Simulering av en Trebuchet
TNM085

Samuel Svensson | samsv787
Alexander Bergman | alebe181
Carl Håkansson | carha719
Marcus Johansson | marjo561

27 augusti 2019

Sammanfattning

Denna rapport redogör för hur Euler-approximering kan användas för att lösa differentialekvationer för ett fysiskt system. Systemet i detta fall är en trebuchet som avfyrar en projektil genom att en motvikt släpps från andra sidan av hävarmen vilket genererar ett vridmoment. Trebucheten användes så tidigt som 300-talet före Kristus för att belägra fort och var en av de mest förödande belägringsvapnen under denna tidsperiod. I kursen TNM085 - Modelleringsprojekt simulerade projektgruppen systemet i datorprogrammet MATLAB. Samma simulering gjordes sedan med mer avancerad grafik i biblioteket Three.JS och programspråket Javascript.

Innehåll

Figurer

Kapitel 1

Inledning

Redan på 300-talet före Kristus utvecklades i forna Kina avancerade belägringsvapen varav ett av de mest effektiva var inspirationen till det här projektet. En trebuchet (även kallad blida) var en form av katapult med potential att slunga tyngre projektiler över större avstånd. Den spreds senare mot Europa och användes av det Bysantinska riket under 500-talet. Modellen som det Kinesiska och Bysantinska riket använde fungerade genom att män drog i rep som var fästa vid ena änden. Det var först på medeltiden som den mer moderna trebucheten med en motvikt introducerades. Trebucheten kom att användas fram till slutet av 1400-talet och ersattes av kanoner allteftersom krut blev mer tillgängligt [1].

I takt med att datorers beräkningskraft och tillgänglighet ökat under de senaste årtiondena har det blivit allt lättare att genomföra de svåra beräkningar som krävs för att simulera komplexa system. Alternativet innan detta var att konstruera en fysisk modell för testning av systemet, något som var oerhört tidskrävande. Systemen beskrivs med hjälp av differentialekvationer uttryckta i fysikaliska storheter. Genom att lösa differentialekvationerna med numeriska metoder fås ett approximerat resultat [2].

1.1 Syfte

Syftet med projektet var att med hjälp av beräkningar realistiskt simulera och animera en projektil som avfyrats från en trebuchet. All beräkning och simulering gjordes i beräkningsprogrammet *MATLAB* och sedan utvecklades en webbapplikation i Javascript-biblioteket *Three.js* som utgjorde den grafiska implementationen av animationen.

1.2 Avgränsningar

Ett antal begränsningar har gjorts av trebuchetens fysiska aspekter. För att förenkla beräkningarna avsevärt är både motvikten och projektilen fästa direkt i kastarmen, istället för att ha en extra slunga de hänger i. Kastarmen antas även vara helt masslös. Slutligen simuleras inte den påverkan vind skulle ha på projektilens bana, luftmotstånd har emellertid tagits hänsyn till.

Kapitel 2

Systembeskrivning

2.1 Beskrivning av systemet

För att underlätta arbetet så delades systemet upp i två delsystem. Det första systemet är kastarmens rotation, då vridmoment genereras av motvikten när den släpps från ena sidan av hävarmen. Det andra när projektilen färdas genom luften med hjälp av vridmomentet från det första systemet.

2.1.1 Rotationsrörelse

Systemets rotationsrörelse påverkas bland annat av vridmomentet. Ett av systemets två delsystem är kastarmens rotation. I kastarmens ena ände finns motvikten, vars tyngdkraft verkar på kastarmen. Enligt hävstångsprincipen skapar detta ett vridmoment som får kastarmen att rotera [3](p.8). Projektilen finns i kastarmens andra ände, och på samma sätt skapar den ett vridmoment [4]. Då motvikten är mycket tyngre än projektilen gör det resulterande vridmomentet att kastarmen roterar medsols, enligt figur **??**.

Vridmomentet beräknas genom ekvation 2.1.

$$\tau(t) = \boldsymbol{LI} \cdot F(t) \cdot sin(\theta(t)) \tag{2.1}$$

där $L1$ är avståndet mellan rotationspunkten och motvikten, $F(t)$ är motviktens tyngd och $\theta(t)$ är vinkeln mellan denna kraft och riktningen som motvikten rör sig. Tyngdkraften nämnd ovan ges av Newtons andra lag och visas i 2.2.

$$F(t) = ma(t) \tag{2.2}$$

där m är massans vikt och $a(t)$ är gravitationskonstanten g. Vidare så kan hastigheten som den andra änden på hävarmen beräknas med

$$v(t) = L2 \cdot \omega(t) \tag{2.3}$$

Där $L2$ är den ände på hävarmen som projektilen sitter på och $\omega(t)$ är vinkelhastigheten.

Kastarmens rotation påverkas också av dess tröghetsmoment. Kastarmens tröghetsmoment innebär det vridmoment som krävs för att uppnå en viss ändring av dess rotationshastighet per tidsenhet kring dess rotationsaxel [4]. Ekvationen för kastarmens tröghetsmoment med de två vikterna ges i 2.4.

$$I = \sum_{i=1}^{2} m_i \cdot L_i^2 \tag{2.4}$$

där m_i är projektilen samt motviktens massor och L_i är avstånden från rotationsaxeln till projektil och motvikt. Dessa ekvationer kan sedan användas för att beskriva den ordinära differentialekvationen för vinkelaccelerationen $\alpha(t)$ av kastarmens rotation. Differentialekvationen visas i Figur 2.5.

$$\alpha(t) = \frac{((m1 \cdot (L1)^2 + m2 \cdot (L2)^2) \cdot sin(\theta(t)) \cdot (-g) - b \cdot \omega(t))}{I} \tag{2.5}$$

där $\alpha(t)$ är vinkelaccelerationen, $m1$ är motviktens vikt, L1 är avståndet från motvikten till rotationsaxeln, $m2$ är projektilens vikt och L2 är avståndet från projektilen till rotationsaxeln. Vidare är $\theta(t)$ kastarmens vinkel, g är tyngdaccelerationen, b är friktionskonstanten mellan kastarmen och trebuchetens ram, $\omega(t)$ är vinkelhastigheten och I är kastarmens totala tröghetsmoment.

2.1.2 Projektilbana

När motvikten släpps och vridmomentet genereras kommer projektilen vid andra sidan av hävarmen röra sig i en cirkulär rörelse uppåt. När rotationen uppnår en förbestämd vinkel $\theta = \frac{\pi}{4}$ så kommer projektilen att lämna kastarmen, för att sedan börja färdas genom luften med en acceleration som beskrivs av differentialekvationen 2.6.

$$\begin{cases} a_x = \frac{-F_{d_x}}{m_2} \\ a_y = \frac{-g-F_{d_y}}{m_2} \end{cases} \tag{2.6}$$

Under tiden som projektilen färdas så påverkas den av ett varierande luftmotstånd beroende på projektilens egenskaper [5]. Luftmotståndet $F_d(t)$ ges av 2.7.

$$F_d(t) = \frac{\rho v^2(t) C_d A}{2} \tag{2.7}$$

Där ρ är luftens densitet, $v(t)$ är hastigheten i varje kraftkomponent, C_d är en luftmotståndskoefficient och A är projektilens tvärsnittsarea. I detta fall baserades beräkningarna på en jämn sfär vilket innebär att tvärsnittsarean är en cirkel.

2.2 Eulers stegmetod

För att numeriskt beräkna simuleringen användes Eulers stegmetod då systemet beskrevs av en *ordinär differentialekvation* [2]. Med givna initialvärden delas differentialekvationen upp i stegintervall och därefter approximeras lösningen. Differentialekvationen $\dot{x} = f(t, x)$ kan beskrivas i diskret tidsform enligt .

$$x_{n+1} = x_n + h \cdot f(t_n, x_n) \tag{2.8}$$

Där x_{k+1} representerar nästa stegintervall i lösningen och baseras på informationen från det tidigare steget. Steglängden h innefattar längden av tidsintervallet mellan beräkningar. Funktionen $f(t_k, x_k)$ beräknas varje steg (vid en viss tidpunkt t_k och för ett visst värde x_k) för att sedan multipliceras med h. Vidare görs approximering vid funktionens tangent i varje punkt som sedan används för att beräkna nästa punkt genom att följa tangentens riktning. Denna stegmetod är en så kallad *enstegsmetod* vilket innebär att beräkningen baseras på informationen från föregående steg.

2.2.1 Applicering av stegmetod

Med de givna differentialekvationerna 2.5 och 2.6 kan Eulers stegmetod användas för att numeriskt beräkna samt approximera differentialekvationen för både rotationen av kastarmen och projektilens färdbana. Stegberäkningen baseras som tidigare nämnt i Kapitel 2.2 endast på informationen som beräknades vid det tidigare steget h_{n-1}. Differential ekvationerna är definierade som tidsberoende men för att kunna göra den numeriska approximeringen så beskrivs differentialekvationen i diskret form. De fysikaliska krafterna som påverkar trebucheten och som genereras vid rotation kan ses som tidsberoende. För att kunna beräkna dessa krafter och approximera en lösning (då steget är nära $t = t_0$) med en dator krävs det att den beskrivs i sin diskreta form.

Implementeringen av Eulers stegmetod för dessa differentialekvationer utfördes först i datorprogrammet MATLAB och sedan med programspråket Javascript. Utdrag från kod visas i Appendix 5.3.

Kapitel 3

Implementation av modellen

3.1 Simulering av systemet

Initialt genomfördes simuleringen av systemet i programvaran *MATLAB*. Innan utvecklingen av koden startades, sammanställdes alla nödvändiga formler från 2.1.1 och 2.1.2. Dessa kombinerades sedan för att få fram uttryck för de variabler som behövdes för att beskriva systemet. För att kunna simulera kaströrelsen av projektilen korrekt behövdes en utgångsposition och utgångshastighet. För att kunna räkna ut dessa behövdes kastarmens vinkel och vinkelhastighet i kastögonblicket, och dessa kunde i sin tur finnas genom att använda Eulers stegmetod i kombination med 2.5 för kastarmens vinkelacceleration.

Vid approximering av den ordinära differentialekvation så används Eulers stegmetod. Först beräknas den momentana vinkelaccelerationen av kastarmens rotation (från en förbestämd startvinkel) med 2.5. Informationen som fås av denna beräkning kan användas för att numeriskt beräkna momentana vinkelhastigheten och till slut även kastarmens vinkel också. Dessa beräkningar görs som tidigare nämnt i 2.2 för varje steg i approximeringen. De tidigare värdena på vinkelhastigheten och position används för att numeriskt beräkna vinkelaccelerationen för nästa steg (baserat på Eulers stegmetod). Detta implementeras som tidigare nämnt med hjälp av programkod i MATLAB. Hela simuleringen itereras igenom så länge projektilen är i luften. Då kastarmen når en fördefinierad vinkel släpps projektilen, och i detta ögonblick beräknas dess utgångsposition och utgångshastighet. Dessa värden kan sedan användas för att simulera projektilens bana. En mer generell förklaring av koden beskrivs i pseudokod i 3.1. Programkoden som användes för simuleringen kan ses i Appendix 5.3.

Algorithm 1 Calculate $\dot{x} = f(t(t), x(t))$

for $t = 0 : h : t_{final}$ **do**
  $a_{n+1} = f(t_n, o_n)$
  $o_{n+1} = o_n + h \cdot a_{n+1}$
  $t_{n+1} = t_n + h \cdot o_{n+1}$
  $x = sin(t_{n+1}) \cdot lengthofarm$
end for
if $t_n = \theta$ **and** $isThrown = 0$ **then**
  $x1 = x$
  $y1 = y$
  $isThrown = 1$
end if
for $t = 0 : h : t_{final}$ **do**
  $ax_{n+1} = g(n)$
  $ay_{n+1} = g(n)$
  $vx_{n+1} = vx_n + h \cdot ax_{n+1}$
  $vy_{n+1} = vy_n + h \cdot ay_{n+1}$
  $x_{n+1} = x_n + h \cdot vx_{n+1}$
  $y_{n+1} = y_n + h \cdot vy_{n+1}$
end for

Figur 3.1: Algoritmen som användes i simuleringen

Där a_{n+1} och liknande resulterande variabler representerar de nya beräknade värdena från det tidigare steget.

3.2 Animation av systemet

Vid animeringen för systemet så användes webb-biblioteket *Three.js* som är skrivet i programspråket *Javascript* och API:et *WebGL*. Three.js möjliggör 3D-rendering direkt i webbläsaren. [6].

Programkoden är strukturerad genom tre större filer som fyller olika funktioner. *Main.js* initierar scenen och kallar på funktioner från de andra filerna. *Loadmodels.js* ser till att alla 3D-modeller laddas in i scenen på ett korrekt sätt. Här specificeras även objektens positioner, storlekar och texturer. Trebucheten är uppdelad i tre delar (kropp, arm och motvikt) för att förenkla animeringen. För att öka prestandan i animeringen av systemet används en *GLTF-Loader*[7], vilket är ett verktyg som används för att importera modeller i filformatet .glb. Detta filformat är en del av specifikationen *glTF (GL Transmission Format)* som används för att få en effektiv överföring och laddning av 3D-scener samt modeller i en applikation. Specifikationen glTF minimerar både storleken på 3D-tillgångar samt behandlingen som behövs vid körning för att packa upp och använda dessa tillgångar [8]. Specifikationen används till exempel av Facebook för effektivisering vid laddning av 3D-modeller.

Filen *renderFunctions.js* innehåller majoriteten av den konverterade programkoden som skrevs i MATLAB där beräkningar av projektilens koordinater i varje bilduppdatering. En numerisk approximation beräknas genom Eulers stegmetod, precis som i MATLAB, för både rotationsrörelsen och projektilens rörelse när den färdas under simuleringen.

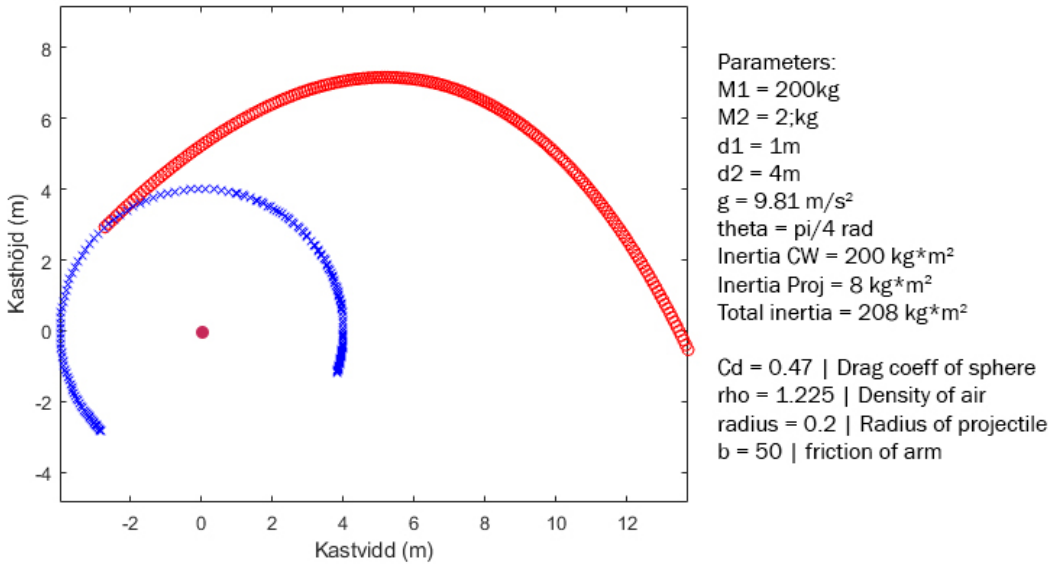
Utöver dessa filer finns även en mindre fil som hanterar ljuskällors positioner och egenskaper runt om

i scenen. Miljön runt trebucheten består av 3D-modeller som är skapade i modelleringsprogrammet *Blender* med tillhörande texturer.

Kapitel 4

Resultat

Från ekvationerna i 2.1.1 och 2.1.2 beräknades projektilens rörelse med hjälp av datorprogrammet MATLAB. Både rörelsen när den roteras med kastarmen och rörelsen den får efter den har lämnat trebucheten simulerades. Dessa rörelser visas i Figur 4.1 med olika färger.



Figur 4.1: Graf som visar projektilens kastbana i MATLAB. Blå bana redogör projektilens rörelse under rotation och röd bana är projektilens translation

Animationerna för systemet implementerades med hjälp av biblioteket Three.JS och programspråket Javascript. Det färdigställda resultatet kan ses i Figur 4.3. Simuleringen består av trebucheten, en bana för projektilen att landa på och en interaktiv användarmeny där användaren kan ändra några av simuleringens parametrar. Dessa är projektilens och motviktens massa, kastarmens längd samt luftmotståndskoefficienten. Användaren kan även välja att följa efter projektilen med kameran för att tydligare se dess kastbana.

När användaren klickar på *Launch* startas animationen och trebucheten slungar iväg projektilen. Projektilens färd beräknas enligt tidigare givna metoder approximativt under simuleringens gång. När projektilen landar så markeras dess position visuellt i form av tydligt uppmärkta ringar. Två ljud har lagts till för att ge ytterligare återkoppling; ett när trebucheten avfyrar projektilen och ett som spelas när projektilen kolliderar med marken. Dessa signalerar för användaren att simuleringen har börjat respektive slutat.

Figur 4.2: Det färdiga resultatet med trebucheten placerad i miljön

Figur 4.3: En närbild på trebucheten

Kapitel 5

Diskussion och slutsatser

5.1 Simulering och animering

I och med den begränsade projekttiden behövdes vissa avgränsningar göras. En av dessa var att slungan som traditionellt är kopplad till änden av kastarmen avlägsnades från beräkningarna. Att ha ett rep som rör sig fritt vid avfyrning innebär att beräkningar av spänning och trådkraft måste tas hänsyn till, vilket hade försvårat arbetet.

För att göra en mer noggrann approximation skulle den numeriska metoden Runge-Kutta kunna ha använts. Detta skulle däremot försämrat prestandan i simuleringen och animationen på grund av att fler beräkningar krävs. Ett alternativ till numeriska metoder skulle ha varit Lagranges ekvationer för att bestämma rörelsen i det mekaniska systemet men med projektgruppens begränsade kunskaper inom detta område så valdes Eulers stegmetod istället.

5.2 Vidareutveckling

Det finns flera potentiella förbättringar med både simuleringen och animationen. Som tidigare nämnts i 5.1 skulle en slunga med ett korrekt fysikaliskt beteende kunna ha implementerats. Vidare skulle vind som påverkar projektilen kunnat simuleras för att öka realismen. Detta skulle resulterat i en mer verklighetstrogen simulering men även mer beräkningstung. På grund av den korta projekttiden tvingades gruppen att begränsa komplexiteten bland de matematiska sambanden och göra flera avgränsningar.

Vidare skulle kastarmens rotation kunnat ha animerats på ett mer verklighetstroget sätt där friktion mellan komponenter tas hänsyn till. Simuleringstiden skulle också kunna förlängas för att till slut låta kastarmen nå sitt jämviktsläge i upprätt tillstånd.

En ytterligare förbättring av den visuella återkopplingen skulle ha varit att visa en tydlig text på antal meter som projektilen färdats när den landat. Användaren hade då kunnat jämföra vilka kastlängder som kan uppnås med olika kombinationer av parametervärden.

5.3 Slutsats

Projektgruppen lyckats utveckla en fungerande webbapplikation som simulerar avfyrningen av en trebuchet med hjälp av numeriska metoder.

Litteraturförteckning

[1] Shawn Rutan och Becky Wieczorek, *Modern Siege Weapons: Mechanics of the Trebuchet*, 2005, hämtad: 2019-03-14

[2] Brian Bradie, *A Friendly Introduction to Numerical Analysis*, Upper Saddle River, New Jersey: Pearson Prentice Hall, 2006, hämtad: 2019-03-14

[3] Donald B. Siano, *Trebuchet Mechanics - The Algorithmic Beauty of the Trebuchet*, Upper Saddle River, New Jersey: Pearson Prentice Hall, 2013-11-16, hämtad: 2019-03-14
http://www.algobeautytreb.com/trebmath356.pdf

[4] Randy Kobes, *Relation Between Torque and Angular Acceleration*, 1997-10-09, hämtad: 2019-03-14
https://theory.uwinnipeg.ca/physics/rot/node5.html-SECTION00840000000000000000

[5] Matthew West, *Projectiles with air resistance*, 2015, hämtad: 2019-03-14
http://dynref.engr.illinois.edu/afp.html

[6] mrdoob, *Three.js Loader*, 2019, hämtad: 2019-03-14
https://threejs.org/docs/index.html-api/en/loaders/Loader

[7] mrdoob, *Three.js GLTF-Loader*, 2019, hämtad: 2019-03-14
https://threejs.org/docs/index.html-examples/loaders/GLTFLoader

[8] The Khronos® Group Inc, *glTF Overview*, 2019, hämtad: 2019-03-14
https://www.khronos.org/gltf/

Appendix

```
for t=0:delta_t:2
    %ROTATION
    %a1 = Angular acceleration
    a1 = (((m1*d1) - (m2*d2))*sin(t1)*-g-(b*o1))/Itot;

    %o1 = Angular velocity
    o1 = o1+(delta_t * a1);

    %t1 = Angle position
    t1 = t1 + (delta_t * o1);

    x = sin(t1)*d2; %Convert Polar -> Cartesian
    y = cos(t1)*d2; %Convert Polar -> Cartesian



    %THROW EVENT
    if t1==theta && pThrown == 0
        x1=x;
        y1=y;
        pThrown = 1;

    end
    %PROJECTILE PATH
    %Projectile acceleration w/ air resistance
    ax = -(Fdrag_x)/m2;
    ay = -g-(Fdrag_y)/m2;

    %Projectile velocity
    vx = vx + delta_t * ax;
    vy = vy + delta_t * ay;

    %Projectile position
    x1 = x1 + delta_t * vx;
    y1 = y1 + delta_t * vy;

    xlabel('Range');
    ylabel('Height');

    plot(x,y,'xb',x1,y1, 'or');
    axis equal;
    hold on;
end
```

Figur 1: En del av implementationen i MATLAB.

```
//Do simulation loop while projectile is above ground (ground: y = 0)
while (y[i] >= 0)
{
    //Rotation
    // Torque / Moment of inertia = Angular acceleration
    aa = (((m1*d1) - (m2*d2))*Math.sin(tp)*-g-(b*ov))/Itot;
    // Euler (angular acc -> angular velocity)
    ov = ov + (delta_t * aa);
    // Euler (angular velocity -> angular position)
    tp = tp + (delta_t * ov);
    xR = Math.sin(tp)*d2; // Convert Polar -> Cartesian
    yR = Math.cos(tp)*d2; // Convert Polar -> Cartesian

    if (tp==theta){
        x=xR;
        y=yR;
    }
    // Drag force
    Fdrag_x = (rho * C * A * vx^2)/2;
    Fdrag_y = (rho * C * A * vy^2)/2;

    // Trajectory
    // Acceleration with air drag
    ax = - Fdrag_x * vx;
    ay = - g - Fdrag_y * vy;

    // Euler (acc -> velocity)
    vx = vx + delta_t * ax;
    vy = vy + delta_t * ay;

    // Euler (velocity -> position)
    x[i+1] = x[i] + delta_t * vx;
    y[i+1] = y[i] + delta_t * vy;

    i++;
}
```

Figur 2: En del av implementationen i Javascript.