FY1008_M05

November 11, 2025

1 Scipy: Scientific Python

1.0.1 Introduksjon

Relevant informasjon er Sec. 1.5 av Scientific Python lecture notes (se også pdf versjonen). Dessuten, kan følgende videoen være nyttige for å få en introduksjon til SciPy: SciPy Video [ev.også av interesse, NumPy Video i samme serie].

SciPy er en open source module (eller bibliotek) for vitenskaplige og tekniske beregninger. Det tilbyr et bredt spekter av matematiske algoritmer og funksjoner som forenkler komplekse beregninger i felt som fysikk, engineering, statistikk og data analyse. ScyPy ha spesialiserte sub-moduler for optimalisering, integrasjon, interpolasjon, linear algebra med mere.

SciPy er bygget på toppen av NumPy, og denne modulens ndarray blir brukt mye av SciPy. For eksempel, inneholder sub+modulen *scipy.linalg* (se nedfor) mye av den samme funksjonaliteten som *numpy.linalg*.

SciPy er hoved modulen for scientific computing i Python! Det er derfor viktig for en fysikker å kjenne til de sentrale rutinene som denne modulen tilbyr. I denne delen av emnet vil vi derfor diskutere noen av de sentrale rutiner fra denne modulen, men vi kan, selvsagt, ikke diskutere dem alle. **Dog, før du bestemmer deg for å implementere en rutine, er det verdt å undersøke at SciPy ikke allerede tilbyr denne.**

Så hva er fordelen med å bruke SciPy?

- *Omfattende funksjonalitet*
  - SciPy tilbyr spesialiserte moduler for ulike vitenskapelige oppgaver, og det brede verktøysettet lar brukere takle problemer innen fysikk, ingeniørfag, finans og mer, noe som gjør det til en allsidig løsning for mange domener.
- *Bygget på NumPy*
  - Den utnytter kraften og hastigheten til NumPy-arrayer og sikrer effektive numeriske beregninger. Denne integrasjonen muliggjør sømløs håndtering av store datasett og komplekse matematiske operasjoner med høy ytelse.
- *Åpen kildekode og fellesskapsdrevet*
  - Som åpen kildekode drar SciPy nytte av kontinuerlige forbedringer og bidrag fra forskere og utviklere over hele verden.
- Brukervennlighet
  - SciPys veldokumenterte API gjør det tilgjengelig for nybegynnere samtidig som det tilbyr avanserte funksjoner for erfarne brukere.

1.1 Innholdet av SciPy

En detaljer oversikt over sub-moduler i SciPy er som følger i alfabetisk orden:

1. scipy.cluster: Vector quantization / Kmeans
2. scipy.constants: *Physical and mathematical constants*
   - **g**: standard acceleration of gravity (m/s^2)
   - **G**: Newtonian constant of gravitation ($m^3/kg\,s^2$)
3. scipy.datasets: *Collection of test datasets*
4. scipy.differentiate: *performing finite difference numerical differentiation of black-box functions*
   - *derivative*: Evaluate the derivative of a elementwise, real scalar function numerically.
5. scipy.fft: *Fourier Transforms*
   - *fft*: Computes the one dimensional discrete Fourier Transform.
   - *ifft*: Computes the inverse Fourier Transform.
   - *fft2*: Computes the two-dimensional Fourier Transform.
6. scipy.fftpack: *Fourier Transforms*
   - Replaced by scipy.fft (so do not use)
7. scipy.integrate: *Integration*
   - *quad*: Performs adaptive quadrature for definite integrals.
   - *dblquad* and *tplquad*: Compute double and triple integrals.
   - *odeint*: Solves ordinary differential equations (ODEs).
   - *solve_ivp*: Another solver for initial value problems of ODEs with various methods.
8. scipy.interpolate: *Interpolation*
   - *interp1d*: One dimensional interpolation with different methods.
   - *griddata*: Interpolates scattered data points on a grid.
   - *BarycentricInterpolator*: Efficient polynomial interpolation method.
9. scipy.io: *Data input and output*
   - *savemat*: Saves a matrix
   - *loadmat*: Load a matrix
10. scipy.linalg: *Linear Algebra routines*
    - *solve*: Solves linear systems of equations.
    - *inv*: Computes the inverse of a matrix.
    - *eigh*: Computes eigenvalues and eigenvectors of Hermitian matrices.
    - *svd*: Performs Singular Value Decomposition.
11. scipy.ndimage: n-dimensional image package
12. scipy.odr: *Orthogonal distance regression*
13. scipy.optimize: *Optimization*

- *minimize*: Finds the minimum of scalar or multi variable functions using various algorithms.
- *root**: Solves equations or systems of nonlinear equations to find roots.
- *curve_fit*: Performs least squares fitting of a function to data.
- *linprog*: Solves linear programming problems.

14. scipy.signal: *Signal Processing*

    - *convolve*: Computes the convolution of two signals.
    - *butter*: Designs Butterworth filters.
    - *find_peaks*: Detects peaks in data.
    - *spectrogram*: Computes the spectrogram of a signal.

15. scipy.sparse: *Sparse Matrices*

    - *csr_matrix*: Compressed Sparse Row matrix format.
    - *csc_matrix*: Compressed Sparse Column matrix format.
    - *spsolve*: Solves sparse linear systems efficiently.

16. scipy.spatial: *Spatial data structures and algorithms*

17. scipy.special: *Special functions*

    - *erf*: Error function
    - *j0*: Bessel function of the first kind of order 0, $J_0(\cdot)$

18. scipy.stats: *Statistics*

    - *norm*: Normal (Gaussian) distribution functions.
    - *ttest_ind*: Performs t tests on two independent samples.
    - *pearsonr*: Computes Pearson correlation coefficient.
    - *describe*: Summarizes data with descriptive statistics.

1.2 Bruk av SciPy

Vi vil nedenfor gå gjennom noen sentrale funksjoner fra SciPy. Det vil sterkt anbefales at dere også ser på følgende SciPy tutorial. Denne referansen er bare en liten del av Scientific Python Lectures v2025. Dette er en god ressurs som er anbefalt å se på.

For å kunne bruke SciPy må man først importere denne module. Dette gjøres på vanlig måte slik som dette:

[ ]:
```
import scipy as sp           # Vi vil bruke den vanlige kortformen sp for SciPy

# SciPy versjon
print(sp.__version__)
```

[128]:
```
# Senere trenger vi også
import numpy as np
import matplotlib.pyplot as plt
```

Hvordan skal man finne informasjon om innholdet en sub-modul? Her kan man bruke *help()* på vanlig måte

[ ]:
```
help(sp.io)
```

Denne samme informasjonen finner du også online på referansesiden for SciPy. Her er informasjonen presentert på en mer oversiktlig og bedre formatert måte. De andre sub-module kan du nå via menyen på venstre side på denne siden.

1.2.1 scipy.constant

[ ]:
```
g=sp.constants.g

print(f"Tyngde akselerasjonen : {g:.10f} m/s^2")
```

1.2.2 scipy.special

Error funksjonen er definert via integralet

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2)\, dt$$

.

[ ]:
```
x  = 0.5
efunc = sp.special.erf(x)

print(f"sp.special.erf:  erf({x:.4f}) = {efunc:.10f}")

# beregner integralet direkte
#efunc_int = sp.integrate.quad(lambda t: np.exp(-t*t),0,x)
#print(f"Via integrering: erf({x:.4f}) = {efunc:.10f}")
```

Det finnes mange andre spesial funksjoner i *scipy.special*, som Gamma, funksjoner, bessel funksjoner, etc. For en oversikt se dokumentasjonen.

1.2.3 scipy.linalg

La oss beregne determinanten til en 3×3 matrise. Dette gjorde vi tidligere med NumPy, men det kan også gjøres via SciPy (selvsagt med samme resultat).

[ ]:
```
import numpy as np

# Definer matrisen
A = np.array([[1, 2, 3],
              [0, 1, 4],
              [5, 6, 0]])

# Numpy beregning
detA_np = np.linalg.det(A)

# Tilsvarende SciPy beregning
detA_sp = sp.linalg.det(A)
```

```
print(f"NumPy:  det(A) = {detA_np:.8f}")
print(f"SciPy:  det(A) = {detA_sp:.8f}")
```

At både NumPy og SciPy tilbyr den samme funksjonaliteten når det kommer til lineær algebra, er nok bare gjort av praktiske årsaker. Utover lineær algebra (og enkelte andre rutiner), er det ikke mange funksjoner som er de samme i de to modulene.

Løsning av det linear systemet av likninger.

[ ]:
```
A = np.array([[1, 2],
              [2, 3]])
b = np.array([14, 23])
x = sp.linalg.solve(A, b)

print("Solution x = ", x)

# Test av resultatet....
np.allclose(A @ x, b)
```

Det er typisk ikke lurt numerisk å beregne løsningen av $A\vec{x} = \vec{b}$ som $\vec{x} = A^{-1}\vec{b}$ da beregningen av den inverse matrisen typisk ikke er numerisk stabil.

1.2.4 scipy.interpolate

Interpolasjon betegner prosessen for å tilnærme en funksjon i et punkt ved hjelp av kjente funksjonsverdier i nærliggende punkter. Man bruker altså kjente funksjonsverdiene i et sett av punkter for å tilnærme ukjente funksjonsverdier mellom disse punktene. Om man søker en tilnærming *utenfor* de kjente punktene kalles prosessen for *ekstrapolasjon*.

[ ]:
```
# Ex 1
# ------------------

# Genererer utgangsdata for x og y = x^2
xmax = 10
x = np.arange(0, xmax)
y = x**2

# plot data
plt.plot(x,y,'*')
plt.xlabel(r'$x$')
plt.ylabel(r'$y=x^2$')
plt.show()
```

La oss så interpolere $y(x)$ hvor vi antar at funksjonen ikke er kjent, men bare funksjonsverdiene generert overfor.

[ ]:
```
# Finn syntax
help(sp.interpolate.interp1d)
```

[ ]:
```
# hva er y(x_ipol)?
# Lage en interpolerende funksjon basert på våre data.
y_ip_func = sp.interpolate.interp1d(x, y)

# generate 5 random punkter
xnew = xmax * np.random.uniform(size=5)
ynew = y_ip_func(xnew)

# Print results
print(xnew)
print(ynew)

# Plot resultater
plt.plot(x,y,'*')
plt.plot(xnew,ynew,'o')
plt.xlabel(r'$x$')
plt.ylabel(r'$y=x^2$')
plt.show()
```

[ ]:
```
# Ex 2:
# ------------------------

# Genererer utgangsdata for x og y
xmax = 4
x = np.linspace(-xmax, xmax, 15)
y = sp.stats.norm(loc=0, scale=1).pdf(x) # standard normal distribution

# Interpolating function
y_ip_func = sp.interpolate.interp1d(x, y)

# plot data
plt.plot(x,y,'*')
plt.show()
```

La oss interpolere disse dataene.

[ ]:
```
# plot original data
plt.plot(x,y,'*',label='original data')
plt.xlabel(r'$x$')
plt.xlabel(r'$y(x)$')

xnew = np.linspace(-xmax, xmax, 50)

# Interpoler via method='nearest'
method ='nearest'
y_ip_func = sp.interpolate.interp1d(x, y, kind=method)
ynew = y_ip_func( xnew)
plt.plot(xnew,ynew,'-',label=method)
```

```
plt.legend()
plt.show()
```

Andre interpoleringsmetoder....

[ ]:
```
# plot original data
plt.plot(x,y,'*',label='original data')
plt.xlabel(r'$x$')
plt.xlabel(r'$y(x)$')

xnew = np.linspace(-xmax, xmax, 50)

for method in ['nearest','linear','quadratic', 'cubic']:
    y_ip_func = sp.interpolate.interp1d(x, y, kind=method)
    ynew = y_ip_func( xnew)
    plt.plot(xnew,ynew,label=method)
# end for

plt.legend()
plt.show()# plot original data
plt.plot(x,y,'*',label='original data')
plt.xlabel(r'$x$')
plt.xlabel(r'$y(x)$')

xnew = np.linspace(-xmax, xmax, 50)

for method in ['nearest','linear','quadratic', 'cubic']:
    y_ip_func = sp.interpolate.interp1d(x, y, kind=method)
    ynew = y_ip_func( xnew)
    plt.plot(xnew,ynew,label=method)
# end for
plt.legend()
plt.show()
```

Disse ulike metoden kalles ofte for *splines*.

Ex: Smoothing Spline (fra extern kilde)

[ ]:
```
# Lag data med støy
measured_time = np.linspace(0, 2*np.pi, 20)
function = np.sin(measured_time)
noise = np.random.normal(loc=0, scale=0.1, size=20)
measurements = function + noise

# Plot
plt.plot(measured_time, measurements, '*',label='measurements')
plt.plot(measured_time, function, '--',label='underlying')
plt.legend()
```

[ ]:
```
# smoothing spline
smoothing_spline = sp.interpolate.make_smoothing_spline(measured_time,␣
↪measurements)
interpolation_time = np.linspace(0, 2*np.pi, 200)
smooth_results = smoothing_spline(interpolation_time)

# Plot
plt.plot(measured_time, measurements, '*',label='measurements')
plt.plot(measured_time, function, '--',label='underlying')
plt.plot(interpolation_time, smooth_results, '-',label='interpolating spline')
plt.legend()
```

Siden vi har langt mer punkter i den *interpolating spline*, kan man bruke denne til å regne ut den deriverte via finite differences.

I neste eksempel skal vi se på noe som på engelsk kalles *Univariate Spline*. Det er en 1D utjevningsspline som tilpasser en gitt gruppe datapunkter. scipy.interpolate.UnivariateSpline brukes til å tilpasse en spline y = spl(x) av grad k til de oppgitte x, y-dataene. Parameteren s spesifiserer antall knuter ved å spesifisere en utjevningsbetingelse.

scipy.interpolate.UnivariateSpline.set_smoothing_factor: Spline-beregning med den gitte utjevningsfaktoren s og med knutene funnet ved siste kall.

[ ]:
```
# Generate data
x = np.linspace(-3, 3, 50)
y = np.exp(-x**2) + 0.1 * np.random.randn(50)
plt.title("Univariate Spline")
plt.plot(x, y, 'g.', ms=8)

# Using the default values for the
# smoothing parameter
spl = sp.interpolate.UnivariateSpline(x, y)
xs = np.linspace(-3, 3, 1000)
plt.plot(xs, spl(xs), 'blue', lw=2)
#plt.plot(xs, np.exp(-xs*xs), '-','blue', lw=2)

# Manually change the amount of smoothing
spl.set_smoothing_factor(0.5)
plt.plot(xs, spl(xs), color='red', lw=2)
plt.show()
```

Ex: sp.interpolate.make_smoothing_spline

[ ]:
```
# Lag støyetete data
measured_time = np.linspace(0, 2*np.pi, 20)
function = np.sin(measured_time)
noise = np.random.normal(loc=0, scale=0.1, size=20)
measurements = function + noise
```

[ ]:
```
#
smoothing_spline = sp.interpolate.make_smoothing_spline(measured_time,␣
↪measurements)
interpolation_time = np.linspace(0, 2*np.pi, 200)
smooth_results = smoothing_spline(interpolation_time)
```

[ ]:
```

```

1.2.5 scipy.optimize

Optimalisering betegner prosessen med med å finne *maksimum* og/eller *minimum* av en funksjon over et interval eller region. Mer spesielt kan dette brukes bl.a. for å finne nullpunkter for likninger eller systemer av slike, samt til mer generell kurvetilpasning.

Dette er et viktig felt, som det brukes mye resurser på å løse. Dog finnes ikke noen generell løsning på problemet. Hvor kan du tenkte deg at denne type informasjon brukes?

Ex 01: Nullpunkter: For å illustrer dette, la oss finne nullpunktet av en triviell kvadratisk funksjon

$$f(x) = (x-1)(x-2) = x^2 - 3x + 2.$$

Dens deriverte er $f'(x) = 2x - 3$.

Vi starter med å definere disse som funksjoner:

[193]:
```
def f(x):
    res = (x-1)*(x-2)
    return res
# end function

def dfdx(x):
    res = 2*x-3
    return res
# end function
```

La oss så forsøke å finne nullpunktene til $f(x)$ ved hjelp av *scipy.optimize*. Vi vil bruke funksjonen scipy.optimize.root_scalar for å finne røttene.... [ev. bruk help(scipy.optimize.root_scalar)].

Siden funksjonen er ikke-lineær, må vi ha en første gjetning for løsningen. La oss først plotte funksjonen $f(x)$ for å se tingenes tilstand:

[ ]:
```
x=np.linspace(-0.5,4)
plt.plot(x,f(x),x,0*x,'--')
plt.show()
```

[ ]:
```
# Gjetning
x0 = 0
```

La oss først forsøke å finne roten UTEN å bruke informasjon om den deriverte; her brukes secant metoden (quasi-Newton method). Dernest bruker vi den deriverte for å finne roten; her brukes newtons metoden.

[ ]:
```
# Nullpunkts beregning (UTEN derivert)
res = sp.optimize.root_scalar(f, x0=x0, x1=1.5)
# Nullpunkts beregning (MED derivert)
#res = sp.optimize.root_scalar(f, x0=x0, fprime=dfdx)

# Resultat
print( res )
print ("\n")   # blank linje

# Print resultat
if (res.converged):
    print ("RESULTAT:")
    print(f"Nullpunkt i x = {res.root:.8f}, funksjons verdi f(x)={f(res.root):.
 ↪8f}")
else:
    print("ERROR:Ingen konvergens")
# end if
```

Ex2: en ikke triviell funksjon

La oss se på funksjonen:

$$f(x) = x^2 + 10\sin(x).$$

Hva er er minimumspunktet for denne funksjonen?

[220]:
```
# funksjonens definisjon
def f(x):
    return x**2 + 10*np.sin(x)
# end function
```

La oss plotte funksjonene i et passe område!

[ ]:
```
x = np.arange(-5, 5, 0.1)
plt.plot(x, f(x))
```

La oss finne minimumspunktet for denne funksjonen ved hjelp av rutinen sp.optimize.minimize_scalar.

[ ]:
```
res = sp.optimize.minimize_scalar(f, bounds=(-2, -1))
res
```

Om man er i flere dimensjoner er ofte funksjonen scipy.optimize.minimize hva man trenger.

Kurve tilpasning

Kurvetilpasning er også en form for minimering. Man forsøker å bestemme parametrene slik at differansen mellom en funksjon og dataene blir minst mulig. Dersom vi betegner (måle) dataene ved $y(x)$, og funksjonen man ønsker å bestemme for $f(x|p)$ hvor p er en vektor som inne holder

parametrene, kam man definere

$$\chi^2(p) = \sum_{n=1}^{N} |f(x_n|p) + y(x_n)|^2$$

Her er N antall data punkter ved x_n som man har målt. Med å minimere $\chi^2(p)$ med hensyn på p finner man verdiene p_0 slik at $f(x|p_0)$ best tilpasser måledataene.

For dette formålet vil vi bruke funksjonen scipy.optimize.curve_fit. La oss illustrere dette via et eksempel.

La oss anta signalet:

$$s(x) = a\cos(bx),$$

hvor a og b er parametre samt at vi vil legge på støy på dette signalet for å simulere målte data. Dernest vil vi forsøke å tilpasse disse "måledataene".

[ ]:
```
# genererte data
x = np.linspace(-5, 5, num=50)  # 50 values between -5 and 5
a, b = 2.9, 1.5
signal =  a * np.cos(b * x)
noise  = 0.15 * np.cos(100 * x)  # High-freq noise
y      = signal + noise

# Plot
plt.plot(x, signal, '--',label='signal')
plt.plot(x, y, '*',label='noisy signal')
plt.legend()
```

La oss gjennomføre kurve tilpasningen for funksjonen

$$f(x) = a\sin(bx + c),$$

hvor a, b, og c er parametrene som man ønsker å tilpasse!

Vi må studere hvordan man definerer tilpasningsfuksjonen (se dokumentasjonen under scipy.optimize.curve_fit).

[ ]:
```
def f(x,a,b,c):
    res = a * np.sin( b * x + c)
    return res
# end function

#
# eller
#

# funksjon    ( .... som bruker Arbitrary Positional Arguments)
def ff(x,*params):
    res = params[0] * np.sin( params[1] * x + params[2])
```

```
    return res
# end function
```

[ ]:
```
# Det 2. output argumentet er kovariansen (brukes til å sette usikkerheter på␣
 ↪de tilpassede dataene)
params, _ = sp.optimize.curve_fit(f, x, y)

# Er resultat korrekt?
aa,bb,cc = params
tilpasning = f(x,aa,bb,cc)

# Plot
plt.plot(x, signal, '.',label='signal')
plt.plot(x, y, '*',label='noisy signal')
plt.plot(x, tilpasning,label='tilpasning')
plt.plot(x, tilpasning -y, '--',label='feil')

plt.legend()
```

[217]:
```
# Teoretiske verdier
ref = [a, b, np.pi/2]  # what we'd expect
```

[ ]:
```
# Er parameter verdiene riktige?
print( " Tilpasset    = ", params )
print( " Teoretisk    = ", ref )


np.allclose(params, ref )

# Set rtol
#np.allclose(params, ref, rtol=1e-2)
```

[ ]:
```
# Hva om vi bruker tilpasningsfunksjonen ff istedet for f?
params, _ = sp.optimize.curve_fit(ff, x, y)     # Vi bruker nå ff

# Er resultat korrekt?
aa,bb,cc = params
tilpasning = f(x,aa,bb,cc)

# Plot
plt.plot(x, signal, '.',label='signal')
plt.plot(x, y, '*',label='noisy signal')
plt.plot(x, tilpasning,label='tilpasning')
plt.plot(x, tilpasning -y, '--',label='feil')

plt.legend()
```

1.2.6 scipy.stats

Denne sub-modulen inneholder ulike rutiner som er nyttige for å gjøre statistikk. F.eks. inneholder den rutiner for mange ulike *probability distribution functions* (PDFs) for kontinuerlige, multivariate (flere variable) og diskrete fordelinger. Videre er det rutiner for å generere random tall tatt fra ulike fordelinger. Dette har noe overlap med vårt tidligere bekjentskap *np.random*, men *sp.stat* er langt mer omfattende.

I tillegg har selvsagt ulike rutiner for å regne ut middelverdier, momenter, etc. samt hypotesetestesting.

Nedenfor gir vi bare noen få eksempler.

[ ]:
```
# PDF for en Uniforme fordeling på intervallet [a,b]
# Notice the syntax here

# parameter
a=2
b=4

x =np.linspace(a-0.5,b+0.5,100)


pdf = sp.stats.uniform(loc=a,scale=(b-a)).pdf(x)
#pdf = sp.stats.uniform.pdf(x,loc=a,scale=(b-a))

# ev
U = sp.stats.uniform(loc=a,scale=(b-a))
PDF = U.pdf(x)

plt.plot(x,pdf,x,PDF,'.')
```

[ ]:
```
# Generate tilfeldige (random) tall fra denne fordelingen

rnd = sp.stats.uniform(loc=a,scale=(b-a)).rvs(size=1000)
RND = U.rvs(size=1000)

plt.plot(rnd)
# plt.plot(RND)
```

[ ]:
```
# Kontroller fordelingen
plt.plot(x,pdf)
plt.hist(rnd, density=True, bins='auto', histtype='stepfilled', alpha=0.2);
 ↪# Merk deg ";" på slutten av linjen
```

[ ]:
```
# hva med middelverdi etc.?
loc, scale = sp.stats.uniform.fit(rnd)

print(" middelverdi (num)  = ", loc+scale/2, " teoretisk : ", (b+a)/2)
```

```
print(" bredde (num)       = ",       scale, " teoretisk : ", b-a)
```

For enkel bruk, finner jeg personlig *np.random* mer intuetiv å bruke enn *sp.stats*.

1.2.7 scipy.integrate

Integrasjons modulen innholder metoder for å regne ut integraler i en og flere dimensjoner, samt for å løse differential likninger. Vi vil nå gi eksempler på dette.

Integrasjon av kjente funksjoner

Her ønsker man å beregne

$$I(a,b) = \int_a^b dx\, f(x),$$

hvor funksjonen $f(x)$ er en kjent analytisk funksjon. Slike integraler beregnes typisk via rutinen *sp.integrate.quad* som er en wrapper til det klassiske FORTRAN biblioteket QUADPACK.

Om integranden $f(x)$ er "well-behave'' over intervallet $[a,b]$ gjøres beregning som følger:

[ ]:
```
# Integranden
def f(x):
    res = np.exp(-x)
    return res
# end function

# Integrasjons grenser
a=0
b=2

# Integrate
Int = sp.integrate.quad(f,a,b)              # som tuple
# Int, error = sp.integrate.quad(f,a,b)     # som floats

# Inspeksjon
ref = np.exp(-a) - np.exp(-b)

print(" I(a,b)    = ", Int[0], "     feil estimat ", Int[1] )
print(" Analytisk : ", ref )
print(" Korrekt resultatet : ", np.allclose( Int[0],ref) )
```

Spørsmål: Hva blir beregnet i følgende eksempel?

[ ]:
```
Int = sp.integrate.quad(f,0,np.inf)
print(" I(a,b)    = ", Int[0], "     feil estimat ", Int[1] )
```

Merk deg at vi kan også kan bruke lambda funksjoner ifm integrasjonen, noe gir kortere kode. Følgende eksempel viser dette:

[ ]:
```
sp.integrate.quad(lambda x: np.sin(2*np.pi*x),0.,0.5)
```

Hva nå om funksjonen vi ønsker å integrere er definert via noen parametre. Hva kan vi gjøre da?

La oss se på funksjonen

$$g(x) = \gamma \exp[-\beta x].$$

Se dokumentasjonen scipy.integrate.quad.

[279]:
```
# definisjon av funksjoner
def g(x, gamma, beta):
    res = gamma * np.exp( - beta * x)
    return res
# end function

def gg(x, *args):
    res = args[0] * np.exp( - args[1] * x)
    return res
# end function
```

[ ]:
```
# Integrasjons grenser
a=0
b=2

# parameter verdier
gamma = 3
beta  = 2

# Integrate
Int = sp.integrate.quad(g, a,b,args=(gamma,beta,))                # bruker g
#Int = sp.integrate.quad(gg,a,b,args=(gamma,beta,))               # bruker gg␣
 ↪(skal gir samme resultat)

# Inspeksjon
ref = (gamma/beta) * ( np.exp(-beta*a) - np.exp(-beta*b) )

print(" I(a,b)    = ", Int[0], "     feil estimat ", Int[1] )
print(" Analytisk : ", ref )
print(" Korrekt resultatet : ", np.allclose( Int[0],ref) )
```

I tillegg til *scipy.integrate.quad* rutinen, finnes det flere ulike integrasjon rutiner i scipy.integrate, men vi vil ikke diskutere dem her.

Om det er en vektor funksjon man ønsker å integere, kan man bruke rutinen *scipy.integrate.quad_vec*.

Hva om integranden ikke er definert for enkelte verdier? Man sier at integranden er *singulær* i et slikt punkt og den kan ikke beregnes i et slikt punkt. I enkelte tilfeller eksisterer allikevel integralet over et interval som innholder et singulært punkt.Dog er dette et unntak, mer enn en regel!

For å illustrere dette skal vi se på funksjonen

$$s(x) = \frac{1}{\sqrt{|1-x|}}$$

Denne funksjonen har et singulært punkt i $x_s = 1$, og $s(x_s)$ eksisterer derfor ikke.

Hvordan kan man beregne integralet (som er vell-definert)

$$I_s = \int_0^2 dx\, s(x).$$

[ ]:
```
def s(x):
    #res =np.log( np.sin(x) )
    #res = 1/np.sqrt(1-x)
    res = 1/np.sqrt( np.abs(1-x) )
    return res
# end function

x =np.linspace(0,2,500)
plt.plot(x,s(x))
```

[ ]:
```
# Hva om vi forsøker?  Hva skjer tror du.....
Int = sp.integrate.quad(s, 0, 2)
```

Man må spesifisere hvor det singulære punktet er, slik at rutinen kjenner til dette og ikke evaluerer integranden i dette punktet.... Igjen for å finne ut hvordan dette skal gjøres må man studere dokumentasjonen for scipy.integrate.quad.

[315]:
```
# Det singulære punktet
xs=1.

Int = sp.integrate.quad(s, 0,2, points=xs )
```

Merk deg at om det "problematiske'' punktet er ett av endepunktene av integrasjons intervallet, vil ofte integrasjonensrutinen *quad* greier å håndtere dette automatisk

[ ]:
```
Int1 = sp.integrate.quad(s, 0, 1)
Int2 = sp.integrate.quad(s, 0,1, points=xs )

print(" Er resultatene like : ", np.allclose( Int1[0], Int2[0]) )
```

Integrasjon i høyere dimensjoner

For eksempel i 2D er et slikt integral

$$I = \int_{a_y}^{b_y} \int_{b_x}^{b_x} f(x,y)\, dxdy.$$

Dersom $f(x,y) > 0$ kan I tolkes som volumet begrenset av flaten $f(x,y)$ og integrasjonsområdet.

[ ]:
```
# enkel funksjon
f = lambda x, y: 2

ax, bx = 0, 2
ay, by = 0, 1

# Forventet resultat
ref = f(0,0)*bx*by

Int = sp.integrate.dblquad(f, 0, 2, 0, 1)

print(" I         = ", Int[0], "     feil estimat ", Int[1] )
print(" Analytisk : ", ref )
print(" Korrekt resultatet : ", np.allclose( Int[0],ref) )
```

Et annet eksempel er

$$I = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \exp[-x^2 - y^2]\, dxdy.$$

[ ]:
```
# et annet eksempelØ Gaussisk funksjon
f = lambda y, x: np.exp(-(x ** 2 + y ** 2))

Int = sp.integrate.dblquad(f, -np.inf, np.inf, -np.inf, np.inf)

# Forventet resultat
ref = np.pi

print(" I         = ", Int[0], "     feil estimat ", Int[1] )
print(" Analytisk : ", ref )
print(" Korrekt resultatet : ", np.allclose( Int[0],ref) )
```

Merk deg at fuksjonen som definerer integranden har y før x, dvs. den er definert som $f(y, x)$!

Integrering av ordinære differentiallikninger (ODEer)

Differentiallikninger er viktige i mange ulike grener av fysikken. Vi skal nå se på hvordan vi kan numerisk løse ODEer. Dette har vi allerede sett på i flere øvinger.

Som du muligens husker, er prosedyren å omforme den høyere ordens differential likningen til et *system av første ordens* ODEer. Dette gjøres ved å innføre nye avhengige variable. Ordenen til en ODE er gitt av den høyeste deriverte som opptrer.

Andre ordens ODEer er spesielt viktig i fysikken, bl.a. da Newtons andre lov er av denne typen. Vi vil derfor behandle dette tilfellet her. La oss anta at $\mathbf{r}(t)$ er den ukjente posisjonen i vårt problem. For eksempel for å kunne bruke *scipy.solve_ivp* må den andre ordens ODEen omskrives til et system av første ordens ODEer. Dette gjøres typisk ved å innføre nye variable for $\dot{\mathbf{r}}(t)$. For å illustrere dette, la oss anta at vi jobber i to dimensjoner slik at $\mathbf{r}(t) = (x(t), z(t))$. Dersom vi definerer (i

dokumentasjonen for *scipy.solve_ivp*):

$$\mathbf{u}(t) = \begin{pmatrix} x(t) \\ z(t) \\ \dot{x}(t) \\ \dot{z}(t) \end{pmatrix}.$$

Med denne definisjonen brukes den opprinnelige ODEen til å definere en vektor funksjon $\mathbf{f}(t, \mathbf{u})$ slik at

$$\dot{\mathbf{u}}(t) = \mathbf{f}(t, \mathbf{u}).$$

Etter dette kan man bruke scipy.solve_ivp til å løse differensiallikningen.

Merk: ODEen trenger ikke å være lineær.

1.2.8 scipy.differentiate

Denne pakken kan brukes til å regne ut den deriverte av ulike funksjoner. I bunn og grunn er beregningen basert på finite-difference approksimasjonen. La oss illustrere dette med noen eksempler.

[53]:
```
#import numpy as np
#import scipy as sp
#import matplotlib.pyplot as plt

#
def f(x):
    return x**3 - 1
# end function

def dfdx(x):
    return 3*x**2
# end function
```

Slik ser funksjonen ut:

[ ]:
```
# Beregn f(x)
N = 1000
x = np.linspace(0,5,N)
y = f(x)

plt.plot(x,y)
```

La oss så generere den deriverte av $f(x)$ ved å bruke scipy.differentiate.derivative

[ ]:
```
# Den deriverte
xx = np.linspace(1,4,25)
#xx = 1
fprime = sp.misc.derivative(f,xx,dx=1E-8)
```

[54]:
```
# ADVARSEL

#from scipy.differentiate import derivative
#from scipy.misc import derivative

# ALLTID controller at du leser dokumentasjonen for den versjonen av SCIPY du␣
 ↪bruker.....

# Se hva som fungere for deg her!
```

[ ]:
```
fprime
```

[ ]:
```
plt.plot(x,dfdx(x))
plt.plot(xx,fprime,'ro')
```

1.2.9 scipy.fft

En Fourier transformasjon brukes for å studere frekvensinnholdet i et signal. I musikk refererer "A" ofte til A4-noten, som har en standardisert frekvens på 440 Hz (svingninger per sekund). Som kjent er enheten for frekvens Hertz (Hz).

Om man tar opp lyden av en F spilt på et orgel, vil man få en tidsserie. Spørsmålet er hvilken frekvens denne tonen svarer til? Dette kan man finne ut ved å benytte en Fourier transform (FT) som ble utviklet av den franske fysikeren og matematikeren Jean-Baptiste Joseph Fourier (1768–1830). Fourier transformasjonen beskriver hvordan et signal kan tilnærmes som en rekke av trigonometriske funksjoner (sinus og cosinus).

Fast Fourier Transform, eller FFT, er en numerisk implementasjon av FT som også er rask. *FFT har blir stemt frem som den viktigste algoritmen fra den 20 århundre!*

Vi vil ikke diskutere dette i detalj her, men bare nedenfor gi noen få eksempler. Du vil garantert senere lære mye mer om Fourier transformer da det brukes mye i fysikk.

La oss laste inn et støyete input signal (en tidsserie) som vi så vil analysere...

[ ]:
```
# Tids variable
time_step = 0.02
t = np.arange(0, 20, time_step)

# Signal
period    = 5.0
sin_signal   = np.sin(2*np.pi*t/period)
noise_signal = np.random.normal(0., 0.5, size=t.size)
signal = sin_signal + noise_signal

# Plot av signalet
plt.plot(t, signal)
```

La oss så analysere frekvens innholdet i dette signalet som vi vil kalle for $s(t)$. Fourier (sinus)

transformen av et slikt signal er

$$S(f) = \int_{-\infty}^{\infty} s(t)\sin(2\pi f t),$$

hvor f er frekvensen til signalet (eller en over dets periode, $f = 1/T$).

[ ]:
```
S = sp.fft.fft(signal)

freq = sp.fft.fftfreq(S.size, d=time_step)
```

[ ]:
```
# Plot av frekvens innholdet i signalet
plt.plot(freq, np.abs(S)**2)
plt.ylabel("Power")
plt.xlabel("frequency [Hz]")

# Zoom lin
plt.xlim(0,0.3)
```

1.2.10 scipy.signal

Se Scientific Python lecture notes for flere detaljer.

1.2.11 scipy.sparse

En *sparse matrise* er en matriser som har mange elementer som er null. På norsk kaller man slike matriser for *glisne matriser*. For eksempel, er matrisen

$$A = \begin{pmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{pmatrix}$$

glissen. Den har -2 på diagonalen og 1 på både sub- og super-diagonalen. For en slik $N \times N$ matrisen er det dermed bare $\mathcal{N} = N + 2(N-1)$ elementer som ikke er null. Dermed er fraksjonen av elementer som ikke er null gitt av $\mathcal{N}/N^2$, et tall som ofte kalles for matrisens sparsity.

For matrisen gitt overfor finner man

[ ]:
```
N=np.linspace(3,1E5,100,dtype=float)

def sparsity(N):
    res =  N + 2*(N-1)
    res = res.copy() / (N*N)
    return res
# end function

SP = sparsity(N)
plt.semilogy(N,SP)
plt.xlabel("N")
```

```
plt.ylabel("Sparsity")
```

For store matriser kan man fort kaste bort mye minne ved å lagre alle null elementene...

scipy.sparse har ulike rutiner for å jobbe med slike matriser.

[ ]:
```
A = np.array([[-2,  1,  0,  0,  0 ],
              [ 1, -2,  1,  0,  0 ],
              [ 0,  1, -2,  1,  0 ],
              [ 0,  0,  1, -2,  1 ],
              [ 0,  0,  0,  1,  -2 ]] )
```

[ ]:
```
# Kan man invertere matrisen
np.linalg.det(A)
```

[ ]:
```
b = np.array([1,2,3,4,5])

sol = np.linalg.solve(A,b)
x_dense = sol

print("Riktig løsning:  ",np.allclose(A@sol,b))
```

La oss så gjøre det samme via bruk av glisne matriser.

[417]:
```
row  = np.array([0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4])
col  = np.array([0, 1, 0, 1, 2, 1, 2, 3, 2, 3, 4, 3, 4])
data = np.array([-2,1, 1,-2, 1, 1,-2, 1, 1, -2, 1, 1, -2])
A_sparse = sp.sparse.coo_matrix((data, (row, col)), shape=(5, 5))
```

[ ]:
```
# La oss forsikre oss om at A_sparse of A har samme innhold
AA = A_sparse.toarray()

# er A = AA
print( " A = AA ? :  ", np.allclose(A,AA))

print ("\n AA = ", AA)
```

Det finnes også andre og mer effektive lagringsformater for glisne matriser. Noen av disse er

- Compressed Sparse Row (CSR)
- Compressed Sparse Column (CSC)
- COOrdinate list (COO)

[ ]:
```
# Konverte matriose lagring til CSC format
A_sparse_2 = A_sparse.tocsc()
```

[ ]:
```
# Løs det glisne likning systemet
x_sparse_2 = sp.sparse.linalg.spsolve(A_sparse_2,b)

# Er løsningen korrekt...
```

```
print("SPSOLVE løsning OK? : ", np.allclose(x_sparse_2, sol))
```

Det finnes en mengde ulike iterative løsere som har ulik konvergensrate:

[ ]:
```
# Løseren bicgstab
x_sparse, _ = sp.sparse.linalg.bicgstab(A_sparse,b)
print("BICGSTAB løsning OK? : ", np.allclose(x_sparse,sol))

# Løseren gmres
x_sparse, _ = sp.sparse.linalg.gmres(A_sparse,b)
print("GMRES løsning OK?    : ", np.allclose(x_sparse,sol))
```