

# Løsning av ligninger

Joakim Sundnes<sup>1,2</sup>

Simula Research Laboratory<sup>1</sup>

Universitetet i Oslo, Institutt for informatikk<sup>2</sup>

Sep 24, 2026

- Oppgave 5.13 fra "A primer on..."
- Mer om plotting; animasjoner (lysbilder fra tirsdag)
- Oppgave 5.39 fra "A primer on..."
- Løsning av ikke-lineære ligninger (Kap. 11 i "Programmering for naturvitenskap...")

# Ikke-lineære ligninger

Ikke-lineære algebraiske ligninger som

$$x = 1 + \sin x$$

$$\tan x + \cos x = \sin 8x$$

$$x^5 - 3x^3 = 10$$

er vanligvis umulige å løse med penn og papir, men kan løses med numeriske metoder. For å gjøre dette skriver vi ligningene på formen

$$f(x) = 0$$

For eksemplene over gir dette (flytt alt over på venstre side)

$$f(x) = x - 1 - \sin x$$

$$f(x) = \tan x + \cos x - \sin 8x$$

$$f(x) = x^5 - 3x^3 - 10$$

# Vi skal implementere metoder for å løse $f(x) = 0$

En løsning  $x$  av  $f(x) = 0$  kalles et nullpunkt eller en *rot* til  $f(x)$

Vi skal se på tre ulike metoder:

- 1 Halveringsmetoden
- 2 Newtons metode
- 3 Sekantmetoden

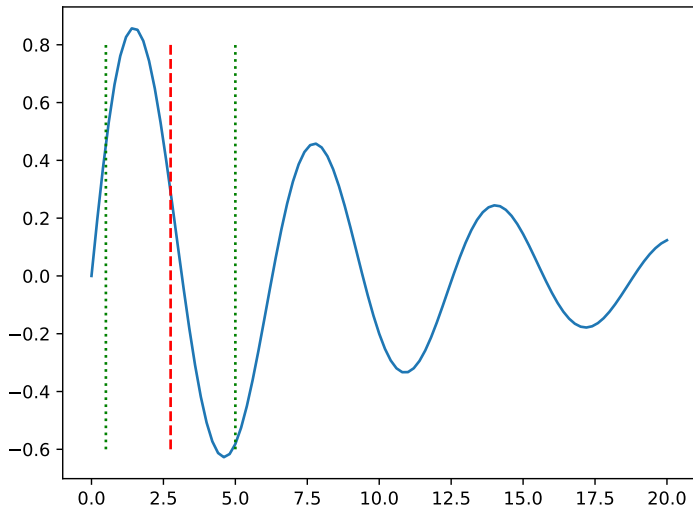
# Alle metodene følger samme oppskrift

- 1 Formulere en metode for å finne et nullpunkt
- 2 Oversette metoden til en presis algoritme
- 3 Implementere algoritmen i Python
- 4 Teste implementasjonen

# Halveringsmetoden (biseksjonsmetoden)

- Start med et intervall  $[a, b]$  der  $f(x)$  skifter fortegn
  - Da må det være (minst) ett nullpunkt i  $[a, b]$
  - Halver intervallet:
    - $m = (a + b)/2$ ; skifter  $f$  fortegn i venstre halvdel  $[a, m]$ ?
    - Ja: fortsett med venstre intervall  $[a, m]$  (sett  $b = m$ )
    - Nei: fortsett med høyre intervall  $[m, b]$  (sett  $a = m$ )
  - Gjenta prosedyren
- 
- Etter å ha halvert det opprinnelige intervallet  $n$  ganger, vet vi at  $f(x)$  må ha et nullpunkt innenfor et (lite) intervall  $2^{-n}(q - p)$
  - Metoden er treg, men veldig sikker
  - Andre metoder (som Newtons metode) kan være raskere, men kan også mislykkes i å finne en rot - halveringsmetoden feiler ikke

# Illustrasjon av halveringsmetoden



# Algoritme/implementasjon av metoden

```
for i in range(0, n+1):  
    m = (a + b)/2  
    if f(a) * f(m) <= 0:  
        b = m # nullpunkt i venstre halvdel  
    else:  
        a = m # nullpunkt i høyre halvdel
```

# Algoritmen kan gjøres mer effektiv

- $f(a)$  blir beregnet på nytt i hver if-test
- Dette er ikke nødvendig hvis  $a$  ikke har endret seg siden forrige gjennomgang av løkken
- På moderne datamaskiner og med enkle formler for  $f(x)$  har disse ekstra beregningene ikke noe å si
- Men innenfor vitenskap og ingeniørfag møter man  $f$ -funksjoner som tar timer eller dager å evaluere i ett punkt, og da betyr det noe å spare noen  $f(a)$ -evalueringer!
- Tommelfingerregel: fjern overflødige beregninger (med mindre koden blir mye mer komplisert og vanskeligere å verifisere)

## Ny, mer effektiv versjon av algoritmen

Idé: lagre  $f(x)$ -evalueringer i variabler

```
f_a = f(a)
for i in range(0, n+1):
    m = (a + b)/2
    f_m = f(m)
    if f_a * f_m <= 0:
        b = m # nullpunkt i venstre halvdel
    else:
        a = m # nullpunkt i høyre halvdel
    f_a = f_m
```

## Hvordan velge $n$ ? Altså, når skal iterasjonen stoppe

- Vi vil at feilen i roten skal være  $\epsilon$  eller mindre
- Etter  $n$  iterasjoner er det opprinnelige intervallet  $[a, b]$  halvert  $n$  ganger, og det nåværende intervallet har lengde  $2^{-n}(b - a)$ . Dette er tilstrekkelig lite hvis
$$2^{-n}(b - a) = \epsilon \quad \Rightarrow \quad n = -\frac{\ln \epsilon - \ln(b - a)}{\ln 2}$$
- Et enklere alternativ: bare gjenta halveringen til lengden på det nåværende intervallet er  $\leq \epsilon$
- Dette gjøres enklest med en while-løkke: `while b - a > epsilon:`
- Vi legger også til en test for å sjekke om  $f$  virkelig skifter fortegn i det opprinnelige intervallet  $[a, b]$

# Endelig versjon av halveringsalgoritmen

```
f_a = f(a)
if f_a * f(b) > 0:
    # feil: f skifter ikke fortegn i [a,b]

i = 0
while b - a > epsilon:
    i = i + 1
    m = (a + b)/2
    f_m = f(m)
    if f_a * f_m <= 0:
        b = m # roten er i venstre halvdel
    else:
        a = m # roten er i høyre halvdel
    f_a = f_m
```

# Implementasjon som en Python-funksjon

```
def bisection(f, a, b, eps):  
    fa = f(a)  
    if fa * f(b) > 0:  
        return None, 0  
        # Alternativ: raise ValueError(  
        # f'Ingen fortegnsskifte i [{a:g},{b:g}]')  
  
    i = 0 # iterasjonsteller  
    while b - a > eps:  
        i += 1  
        m = (a + b)/2.0  
        fm = f(m)  
        if fa * fm <= 0:  
            b = m # roten er i venstre halvdel av [a,b]  
        else:  
            a = m # roten er i høyre halvdel av [a,b]  
            fa = fm  
    return m, i  
  
print(bisection(f=lambda x: 2*x-3, a=0, b=10, eps=1E-5))
```

## Newtons metode

Simpson (1740) kom opp med følgende generelle metode for å løse  $f(x) = 0$  (basert på ideer fra Newton):

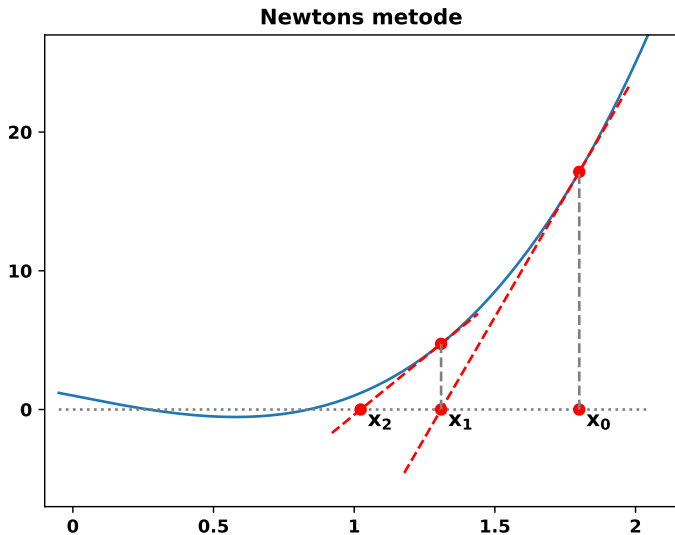
$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}, \quad x_0 \text{ gitt}$$

- Dette er en (ikke-lineær) differensligning
- Bygger på en lokal *linearisering* av  $f(x)$  rundt punktet  $x_n$ :

$$f(x) \approx \tilde{f}(x) = f(x_0) + f'(x_0)(x - x_0)$$

- Vi løser ligningen  $\tilde{f}(x) = 0$  i stedet for  $f(x) = 0$ , og gjentar prosessen med en ny linearisering rundt dette nullpunktet.

# Illustrasjon av Newtons metode



## Newtons metode forts.

- Når  $n \rightarrow \infty$ , håper vi at  $x_n \rightarrow x_s$ , der  $x_s$  løser  $f(x_s) = 0$
- Hvordan velge  $N$  når det vi ønsker er at  $x_N$  er nær  $x_s$ ?
- Trenger et litt annerledes program: simuler til  $f(x) \leq \epsilon$ , der  $\epsilon$  er en liten toleranse
- Newtons metode kan (lett) divergere, så  $f(x) \leq \epsilon$  vil kanskje aldri bli oppfylt!

# Et program for Newtons metode

## Rask implementasjon:

```
def Newton(f, x, dfdx, tol = 1.0E-7, max_n=100):  
    n = 0  
    while abs(f(x)) > tol and n <= max_n:  
        x = x - f(x)/dfdx(x)  
        n += 1  
    return x, n, f(x)
```

## Merk:

- $f(x)$  evalueres to ganger i hver runde av løkken - bare én evaluering er strengt tatt nødvendig (kan lagre verdien i en variabel og gjenbruke den)
- Det kan være nyttig å lagre  $x$ - og  $f(x)$ -verdiene i hver iterasjon (for plotting eller utskrift av en konvergenstabell)

# En forbedret funksjon for Newtons metode

Kun ett  $f(x)$ -kall i hver iterasjon, samt valgfri lagring av  $(x, f(x))$ -verdier under iterasjonene:

```
def Newton(f, x, dfdx, epsilon=1.0E-7, max_n=100,
           store = False):
    f_value = f(x)
    n = 0
    if store: info = [(x, f_value)]
    while abs(f_value) > epsilon and n <= max_n:
        x = x - f_value / dfdx(x)
        n += 1
        f_value = f(x)
        if store: info.append((x, f_value))
    if store:
        return x, info
    else:
        return x, n, f_value
```

# Bruk av Newtons metode

$$e^{-0.1x^2} \sin\left(\frac{\pi}{2}x\right) = 0$$

Løsninger:  $x = 0, \pm 2, \pm 4, \pm 6, \dots$

## Hovedprogram:

```
import sys
import numpy as np

def g(x):
    return np.exp(-0.1 * x**2) * np.sin(np.pi / 2*x)

def dg(x):
    return -2*0.1*x*np.exp(-0.1*x**2)*np.sin(np.pi/2*x) +
           np.pi/2*np.exp(-0.1*x**2)*np.cos(np.pi/2*x)

x0 = float(sys.argv[1])
x, n, val = Newton(g, x0, dg)
print('Beregnet rot:', x)
```

## Fordeler og ulemper med Newtons metode

- Konvergerer vanligvis mye raskere enn halveringsmetoden
- Kan divergere, spesielt hvis første gjett  $x_0$  er langt unna nullpunktet
- Algoritmen trenger den deriverte  $f'(x)$ , som det ikke alltid er mulig å beregne analytisk.

Alternative løsninger:

- Estimer den deriverte med

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

- Bruk en alternativ metode, f.eks. sekantmetoden

- Newtons metode vil gi en sekvens av løsninger  $x_0, x_1, \dots$  og tilhørende funksjonsverdier  $f(x_0), f(x_1), \dots$ ,
- Vi kan bruke disse punktene  $(x_0, f(x_0)), (x_1, f(x_1)), \dots$  til å estimere den deriverte for neste iterasjon:

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}, \quad (1)$$

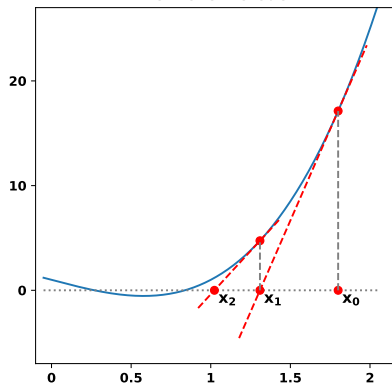
Hvis vi setter inn denne approksimasjonen av  $f'(x)$  i algoritmen for Newtons metode får vi: metode

$$x_{n+1} = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}, \quad (2)$$

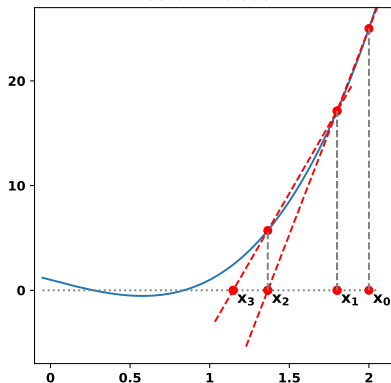
- Vi trenger to startverdier  $x_0$  og  $x_1$ , men ikke den deriverte
- Sekantmetoden konvergerer vanligvis litt tregere enn Newtons metode, men kan likevel være mer effektiv
- Trenger bare en evaluering av  $f(x)$  for hver iterasjon

# Sekantmetoden - illustrasjon

Newton's metode



Sekantmetoden



Oppgave A.10 fra "A primer on..."

# Ligningsløserne fra SciPy

Løsning av ikkelineære ligninger er en veldig vanlig oppgave, og det finnes derfor en del ferdige verktøy man kan bruke. Løserne i SciPy-biblioteket er enkle å bruke, og virker bra på enkle problemer. Funksjonen for å løse skalare ligninger heter `root_scalar`:

```
from scipy.optimize import root_scalar
import numpy as np

def f(x):
    return x**3 - 1

def dfdx(x):
    return 3*x**2

# Nullpunkter
nullpunkt_halverings = root_scalar(f,method='bisect',bracket=[0,5])
nullpunkt_newton = root_scalar(f,method='newton',fprime=dfdx, x0=5)
nullpunkt_secant = root_scalar(f,method='secant',x0=5,x1=4)
```

## Mer om ligningsløserne fra SciPy

- For enkle ligninger virker alle løserne i SciPy fint, uten merkbare forskjeller i effektivitet
- For mer krevende ligninger kan noen av løserne divergere eller gi feil svar:  $\Rightarrow$  For å velge riktig metode og bruke den riktig trenger vi å vite hvordan de ulike metodene virker
- Den beste måten å få denne forståelsen er å programmere metodene selv, som vi har gjort til nå