# Gruppeopgave 2:
# Modellering og Simulering af Fysiske Systemer

Kenny Erleben Sarah Niebe

3. december 2010

## Introduktion

I denne uge vil I blive introduceret til en række værktøjer og arbejdsmetoder der benyttes når man modellerer og simulerer fysiske systemer. Til opgave 2 og 3 skal der implementeres en (lille) række funktioner. Vi anbefaler at I implementerer disse i MatLab, da alle eksempelfiler er givet i dette format. MatLab kan hentes på KUnet og installeres via KUs licens. Ugen vil byde på følgende:

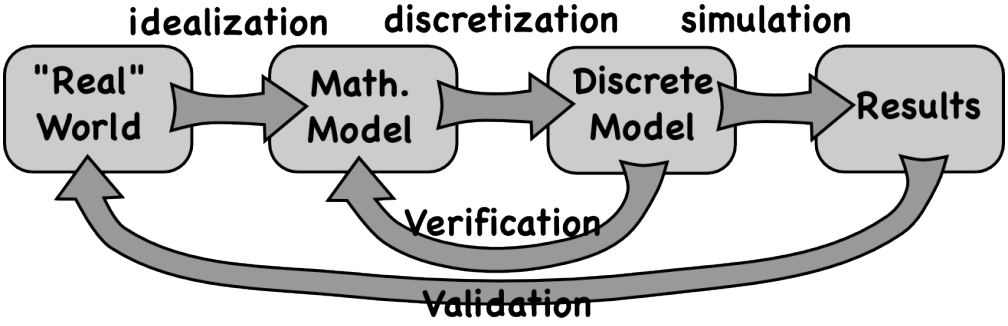
- Koncepter og terminologi omkring computereksperimenter vil blive berørt
- Forskel på en data-drevet og metode-drevet modelleringsprocess vil blive forklaret
- Modellerings- og simuleringsprocesser vil blive beskrevet vha. flow-diagrammer
- Der vil blive givet eksempler på verificering og validering
- I vil skulle læse en videnskabelig artikel og vurdere det videnskabelige arbejde i denne
- I vil implementere jeres egen simulator som skal verificeres og valideres
- I vil implementere en kant-finder til at analysere billeder som skal verificeres og valideres

## Opgave 1: Analyse af modellerings og simulerings processer

I denne opgave skal I anvende data-flow diagrammet fra Figur 1, som et redskab til at analysere to tekster.

### 1.a) K. Erleben - Simulering og modellering af robotter og mennesker

- Identificér hvilke bokse (modeller/data) og pile (processer) fra diagrammet som artiklen behandler, samt hvilke artiklen ikke kommer nærmere ind på.




Figur 1: Data-flow diagram.

- Overvej over hvordan I selv ville håndtere de processer (pile) som artiklen ikke omhandler, beskriv og begrund jeres bud på hvordan I ville håndtere disse processer.

### 1.b) Hsu og Keyser - Piles of Objects

I kan med fordel læse artiklen "How to read a paper" af S. Keshav før I læser "Piles of objects".

- Beskriv resultatet af jeres analyse af artiklen (Bemærk at denne artikel dækker alle bokse og pile fra flow-diagrammet.)
- På baggrund af jeres resultater overvej om I syntes det videnskabelige arbejde i artiklen er i orden, begrund jeres vurdering og giv eventuelt eksempler til forbedringer, hvis I mener der er nogen processer som ikke er i orden.

## Opgave 2: Bold simulator

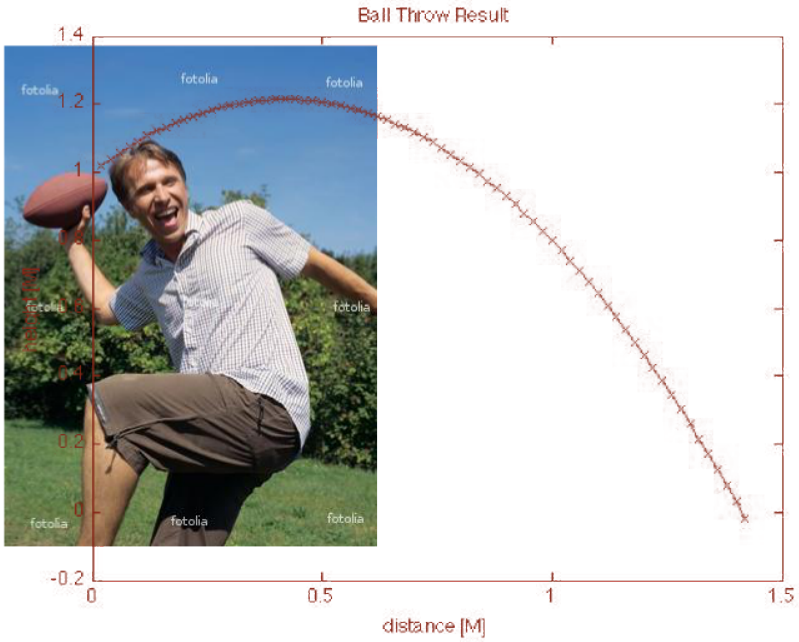
I denne opgave skal der implementeres en simulator som kan simulere bold kast. Vi laver følgende idealiserende og simplificerende antagelser:

- Den eneste udefra kommende påvirkning af bolden er kastet og tyngdeaccelerationen; vi ser altså bort fra påvirkninger fra eksempelvis vindmodstand
- Bolden er symmetrisk rund og af en størrelse som er lille sammenlignet med den afstand man kan kaste bolden, vi kan derfor antage at bolden kan beskrives som en punkt partikel med en given masse

Med disse antagelser kan vi beskrive banekurven i et 2D plan frem for et 3D rum.

Vi vælger en x-akse i planet som er horizontal med jordens overfalde og en y-akse til angive boldens højde over jorden. Boldens position angives ved koordinatsættet $(x, y)$ som en funktion af tiden, boldens hastighed er angivet ved vektoren $(v_x, v_y)$ som ligeledes

er en funktion af tiden, se Figur 2. Til tiden $t = 0$, når kasteren slipper bolden, så er boldens position givet ved $(x_0, y_0)$ og hastigheden ved $(v_{x0}, v_{y0})$.




Figur 2: Illustration af en banekurve i et 2D plan, bemærk at det er boldens position der er angivet.

Vi anvender klassisk mekanik, hvorfra vi kan beskrive boldens bevægelse ved hjælp af Newtons 2. lov:

$$m\frac{d^2x}{dt^2} = 0 \tag{1a}$$
$$m\frac{d^2y}{dt^2} = -g \tag{1b}$$

hvor $m$ er boldens masse og $g$ er størrelsen af tyngdeaccelerationen. Brøken $\frac{d^2x}{dt^2}$ er den anden afledede af $x$ i forhold til $t$, ligeledes er $\frac{d^2y}{dt^2}$ den anden afledede af $y$ i forhold til $t$. Da $x$ og $y$ er positioner er deres anden afledede derfor accelerationen. Den første afledte af $x$ of $y$ er dermed hastigheden, det skrives som:

$$\frac{dx}{dt} = v_x \tag{2a}$$
$$\frac{dy}{dt} = v_y \tag{2b}$$

Ved indsættelse i Newtons anden lov (Ligning 1) får vi:

$$
\begin{aligned}
m\frac{dv_x}{dt} &= 0 && \text{(3a)}\\
m\frac{dv_y}{dt} &= -g && \text{(3b)}\\
\frac{dx}{dt} &= v_x && \text{(3c)}\\
\frac{dy}{dt} &= v_y && \text{(3d)}
\end{aligned}
$$

med startbetingelserne

$$
\begin{aligned}
x(0) &= x_0 && \text{(4a)}\\
y(0) &= y_0 > 0 && \text{(4b)}\\
v_x(0) &= v_{x_0} && \text{(4c)}\\
v_y(0) &= v_{y_0} && \text{(4d)}
\end{aligned}
$$

Denne matematiske model (first order ordinary differential equation) kan nu diskretiseres ved at approksimere de tidsafledede ved hjælp af en finite difference metode:

$$
\begin{aligned}
\frac{dv_x}{dt} &\approx \frac{v_x(t+\Delta t)-v_x(t)}{\Delta t} && \text{(5a)}\\
\frac{dv_y}{dt} &\approx \frac{v_y(t+\Delta t)-v_y(t)}{\Delta t} && \text{(5b)}\\
\frac{dx}{dt} &\approx \frac{x(t+\Delta t)-x(t)}{\Delta t} && \text{(5c)}\\
\frac{dy}{dt} &\approx \frac{y(t+\Delta t)-y(t)}{\Delta t} && \text{(5d)}
\end{aligned}
$$

Hvor $\Delta t$ er en tidsskridt størrelse vi vælger. Indsættes approksimeringerne fra Ligning 5 i Ligning 3 fås den diskrete model:

$$
\begin{aligned}
v_x(t+\Delta t) &= v_x(t) && \text{(6a)}\\
v_y(t+\Delta t) &= v_y(t) - \frac{\Delta t}{mg} && \text{(6b)}\\
x(t+\Delta t) &= x(t) + \Delta t v_x(t) && \text{(6c)}\\
y(t+\Delta t) &= y(t) + \Delta t v_y(t) && \text{(6d)}
\end{aligned}
$$

Ovenstående er en model som for en given værdi for $t$ kan give et estimat for boldens position og hastighed til tiden $t + \Delta t$. Da vi kender værdierne i $t = 0$ (Ligning 4) kan vi iterativt beregne approksimerede værdier for alle værdier af $t$.

### 2.a) `ball_step`

Implementer funktionen `ball_step.m` der tager $x(t), y(t), v_x(t), v_y(t), m, g, \Delta t$ som input og giver $x(t+\Delta t), y(t+\Delta t), v_x(t+\Delta t), v_y(t+\Delta t)$ som output. Der er givet en skabelon til funktionen i Figur 6.

### 2.b) `ball_simulate`

Implementer funktionen `ball_simulate.m` der tager $x_0, y_0, v_x(0), v_y(0), m, g, \Delta t$ som input. Der skal implementeres en løkke der kalder `ball_step` og beregner positioner indtil $y < 0$. Output skal være en matrix af positioner.
Koden i `ball_thrower.m` (Figur 8) kører `ball_simulate.m` med passende værdier og plotter boldens kurve. Der er givet en skabelon til funktionen i Figur 7.

### 2.c) Verificering og validering

I har nu programmet jeres egen simulator og kan nu udføre numeriske eksperimenter med den.

1. Udtænk og beskriv hvordan I vil verificere den diskrete model og validere jeres nye computer simulator (Hint: prøv at opstille en eksperiment plan ved at lave hypoteser om eksempelvis effekten af at ændre på parametre som $\Delta t, m, g$.)

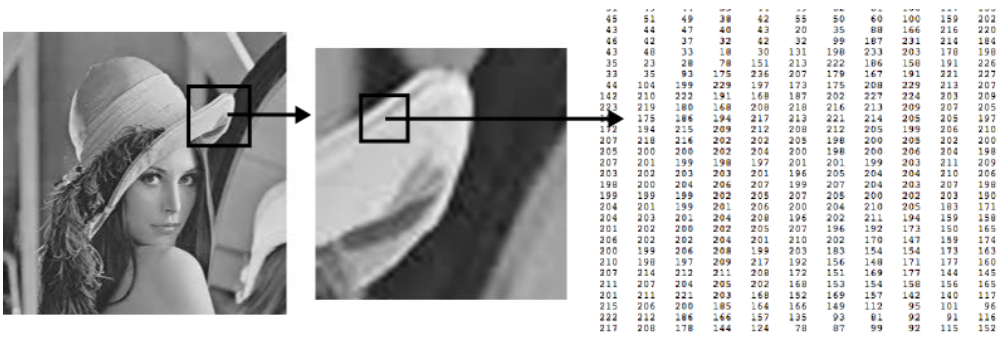
2. Udfør dine eksperimenter, beskriv og diskuter jeres resultater (tag stilling til om I syntes det er en god eller dårlig simulator I har lavet)

## Opgave 3: Kant finder

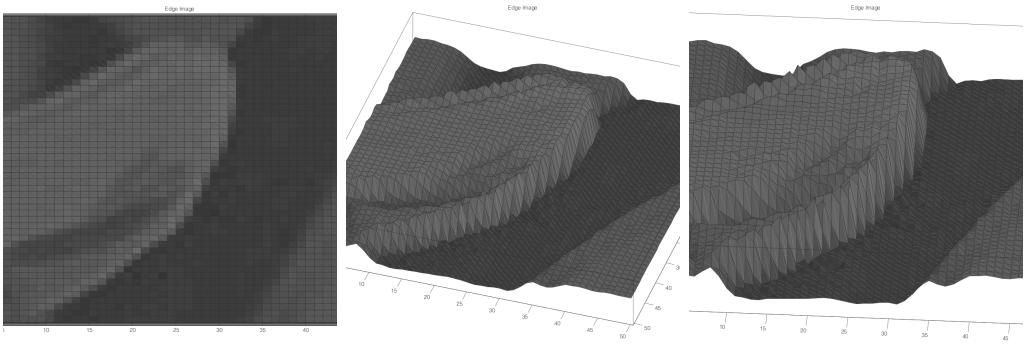
I denne opgave skal I implementere en kant finder, der kan finde kanter i et sort-hvidt billede af den "rigtige" verden. Et billede kan repræsenteres ved et 2D gitter af talværdier (en matrix). Hvert tal i dette gitter kaldes en pixel og værdien angiver intensiteten i den tilhørende pixel, vi kalder den for pixelværdien. Således vil en helt sort pixel have pixelværdien 0, mens en helt hvid pixel har pixelværdien 255. Pixels med gråtoner kan antage pixelværdier i intervallet $]0; 255[$.

Et billede kan opfattes som et slags højdekort af et "landskab", hvis vi tegner billedet som et gitter i et 3D koordinat system, hvor $x$ og $y$ aksen er pixel indices og $z$ aksen er pixel værdien (=højden af landskabet).

Fra vores intuition om landskaber og ved at kigge på billeder får vi ideen om at kanter findes der hvor der findes en væg imellem to flade områder eller der er en højde forskel imellem to områder. Fra dette kan vi også indse at vores nyfundne definition af kanter i billeder er lokal, dvs. den afhænger kun af et lille område af billedet og ikke hele billedet.
Når vi skal teste en pixel i billedet for om det er en kant så vil vi gerne sammenligne hvor meget dens nabo område ligner vores idealiseret geometriske form for en kant. Vi vælger nu kun at kigge i et område som inkluderer de pixels der ligger lige omkring den



Figur 3: Lena.jpg: Tre niveauer af forstørrelse, sidste niveau viser pixelværdierne



Figur 4: Det andet niveau af forstørrelse fra forrige FIgur. Plottet er lavet ved hjælp af MatLab funktionen `surface`, figuren viser 3D plottet fra tre vinkler.

pågældende pixel.
For at afgøre om vores pixel er en kant vil vi sammenligne dens område med en skabelon for en kant. For at holde kant finderen simpel, kigger vi på lokale ændringer i $x$ retningen og $y$ retningen. En kant skabelon kan opskrives som et lille maskebillede:

$$E_x = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}, E_y = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{7}$$

Maskerne $E_x$ og $E_y$ kaldes Sobel operatorer, og er en tilnærmelse af de afledte for den midterste pixel i henholdsvis $x$ og $y$ retningen. Nu kan vi udføre sammenligningen med en pixel ved at *folde* naboområdet med skabelonerne, her er et tal eksempel for $E_x$:

$$\begin{bmatrix} 10 & 11 & 12 \\ 20 & 11 & 0 \\ 100 & 11 & 10 \end{bmatrix} \otimes \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} = 78 \tag{8}$$

hvor $\otimes$ angiver at hver pixel i naboområdet er ganget med den tilsvarende pixel i Sobel operatoren og at disse produkter er lagt sammen:

$$10*(-1)+11*(-2)+12*(-1)+20*(0)+11*(0)+0*(0)+100*1+11*2+10*1=78$$

For et ensartet naboområde (hvor alle pixelværdier er ens) vil dette give nul, overbevis dig selv ved at udføre følgende:

$$\begin{bmatrix} 42 & 42 & 42 \\ 42 & 42 & 42 \\ 42 & 42 & 42 \end{bmatrix} \otimes \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{9}$$

Med andre ord, en høj værdi indikerer en høj sandsynlighed for en kant. For at detektere kanter i begge retninger foldes naboområderne med begge Sobel operatorer.

### 3.a) `edge_detector`

Implementer funktionen `edge_detector.m` som tager et talgitter (matrix) som input og returnerer et nyt talgitter som output. Funktionen skal indeholde en løkke der løber igennem alle pixels og danner en nabomatrix for hver pixel, vær opmærksom på problemer i gitterets yderste pixels (hint: en løsning kan være at starte i pixel $2,2$ og slutte i $n-1, n-1$). Husk at MatLab bruger 1-indeksering i stedet for den sædvanlige 0-indeksering.
Hver nabomatrix skal foldes med de to Sobel operatorer, resultatet af de to foldninger skal lægges sammen således at:

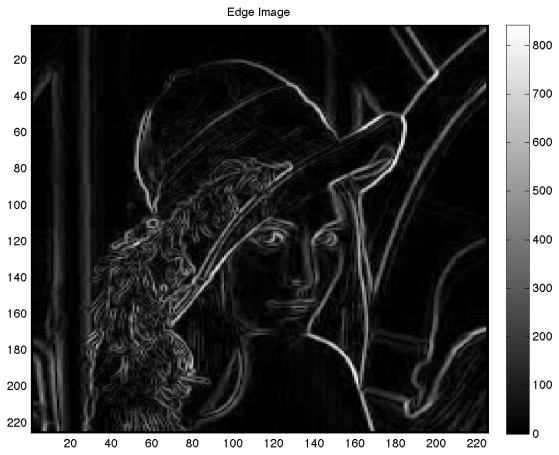
$$resultat = \sqrt{(nabomatrix \otimes E_x)^2 + (nabomatrix \otimes E_y)^2}$$

Den ovenstående resultatværdi lægges i output talgitter på den plads som svarer til den pixel som nabomatrixen er udregnet for. Der er givet en skabelon til funktionen i Figur 9.
Koden i `run_edge_detector.m` (Figur 10) indlæser billedet `lena.jpg` og gemmer det i talgitteret `I`, `edge_detector` køres på `I` og passende figurer plottes.

### 3.b) Verificering og validering

I har nu jeres egen simple kant finder og kan anvende den til at analysere billeder af den rigtige verden.

- Overvej hvordan jeres kant finder skal verificeres og valideres? Beskriv hvordan I vil gøre dette, eventuelt ved at opstille en eksperiment plan.
- Udfør jeres eksperimenter og beskriv jeres resultater. Er det en god eller dårlig kant finder?




Figur 5: Kanter fundet i `lena.jpg` ved hjælp af de to Sobel operatorer

## Litteratur


- *Kenny Erleben* Simulering og modellering af robotter og mennesker. Den digitale revolution – fortællinger fra datalogiens verden, pp. 68-79. 2010.
- *S. Keshav* How to read a paper, SIGCOMM Comput. Commun. Rev. vol . 37. no. 3, 2007 (http://portal.acm.org/citation.cfm?id=1273458).
- *Shu-Wei Hsu and John Keyser* Piles of Objects, to be published at SIGGRAPH ASIA 2010 (http://students.cs.tamu.edu/swhsu/proj/piles/).


## Matlab Programming Solutions

```
1 function [x, y, vx, vy] = ball_step(x, y, vx, vy, dt, m, g)
2
3 x = 0; error('Udregn den opdaterede vaerdi for x vha lign. 6')
4 y = 0; error('Udregn den opdaterede vaerdi for y vha lign. 6')
5 vy = 0; error('Udregn den opdaterede vaerdi for vy vha lign. 6')
6
7 end
```

Figur 6: `ball_step.m`

```
function [X Y] = ball_simulate(x0, y0, vx0, vy0, dt, m, g)

x  = x0;
y  = y0;
vx = vx0;
vy = vy0;

X  = [];
Y  = [];

while y>0

  [x y vx vy] = ball_step(x,y,vx,vy,dt,m,g);

  X = [X x];
  Y = [Y y];
end

end
```

Figur 7: `ball_simulate.m`: referenceløsning

```
clear all;
close all;

m = 1;
g = 9.81;
dt = 0.01;

x0 = 0;
y0 = 1;
vx0 = 2;
vy0 = 2;

[X Y] = ball_simulate( x0, y0, vx0, vy0, dt, m, g  );

figure(1);
clf;
plot(X,Y,'-xr');
hold on;
xlabel('distance [M]');
ylabel('height [M]');
title('Ball Throw Result');
hold off;
```

Figur 8: `ball_thrower.m`

```
function J = edge_detector(I)

[M, N] = size(I);
J = zeros( size(I) );

Ex = 0; error('Indsaet Sobel operator for Ex')
Ey = 0; error('Indsaet Sobel operator for Ey')

for i=2:M-1
  for j=2:N-1

    A = 0; error('Konstruer nabomatrix for pixel (i,j)')
    val1 = 0; error('Fold med Sobel operator for Ex')
    val2 = 0; error('Fold med Sobel operator for Ey')
    J(i,j) = 0; error('Udregn resultat')

  end
end

end
```

Figur 9: `edge_detector.m`

```
clear all;
close all;

RGB = imread('lena_part.jpg');
I = double( rgb2gray(RGB) );
J = edge_detector(I);

figure(1);
clf;
imagesc( I );
hold on;
axis tight;
colorbar;
colormap gray;
title('Input Image');
hold off;

figure(2);
clf;
imagesc( J );
hold on;
axis tight;
colorbar;
colormap gray;
title('Edge Image');
hold off;
print('-f2', '-dpng' ,'my_edges');
```

Figur 10: `run_edge_detector.m`