Objektorientert programmering og løsning av ODE'er

Ole Christian Lingjærde, Institutt for Informatikk, UiO

1. november 2021

Agenda for denne og neste uke

- Kjapp repetisjon av sentrale klasse-begreper
- Programmering med klasser og subklasser (OOP)
- Løsing av en differensiallikning i Python
- Løsing av flere differensiallikninger samtidig
- Innføring i modulen ODESolver

Repetisjon av klasser

Klasser kan brukes til å holde på data:

```
class K:
   def __init__(self, a, b):
      self.a = a
      self.b = b
```

Bruk av klassen:

```
p = K(2,6)     # Her setter vi data inn
print(p.a)     # Her henter vi data ut
print(p.b)
```

Repetisjon av klasser

Klasser kan ha flere instanser:

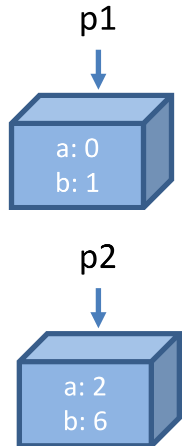
```
# Vi definerer klassen:
class K:
   def __init__(self, a, b):
      self.a = a
      self.b = b

# Vi lager to instanser av klassen:
p1 = K(0, 1)
p2 = K(2, 6)

# Vi ser på innholdet:
print(p1.a)    # Skriver ut 0
print(p1.b)    # Skriver ut 1
print(p2.a)    # Skriver ut 2
print(p2.b)    # Skriver ut 6
```

Merk: totalt fire verdier er lagret!

Repetisjon av klasser

Klasser kan ha flere instanser:

```
# Vi definerer klassen:
class K:
   def __init__(self, a, b):
      self.a = a
      self.b = b

# Vi lager to instanser av klassen:
p1 = K(0, 1)
p2 = K(2, 6)

# Vi ser på innholdet:
print(p1.a)    # Skriver ut 0
print(p1.b)    # Skriver ut 1
print(p2.a)    # Skriver ut 2
print(p2.b)    # Skriver ut 6
```



Totalt fire verdier er lagret!

Repetisjon av klasser

Klasser kan ha flere funksjoner:

```
# Vi definerer klassen:
class K:
   def __init__(self, a, b):
      self.a = a
      self.b = b

   def verdi(self, x):
      return self.a * x + self.b

# Vi lager to instanser:
p1 = K(0, 1)
p2 = K(6, 8)
print(p1.verdi(1))   # Skriver ut 1
print(p2.verdi(1))   # Skriver ut 14
```

Repetisjon av klasser

Viktig: husk å bruke *self*:

```
class K:
   def __init__(self, a, b):
      self.a = a
      self.b = b

   def verdi(x):
      return a * x + b  # Feil (må stå self.a og self.b)
```

Spesialmetoder i Python:

```
a.__init__(self, args)      # Konstruktør
a.__del__(self)             # Destruktør
a.__call__(self, args)      # Funksjonskall
a.__str__(self)             # Tekstrepresentasjon
a.__repr__(self)            # a == eval(repr(a))
a.__add__(self, b)          # a + b
a.__sub__(self, b)          # a - b
a.__mul__(self, b)          # a * b
a.__div__(self, b)          # a / b
a.__pow__(self, b)          # a ** b
a.__lt__(self, b)           # a < b
a.__le__(self, b)           # a <= b
a.__gt__(self, b)           # a > b
a.__ge__(self, b)           # a >= b
a.__eq__(self, b)           # a == b
a.__ne__(self, b)           # a != b
```

Repetisjon av klasser

Du <u>bør</u> nå ha forstått følgende:

- Hvordan lage en enkel klasse
- Hvordan lage en konstruktør
- Hvordan lage instanser av en klasse
- Forstå når konstruktøren utføres
- Hvordan lage ekstra metoder i en klasse
- Hvordan lage spesialmetoder og når de utføres

I fortsettelsen kommer vi til å bruke begrepene over hele tiden. Henger du etter nå, må du oppdatere deg raskt.

Eksempel: komplekse tall

Vi ønsker å definere en klasse `Complex` for komplekse tall.

- De komplekse tallene $x = 1 + 2i$ og $y = 2 + i$ skal kunne lages slik: `x = Complex(1,2)` og `y = Complex(2,1)`
- Løsning: lager en klasse med instansvariabler `real` og `imag`
- Vi ønsker også å kunne summere ved å skrive `x + z`
- Løsning: vi bruker spesialmetoden `__add__(self,z)`

Eksempel (forts.)

La $x = a + bi$ og $z = c + di$ være to komplekse tall. Etter vanlige regneregler er da:

$$x + z \quad = \quad (a + c) + (b + d)i$$

Anta at vi "sitter inni" objektet x.

Vi ser da attributtene `self.real` og `self.imag`.

Får vi tilsendt et annet komplekst tall z kan vi addere dem slik:

```
svar = Complex(self.real + z.real,
self.imag + z.imag)
```

Implementasjon

```python
class Complex:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __str__(self):
        s = f"{self.real} + {self.imag}i"
        return s

    def __add__(self, z):
        real = self.real + z.real
        imag = self.imag + z.imag
        res  =  Complex(real,  imag)
        return res

# Eksempel på bruk:
x = Complex(1,2)
y = Complex(2,1)
z = x + y
print(z)
```

Utvidelse til alle fire regnearter

Vi kan bruke samme triks som over til å implementere alle fire regnearter.

- For å subtrahere: bruk spesialfunksjonen `__sub__`
- For å multiplisere: bruk spesialfunksjonen `__mul__`
- For å dividere: bruk spesialfunksjonen `__div__`

Eksempel B: Python-kode

```
class Complex:
    <Alt tidligere som før>

    def __sub__(self, z):
        real = self.real - z.real
        imag = self.imag - z.imag
        res = Complex(real, imag)
        return res

    def __mul__(self, z):
        real = self.real*z.real - self.imag*z.imag
        imag = self.real*z.imag + self.imag*z.real
        res = Complex(real, imag)
        return res

    def __div__(self, z):
        r = z.real**2 + z.imag**2
        real = (self.real*z.real + self.imag*z.imag)/r
        imag = (self.imag*z.real - self.real*z.imag)/r
        res = Complex(real, imag)
        return res
```

Eksempler på bruk

```
x = Complex(1,1)  # x = 1 + i
y = Complex(2,3)  # y = 2 + 3i

print(x + y)      # 3 + 4i
print(x - y)      # -1 - 2i
print(x * y)      # -1 + 5i
print(x / y)      # 0.384615 + -0.0769231i

print(x * y / y)  # 1 + 1i
```

Funksjoner med parametre

I matematiske funksjoner er det ofte hensiktsmessig å skille mellom *variabler* og *parametre*. Tenk på parametre som konstanter som må gis verdi før utregning.

Eksempel: funksjon med parametre v_0 og $g = 9.81$:

$$f(t; v_0) = v_0 t - \frac{1}{2} g t^2$$

Vi trenger åpenbart både t, v_0 og $g = 9.81$ for å evaluere f, men hvordan programmere det i praksis?

Løsning A

Vi lar t, v_0 og g være argumenter til funksjonen:

```
t = 0.5
v0 = 50.0
g = 9.81
verdi = f(t, v0, g)
```

Kode:

```
def f(t, v0, g):
    return v0*t - 0.5*g*t**2
```

Løsning B

Vi lar *t* og v_0 være argumenter til funksjonen:

```
t = 0.5
v0 = 50.0
verdi = f(t, v0)
```

Kode:

```
def f(t, v0):  g = 9.81
   return v0*t - 0.5*g*t**2
```

Løsning C

Vi lar kun *t* være argument til funksjonen:

```
t = 0.5
verdi = f(t)
```

Men hvordan får vi fortalt programmet hva verdien til *v*0 er, hvis *v*0 ikke er argument til funksjonen? Svar: bruk en klasse.

```
class F:
    def __init__(self, v0):
        self.v0 = v0
        self.g = 9.81

    def __call__(self, t):
        return self.v0*t - 0.5*self.g*t**2

v0 = 50.0
f = F(v0)        # Lag funksjonen f, med v0=50.0

t = 0.5
verdi = f(t)    # Bruk funksjonen f
```

Implementere funksjoner med mange parametre

Gitt en funksjon med $n + 1$ parametre og en uavhengig variabel:

$$f(x; p_0, \ldots, p_n)$$

er det smart å bruke en klasse til å implementere `f`, hvor $p_0, \ldots, p_n$ er attributter i klassen.

```
class MyFunc:
    def __init__(self, p0, p1, p2, ..., pn):
        self.p0 = p0
        self.p1 = p1
        ...
        self.pn = pn

    def __call__(self, x):
        return ...
```

Eksempel: funksjon med fire parametre

$$v(r; \beta, \mu_0, n, R) = \left(\frac{\beta}{2\mu_0}\right)^{\frac{1}{n}} \frac{n}{n+1} \left(R^{1+\frac{1}{n}} - r^{1+\frac{1}{n}}\right)$$

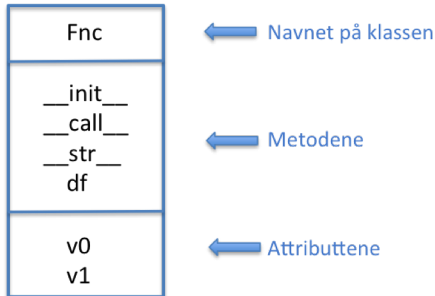
```
class VelocityProfile:
    def __init__(self, beta, mu0, n, R):
        self.beta, self.mu0, self.n, self.R = \
        beta, mu0, n, R

    def __call__(self, r):
        beta, mu0, n, R = \
        self.beta, self.mu0, self.n, self.R
        n = float(n)  # ensure float divisions
        v = (beta/(2.0*mu0))**(1/n)*(n/(n+1))*\
            (R**(1+1/n) - r**(1+1/n))
        return v

v = VelocityProfile(R=1, beta=0.06, mu0=0.02, n=0.1)
print(v(0.1))
```

UML-diagrammer

UML (Unified Modeling Language) er en visuell måte å fremstille og dokumentere datamodeller på.

Vi kan bruke UML-diagrammer til å visualisere innholdet i klasser, og relasjoner mellom klasser.

Eksempel (for klassen vi nettopp definerte):



Objektorientert programmering (OOP)

- Alt i Python er objekter, så teknisk sett er all Python-programmering objektbasert.
- I objektorientert programmering (OOP) går vi ett skritt videre.
- OOP utnytter en svært nyttig egenskap ved klasser: de kan settes sammen som byggeklosser!
- Hvis vi har definert en klasse `class A` så kan vi definere en ny klasse `class B(A)`.
- Da blir klassen `B` en *utvidelse* av klassen `A`
- Vi sier at `B` *arver* data og metoder fra `A`
- Vi sier også at `B` er subklasse av `A`, og at `A` er superklasse til `B`

Prinsipp A: Klasser kan arve fra andre klasser

```
class A:
    def __init__(self, v0, v1):
        self.v0 = v0
        self.v1 = v1

    def f(self, x):
        return x**2

class B(A):
    def g(self, x):
        return x**4

class C(B):
    def h(self, x):
        return x**6
```

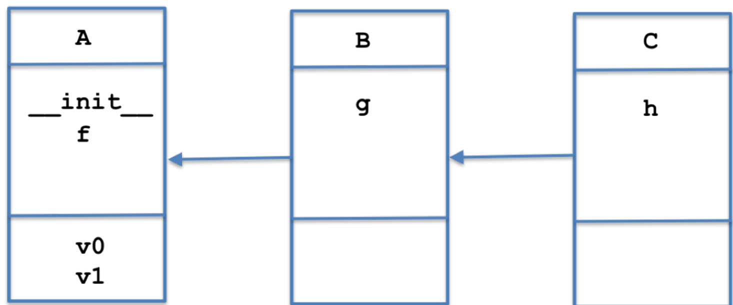
Vi har nå definert tre klasser:

`A` : to attributter (v0, v1) og to metoder `(__init__, f)`

`B` : to attributter (v0, v1) og tre metoder `(__init__, f, g)`

`C` : to attributter (v0, v1) og fire metoder `(__init__, f, g, h)`

UML-diagram



Innholdet i klassene A, B og C

```
# I objekter av A har vi attributtene v0, v1 og metoden f:
p = A(2.7, 5.2)
print(p.v0)          # Utskrift: 2.7
print(p.v1)          # Utskrift: 5.2
print(p.f(3.0))      # Utskrift: 9.0

# I objekter av B har vi det samme + metoden g:
p = B(2.7, 5.2)
print(p.v0)          # Utskrift: 2.7
print(p.v1)          # Utskrift: 5.2
print(p.f(3.0))      # Utskrift: 9.0
print(p.g(3.0))      # Utskrift: 81.0

# I objekter av C har vi det samme + metoden h:
p = C(2.7, 5.2)
print(p.v0)          # Utskrift: 2.7
print(p.v1)          # Utskrift: 5.2
print(p.f(3.0))      # Utskrift: 9.0
print(p.g(3.0))      # Utskrift: 81.0
print(p.h(3.0))      # Utskrift: 729.0
```

Prinsipp B: Subklasser kan overkjøre metoder i superklasser

```
class A:
    def __init__(self, a):
        self.a = a

    def skrivut(self):
        print("Klasse A")

class B(A):
    def __init__(self, b):    # Overkjører __init__ i class A
        self.b = b

    def skrivut(self):        # Overkjører skrivut i class A
        print("Klasse B")

# Eksempler på bruk
p = A(3)       # Lag objekt av superklassen A
p.skrivut()    # Utskrift: "Klasse A"
p = B(4)       # Lag objekt av subklassen B
p.skrivut()    # Utskrift: "Klasse B"
```

Hensikten med å overkjøre metoder

- Subklasser kan brukes for å legge til ny funksjonalitet
- Subklasser kan også brukes for å restriktere funksjonaliteten i klassen det arves fra
- Utskrift og andre funksjoner kan være nødvendig å endre i subklasser
- Praktisk trening er helt nødvendig

Prinsipp C: Overkjørte metoder finnes fortsatt

```
class A:
    def __init__(self, a):
        self.a = a

    def skrivut(self):
        print("Klasse A")

class B(A):
    def __init__(self, a, b):
        A.__init__(self, a)    # Kall __init__ i superklassen
        self.b = b

    def skrivut(self):
        A.skrivut(self)         # Kall skrivut i superklassen
        print("Klasse B")

p = B(3,4)   # Lag objekt av subklassen B
p.skrivut() # Utskrift: 'Klasse A' + linjeskift + 'Klasse B'
```

Prinsipp D: Vi kan ha mange nivåer av subklasser

```
class A:
    def __init__(self, a):
        self.a = a
    def skrivut(self):
        print(f"a = {self.a}")

class B(A):
    def __init__(self, a, b):
        A.__init__(self, a)
        self.b = b
    def skrivut(self):
        print(f"a = {self.a}, b = {self.b}")

class C(B):
    def __init__(self, a, b, c):
        B.__init__(self, a, b)
        self.c = c
    def skrivut(self):
        print(f"a = {self.a}, b = {self.b}, c = {self.c}")

p1 = A(1)
p1.skrivut()      # a = 1
p2 = B(1,2)
p2.skrivut()      # a = 1, b = 2
p3 = C(1,2,3)
p3.skrivut()      # a = 1, b = 2, c = 3
```

Prinsipp E: Vi kan alltid finne ut hvor vi er i hierarkiet

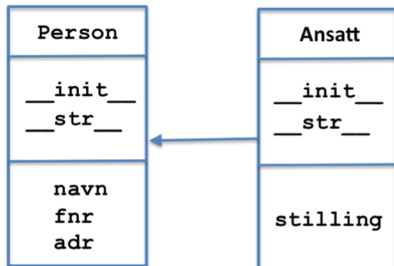
Anta at klassene A, B, C er definert som på forrige slide.

```
# Lag objekter av A og B
p = A(1)
q = B(1,2)

# Hvilke klasser er et objekt en instans av?
isinstance(p, A)   # True
isinstance(p, B)   # False
isinstance(q, A)   # True
isinstance(q, B)   # True

# Hvilken unike klasse tilhører objektet p?
print(p.__class__    == A)    # True
print(p.__class__    == B)    # False
print(q.__class__    == A)    # False
print(q.__class__    == B)    # True

# Alternativ til over
print(p.__class__.__name__    == "A")     #  True

# Er klassen B en subklasse av klassen A?
issubclass(B, A)   # True
issubclass(A, B)   # False

# Finn navnet på superklassen til et objekt
print(q.__class__.__bases__[0].__name__)   # A
```

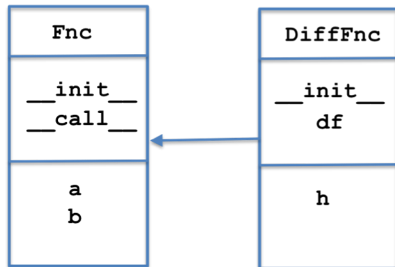
Eksempel A: Person - Ansatt

```
class Person:
    def __init__(self, navn, fnr, adr):
        self.navn = navn
        self.fnr = fnr
        self.adr = adr

    def __str__(self):
        s = f"Navn: {self.navn}\nFnr: {self.fnr}\nAdresse: {self.adr}\
        return s

class Ansatt(Person):
    def __init__(self, navn, fnr, adr, stilling):
        Person.__init__(self, navn, fnr, adr)
        self.stilling = stilling

    def __str__(self):
        s1 = Person.__str__(self)
        s2 = f"Stilling: {self.stilling}\n"
        return(s1 + s2)

p = Person("Rex", "18050012345", "Slottet")
print(p)

p = Ansatt("Rex", "18050012345", "Slottet", "Konge")
print(p)
```

UML-diagram



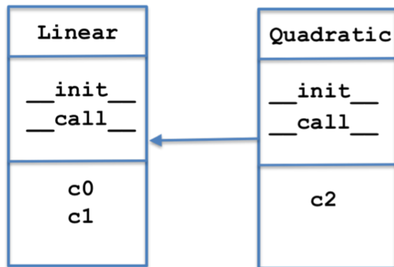
UML-diagram



Eksempel B: Lineært polynom - kvadratisk polynom

```
class Linear:
    def __init__(self, c0, c1):
        self.c0 = c0
        self.c1 = c1

    def __call__(self, x):
        return self.c0 + self.c1*x

class Quadratic(Linear):
    def __init__(self, c0, c1, c2):
        Linear.__init__(self, c0, c1)
        self.c2 = c2

    def __call__(self, x):
        return Linear.__call__(self, x) + self.c2*x**2

# Test av klassene
p = Quadratic(3, 4, 5)
print(p(2.5))           # 44.25
```

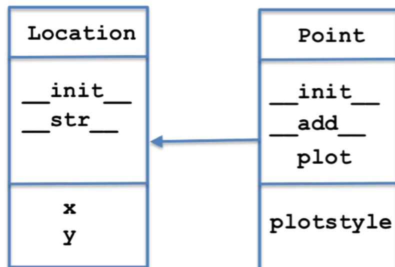
UML-diagram



Eksempel C: Punkt - vektor

```
import matplotlib.pyplot as plt

class Location:
    def __init__(self, x, y):
        self.x = x; self.y = y

    def __str__(self):
        return f"({self.x}, {self.y})"

class Point(Location):
    def __init__(self, x, y, plotstyle="ro"):
        Location.__init__(self, x, y)
        self.plotstyle = plotstyle

    def __add__(self, p):
        xnew = self.x + p.x
        ynew = self.y + p.y
        return Point(xnew, ynew)

    def plot(self):
        plt.plot(self.x, self.y, self.plotstyle)

p1 = Point(3,4); p2 = Point(1,1); p3 = Point(2.5, 1.5)
p4 = p1 + p3
p1.plot(); p2.plot(); p3.plot(); p4.plot()
```

UML-diagram



Større eksempel: klassehierarki for numerisk derivasjon

Hvis funksjonen $f(x)$ er deriverbar i punktet x, så vet vi at

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

og derfor er (for $h > 0$ liten):

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

Formelen over er alt vi trenger for å regne ut deriverte i Python!

Finne den deriverte i et punkt x

```python
# Definer en kvadratisk funksjon
def g(x):
    return x**2 + 5*x + 1

# Definer den (eksakte) deriverte
def dg(x):
    return 2*x + 5

# Definer funksjon som finner numerisk derivert i et punkt
def deriv(f, x):
    h = 1e-5
    return (f(x+h)-f(x))/h

# Sammenlikn løsninger
print(f"Eksakt: g'(0)={dg(0)}    Numerisk: g'(0)={deriv(g,0)}")
```

Finne hele derivertfunksjonen

Løsningen på forrige slide har én svakhet:

Vi finner ikke egentlig derivertfunksjonen $g'(x)$, vi bare regner ut hva den deriverte er i et bestemt punkt x. For hvert nytt punkt x må vi kalle på funksjonen *deriv* og må oppgi navnet på funksjonen som skal deriveres:

```
deriv(g, 0.5)     # Den deriverte i  x=0.5
deriv(g, 1.5)     # Den deriverte i  x=1.5
deriv(g, 4.3)     # Den deriverte i  x=4.3
```

Kan vi i stedet få Python til å finne en funksjon *dg* som oppfører seg akkurat som den deriverte? Vi vil at dette skal virke:

```
dg = Derivative(g)
dg(0.5)  # Den deriverte i  x=0.5
dg(1.5)  # Den deriverte i  x=1.5
dg(4.3)  # Den deriverte i  x=4.3
```

Implementasjon

```
# Lag en klasse som implementerer derivertfunksjonen til f
class Derivative:
  def __init__(self, f, h=1E-5):
    self.f = f
    self.h = float(h)

    def __call__(self, x):
      f, h = self.f, self.h
      return (f(x+h) - f(x))/h

# Lag en konkret funksjon g(x)
def g(x):
  return x**2 + 5*x + 1

# Finn derivertfunksjonen g'(x)
dg = Derivative(g)

# Derivertfunksjonen kan brukes som en vanlig funksjon
dg(0.5)  # Den deriverte i x=0.5
dg(1.5)  # Den deriverte i x=1.5
dg(4.3)  # Den deriverte i x=4.3
```

Utvidelse: flere måter å beregne deriverte på

Formelen vi har brukt for den deriverte er bare en av flere muligheter. Her er noen ulike alternativer:

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

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

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

$$f'(x) \approx \frac{4}{3}\frac{f(x+h) - f(x-h)}{2h} - \frac{1}{3}\frac{f(x+2h) - f(x-2h)}{4h}$$

Vi kan implementere hver av dem som en Derivative-klasse.

Implementasjon

```
class Forward1:
  def __init__(self, f, h=1E-5):
    self.f, self.h = f, h

  def __call__(self, x):
    f, h = self.f, self.h
    return (f(x+h) - f(x))/h

class Central2:
  def __init__(self, f, h=1E-5):
    self.f, self.h = f, h

  def __call__(self, x):
    f, h = self.f, self.h
    return (f(x+h) - f(x-h))/(2*h)

class Central4:
  def __init__(self, f, h=1E-5):
    self.f, self.h = f, h

  def __call__(self, x):
    f, h = self.f, self.h
    return (4/3)*(f(x+h)-f(x-h))/(2*h)-(1/3)*(f(x+2*h)-f(x-2*h))/(4*h)
```

Implementasjon med subklasser

```
class Diff:
  def __init__(self, f, h=1E-5):
    self.f, self.h = f, h

class Forward1(Diff):
  def __call__(self, x):
    f, h = self.f, self.h
    return (f(x+h) - f(x))/h

class Central2(Diff):
   def __call__(self, x):
     f, h = self.f, self.h
    return (f(x+h) - f(x-h))/(2*h)

class Central4(Diff):
   def __call__(self, x):
     f, h = self.f, self.h
    return (4/3)*(f(x+h)-f(x-h))/(2*h)-(1/3)*(f(x+2*h)-f(x-2*h))/(4*h)
```

Løsning av ordinære differensiallikninger

- ODE = Ordinary Differential Equation
- Likning hvor den ukjente er en funksjon $u(t)$
- Differential: knytter sammen $u(t)$, $u'(t)$ (og evt høyereordens deriverte)
- Ordinary: ser bare på deriverte i én variabel (f.eks. t)

Vi kommer til å se på to varianter av ODE'er:

- Skalar ODE: en likning
- Vektor ODE: flere likninger (likningssystem)

Eksempel A

Anta at vi skal finne funksjonen $u(t)$ når vi vet at

$$u'(t) = t^3$$

Vi kan integrere på begge sider:

$$u(t) = \frac{1}{4}t^4 + C$$

Hvis vi i tillegg har en *initialbetingelse* $u(0) = 1$ kan vi finne C:

$$u(t) = \frac{1}{4}t^4 + 1$$

Eksempel B

Anta at vi skal finne funksjonen $u(t)$ når vi vet at

$$u'(t) = f(t)$$

Vi kan igjen integrere begge sider:

$$u(t) = \int f(t)\,dt + C$$

Har vi en initialbetingelse $u(0) = u_0$ kan vi igjen finne C.

Vi har sett tidligere i kurset hvordan man kan regne ut integraler numerisk.

Eksempel C

Eksempel A og B var veldig enkle differensiallikninger, siden vi kunne finne $u(t)$ bare ved å integrere på begge sider. Vi ser nå på en likning hvor u(t) også forekommer i høyresiden:

Anta at vi skal finne funksjonen $u(t)$ når vi vet at

$$u'(t) = \alpha u(t)$$

Vi prøver som før å integrere på begge sider:

$$u(t) = \alpha \int u(t)\, dt + C$$

Vi har funnet et uttrykk for $u(t)$, men høyresiden inneholder den ukjente funksjonen!

Hvordan løse Eksempel C?

Eksempel C er såpass enkel at vi kan løse den matematisk:

Vi skal finne $u(t)$ når

$$u'(t) = \alpha u(t)$$

Vi flytter om:

$$\frac{u'(t)}{u(t)} = \alpha$$

Vi får en smart innsikt og ser at dette kan skrives

$$(\ln u(t))' = \alpha$$

Vi integrerer på begge sider:

$$(\ln u(t)) = \int \alpha \, dt = \alpha t$$

Vi tar $exp(.)$ på begge sider:

$$u(t) = e^{\alpha t}$$