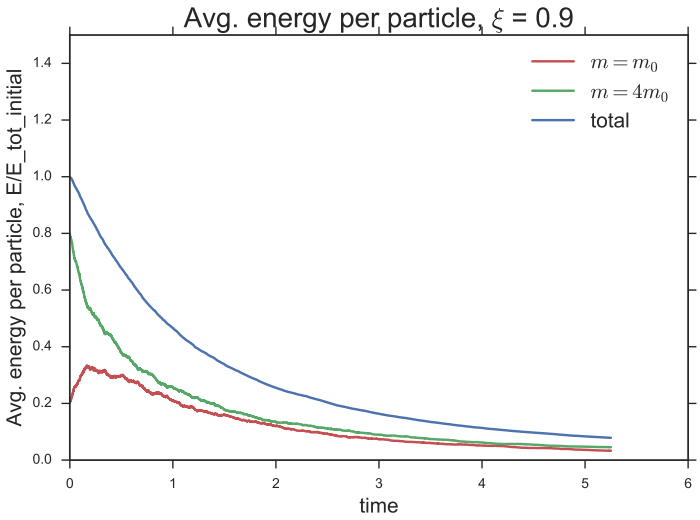


Figure 7: 1000 partikler, halvparten med masse $m = m_0$ og andre halvparten med masse $m = 4m_0$. $\xi = 0.9$, altså uelastisk støt. Total energi er ikke bevart og konvergerer mot 0.

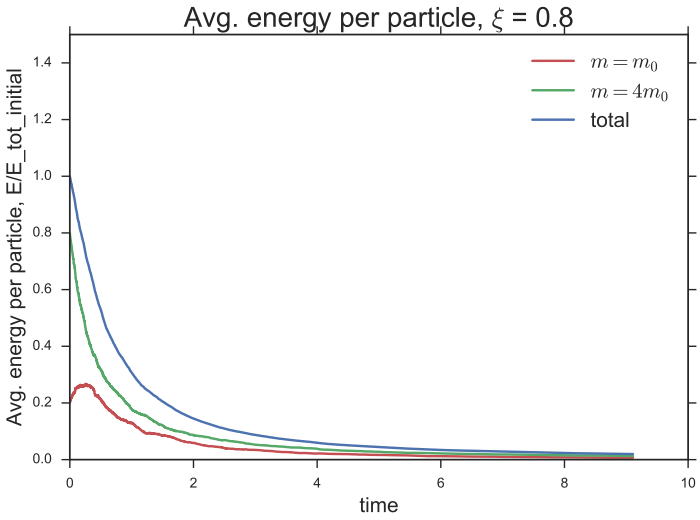


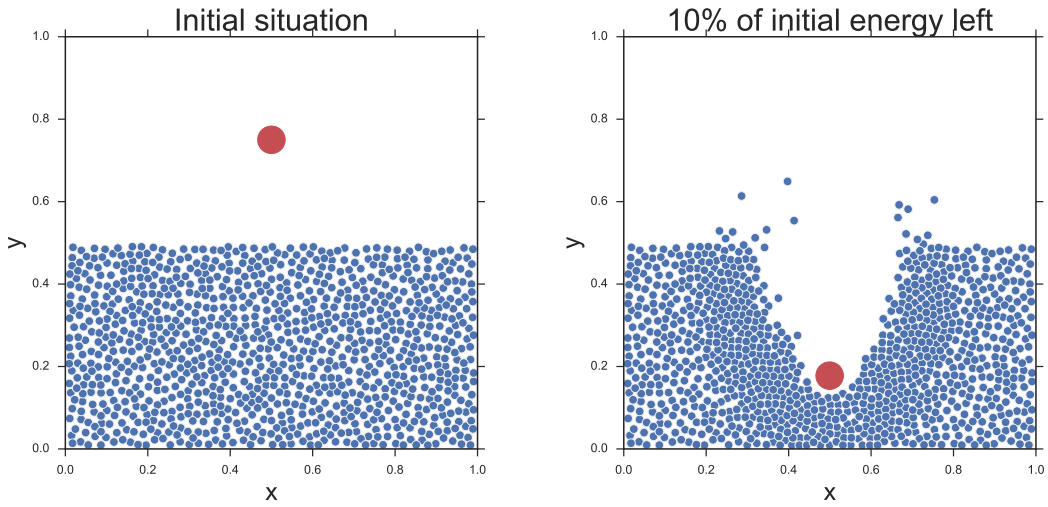
Figure 8: 1000 partikler, halvparten med masse $m = m_0$ og andre halvparten med masse $m = 4m_0$. $\xi = 0.8$, altså uelastisk støt. Total energi er ikke bevart og konvergerer mot 0.

Jeg ser av figur 8 og 7 viser at jo lavere støtfaktoren ('restitution coefficient') er, jo raskere konverterer energien mot 0, som betyr at partiklene kommer fortere til ro.

Oppgave 5

Koden til denne oppgaven er under 'Problem 5' i python-koden. Initialsystemet består av 1000 partikler i den nedre halvdelen av boksen med startfart 0. Et prosjektil er plassert i posisjon $\mathbf{x} = [0.5, 0.75]$ med en fart $\mathbf{v} = [0, -v_0]$ mot 'veggen' av partikler i nedre halvdel. Figur 9a viser dette oppsettet. Partiklene som danner 'veggen' har en radius $r = 0.009$ slik at pakketettheten er 0.508. t_c er satt til 10^{-6}. Algoritmen for å plassere partiklene gikk ut på

å velge en $x \in (r, 1-r)$ og $y \in (r, 0.5-r)$ uniformt tilfeldig. intervallet er valgt slik for å unngå at en partikkel blir plassert inne i en av de ytre veggene. Koordinatene x og y forkastes dersom avstanden til en annen partikkel som allerede er lagt er under $2r$. (Tilsvarende algoritme er brukt på oppgavene 2—4, men da er y trukket fra intervallet $(r, 1-r)$). Jeg brukte `NumPy.random` for å velge x og y. Støtfaktoren $\xi = 0.5$.



(a) Startsituasjon. Veggen består av 1000 partikler. Pakketetthet er 0.508.

(b) Situasjoen når 10% av startenergien er igjen.

Figure 9: Prosjektil med masse $M = 25m$, radius $R = 5r$ og startfart $\mathbf{v} = [0, -v_0]$ treffer en 'vegg' av 1000 stillestående partikler med masse m og radius r.

Jeg lar systemet utvikle seg til den totale energien er 10% av hva den var opprinnelig. For å finne ut hvor stort krateret har blitt teller jeg hvor mange partikler som har flyttet seg en avstand mer enn r fra opprinnelig startposisjon. Figur 9b viser situasjonen (posisjonen til partiklene) når 10% av startenergien er igjen.

For den siste delen av oppgaven valgte jeg å se på kraterstørrelse som funksjon av radius på prosjektilet. Jeg kjørte simuleringen med samme utgangspunkt, men endret radius på prosjektilet fra $r_{\min} = r$ (samme størrelse som partiklene i 'veggen') til $r_{\max} = 0.2$ (prosjektilet opptar nesten hele den øvre halvdelen av boksen) over 11 steg. Kraterstørrelsen ble regnet ut som over. Figur 10 viser kraterstørrelse som funksjon av relativ radius på prosjektilet.

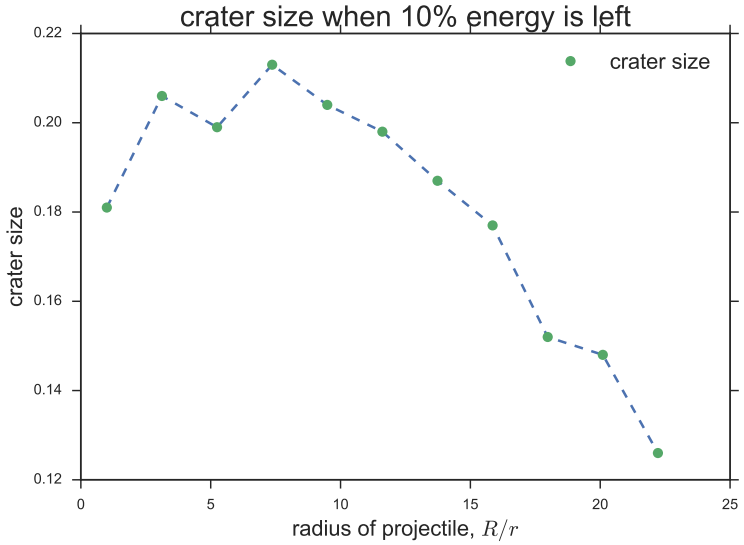
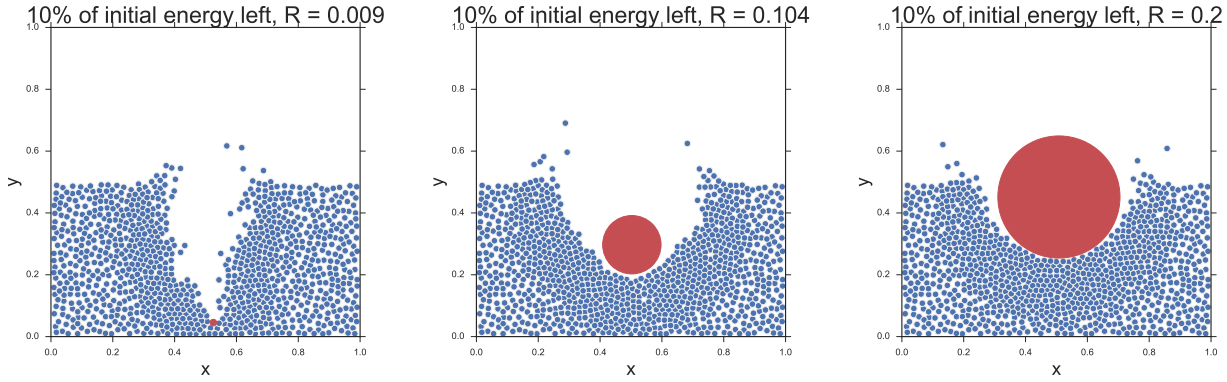


Figure 10: Kraterstørrelse som funksjon av relativ radius på prosjektilet, R/r, der r er radius til partiklene i veggen.

Figur 11 viser krater og prosjektil når 10% energi er igjen for tre utvalgte prosjektilstørrelser. Jeg ser av Figur 10 at det virker som kraterstørrelsen blir mindre jo større radien er med unntak av når radien er ca. like stor som partiklene i veggen. En mulig årsak er at når prosjektilet er såpass lite vil det lage et dypt krater (Figur 11a) og antall partikkel-'ytre vegg i boksen' (y = 0) kollisjoner begynner å øke. Dette gjør at prosjektilet raskt mister energi uten å kollidere med så mange andre partikler og krateret blir lite. Når prosjektilet er stort (Figur 11c) vil det kollidere med mange partikler tidlig og vil dermed ikke lage så stort krater før energien i systemer er 10% av hva det var.



(a) Minnste størrelse på prosjektil. Prosjektilet er like stort som de andre partiklene i 'veggen'. R = r.

(b) Medium størrelse på prosjektil. Radius på prosjektilet er 0.027

(c) Største størrelse på prosjektil. Radius på prosjektilet er 0.2

Figure 11: Krater dannet av prosjektiler med ulik størrelse.

Kommentarer

En kommentar angående optimalisering av koden: Jeg la merke til at kjøretiden ble betydelig raskere (fra størrelsesorden minutter til sekunder) når jeg benyttet meg (til en viss grad) av vektor-operasjoner fra `NumPy` ved utregning av tiden til neste kollisjon samt avstanden til de allerede plasserte partiklene i boksen. Bruk av `heapq` for å legge til og trekke ut elementer fra prioritetskøen gjør at denne delen av koden fikk kjøretid $\mathcal{O}(\log n)$, som er mye raskere enn hva jeg ville klart å implementere selv. Ved simuleringer av fysiske systemer inngår det mange utregninger, og jeg ser definitivt nytten av å kunne ta i bruk optimaliserte biblioteker for å utføre spesifikke deler av koden.

Acknowledgements

Jeg har diskutert resultatene og plottene med Stian Hartman.