1.25 ODESolver og løsing av ODE-systemer

Ole Christian Lingjærde, Institutt for Informatikk, UiO

2. november 2020

Dagens tema

- Strukturen til ODESolver
- Eksempler på løsing av skalare ODE'er
- Oppgave E.21 og E.22
- Løsing av ODE-systemer
- Eksempler på løsing av ODE-systemer
- Gjennomgang av modulen ODESolver
- SIR-modellen

Strukturen til ODESolver

```
class ODESolver:
   def __init__(self, f):
      ...osv...

   def set_initial_condition(self, U0):
      ...osv...

   def solve(self, time_points):
      ...osv...

class ForwardEuler(ODESolver):
   def advance(self):
      u=self.u; t=self.t; f=self.f; k=self.k
      dt = t[k+1]-t[k]
      return u[k] + dt * f(u[k], t[k])

class RungeKutta4(ODESolver):
   def advance(self):
      ....osv...
```

Hvordan løse skalare ODE'er med ODESolver

Trinn 1: Finn $f(u,t)$ og implementer. Eksempel:

$$u'(t) = 2u(t)^3 \quad \Rightarrow \quad \texttt{f = lambda u,t: 2*u**3}$$

Trinn 2: Velg løsningsmetode. Eksempel:

```
metode = ForwardEuler(f)
```

Trinn 3: Sett initialbetingelse. Eksempel:

```
metode.set_initial_condition(U0=5)
```

Trinn 4: Velg tidspunkt. Eksempel:

```
timepoints = np.linspace(0, 5, 500)
```

Trinn 5: Løs likningen. Eksempel:

```
u,t = metode.solve(timepoints)
```

Eksempel 1

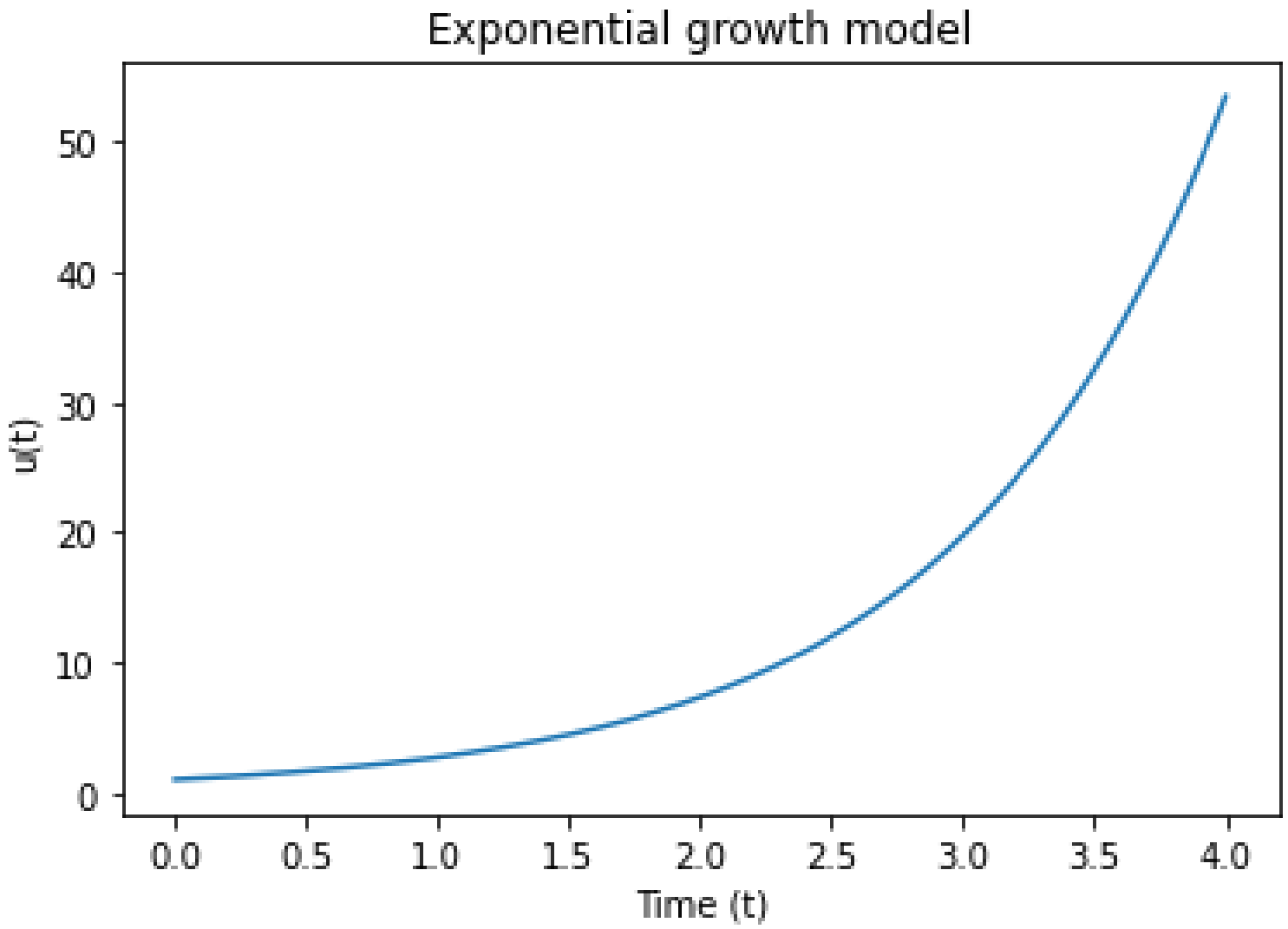
ODE: $u'(t) = u(t); \qquad u(0) = 1$

```
# Importer nødvendige moduler
from ODESolver import *
import numpy as np

# Vi har f(u,t)=u og implementerer:
f = lambda u,t: u

# Vi bruker Forward Euler metoden:
metode = ForwardEuler(f)

# Vi setter initialbetingelsen:
metode.set_initial_condition(U0=1)

# Vi angir tidspunkter hvor løsning skal beregnes:
timepoints = np.linspace(0, 4, 400)

# Vi løser likningen:
u,t = metode.solve(timepoints)
```

Kode for å plotte løsningen

```
import matplotlib.pyplot as plt
plt.plot(t,u)
plt.title("Exponential growth model")
plt.xlabel("Time (t)")
plt.ylabel("u(t)")
plt.show()
```

Plott av løsningen

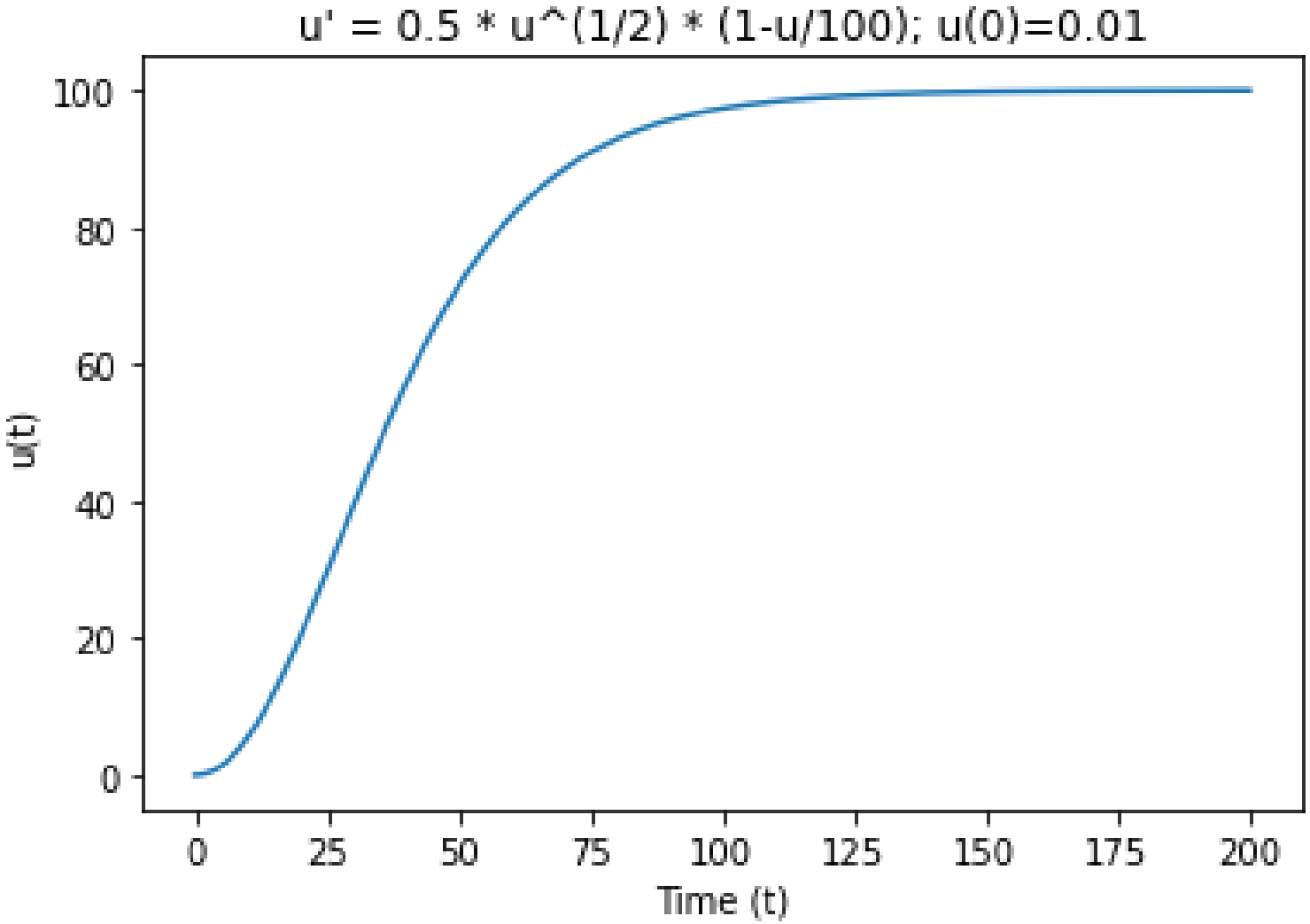


Eksempel 2

ODE: $u'(t) = (1/2)\sqrt{u(t)}(1 - \frac{u(t)}{100}); \qquad u(0) = 0.01$

```
from ODESolver import *
import numpy as np
import matplotlib.pyplot as plt

class Fnc:
   def __init__(self, alpha, R):
      self.alpha = alpha
      self.R = R

   def __call__(self, u, t):
      return self.alpha * np.sqrt(u) * (1-u/self.R)

metode = ForwardEuler(Fnc(alpha=0.5, R=100))
metode.set_initial_condition(U0=0.01)
timepoints = np.linspace(0, 200, 400)
u,t = metode.solve(timepoints)

plt.plot(t,u)
plt.title("u' = 0.5 * u^(1/2) * (1-u/100); u(0)=0.01")
plt.xlabel("Time (t)")
plt.ylabel("u(t)")
plt.show()
```

Løsning

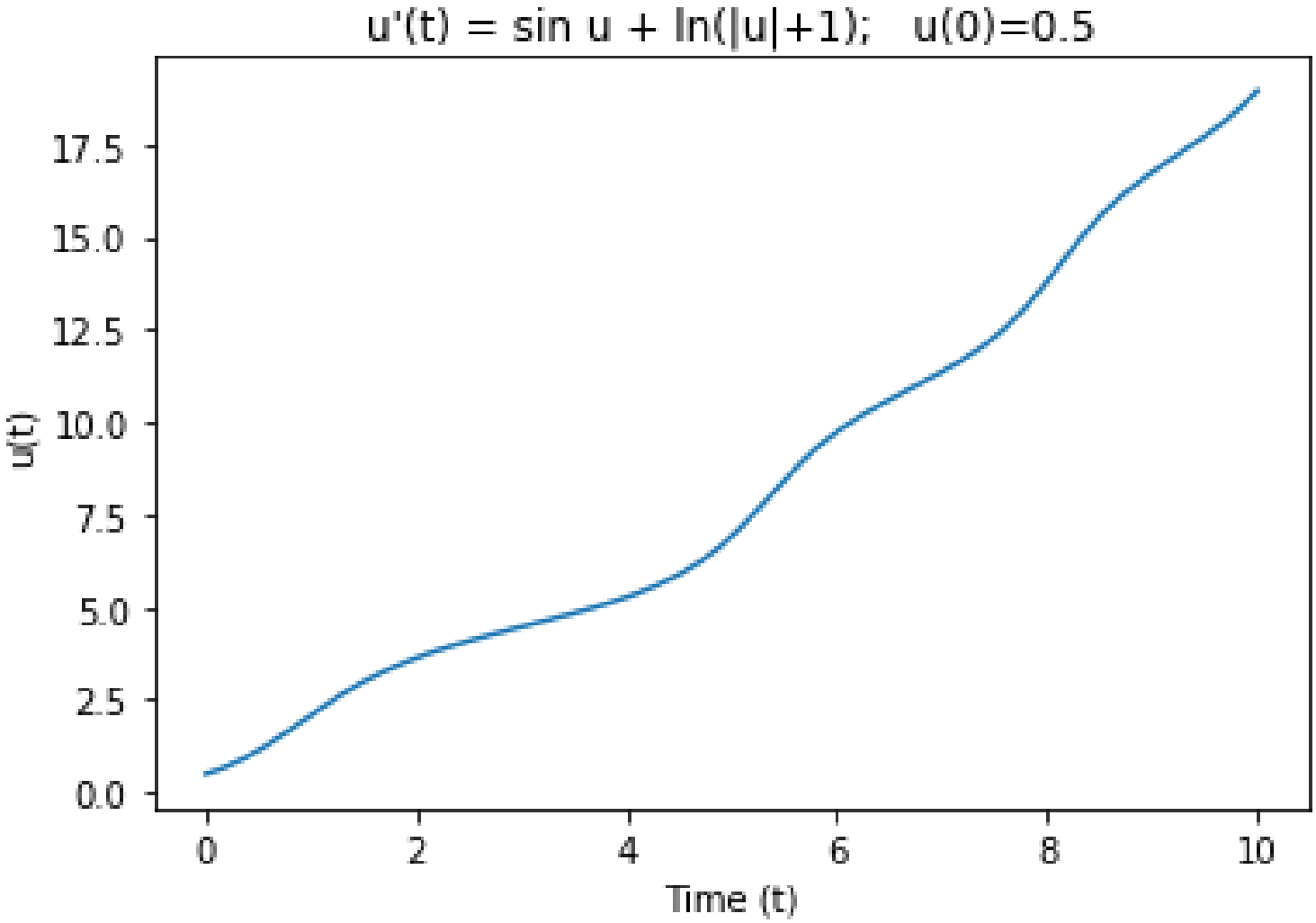

u' = 0.5 * u^(1/2) * (1-u/100); u(0)=0.01
100
80
60
40
20
0
u(t)
0
25
50
75
100
125
150
175
200
Time (t)

Eksempel 3

ODE: $u'(t) = \sin u(t) + \ln(|u(t)| + 1)$, $u(0) = 0.5$

```
from ODESolver import *
import numpy as np

# Vi implementerer f(u,t)=\sin u + \ln(|u|+1) som funksjon:
f = lambda u,t: np.sin(u) + np.log(abs(u)+1)

# Vi velger RungeKutta-metoden:
metode = RungeKutta4(f)

# Sett initialbetingelsen:
metode.set_initial_condition(U0=0.5)

# Angi tidspunkter hvor løsning skal beregnes:
timepoints = np.linspace(0, 10, 500)

# Løs likningen:
u,t = metode.solve(timepoints)
```

Løsning



Oppgave E.21 og E.22

Exercise E.21: Code the 4th-order Runge-Kutta method; function
Use the file `ForwardEuler_func.py` from Sect. E.1.3 as starting point for implementing the famous and widely used 4th-order Runge-Kutta method (E.41)–(E.45). Use the test function involving a linear $u(t)$ for verifying the implementation. Exercise E.23 suggests an application of the code.
Filename: `RK4_func`.

Exercise E.22: Code the 4th-order Runge-Kutta method; class
Carry out the steps in Exercise E.21, but base the implementation on the file `ForwardEuler.py` from Sect. E.1.7.
Filename: `RK4_class`.

4'de ordens Runge-Kutta

Oppdateringsregel:

$$u_{k+1} = u_k + \frac{1}{6}\left(K_1 + 2K_2 + 2K_3 + K_4\right)$$

hvor

$$\begin{aligned}
K_1 &= \Delta t\ f(u_k, t_k) \\
K_2 &= \Delta t\ f(u_k + \tfrac{1}{2}K_1,\ t_k + \tfrac{1}{2}\Delta t) \\
K_3 &= \Delta t\ f(u_k + \tfrac{1}{2}K_2,\ t_k + \tfrac{1}{2}\Delta t) \\
K_4 &= \Delta t\ f(u_k + K_3,\ t_k + \Delta t)
\end{aligned}$$

Løsing av ODE-systemer

Anta at vi har følgende system hvor $x(t)$ og $y(t)$ er ukjente:

$$\begin{aligned} x'(t) &= y(t) \\ y'(t) &= -x(t) \end{aligned}$$

Vi vil finne løsning for $t \in [0, 6]$ når $x(0) = 1$, $y(0) = 0$:

- La $t_k = k \cdot dt$, $k = 0, 1, \ldots, n$.
- Hvis $y(t)$ antas kjent kan vi finne $x(t)$ med Forward-Euler:

$$x(t_{k+1}) = x(t_k) + dt \cdot y(t_k)$$

- Hvis $x(t)$ antas kjent kan vi finne $y(t)$ med Forward-Euler:

$$y(t_{k+1}) = y(t_k) - dt \cdot x(t_k)$$

- På vektorform:

$$\begin{pmatrix} x(t_{k+1}) \\ y(t_{k+1}) \end{pmatrix} = \begin{pmatrix} x(t_k) \\ y(t_k) \end{pmatrix} + dt \cdot \begin{pmatrix} y(t_k) \\ -x(t_k) \end{pmatrix}$$

Vektor-ODE'er

Hvis vi definerer

$$\mathrm{u}(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}, \quad \mathrm{f}(\mathrm{u}, t) = \begin{pmatrix} u_2 \\ -u_1 \end{pmatrix}$$

så kan oppdateringsregelen

$$\begin{pmatrix} x(t_{k+1}) \\ y(t_{k+1}) \end{pmatrix} = \begin{pmatrix} x(t_k) \\ y(t_k) \end{pmatrix} + dt \cdot \begin{pmatrix} y(t_k) \\ -x(t_k) \end{pmatrix}$$

skrives slik:

$$\mathrm{u}(t_{k+1}) = \mathrm{u}(t_k) + dt \cdot \mathrm{f}(\mathrm{u}(t_k), t_k)$$

Implementasjonen er nesten identisk med den vi tidligere har sett på for skalare ODE'er - vi må bare huske at u nå er en vektor og at $\mathrm{f}(\mathrm{u}, t)$ returnerer en vektor.

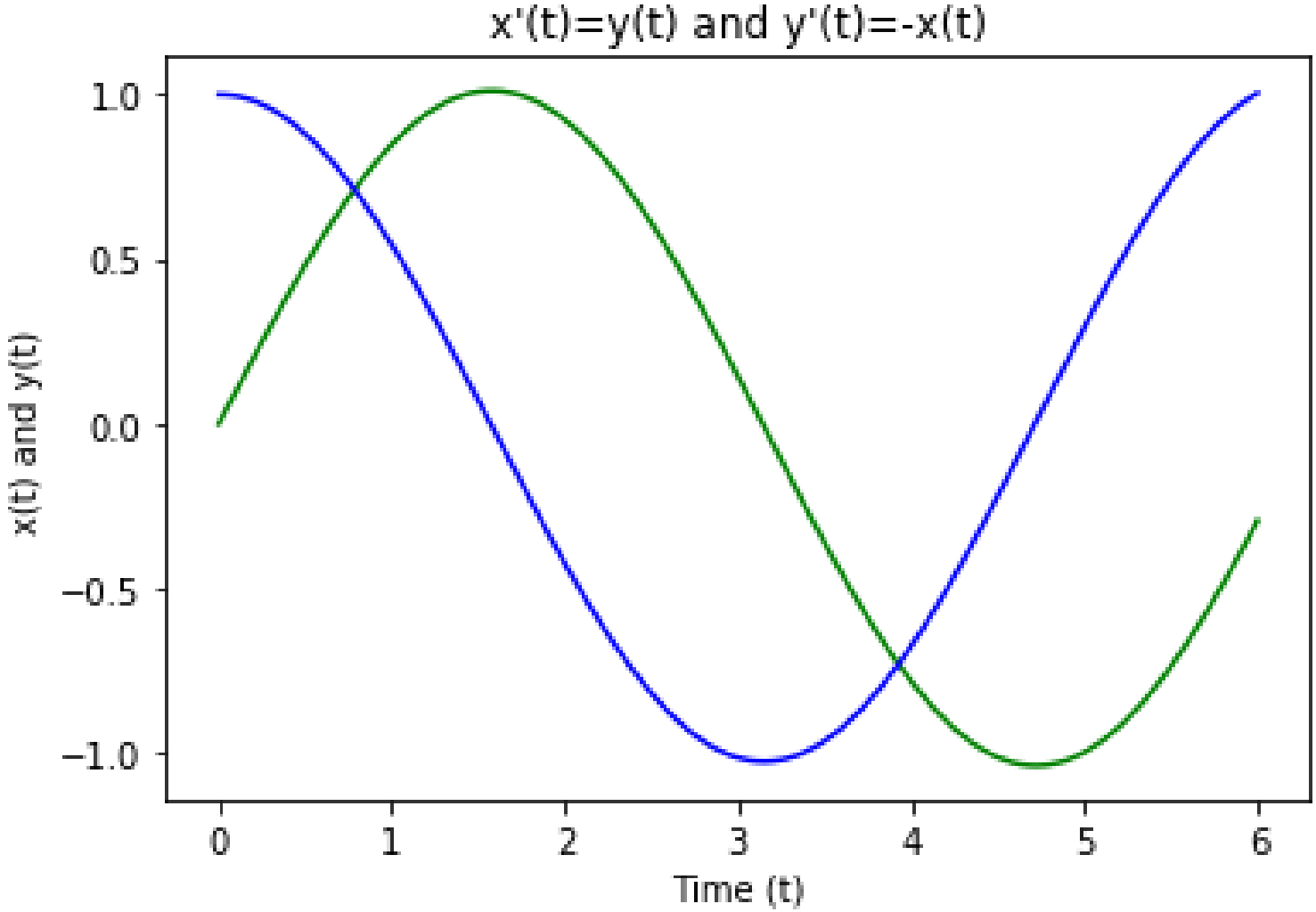
Eksempel 1

ODE-system:

$$\begin{aligned} x'(t) &= y(t) \quad , \quad x(0) = 0 \\ y'(t) &= -x(t) \quad , \quad y(0) = 1 \end{aligned}$$

```
from ODESolver import *
import numpy as np

# Vi har f(u,t)=(u2,-u1) og vi implementerer den som
def f(u,t):
   return np.array([u[1], -u[0]])

# Vi bruker Forward Euler metoden og lager instans:
metode = ForwardEuler(f)

# Vi setter initialbetingelser:
metode.set_initial_condition(U0=[0,1])

# Vi angir tidspunkter hvor løsning skal beregnes:
timepoints = np.linspace(0, 6, 400)

# Vi løser likningen:
u,t = metode.solve(timepoints)
```

Løsning



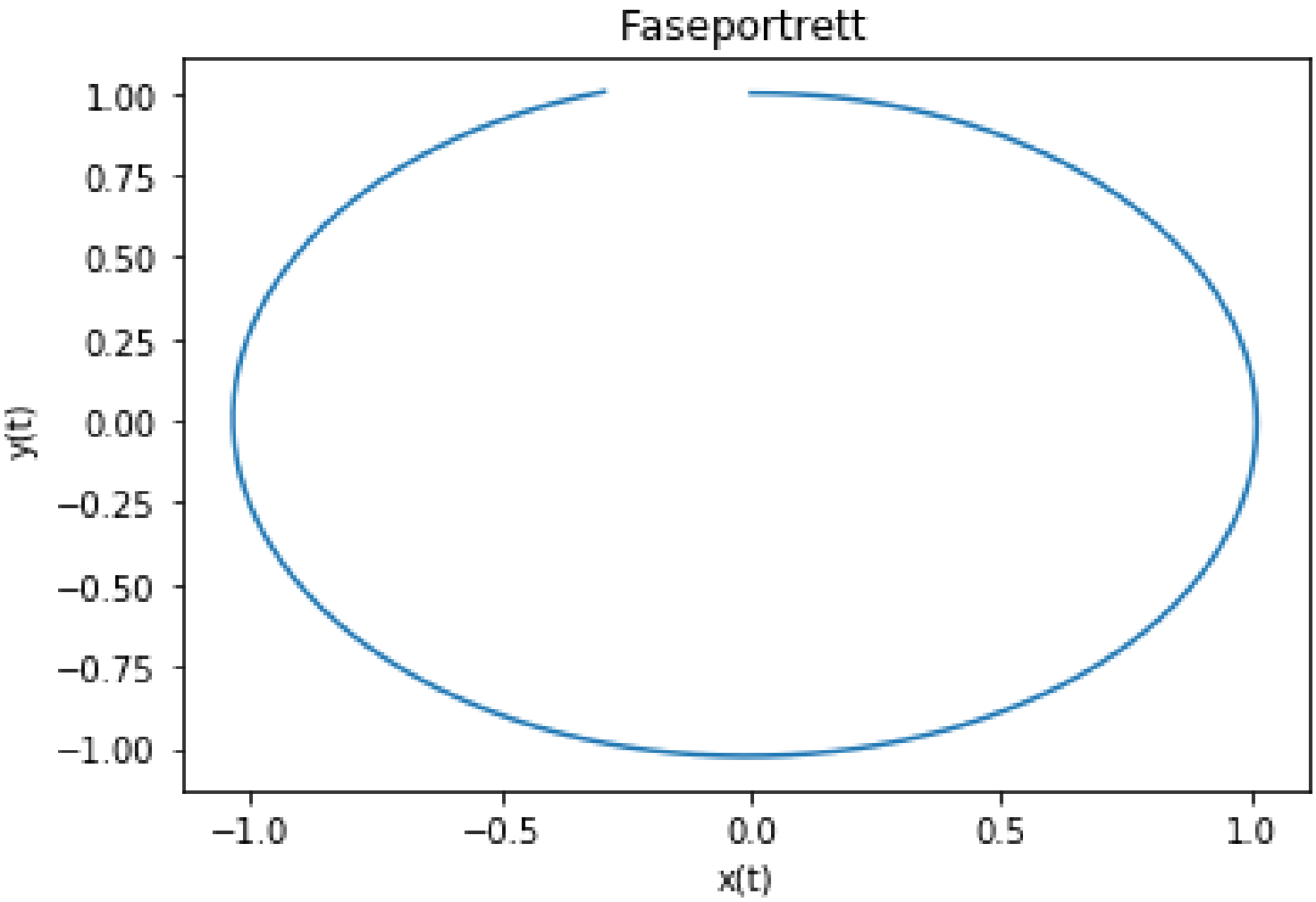
Faseportretter

- Når vi har løst et system med flere ODE'er er det ofte av interesse hvordan tilstanden til systemet forandres over tid.
- Eksempel: har vi funnet $x(t)$ og $y(t)$ så kan vi plotte de to mot hverandre. Det kalles et

faseportrett.

```
import matplotlib.pyplot as plt

plt.plot(u[:,0], u[:,1])
plt.title("Faseportrett")
plt.xlabel("x(t)")
plt.ylabel("y(t)")
plt.show()
```

Faseportrettet til $x'(t) = y(t)$ og $y'(t) = -x(t)$



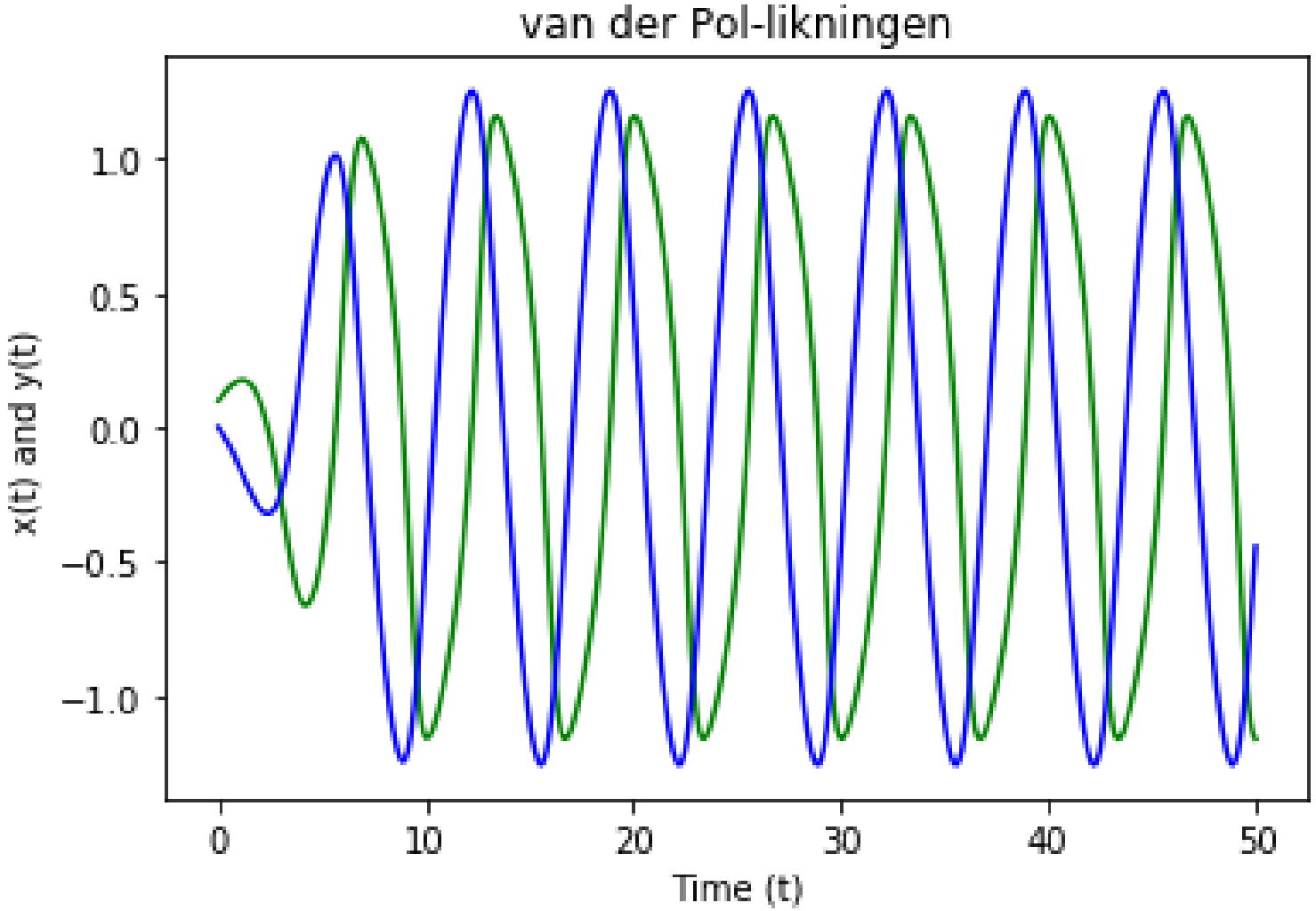
Eksempel 2

ODE-system (*van der Pol likningen*):

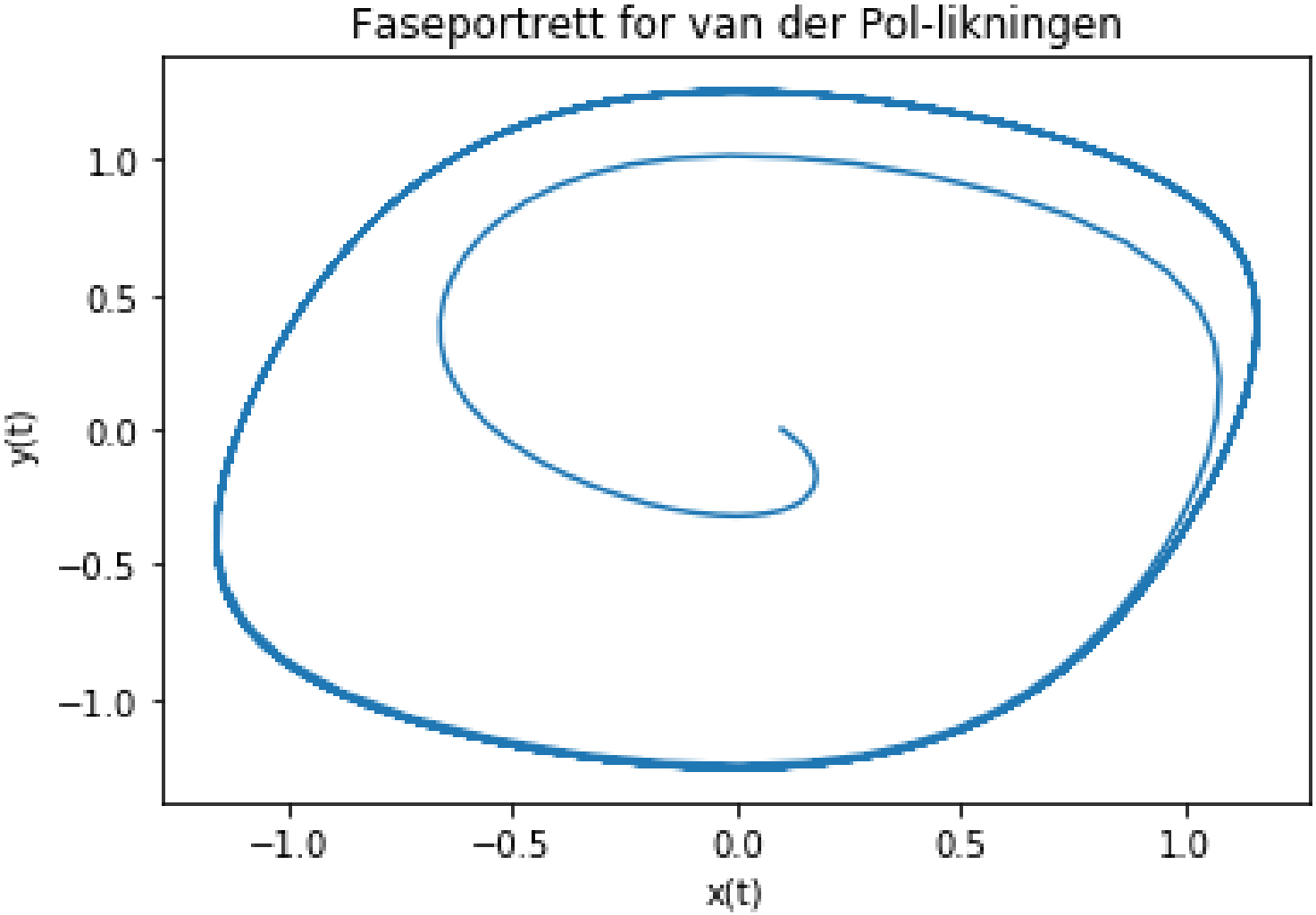
$$
\begin{aligned}
x'(t) &= y(t) - x(t)^3 + x(t) \quad , \quad x(0) = 0.1 \\
y'(t) &= -x(t) \quad , \quad y(0) = 0
\end{aligned}
$$

```
from ODESolver import *
import numpy as np

def f(u,t):
   return np.array([u[1]-u[0]**3+u[0], -u[0]])

metode = RungeKutta4(f)
metode.set_initial_condition(U0=[0.1, 0.0])
timepoints = np.linspace(0, 50, 400)
u,t = metode.solve(timepoints)
```

Løsning


van der Pol-likningen
1.0
0.5
0.0
−0.5
−1.0
x(t) and y(t)
0
10
20
30
40
50
Time (t)

Faseportrett



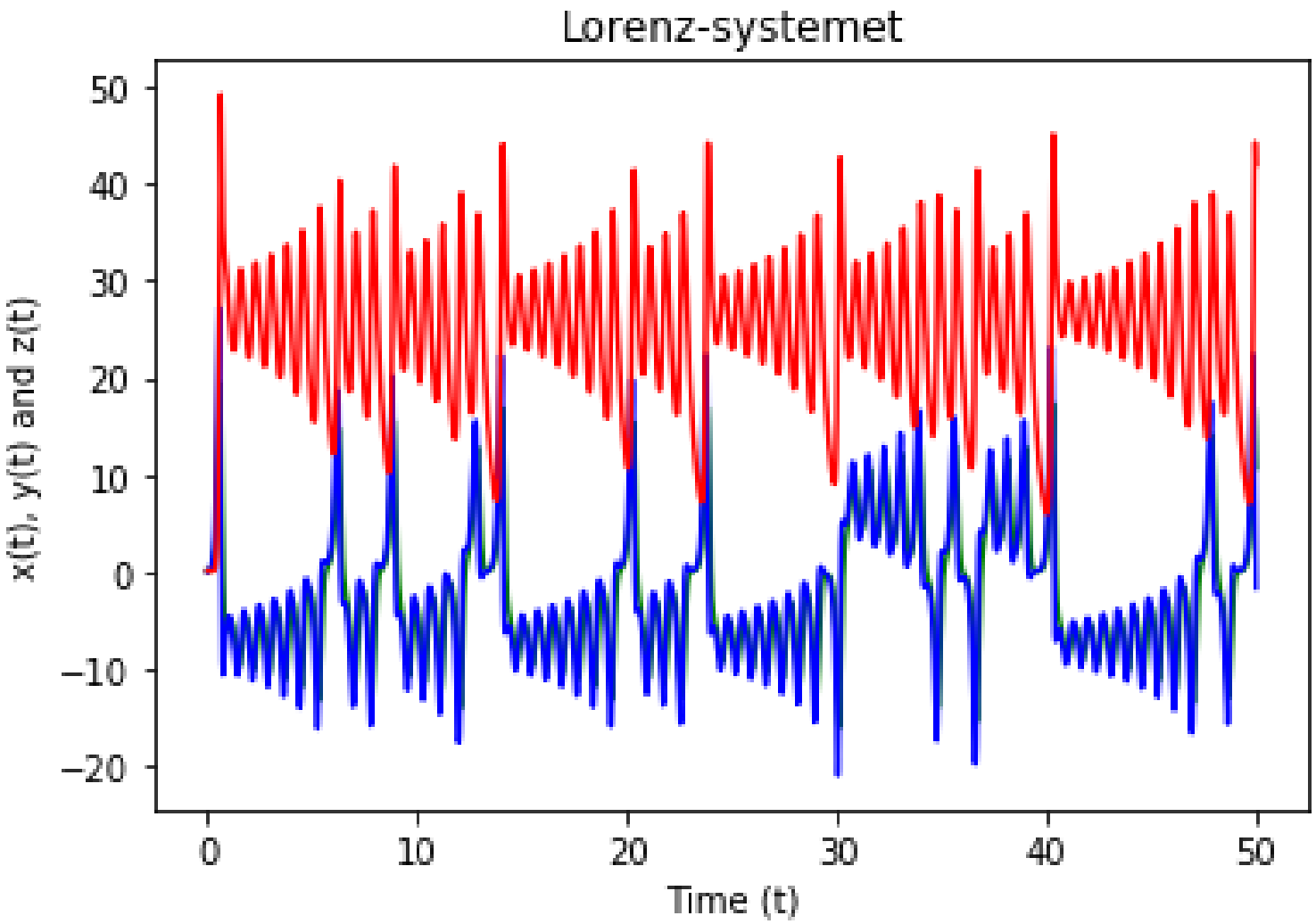
Eksempel 3

ODE-system (*Lorenz-systemet*):

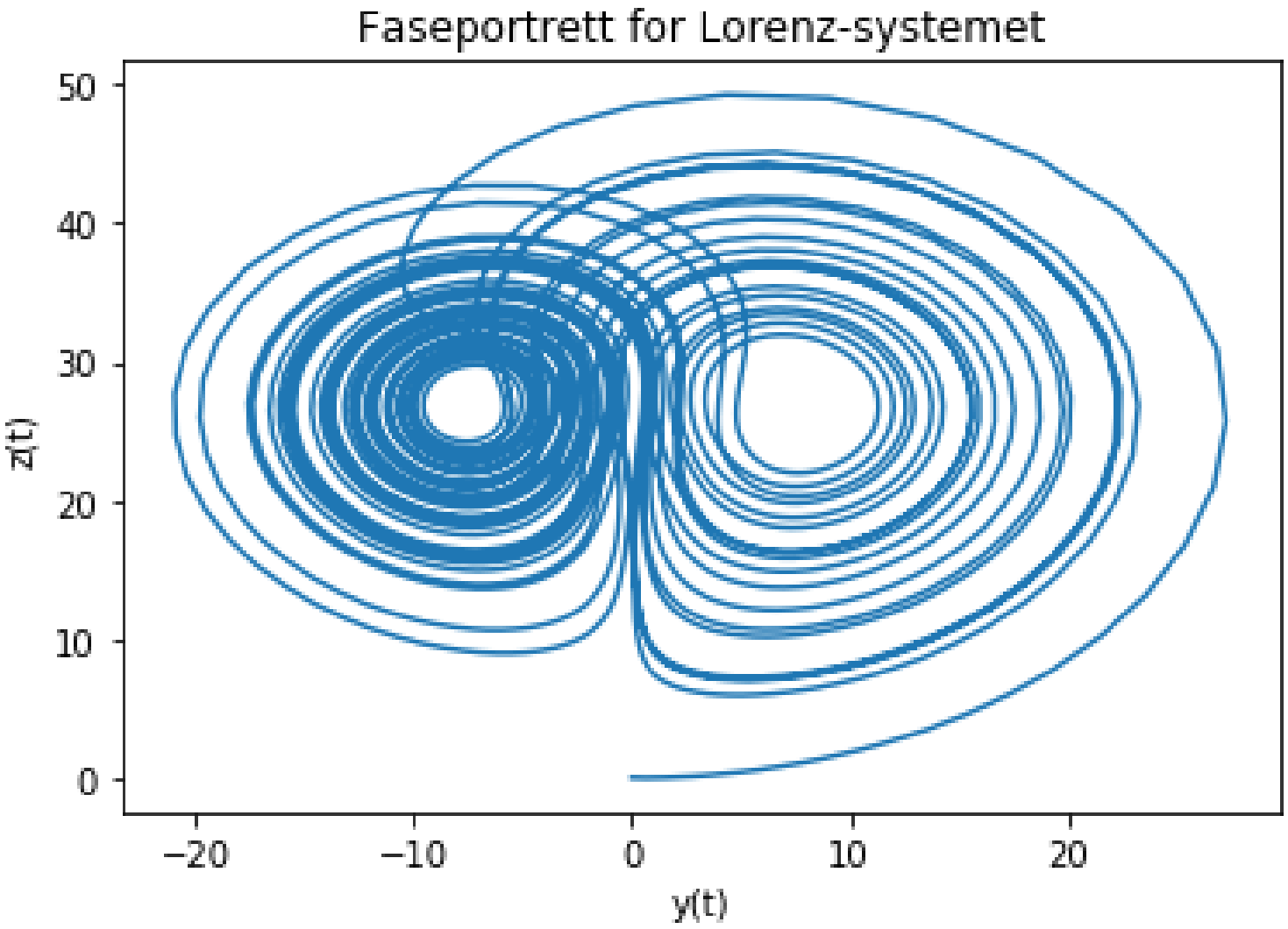
$$
\begin{aligned}
x' &= 10(y - x) \\
y' &= 28x - y - xz \\
z' &= xy - (8/3)z
\end{aligned}
$$

```
from ODESolver import *
import numpy as np

def f(u,t):
    u1 = u[0]; u2 = u[1]; u3 = u[2]
    return np.array([10*u2-10*u1, 28*u1-u2-u1*u3, u1*u2-(8/3)*u3])

# Vi bruker Runge-Kutta metoden
metode = RungeKutta4(f)
metode.set_initial_condition(U0=[0.1, 0.0]))
timepoints = np.linspace(0, 50, 4000)
u,t = metode.solve(timepoints)
```

Løsning


Lorenz-systemet
50
40
30
20
10
0
−10
−20
x(t), y(t) and z(t)
0
10
20
30
40
50
Time (t)

Faseportrett



Gjennomgang av klassen ODESolver

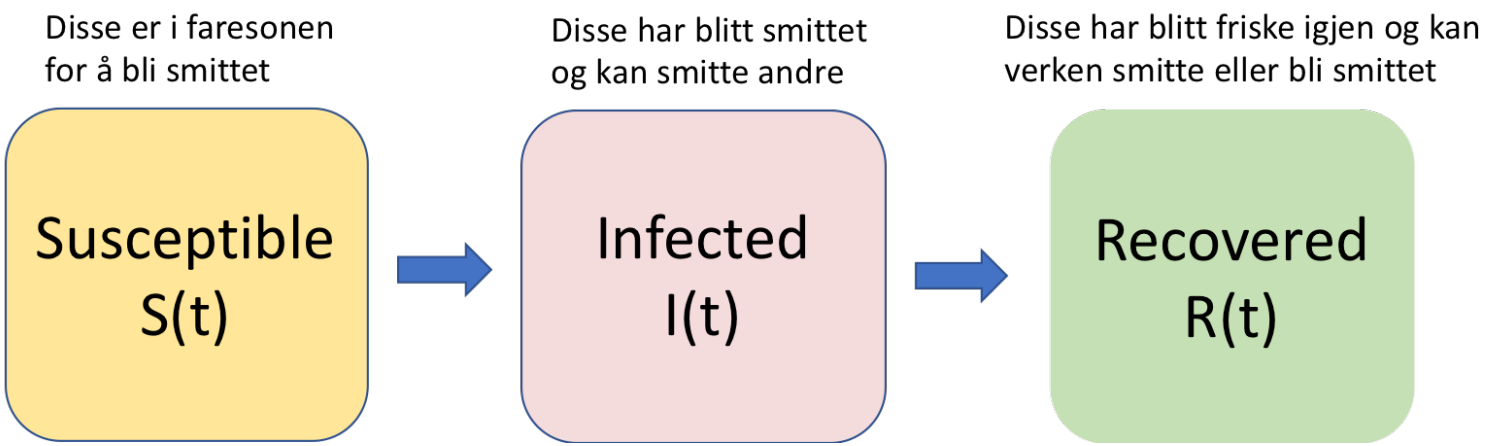
Modulen ODESolver er pensum:

- Du bør forstå innholdet i modulen såpass godt at du kan gjøre enkle modifikasjoner eller utvidelser av den.
- Du skal også kunne anvende modulen til å løse skalare ODE'er og systemer av ODE'er.

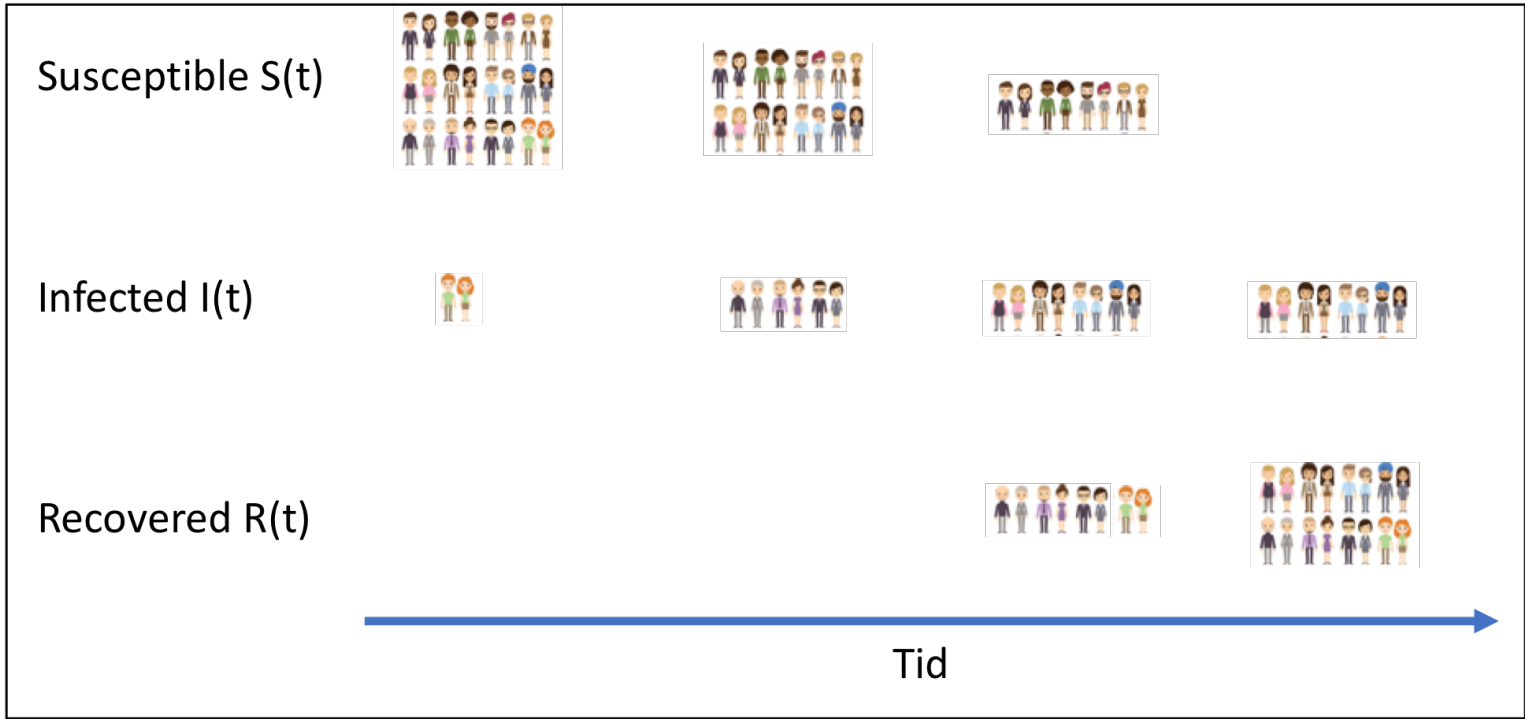
SIR-modeller

Forestill deg at du skal lage en datasimulering over hvordan et utbrudd av en smittsom sykdom utvikler seg over tid. Det er nyttig å holde rede på tre tall på hvert tidspunkt:

- $S(t)$: antall i faresone for å bli smittet
- $I(t)$: antall smittede og smittefarlige
- $R(t)$: antall som har blitt friske igjen



Utvikling av smittsom sykdom over tid



Kermack-McKendrick-modellen

Dette er en modell for hvordan $S(t)$, $I(t)$ og $R(t)$ endrer seg over tid under et sykdomsutbrudd. Modellen består av tre ODE'er:

$$\begin{aligned} S'(t) &= -\beta S(t)I(t) \\ I'(t) &= \beta S(t)I(t) - \nu I(t) \\ R'(t) &= \nu I(t) \end{aligned}$$

Disse likningene sier:

- Antall som smittes ved tid t er $\beta S(t)I(t)$
- Dette kommer til fratrekk på $S(t)$ og tilskudd på $I(t)$
- En viss prosent av de smittede blir friske igjen
- Dette kommer til fratrekk på $I(t)$ og tilskudd på $R(t)$

f(u,t) for SIR-modellen

Siden SIR-modellen har to parametre (β og ν) implementerer vi den best som en klasse:

```
class SIR:
    def __init__(self, beta, nu):
            self.beta = beta
            self.nu = nu

    def __call__(self, u, t):
        beta = self.beta; nu = self.nu
        S = u[0]; I = u[1]; R = u[2]
        f1 = -beta * S * I
        f2 = beta * S * I - nu * I
        f3 = nu * I
        return np.array([f1, f2, f3])
```

Med klassen over kan vi lett lage f(u,t) funksjoner med gitte parameterverdier. Eksempel: hvis vi skal ha beta = 0.1 og nu = 0.01 så kan vi skrive `f = SIR(beta=0.1, nu=0.01)`.

Løsning av SIR-modellen

```
# Definer høyresiden f(u,t) i ODE'en:
class SIR:
    ... se forrige slide ...

# Velg parameterverdier:
f = SIR(beta=0.1, nu=0.01)

# Velg Runge-Kutta til å løse likningene:
metode = RungeKutta4(f)

# Velg initialbetingelser
metode.set_initial_condition(U0=[0, 1, 0])

timepoints = np.linspace(0, 100, 500)
u,t = metode.solve(timepoints)
```