

Følger og differensligninger

Joakim Sundnes^{1,2}

Simula Research Laboratory¹

Universitetet i Oslo, Institutt for informatikk²

Sep 28, 2026

Plan for uke 40

Tirsdag 29. sept:

- Oppgave 5.14, 5.29 fra "A primer on..."
- Løsning av likninger (lysbilder fra forrige uke)
 - Newton's metode
- Oppgave A.10 fra "A primer on..."
- Diskret modellering (differensligninger) (Kap. 13 i "Programmering for naturvitenskap...")

Torsdag 1. oktober:

- Oppgave A.7, A.19 fra "A primer on..."
- Systemer av differensligninger
- Oppgave A.4 fra "A primer on..."

Tallfølger er viktige i matematikk:

$$x_0, x_1, x_2, \dots, x_n, \dots,$$

Eksempel: alle oddetall

$$1, 3, 5, 7, \dots, 2n + 1, \dots$$

For denne tallfølgen har vi en formel for det n -te leddet:

$$x_n = 2n + 1$$

og vi kan skrive tallfølgen mer kompakt som

$$(x_n)_{n=0}^{\infty}, \quad x_n = 2n + 1$$

Andre eksempler på tallfølger

$$1, 4, 9, 16, 25, \dots \quad (x_n)_{n=0}^{\infty}, \quad x_n = n^2$$

$$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \dots \quad (x_n)_{n=0}^{\infty}, \quad x_n = \frac{1}{n+1}$$

$$1, 1, 2, 6, 24, \dots \quad (x_n)_{n=0}^{\infty}, \quad x_n = n!$$

$$1, 1+x, 1+x+\frac{1}{2}x^2, 1+x+\frac{1}{2}x^2+\frac{1}{6}x^3, \dots \quad , x_n = \sum_{j=0}^n \frac{x^j}{j!}$$

- Følger kan brukes til å modellere (simulere) fenomener fra den virkelige verden, som for eksempel:
 - Vekst av penger i banken
 - Størrelsen på en populasjon (biologi)
 - Antall smittede under en epidemi
 - ...
- Vanligvis umulig å utlede en formel for det n -te leddet, men vi kan utlede en ligning for hvordan leddet x_n relaterer seg til det forrige leddet x_{n-1}
- Slike ligninger kalles *differensligninger*

Hvorfor studere differensligninger?

- Differensligninger er vanskelige og/eller kjedelige å løse for hånd, men enkle å løse med løkker og arrayer i et dataprogram
- Mange viktige anvendelser:
 - Simulering av dynamiske prosesser
 - Effektiv beregning av følger og approksimasjoner
 - Nært beslektet med løsningen av *differensialligninger*

Problem:

Sett x_0 penger i en bank i år 0. Hva er verdien etter N år hvis renten er p prosent per år?

Løsning:

Vi har følgende formel for hvordan verdien i år n , dvs x_n , av verdien året før, x_{n-1} :

$$x_n = x_{n-1} + \frac{p}{100} x_{n-1}$$

Hvordan løser vi denne ligningen for x_n ? Start med x_0 , beregn x_1 , x_2 , ...

Simulering av differensligningen for renter

Program for $x_n = x_{n-1} + (p/100)x_{n-1}$:

```
import numpy as np
import matplotlib.pyplot as plt

x0 = 100                # startbeløp
p = 5                   # rente
N = 40                   # antall år

years = range(N+1)
x = np.zeros(N+1)

x[0] = x0
for n in years[1:]:
    x[n] = x[n-1] + (p/100.0)*x[n-1]

plt.plot(years, x, 'ro')
plt.xlabel('År')
plt.ylabel('Beløp')
plt.show()
```

Vi trenger ikke å lagre hele følgen, men det er praktisk for programmering og senere plotting

- Det forrige programmet lagrer alle x_n -verdiene i en NumPy-array
- For å beregne x_n , trenger vi bare én tidligere verdi, x_{n-1}

Derfor kunne vi lagret bare de to siste verdiene i minnet:

```
x_old = x0
for n in years[1:]:
    x_new = x_old + (p/100.) * x_old
    x_old = x_new  # x_new blir x_old i neste steg
```

Men programmering med en array $x[n]$ gjør det enkelt å plote følgen, så vi vil fortsette å bruke arrayer i eksemplene

Hvorfor bruke differensligningen...?

Vi har en eksakt formel: $x_n = x_0(1 + \frac{p}{100})^n$

Hva hvis renten p endrer seg over tid?

- Formelen kan ikke lenger brukes
- Enkelt å inkludere i programmet: bare fyll en array p med korrekt rente $p[n]$ for år n
- Modifisert program:

```
p = np.zeros(N)  #eller p = np.ones(N)

# fyll p[n] med korrekte verdier

x[0] = x0
for n in years[1:]:
    x[n] = x[n-1] + (p[n-1]/100.0)*x[n-1]
```

Hvorfor bruke differensligningen...?

Vi har en eksakt formel: $x_n = x_0(1 + \frac{p}{100})^n$

Hva hvis renten p endrer seg over tid?

- Formelen kan ikke lenger brukes
- Enkelt å inkludere i programmet: bare fyll en array p med korrekt rente $p[n]$ for år n
- Modifisert program:

```
p = np.zeros(N)  #eller p = np.ones(N)

# fyll p[n] med korrekte verdier

x[0] = x0
for n in years[1:]:
    x[n] = x[n-1] + (p[n-1]/100.0)*x[n-1]
```

Fibonacci-tallfølgen

Intet programmerings- eller mattekurs er komplett uten et eksempel på Fibonacci-tall:

$$x_n = x_{n-1} + x_{n-2}, \quad x_0 = 1, \quad x_1 = 1$$

Matematisk klassifisering

Dette er en *homogen differensligning av andre orden* (andre orden betyr tre nivåer: n , $n-1$, $n-2$). Denne klassifiseringen er viktig for matematisk løsningsteknikk, men ikke for simulering i et program.

Fibonacci utledet følgen for å modellere rottepopulasjoner, men tallfølgen har mange spesielle matematiske egenskaper og har derfor fått mye oppmerksomhet fra matematikere.

Program for å generere Fibonacci-tall

```
import numpy as np
import sys

N = int(sys.argv[1])
x = np.zeros(N+1, int)

x[0] = 1
x[1] = 1
for n in range(2, N+1):
    x[n] = x[n-1] + x[n-2]
print(n, x[n])
```

Nytt eksempel: eksponentiell vekst med begrensede miljøressurser

Modellen for vekst av penger i en bank har en løsning av typen

$$x_n = x_0 C^n$$
$$C = (1 + p/100)$$

Dette kan skrives som

$$x_n = x_0 e^{n \ln C}$$

som er eksponentiell vekst i tid (n)

- Populasjoner av mennesker, dyr og celler viser også samme type vekst så lenge det er ubegrensede ressurser (plass og mat)
- De fleste miljøer kan bare understøtte et maksimalt antall R individer
- Hvordan kan vi modellere denne begrensningen?

Modellering av vekst i et miljø med begrensede ressurser

I starten, når det er nok ressurser, er veksten eksponentiell:

$$x_n = x_{n-1} + rx_{n-1} \quad (r = p/100)$$

Vekstraten r må avta mot null når x_n nærmer seg R . Den enkleste formuleringen av $r(x_n)$ som gir denne effekten er en lineær funksjon:

$$r(x_n) = \rho \left(1 - \frac{x_n}{R}\right)$$

Her har vi $r(n) \approx \rho$ når $x_n \ll R$, og $r(n) \rightarrow 0$ når $x_n \rightarrow R$.

Logistisk vekstmodell:

$$x_n = x_{n-1} + \rho x_{n-1} \left(1 - \frac{x_{n-1}}{R}\right)$$

(Dette er en *ikke-lineær* differensligning)

I et program er det enkelt å introdusere logistisk i stedet for eksponentiell vekst, bare erstatt

$$x[n] = x[n-1] + r * x[n-1]$$

med

$$x[n] = x[n-1] + \text{rho} * x[n-1] * (1 - x[n-1]/R)$$

Differensligninger må ha en startbetingelse

- I matematikk trengs en startbetingelse x_0 for å sikre en unik løsning x_n
- Startbetingelsen er enda viktigere når vi løser ligningen med programmering; ellers kan vi ikke starte programmet (x_0 trengs for å beregne x_1)
- Men: hvis du glemmer $x[0] = x_0$ i programmet, får du ofte $x[0] = 0$ (siden vi bruker $x = \text{zeros}(N+1)$), som (vanligvis) ikke gir svaret vi ønsker!

Systemer av differensligninger

- Noen anvendelser kan kreve flere variabler som er koblet
- Én variabel ved steg n avhenger av verdien til flere variabler ved steg $n - 1$
- Slike følger kan beskrives av *systemer av differensligninger*
- Eksempler:
 - Følge med på verdien av et lån (x) og de månedlige avdragene (y) (Oppg A.4 fra "A primer on..." av H.P. Langtangen)
 - Populasjonsdynamikk for arter som konkurrerer om de samme ressursene, eller hvor en art spiser en annen

Eksempel 1: avdrag på et lån

- Oppgave A.4

Eksempel: Lotka-Volterra-ligningene

Modellerer dynamikken til to populasjoner; byttedyr x_n og rovdyr y_n .

$$x_n = x_{n-1} + ax_{n-1} - bx_{n-1}y_{n-1},$$

$$y_n = y_{n-1} + dbx_{n-1}y_{n-1} - cy_{n-1}.$$

- a ; vekstrate for byttedyr i fravær av rovdyr (eksponentiell vekst)
- b ; dødsrate for byttedyr per møte mellom byttedyr og rovdyr
- c ; naturlig dødsrate for rovdyr i fravær av mat (byttedyr)
- d ; effektiviteten av å omgjøre spiste byttedyr til rovdyr

Fullstendig Lotka-Volterra-kode

```
import numpy as np
import matplotlib.pyplot as plt

x0 = 100                # startpopulasjon byttedyr
y0 = 8                  # startpopulasjon rovdyr
a = 0.0015
b = 0.0003
c = 0.006
d = 0.5
N = 10000               # antall tidsenheter (dager)
time = range(N+1)
x = np.zeros(len(time))
y = np.zeros_like(x)

x[0] = x0
y[0] = y0

for n in time[1:]:
    x[n] = x[n-1] + a * x[n-1] - b * x[n-1] * y[n-1]
    y[n] = y[n-1] + d * b * x[n-1] * y[n-1] - c * y[n-1]

plt.plot(time, x, label = 'Byttedyr')
plt.plot(time, y, label = 'Rovdyr')
plt.xlabel('Tid')
plt.ylabel('Populasjon')
plt.legend()
plt.show()
```

Har du noen gang tenkt over hvordan $\sin x$ egentlig beregnes?

- Hvordan kan du *beregne* $\sin x$, $\ln x$, e^x uten en kalkulator eller et program?
- Disse funksjonene ble opprinnelig definert for å ha visse ønskede matematiske egenskaper, men uten en algoritme for hvordan man evaluerer funksjonsverdier
- Idé: tilnærme $\sin x$, osv. med polynomer, siden de er enkle å beregne (sum, multiplikasjon), men hvordan??

Ville du forventet disse matematiske resultatene?

Utrolig resultat av Gregory, 1667:

$$\sin x = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

Enda mer utrolig resultat av Taylor, 1715:

$$f(x) = \sum_{k=0}^{\infty} \frac{1}{k!} \left(\frac{d^k}{dx^k} f(0) \right) x^k$$

For en hvilken som helst $f(x)$, hvis vi kan kjenner den deriverte til en funksjon i punktet $x = 0$, kan vi evaluere f for en hvilken som helst x , bare ved hjelp av addisjon og multiplikasjon.

I praksis brukes en endelig rekke:

$$f(x) \approx \sum_{k=0}^N \frac{1}{k!} \left(\frac{d^k}{dx^k} f(0) \right) x^k$$

Det mest populære valget er faktisk $N = 1$, som er grunnlaget for løsningsmetoder for differensiallikninger, og har vært essensielt i utviklingen av fysikk og teknologi.

Eksempel 1: $\exp(x)$

Eksponentialfunksjonen har seg selv som derivert:

$$\frac{d^k \exp(x)}{dx^k} = \exp(x) \text{ for alle } k$$

Vi har at $\exp(0) = 1$, og setter inn i formelen

$$f(x) = \sum_{k=0}^{\infty} \frac{1}{k!} \left(\frac{d^k}{dx^k} f(0) \right) x^k$$

som gir

$$\begin{aligned} \exp(x) &= \sum_{k=0}^{\infty} \frac{x^k}{k!} \\ &\approx 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 \\ &\approx 1 + x \end{aligned}$$

Eksempel 2: $\sin(x)$

For $\sin(x)$ har vi

$$\frac{d^k \sin(x)}{dx^k}(x=0) = \begin{cases} 0 & \text{for } k = 0, 2, 4, \dots \\ 1 & \text{for } k = 1, 5, 9, \dots \\ -1 & \text{for } k = 3, 7, 11, \dots \end{cases}$$

Satt inn i formelen

$$f(x) = \sum_{k=0}^{\infty} \frac{1}{k!} \left(\frac{d^k}{dx^k} f(0) \right) x^k$$

får vi

$$\sin(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

Taylor-polynomer rundt et vilkårlig punkt

De forrige Taylor-polynomene er mest nøyaktige nær $x = 0$. Vi kan gjøre polynomene nøyaktige rundt et hvilket som helst punkt $x = a$:

$$f(x) \approx \sum_{k=0}^N \frac{1}{k!} \left(\frac{d^k}{dx^k} f(a) \right) (x - a)^k$$

Taylor-polynom som en differensligning

Taylor-rekken for e^x rundt $x = 0$ er

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Definer

$$e_n = \sum_{k=0}^{n-1} \frac{x^k}{k!} = \sum_{k=0}^{n-2} \frac{x^k}{k!} + \frac{x^{n-1}}{(n-1)!}$$

Vi kan formulere summen i e_n som følgende differensligning:

$$e_n = e_{n-1} + \frac{x^{n-1}}{(n-1)!}, \quad e_0 = 0$$

Mer effektiv beregning: Taylor-polynomet som to differensligninger

Vi har

$$\frac{x^n}{n!} = \frac{x^{n-1}}{(n-1)!} \cdot \frac{x}{n}$$

Hvis vi lar $a_n = x^n/n!$ kan vi effektivt beregne a_n med

$$a_n = a_{n-1} \frac{x}{n}, \quad a_0 = 1$$

Nå kan vi oppdatere hvert ledd via a_n -ligningen og summere leddene via e_n -ligningen:

$$e_n = e_{n-1} + a_{n-1}, \quad e_0 = 0, \quad a_0 = 1$$
$$a_n = \frac{x}{n} a_{n-1}$$