# SymPy introduction v. 1.0

Johann Bock Severin

November 2019

## Contents

# 1 Introduktion til Sympy

Jeg har benyttet lidt tid på at lære SymPy og i tilfælde af, at nogle vil gøre nummeret efter, har jeg her prøvet at samle min viden på en lidt mere overskuelig måde. Dette vil dog på ingen måde være særlig dybdegående og har kun til hensigt, at man kan bruge SymPy-biblioteket til dag-til-dag opgaver og regnerier.

## 1.1 Notebookens opbygning

Jeg har bygget notebooken op således at den først giver det helt basale sympy, hvorefter jeg hhv. går lidt mere i dybden med Algebra og Plotning som forudsætning for at kunne benytte SymPy til at Calculus. Her vil vi altså bruge værktøjet til at differentiere, integrere, løse ordinære differential ligniner og vil derefter udvide disse egenskaber til flere dimensioner. Denne notebook indeholder dog hverken Lineær Algebra, Vektor Calculus eller Partielle Differential Ligninger, og hvis dette er interesse vil jeg i første omgang henvise til dokumentationen, som man kan finde her.

Desuden antager jeg i løbet af denne Notebook, at læseren har kendskab til Python syntaks og har lugtet til libaries som Numpy og Matplotlib.

## 1.2 Import af SymPy

Først importere vi Sympy. Vi kalder den igennem dette dokument fra sp, dog ser man andre steder, at man referere til SymPy som sym eller blot importere de enkelte dele, når man skal bruge dem. Dette vil vi også gøre, hvis det giver mening. Men da vi bruger SymPy igennem hele dokumentet virker det mere naturligt blot at importere det. Det næste vi gør er at aktiverer SymPy's printer, der lader os få vores ligninger i Latex-format. Dette gøres ved fukntionen init_printing(True). Til sidst importere vi desuden display og Math fra IPython for at kunne vise vores ligninger.

[1]:
```
import sympy as sp
from IPython.display import display, Math
%matplotlib inline
sp.init_printing(True)                                    #Aktivering a␣
 ↪pretty-printing
```

## 1.3 Symboler

Integrationen af symbolsk algebra i Python gør, at man benytter Python syntaks til at skrive. Der er altså nogle forskellige ting, vi bliver nødt til at definere. Først bliver vi nødt til at definere, hvad et symbol er. Dette gøres ved at skrive sp.symbols() og give den en liste med repræsentationerne af symbolet. Eksempelvis kan man definere a, b og som:

[2]:
```
a, b, c = sp.symbols("a b c")
a + b
```

[2]:
$$a + b$$

Hvis man nu vil have græske bogstaver eller andre symboler kan man blot skrive det i repræsentationen. Eksempelvis $\omega, \phi$ eller $\dot{x}$ kan skrives som:

[3]:
```
omega, phi, x_dot = sp.symbols("\omega \phi \dot{x}")
display(omega - x_dot * phi)
```

$-\dot{x}\phi + \omega$

Sammensætter man symbolerne (som vist ovenfor) danner man ‘‘expressions’’. Vi kan eksempelvis definerer D, som at være $a \cdot b \cdot c$. Vær her opmærksom på, at man skal benytte Python syntaks og altså ikke blot kan skrive ‘‘abc’’ men man skal bruge "*".

[4]:
```
D = a * b * c
D
```

[4]: $abc$

Ligeledes kan man benytte andre basale regneoperationer som: +, -, *, /. Og ** for potens ligesom i resten af Python. Ønsker man mere avancerede regneredskaber, finder man dem som regel i SymPy-modulet, og man skal starte med at skrive ‘‘sp.’’ foran. Dette kunne eksempelvis være kvadratrødder, trigonometriske funktion, exponentiel funktion og logaritmer. Dette er vist under:

[5]:
```
sp.sqrt(a) + sp.cos(a) + sp.sin(a * b) + sp.exp(b/a) + sp.log(c/a**b) #Eksempel␣
↪på ikke-basale regneoperationer.
```

[5]: $\sqrt{a} + e^{\frac{b}{a}} + \log\left(a^{-b}c\right) + \sin\left(ab\right) + \cos\left(a\right)$

Som man kan se, benytter SymPY lejligheden til at rykke lidt rundt på ledene og faktorerne, men udtrykket er præcis det samme, som man har givet det. Ønsker man andre regneoperationer kan man finde dem i dokumentationen: her.

## 1.4 Værdier

Nogle gange har vi også lyst til at bruge forskellige værdier i vores udtryk, eksempelvis når vi skal evaluere det. SymPy understøtter forskellige typer, hvis du blot indsætter et tal for du en Float (blot et decimal tal), men hvis du eksempelvis skal bruge komplekse tal eller holde noget på brøk-form skal man indskrive det på en bestemt måde. Hvis vi vil indskrive en brvøk, skal vi bruge sp.Rational(*tæller*, *nævner*)

[6]:
```
p, q = sp.Rational(1,4), sp.Rational(3, 6)
display(p, q)
```

$\frac{1}{4}$

$\frac{1}{2}$

Og vi kan nu lave forskellig brøkregning. Bemærk desuden, at den selv har reduceret brøken.

Ligeledes kan vi benytte SymPy til at regne på komplekse tal. Dette gøres ved at bruge sp.I for $i = \sqrt{-1}$. Hvis man skal regne meget på komplekse tal, kan man dog blot importere I ved nedenstående kommando, slipper man for at skrive sp. foran:

[7]:
```
from sympy import I
```

Og vi kan nu blot regne på komplekse udtryk, som vi ville regne på hvad som helst andet.

[8]:
```
a, b, c, d = sp.symbols("a b c d")
x = a * I + b
y = c * I + d
display(x, y)
```

$ia + b$

$ic + d$

[9]:
```
x / y
```

[9]:
$$\frac{ia + b}{ic + d}$$

Eller vi kan indsætte værdier på pladserne:

[10]:
```
z = 3 * I + 4
z ** 5
```

[10]:
$$(4 + 3i)^5$$

Vi kan desuden finde den kompleks konjugerede af et udtryk, ved at skrive .conjugate() på et kompleks udtryk:

[11]:
```
(z ** 5).conjugate()
```

[11]:
$$(4 - 3i)^5$$

[12]:
```
expr = sp.sin(a * 2 * sp.pi)                    #Vi kan først definere en␣
↪"expression"
expr.subs(a, sp.Rational(1, 3))                 #og dernæst evaluere det med a = 1 /␣
↪3
```

[12]:
$$\frac{\sqrt{3}}{2}$$

## 1.5 Evaluering af Expressions

Denne del vil gennemgå mulige måder at evaluere vores Expressions på. Dette kan gøres både numerisk og symbolsk.

### 1.5.1 Symbolsk Evaluering

Hvis vi ønsker at gøre det algebraisk benytter vi expression-metoden .subs(). Denne skal gives input, der fortæller den, hvilken substitution man vil lave. Eksempelvis hvis vi vil evaluere $\sin(a \cdot 2\pi)$ er, når $a = 1/3$, så skriver vi følgende:

[13]:
```
expr = sp.sin(a * 2 * sp.pi)                    #Vi kan først definere en␣
↪"expression"
expr.subs(a, sp.Rational(1, 3))                 #og dernæst evaluere det med a = 1 /␣
↪3
```

[13]:
$$\frac{\sqrt{3}}{2}$$

Det er værd at bemærke, at 1/3 evalueres af Python og indsætter $0.\bar{3}$, mens sp.Rational(1,3) fortæller sympy, at $a = 1/3$ og altså ikke $0.\bar{3}$. Yderligere kan man også se, at $\pi$ er en del af sympy.

Hvis vi nu istedet har fundet ud af, at $a = b$ kan vi også benytte .subs() til dette. Eksempelvis, hvis vi har $(a+b)/ab$. Så skriver vi blot:

[14]:
```
expr = (a + b) / (a * b)
expr.subs(b, a)
```

[14]:
$$\frac{2}{a}$$

Ønsker man at substituere flere udtryk på en gang, så gøres dette med en liste, der består parvist af tuples med (symbol, værdi). Eksempelvis, hvis vi ved at $a = b$ og at $a = 8/9$ og vil evaluere $\sin\left(\frac{a+b}{ab} \cdot \pi\right)$

[15]:
```
expr = sp.sin((a+b)/(a*b) * sp.pi)
expr.subs([(b, a), (a, sp.Rational(8,9))])
```

[15]:
$$\frac{\sqrt{2}}{2}$$

### 1.5.2 Numerisk evaluering

Vi har været heldig med overstående evalueringer, at vi har kunne regne dem exact. Men hvis vi nu havde $\sin(a) + a$ og $a = 1/7$, giver det os ikke så meget.

[16]:
```
expr = (sp.sin(a) + a).subs(a, sp.Rational(1, 7)) #Bemærk, at parantesen omkring␣
↪sin(a) + a markerer en expression.
display(expr)                                     #Dette kan være en genvej i␣
↪stedet for at definere en expr hver gang
```

$$\sin\left(\frac{1}{7}\right) + \frac{1}{7}$$

Vi kan nu istedet benytte expression-metoden .evalf(). Denne tager som udgangspunkt præcisionen som argument. Og .evalf(2) evaluere altså vores udtryk til 2 decimaler, mens .evalf(10) evaluere det til 10 osv.

[17]:
```
two_digs = expr.evalf(2)
ten_digs = expr.evalf(10)
display(two_digs)
```

```
display(ten_digs)
```

0.29

0.2852288726

Ønsker vi at springe steppet over, hvor vi indsætter $a = 1/7$ kan vi gøre dette ved at benytte keyword-arguments ‘‘subs’’ til .evalf(). Denne tager et dictionairy som input, hvor man giver den symboler og tilsvarende værdier, som de skal erstattes med.

[18]:
```
expr = sp.sin(a) + a
expr.evalf(5, subs = {a: sp.Rational(1,7)})  #Her kunne vi lige så godt skrive 1/
↪7, men det er bedre at lade Sympy
                                             #styrer precisionen af vores␣
↪udtryk, end at lade Python give en float.
```

[18]: 0.28523

### 1.5.3 Numerisk evaluering som funktion

Man kan ofte komme ud for at skulle bruge sit udtryk hundrede hvis ikke tusinder eller millioner af gange. Her kan man med fordel benytte funktionen sp.lambdify(), der laver ens expression om til en integreret Python funktion, om man nu bruge den sammen med eksempelvis numpy arrays.

Forestil dig eksempelvis, at vi har været i lab for at tage periode_tiden på en pendul ved forskellige snorlængder. Dette er relateret ved $T = 2 \pi \sqrt{\frac{L}{g}}$. Selvfølgelig kun for små udsving. Hvis vi opskriver dette i Sympy som :

[19]:
```
L, g = sp.symbols("L g")
T = 2 * sp.pi * sp.sqrt(L/g)
display(T)
```

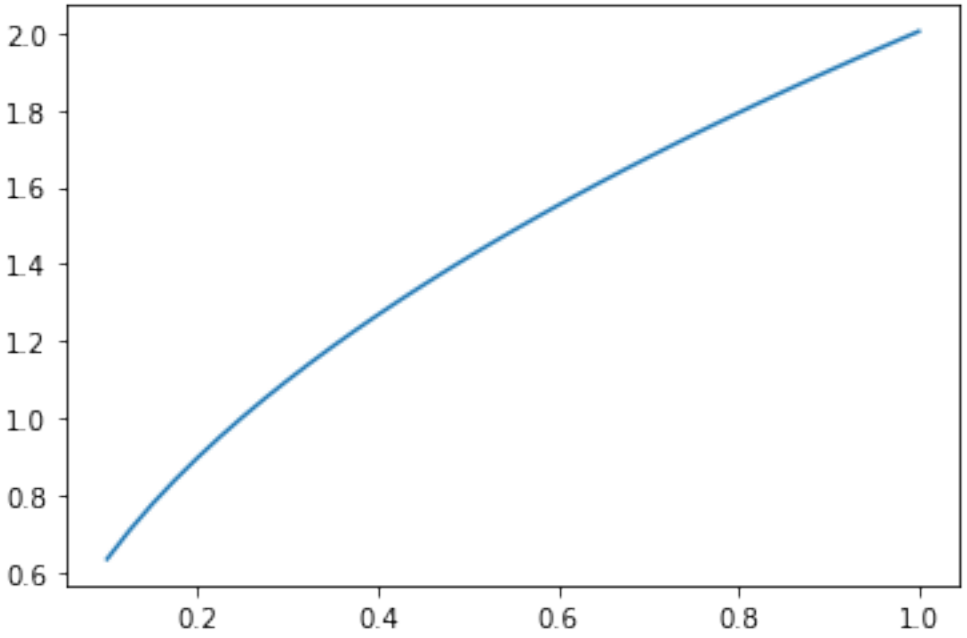
$$2\pi\sqrt{\frac{L}{g}}$$

Vi ønsker nu at konvertere T(L, g) til en Numpy funktion. Vi benytter her lambdify(). Dennes første argument er de variable størrelser, som funktionen skal have. Dernæst skal man give den expression, som skal omdannes til en Numpy funktion. Til sidst skal vi give den, hvilket modul den skal konvertere funtkionen til. Her vil ‘‘numpy’’ ofte være et godt valg.

[20]:
```
T_numpy = sp.lambdify([L, g], T, "numpy")
```

Vi kan nu benytte dette som vi ville bruge en almindelig python funktion. Så hvis, vi har prøvet snorlængderne fra $0.1m$ – $1m$ med intervalstørrelse på $2.5cm$ kan vi blot indsætte det som:

[21]:
```
from numpy import arange
from matplotlib.pyplot import plot

g = 9.82
Ls = arange(0.1, 1.025, 0.025)

Ts = T_numpy(Ls, g)

plot(Ls, Ts);
```




Det er værd at bemærke, at 1/3 evalueres af Python og indsætter $0.\bar{3}$, mens sp.Rational(1,3) fortæller sympy, at $a = 1/3$ og altså ikke $0.\bar{3}$. Yderligere kan man også se, at $\pi$ er en del af sympy.

Hvis vi nu istedet har fundet ud af, at $a = b$ kan vi også benytte .subs() til dette. Eksempelvis, hvis vi har $(a+b)/ab$. Så skriver vi blot:

# 2 Algebra

Nu skal vi så rent faktisk til at benytte værktøjet. Der hvor et symbolsk værktøj virkelig kommer en til gavn er nu man selv vil undgå at reducere udtryk eller løser ligninger. Denne sektion vil derfor præsenterer det grundlæggende algebra, som man kunne finde på at bruge sit symbolskeværktøj til.

## 2.1 Reduktion

At reducere et udtryk er ofte dér, man river sig selv lang tid håret, og hvor et symbolsk værktøj kan komme en til tjeneste. Jeg vil her kun gennemgå den mest universelle reducering i SymPy for mere specifikke reductioner som at udvide, faktorisere, omskrivninger kan man se dokumentationen her. De fleste gange vil man dog kunne slippe afsted med at bruge sp.simplify(), man skal dog være opmærksom på, at det kan være meget forskelligt, hvad der er det mest reduceret udtryk, og man skal altså tage outputtet af denne funktion med et gran salt.

Man benytter simplify på en expression og SymPy prøver så at træffe en beslutning om, hvad det simpleste udtryk er. Nogle eksempler på brugen af sp.simplify() kan findes herunder:

Først kan vi prøve trigonometriske funktioner:

[22]:
```
x = sp.symbols("x")
expr = sp.cos(x) * sp.sin(x) * (sp.cos(x) + sp.sin(x))
print("Expression:")
display(expr)

reduced = sp.simplify(expr)
print("Reduceret:")
display(reduced)
```

Expression:

$$(\sin(x) + \cos(x))\sin(x)\cos(x)$$

Reduceret:

$$\sqrt{2}\sin(x)\sin\left(x + \frac{\pi}{4}\right)\cos(x)$$

Eller måske er vi heldige med en polynomie division:

[23]:
```
Poly1 = 3 * x ** 4 - x** 2 - 4
Poly2 = - 3 * x**2 + 4
expr = Poly1 / Poly2
print("Expression:")
display(expr)

reduced = sp.simplify(expr)
print("Reduceret:")
display(reduced)
```

Expression:

$$\frac{3x^4 - x^2 - 4}{4 - 3x^2}$$

Reduceret:

$$-x^2 - 1$$

Vi kan ligeledes bruge dette til at reducere komplekse udtryk:

[24]:
```
a, b, c, d = sp.symbols("a b c d")
x = 5 * sp.I + 3
y = 4 * sp.I / 5 - 2
z = x / y ** 2
display(z)
display(sp.simplify(z))
```

$$\frac{3 + 5i}{\left(-2 + \frac{4i}{5}\right)^2}$$

$$-\frac{925}{3364} + \frac{4125i}{3364}$$

## 2.2 Ligninger

Før vi kan begynde at løse ligninger skal vi først have introduceret en lighed. Dette hedder i SymPy's sprog en equality og kan defineres som, sp.Eq(*venstre side*, *højre side*). Hvis vi eksempelvis har pythagoras, kan vi opskrive den lighed som:

[25]:
```
a, b, c = sp.symbols("a b c")
Pyth = sp.Eq(c ** 2, a ** 2 + b ** 2)
display(Pyth)
```

$$c^2 = a^2 + b^2$$

Man kan ligeledes sammensætte ligheder ved først at danne expressions:

[26]:
```
theta = sp.Symbol("\\theta")
expr = a**2 + b**2- 2*a * b * sp.cos(theta)
display(expr)

cos_relation = sp.Eq(c**2, expr)
display(cos_relation)
```

$$a^2 - 2ab\cos\left(\theta\right) + b^2$$

$$c^2 = a^2 - 2ab\cos\left(\theta\right) + b^2$$

Nu når vi har lighederne på plads, er det blivet tid til at lade SymPy regne for os. SymPy giver os to forskellige værktøjer til at løse ligninger, og det er lidt forskelligt, hvad de hver især er god til. Her vil jeg derfor præsentere de to forskellige:

### 2.2.1 Solveset

Den første metode hedder solveset(). Dette er den nye version og ultimativt ønsker SymPy-teamet at konvertere helt over til denne funktion, det kan dog ikke alt endnu, men for at være fremtidssikret starter vi med denne, og går tilbage, hvis vi ikke føler, at vi ikke får, hvad vi havde håbet på.

Lad os prøve at løse en af overstående ligninger. Lad os tage helt standard Pythagoras $a^2 + b^2 = c^2$ (defineret ved Pyth over over). Hvis vi nu kender $c = 5$ og $a = 3$ og nu ønsker at finde $b$. Vi omskriver først vores ligning:

[27]:
```
display(Pyth)
Pyth_subbed = Pyth.subs([(a, 3), (c, 5)])
display(Pyth_subbed)
```

$$c^2 = a^2 + b^2$$

$$25 = b^2 + 9$$

Vi benytter nu solveset() til at løse denne. Denne funktion virker ved, at man giver den en ligning eller et udtryk (som den så vil sætte lig 0) og en variabel, som man ønsker at løse for. Vi kan gøre dette ved:

[28]:
```
solution = sp.solveset(Pyth_subbed, b)
display(solution)
```

$$\{-4, 4\}$$

Vi har altså nu fundet løsningerne til ligningen. I pythagoras sætning er vi dog klar over, at vi skal bruge positive tal. Vi kan indskrænke løsningsdomænet ved at give solveset et "domain"-keyword. Denne tager et SymPy-set, som er en lidt indviklet størrelse, men i langt de fleste tilfælde, kan man slippe afsted med at bede om reelle tal ved at skrive "sp.Reals" eller ved at give et interval med "sp.Interval(*fra*, *til*)". Hvor man kan bruge sp.oo for uendelig. Ønsker man mere avanceret løsningsdomæner kan man læse dokumentationen om set her. For nu benytter vi dog dette til kun at løse for positive tal:

[29]:
```
pos_solution = sp.solveset(Pyth_subbed, b, sp.Interval(0, sp.oo))
display(pos_solution)
```

$$\{4\}$$

Andet eksempel, hvor vi kan blive nødt til at indskrænke løsningerne er, hvis vi nu leder efter rødder i et fjerdegrads polynomium:

[30]:
```
x = sp.Symbol("x")
expr = x ** 4 - 1
solutions = sp.solveset(expr, x) #Husk at give den en expression sætter denne␣
 ↪lig med 0 og løser.
display(solutions)
```

$$\{-1, 1, -i, i\}$$

Vi kan nu indskrænke domænet til at være reelle tal ved at skrive:

[31]:
```
solutions_real = sp.solveset(expr, x, sp.Reals)
display(solutions_real)
```

$\{-1, 1\}$

Solveset virker også, hvis vi ikke nødvendigvis har en numerisk værdi for alle symboler, men istedet vil gøre det mere generisk. Hvis vi nu vil finde $b$ i vores cosinus relation defineret tidligere, kan vi skrive følgende:

[32]:
```
display(cos_relation)
sol_b = sp.solveset(cos_relation, b)
display(sol_b)
```

$c^2 = a^2 - 2ab\cos(\theta) + b^2$

$$\left\{a\cos(\theta) - \sqrt{a^2\cos^2(\theta) - a^2 + c^2}, a\cos(\theta) + \sqrt{a^2\cos^2(\theta) - a^2 + c^2}\right\}$$

Solveset() benytter hele set-strukturen fra SymPy for at give os de generelle fuldstændige løsninger. Selvom dette er meget skønt, ender vi dog hurtig med en lidt for general løsning. Dette sker eksempelvis, hvis vi prøver at løse for $\theta$ direkte:

[33]:
```
sp.solveset(cos_relation, theta)
```

[33]:
$$\{\theta \mid \theta \in \mathbb{R} \wedge -a^2 + 2ab\cos(\theta) - b^2 + c^2 = 0\}$$

Hvilket jo er rigtig nok, men siger os måske ikke lige helt så meget, som vi ikke i forvejen vidste. Det leder os videre til den anden metode til at løse ligninger

### 2.2.2 Solve

Solve() er den ældre funktion, som på trods af at være mindre general oftest giver os det, som vi rent faktisk leder efter. Solve tager sit input på samme måde som solveset.

[34]:
```
sp.solve(cos_relation, theta)
```

[34]:
$$\left[-\operatorname{acos}\left(\frac{a^2 + b^2 - c^2}{2ab}\right) + 2\pi,\ \operatorname{acos}\left(\frac{a^2 + b^2 - c^2}{2ab}\right)\right]$$

solve() tager input på samme måde som solveset, man skal dog ikke give den et domæne, i stedet kan den tage en liste af ligheder (eller uligheder) og på den måde løse. Hvis vi prøver at finde vores sidelænge $b$ på samme måde som med solveset() vil den i solve-sprog lyde:

[35]:
```
display(Pyth_subbed)
sol_b = sp.solve([Pyth_subbed, b > 0], b)
display(sol_b)
```

$25 = b^2 + 9$

$b = 4$

På mange måder er solve() altså det nemmere værktøj at bruge. Man skal dog være opmærksom nogle steder. Eksempelvis hvis man løser $\sin(\theta) = 1$:

[36]:
```
sp.solve(sp.Eq(sp.sin(theta), 1), theta)
```

[36]:
$$\left[\frac{\pi}{2}\right]$$

Her får vi altså kun en løsning, da solve() kun kigger på argumenter mellem 0 og $2\pi$ i trigonometriske funktioner.

## 2.3 Ligningssystemer

Hvis du har besluttet dig for at benytte solve er det ret nemt at gå fra en ligning til flere: du skal blot give denne ligningerne og de variable som en liste. Lad os kigge på et helt simpelt lineært ligningssystem:

[37]:
```
#Vi bruger x og y som variable
x, y = sp.symbols("x y")

#Definerer nu to lineære udtryk
lin1 = 3*x + 2 * y -1
lin2 = x + y - 1

#Hvis vi ønsker at finde x, y der løser begge to skriver vi nu blot:
sp.solve([lin1, lin2], [x, y])
```

[37]:
$$\{x : -1,\ y : 2\}$$

Hvis vi har et lidt mere avanceret udtryk, der giver flere løsninger, såsom: $\cos(x)\sin(y) = 0$ og $x + y = 1$ gør vi det ligeledes:

[38]:
```
Eq1 = sp.Eq(sp.cos(x)*sp.sin(y), 0) #Dette er svarende til blot at indsætte Eq1
↪= sp.cos(x)*sp.sin(y)
Eq2 = sp.Eq(x + y, 1)

sp.solve([Eq1, Eq2], [x, y])
```

[38]:
$$\left[(1,\ 0),\ \left(-\frac{3\pi}{2},\ 1+\frac{3\pi}{2}\right),\ \left(-\frac{\pi}{2},\ 1+\frac{\pi}{2}\right),\ (1-\pi,\ \pi)\right]$$

Solve() kan dog ikke håndterer, at vi prøver at sætte begrænsninger på vores variabler, når vi løser ligningssystemer.

Vi vil nu rette øjnene mod solvesets måde at håndtere ligningssystemer. Her er der to funktioner, først linsolve(), som skal bruge til lineære ligningssystemer og nonlinsolve(), som skal bruges, når vi ikke behandler et lineært ligningssystem.

Har vi et lineært ligningssystem, som det, der består af lin1 og lin2 i overstående eksempel, løses det på præcis samme måde med linsolve():

[39]:
```
sp.linsolve([lin1, lin2], [x, y])
```

[39]:
$$\{(-1,\ 2)\}$$

Eller hvis vi kigger på Eq1 og Eq2 i overstående skal vi benytte nonlinsolve:

[40]:
```
sp.simplify(sp.nonlinsolve([Eq1, Eq2], [x, y])) #Her er simplify brugt, så vi␣
↪ikke fylder 3 sider med løsningen
```

[40]:
$$\left\{\left(-2n\pi+\frac{\pi}{2},\ \left\{2n\pi-\frac{\pi}{2}+1 \mid n\in\mathbb{Z}\right\}\right),\left(-2n\pi-\frac{\pi}{2},\ \left\{2n\pi+1+\frac{\pi}{2} \mid n\in\mathbb{Z}\right\}\right),(-2n\pi+1,\ \{2n\pi \mid n\in\mathbb{Z}\}\right.$$

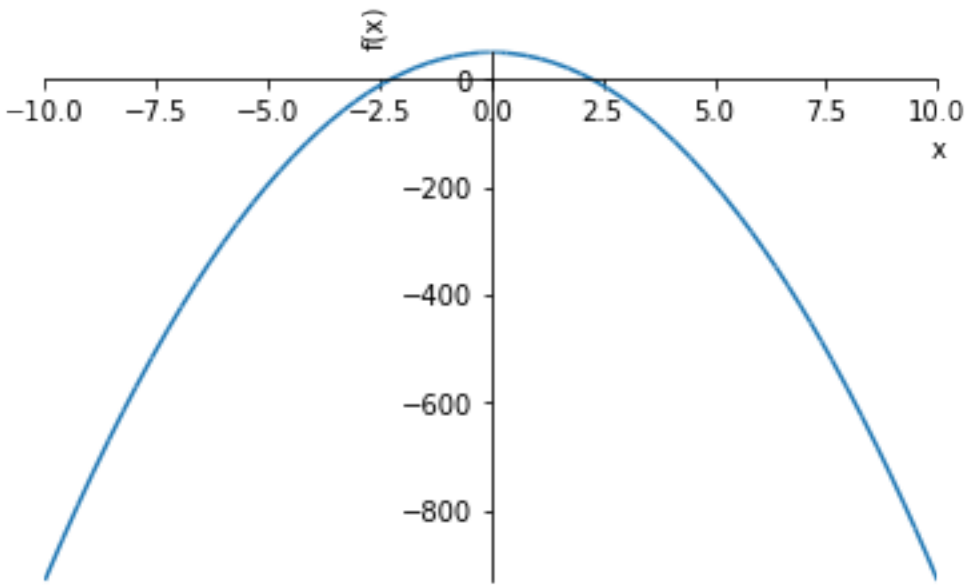
# 3 Plotning

Lad os nu kigge på, hvordan vi visualisere vores udtryk. Det er ofte her, hvor et værktøj som SymPy virkelig kan værdsættes. SymPy bruger modulet sympy.plotting til at plotte, men benytte en Matplotlib backend. Det er altså helt klart en fordel at være fortrolig med dette værktøj, hvis man vil gøre sine grafer nogenlunde præsentable.

## 3.1 Plot simple udtryk

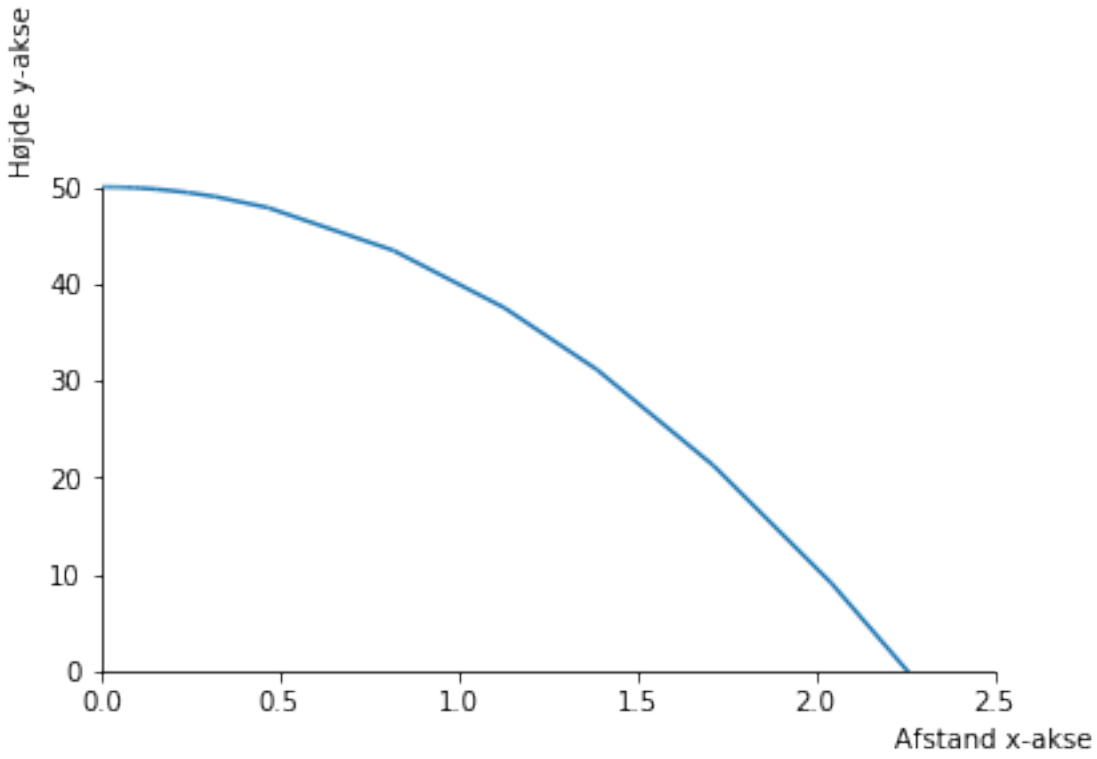
Til at starte med vil vi kigge på den simple funktion "plot", som hvis importeret gennem SymPy lader os visualisere algebraiske udtryk. Lad os starte med at visualisere en parabel.

[41]:
```
from sympy.plotting import plot

x = sp.Symbol("x")
polynomie = 50 - 9.8 * x ** 2

poly_plot = plot(polynomie)
```



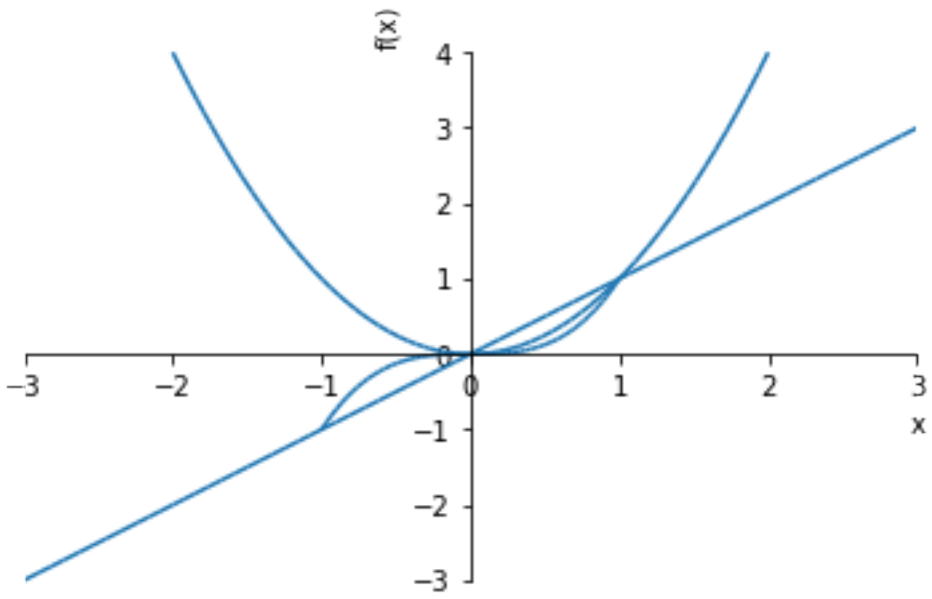

Det er altså forholdsvis ligetil at komme i gang med. Vi kan nu manipulere plottet som vi ville gøre med matplotlib:

[42]:
```
poly_plot.xlim = (0, 2.5)
poly_plot.ylim = (0, 60)
poly_plot.xlabel = "Afstand x-akse"
poly_plot.ylabel = "Højde y-akse"
poly_plot.show()
```




Vi kan også skrive vores limits ind fra starten. plot kan tage tupler bestående af (*udtryk*(*variabel*, *start*, *slut*))

[43]:
```
poly1 = x
poly2 = x**2
poly3 = x**3

plot((poly1, (x, -3, 3)), (poly2, (x, -2, 2)), (poly3, (x, -1, 1)));
```



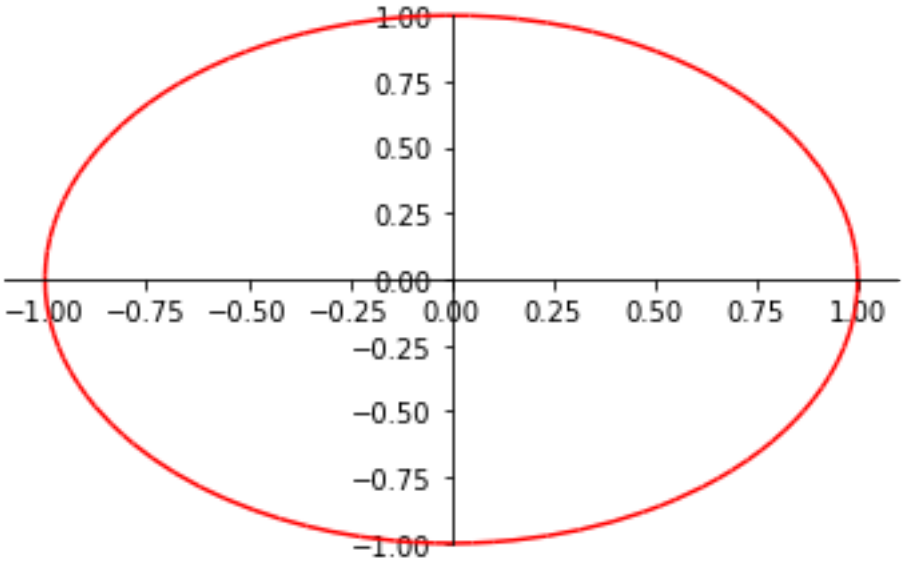

Der er desuden en masse andre key_words som er rare at kende. Såsom line_color, title, nb_of_points, osv.

### 3.2 Parametriske plot

Ud over at plotte simple udtryk, så kan vi også bruge sympy til at plotte parametriske funktioner. Dette gøres med funktionen plot_parametric(). Dette kræver at vi har parametriseret vores $x,y$ som funktioner af en anden parameter, lad os bruge $u$. Hvis vi altså nu har $x(u) = \cos(u), y(u) = \sin(u)$, og $u \in [0, 2\pi[$. Så kan vi plotte det ved at skrive:

plot_paramteric(*x(u), y(u)* , (*u, start, slut*))

[44]:
```
u = sp.symbols("u")
x = sp.cos(u)
y = sp.sin(u)

from sympy.plotting import plot_parametric
circle = plot_parametric(x, y, (u, 0, 2*sp.pi), line_color = 'r');
```

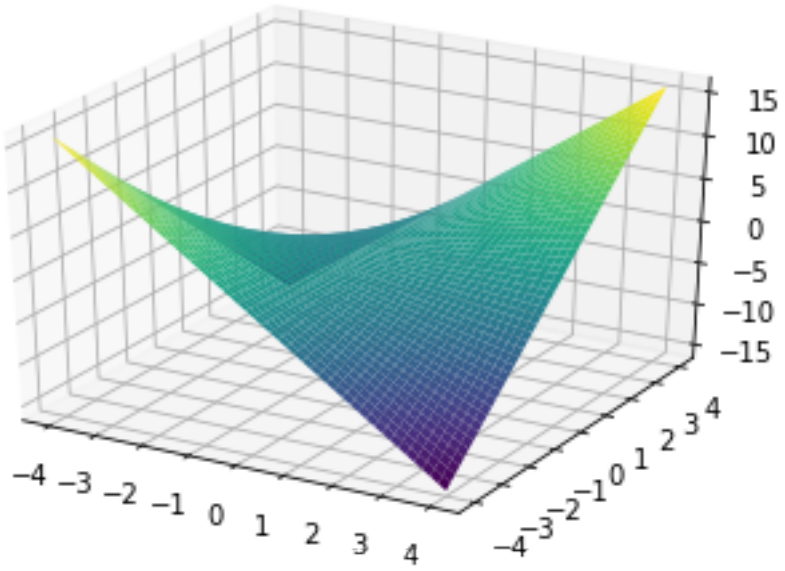



### 3.3 3D - plots

Både plot og plot_parametric har 3d udgaver. Ønsker vi at lave et 3d plot benytter vi nu plot3d. Vi kan eksempelvis plotte $f(x,y) = x \cdot y$, kalder vi:

plot3d(*udtryk, (var1, start, slut), (var2, start, slut)*)

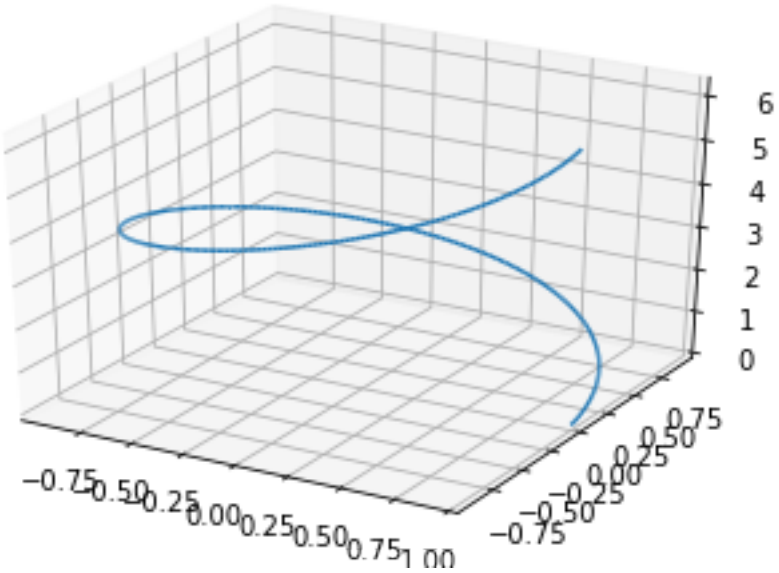
[45]:
```
from sympy.plotting import plot3d
x, y = sp.symbols("x y")
produkt = x * y
plot3d(produkt, (x, -4, 4), (y, -4, 4));
```




*Til 3d plots er det en god idé at bruge en anden backend ind inline, så man interaktivt kan rotere sit plot og undersøge det lidt grundigere.*

Ligelededs kan vi parametrisere hhv. en linje og en overflade med plot3d_parametric_line og plot3d_parametric_surface. For linjerne virker det præcis, som man forestiller sig, man skal bare tilføje en ekstra ligning. Lad os her bare tilføje $z = u$, så kan vi pakke vores cirkel ud i en spiral:

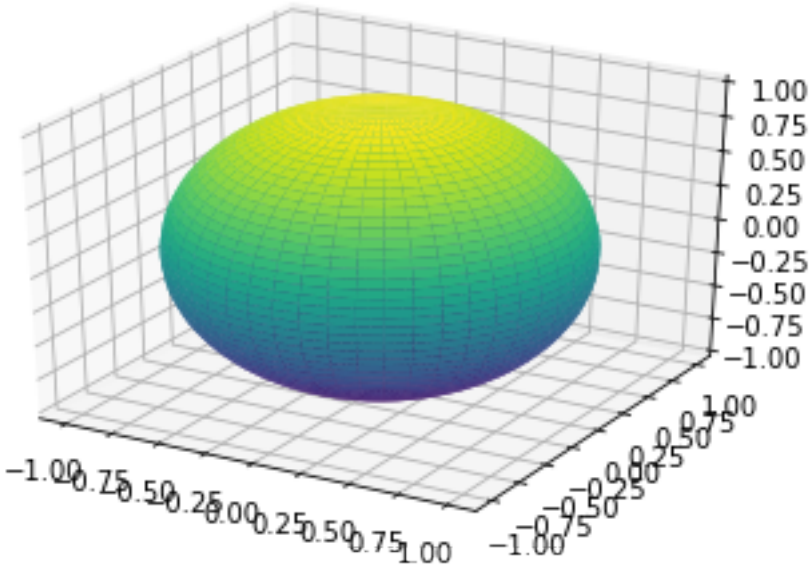
[46]:
```
from sympy.plotting import plot3d_parametric_line
u = sp.symbols("u")
x = sp.cos(u)
y = sp.sin(u)

plot3d_parametric_line(x, y, u, (u, 0, 2 * sp.pi))
```




[46]: <sympy.plotting.plot.Plot at 0x1fd1860c708>

Hvis vi derimod vil have en overflade, skal vi have to parametre. Lad os forsætte i samme tema og lave en kugle ved at define: $x = \cos(u)\sin(v), y = \sin(u)\sin(v)$ og $z = \cos(v)$, med $u \in [0, 2\pi] og v \in [0, \pi]$:

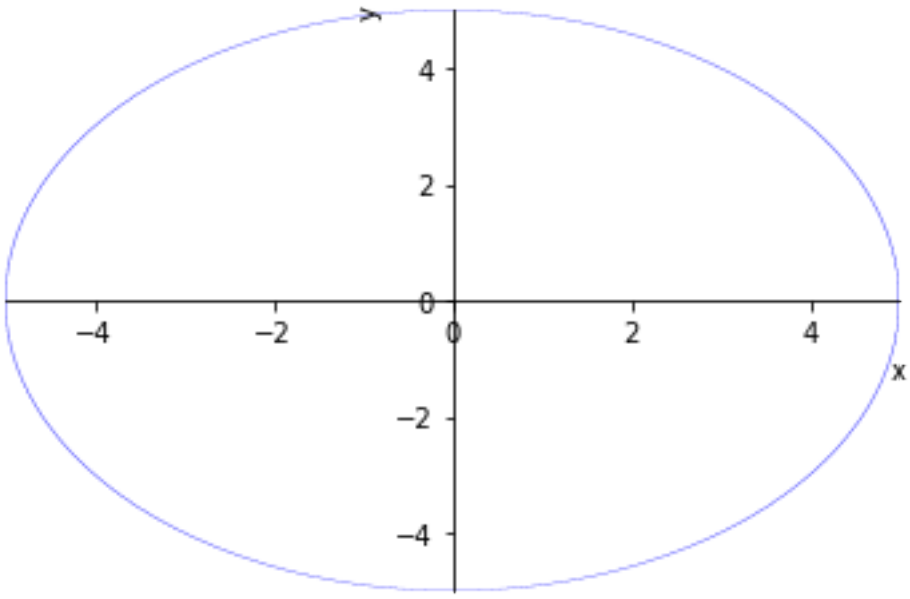
[47]:
```
u, v = sp.symbols("u v")

x = sp.cos(u) * sp.sin(v)
y = sp.sin(u) * sp.sin(v)
z = sp.cos(v)

from sympy.plotting import plot3d_parametric_surface
plot3d_parametric_surface(x, y, z, (u, 0, 2 * sp.pi), (v, 0, sp.pi));
```




### 3.4 Implicit plotning

Den sidste plotting metode, som jeg vil gå over her er at plotte implicit. Her plotter vi altså alle x, y, der opfylder et vist udtryk. Vi kan ligeledes lave vores cirkel på denne måde vil at skrive følgende:

[48]:
```
x, y = sp.symbols("x y")
r = 5
equation = sp.Eq(x ** 2 + y**2, r**2)

from sympy.plotting import plot_implicit
plot_implicit(equation);
```




Som sluttetelig note vil jeg dog sige, at sympy.plotting er en meget skrabet version af matplotlib. Hvis du skal præsentere dit data, eller vil udføre mere nøje analyse kan du benytte lambdafunktionerne og benytte numerisk plotting ved direkte brug af matplotlib biblioteket.

# 4 Calculus - En variabel

Nu skal vi over, hvor det virkelig begynder at blive sjovt. I dette afsnit vil jeg gå igennem, hvordan man bruger sympy til basal calculus. Heribland grænser, differentiering og integration. Det skal siges, at der, hvor sympy virkelig bliver kraftfuld ift. programmer som Maple er, når man kombinerer den integrerede calculus med eksempelvis lambdify, da dette gør afstanden mellem teori og praksis lidt kortere.

## 4.1 Grænser

Grænser er i sympy virkelig nemt. Her skal vi blot importere funktionen limits() og give den et udtryk at slå sig løs på. Lad mig vise et par eksempler:

Først kan vi prøve med noget simpelt, eksempelvis $\lim_{x\to\text{inf}} e^{-x}$

Vi giver limit() følgende argumenter:

limit(*udtryk*, *variabel*, *grænse*)

[49]:
```
from sympy import limit
x = sp.Symbol("x")
expr = sp.exp(-x)
limit(expr, x, sp.oo)      # Husk at sp.oo betyder uendelig
```

[49]:
$$0$$

Man kan måske prøve med nogle mere avncerede. Herunder prøver jeg først et polynomium-stykke og dernæst den klassisk $\sin(x)/x$. De gøres på præcis samme måde:

[50]:
```
poly1 = x ** 4 + x ** 2 + 1
poly2 = 3* x ** 4 - 19 * x **3  - x** 2
poly_div = poly1 / poly2

limit(poly_div, x, sp.oo)
```

[50]:
$$\frac{1}{3}$$

[51]:
```
expr = sp.sin(x)/x

limit(expr, x, 0)
```

[51]:
$$1$$

Nogle udtryk kan det dog blive nødvendigt at definere retningen, vi tager grænsen fra. Eksempelvis $\lim_{x\to 0} 1/x$.

[52]:
```
limit(1 / x, x, 0)
```

[52]:
$$\infty$$

Her bliver altså bare den højre-sidet grænse taget. Man skal altså være opmærksom! Man kan dog specificere retningen ved at give den et ekstra argument, som enten kan være ‘‘+’’ eller ‘‘-’’.

[53]:
```
display(limit(1/x, x, 0, "+"))
display(limit(1/x, x, 0, "-"))
```

$\infty$

$-\infty$

Her får vi altså hhv. positiv og negativ uendelig som forventet.

## 4.2 Differentiering

Differentiering er udtryk er forholdsvis ligetil. Her skal vi bare benytte diff-funktionen. Denne tager som input et udtryk og en varibel og differentiere så udtrykket ift. den givne variabel.

[54]:
```
a, b, x = sp.symbols("a b x")
expr = b * x ** a
print("Expression:")
display(expr)
diss_expr = sp.diff(expr, x)
print("Differentieret:")
display(diss_expr)
display(sp.simplify(diss_expr))
```

Expression:

$bx^a$

Differentieret:

$\frac{abx^a}{x}$

$abx^{a-1}$

Ønsker vi at differentiere en funktion flere gange, giver vi den blot et ekstra arguemnt med antallet af gange, som vi vil differentiere den:

[55]:
```
A, omega, t = sp.symbols("A \omega t")
expr = A * sp.cos(omega * t)
dif_expr = sp.diff(expr, t, 5)
display(dif_expr)
```

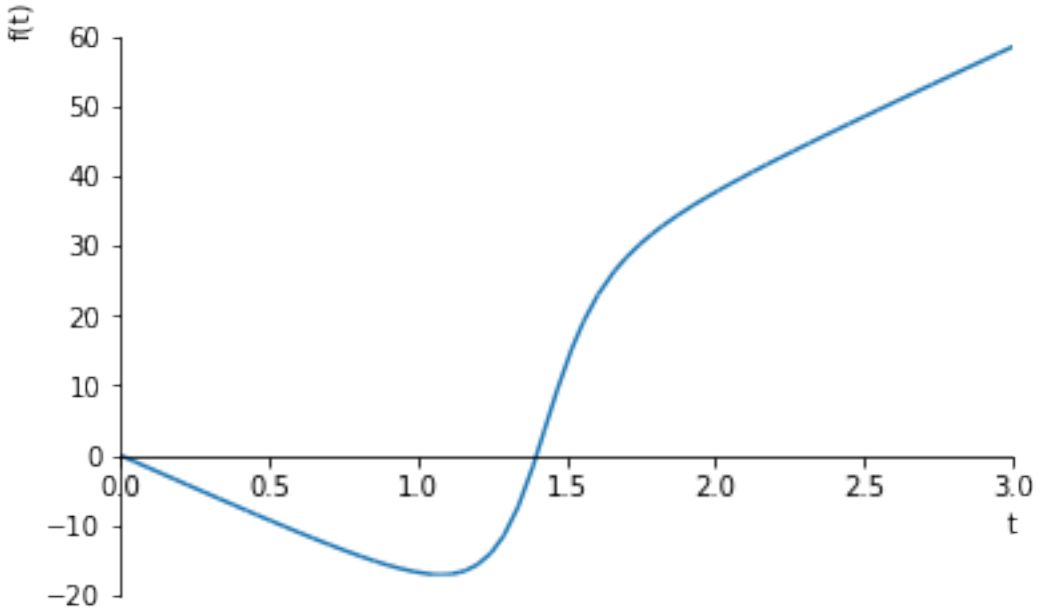
$-A\omega^5 \sin(\omega t)$

Sympy tager også højde for kædereglen. Hvis vi nu har en position defineret som $x = 4t$ og $y = -9.8t^2 + 20$ og vil vide, hvordan størrelsen af hastigheden ændrer sig ift. til tiden. Altså $\frac{d|\vec{r}|}{dt}$.

[56]:
```
x, y, t = sp.symbols("x y t")
x = 4 * t
y = 20 - 9.8 * t ** 2
r = sp.sqrt(x ** 2 + y**2) #Egentlig sammensætter vi bare et udtryk af to
display(r)
print(" ")
v = sp.diff(r, t)
display(v)

from sympy.plotting import plot
plot(v, (t, 0, 3));
```

$$\sqrt{16t^2 + (20 - 9.8t^2)^2}$$

$$\frac{-19.6t\left(20 - 9.8t^2\right) + 16t}{\sqrt{16t^2 + (20 - 9.8t^2)^2}}$$




I en dimension er det egentlig alt, hvad man har brug for at vide, vi vender dog tilbage til denne funktion, når vi taler calculus i flere dimensioner.

### 4.2.1 Taylor serier

I SymPy er der også indbygget en taylor-serie udvikling. Denne går bare under sp.series() og skrives som sp.series(*udtryk*, x = *variabel*, x0 = *omegn*, n = *orden*). Vi kan eksempevis udvide $e^{ix}$: (hvor $i$ skrives som sp.I)

[57]:
```
x = sp.symbols('x')
expr = sp.exp(sp.I * x)
taylor_exp = sp.series(expr, x, 0, 5) #Vi behøves selvfølgelig ikke skrive␣
 ↪keywords, hvis vi giver dem i den rigtige rækkefølge
display(taylor_exp)
```

$$1 + ix - \frac{x^2}{2} - \frac{ix^3}{6} + \frac{x^4}{24} + O\left(x^5\right)$$

Ligeledes kan vi udvide vores serie omkring et andet punkt. Eksempelvis $\frac{3}{1-3r}$ for et $r > 1/3$.

[58]:
```
r = sp.symbols('r')
expr = 3 / (1-3*r)
taylor_expr = sp.series(expr, r, 1, 4)
display(taylor_expr)
```

$$-\frac{15}{4} - \frac{27\left(r-1\right)^2}{8} + \frac{81\left(r-1\right)^3}{16} + \frac{9r}{4} + O\left(\left(r-1\right)^4; r \to 1\right)$$

Hvis vi ønsker at benytte denne formel til noget andet eksempelvis sammen med lambdify eller til at plotte, kan vi benytte .removeO() på vores expression:

[59]:
```
taylor_expr.removeO()
```

[59]:
$$\frac{9r}{4} + \frac{81\left(r-1\right)^3}{16} - \frac{27\left(r-1\right)^2}{8} - \frac{15}{4}$$

### 4.3 Integration

Nu er det tid til at kigge på integration. Der er to slags integrationer vi skal kigge på: det ubestemt og det bestemte. De mindre dog meget om hinanden i SymPy, så lad os blot starte med det ubestemte integrale, altså stamfunktionen.

Stamfunktionen findes ved at tage kalde integrate(*udtryk*, *variabel*). Lad os prøve på noget simpelt:

[60]:
```
x = sp.Symbol("x")
expr = x ** 4
display(expr)
stamfunktion_expr = sp.integrate(expr, x)
display(stamfunktion_expr)
```

$$x^4$$

$$\frac{x^5}{5}$$

Ønsker vi at definere det bestemte integrale, skal vi blot erstatte vores *variabel* med (*variable*, *a*, *b*), hvor *a* og *b* er grænserne for integralet:

[61]:
```
t = sp.Symbol("t")
expr = sp.sin(t) ** 2
display(Math("\int_0^{6\pi} \sin^2(t) dt = "))
display(sp.integrate(expr, (t, 0, 6 * sp.pi)))
```

$$\int_0^{6\pi} \sin^2(t)dt =$$

$$3\pi$$

Slutteligt kan vi også regne uegentlige integraler, altså integraler med $\infty$ som grænse. Her husker vi at uendelig skrives som sp.oo (med to små 'o', altså bogstavet):

[62]:
```
x = sp.Symbol("x", real = True) # x defineres til reelle tal
a = sp.Symbol("a", positive = True) # a defineres kun til at være positive tal,␣
 ↪så vi altid har en grænse
gauss = sp.exp(- x ** 2 * a)
display(Math("\int_{-\infty}^\infty e^{- ax^2}dx = "))
print(" ") #slam linje-skift
display(sp.integrate(gauss, (x, -sp.oo, sp.oo)))
```

$$\int_{-\infty}^{\infty} e^{-ax^2}dx =$$

$$\frac{\sqrt{\pi}}{\sqrt{a}}$$

### 4.4 Ordinære differentialligninger

Lad os nu kigge på SymPys redskaber til at løse ordinære differentialligninger. En del ordinære differentialligninger er understøttet i SymPy, man kan finde en liste over dem her.

Indtil nu har vi dog udelukkende kigget på algebraiske symboler, for at løse differentialligninger, skal vi dog bruge funktionsudtryk. Funktioner defineres blot ligesom symboler, vi benytter bare sp.Function og dernæst laver i en parentes med en variabel, som funktion afhænger af.

[63]:
```
x = sp.Symbol('x') #Variabel
f = sp.Function('f')(x) #Funktion
display(f)
```

$$f(x)$$

Vi kan nu hhv. differentiere og integrere $f(x)$ ved at skrive .diff(*variabel*) eller .integrate(*variabel*) efter vores funktion:

[64]:
```
display(f.diff(x))
display(f.integrate(x))
```

$$\frac{d}{dx}f(x)$$

$$\int f(x)\,dx$$

Yderligere egentskaber ved funktion-klassen kan findes her.

Men lad os nu vende tilbage til differentialligninger. Hvis vi nu eksempelvis har formlen for en simpel harmoniske oscillator:

$$\frac{d^2}{dt^2}x(t) - w^2x(t) = 0$$

kan vi skrive dette op med sp.Eq() præcis som vi gjorde, da vi løste almindelige ligningerne:

[65]:
```
omega, t = sp.symbols("\omega t")
x = sp.Function("x")(t)

ODE = sp.Eq(x.diff(t, 2) + omega ** 2 * x, 0) # 2-tallet i .diff er antallet af␣
 ↪gange vi differentiere.
display(ODE)
```

$$\omega^2x(t) + \frac{d^2}{dt^2}x(t) = 0$$

Vi benytter nu dsolve() på samme måde som vi brugte solve() til at løse algebraiske ligninger.

[66]:
```
solution = sp.dsolve(ODE, x)
display(solution)
```

$$x(t) = C_1e^{-i\omega t} + C_2e^{i\omega t}$$

Det er altså forholdvis lige til at løse differentialligninger. dsolve() har dog en del ekstra arguemnter, som man kan give den. Her er det helt sikkert værd af fremhæve "ics"-argumentet, som lader os give dsolve() vores randbetingelser. Disse skal gives som et dictionairy på følgende måde: {*funtion(hvor)*:*hvad*, *funktion(hvor)*:*hvad* osv.}. Lidt sløset format, men herunder kan man se eksempel, når vi kender start betingelserne $x(0) = 0$ og $\frac{dx}{dt}\big|_{t=0} = 1$. Vi husker, at .subs(*variabel*, *værdi*) er den måde vi algebraisk indsætter en værdi i vores udtryk.

[67]:
```
solution = sp.dsolve(ODE, x, ics = {x.subs(t, 0):0, x.diff(t, 1).subs(t, 0): 1})
display(solution)
print(" ")
display(sp.simplify(solution))
```

$$x(t) = -\frac{ie^{i\omega t}}{2\omega} + \frac{ie^{-i\omega t}}{2\omega}$$

$$x(t) = \frac{\sin(\omega t)}{\omega}$$

Vi kan også indsætte ukendte værdier, som eksempelvis $x_0$ og $v_0$:

[68]:
```
x0, v0 = sp.symbols("x_0 v_0")
solution = sp.dsolve(ODE, x, ics = {x.subs(t, 0):x0, x.diff(t, 1).subs(t, 0):␣
 ↪v0})
display(solution)
print(" ")
display(sp.simplify(solution))
```

$$x(t) = \left(\frac{x_0}{2} - \frac{iv_0}{2\omega}\right) e^{i\omega t} + \frac{(\omega x_0 + iv_0)\, e^{-i\omega t}}{2\omega}$$

$$x(t) = \frac{\left(\omega x_0 + iv_0 + (\omega x_0 - iv_0)\, e^{2i\omega t}\right) e^{-i\omega t}}{2\omega}$$

# 5 Calculus - Flere variable

Umiddelbart er dette til mange formål ikke sværere end at gøre det i en dimension. Hvis vi vil differentiere skal vi bare angive variablen, og så laver vi en partiel differentering. Der er dog nogle få ting, som giver mening at nævne.

Først kan vi partiel differentiere med hensyn til flere variable ved at give sp.diff() flere input eksempelvis først $x$ så $y$ som følge:

[69]:
```
x, y = sp.symbols("x y")
f = x * y ** 2
display(f)
f_diff = f.diff(x, y)
display(f_diff)
```

$$xy^2$$

$$2y$$

Det er desuden muligt at beregne hessematricen. Dette gøres blot med sp.hessian(*funktion, liste med variable*). Hvis vi eksempelvis vil gøre det for overstående funktion skriver vi blot:

[70]:
```
Hesse = sp.hessian(f, [x, y])
Hesse
```

[70]:
$$\begin{bmatrix} 0 & 2y \\ 2y & 2x \end{bmatrix}$$

Integration udvides ligeledes fra en til flere dimensioner ved at give denne flere inputs. Lad os integrere $sin^2(x)cos^2(y)$ over $x, y \in [0, \pi]$:

[71]:
```
f = sp.sin(x) ** 2 * sp.cos(y) ** 2
f.integrate((x, 0 , sp.pi), (y, 0, sp.pi))
```

[71]:
$$\frac{\pi^2}{4}$$

Og ligeledes ubestemte integraler:

[72]:
```
f.integrate(x, y)
```

[72]:
$$\left(\frac{x}{2} - \frac{\sin(x)\cos(x)}{2}\right)\left(\frac{y}{2} + \frac{\sin(y)\cos(y)}{2}\right)$$

## 5.1 Koordinatsystemer og gradienter

Her vil vi kort kigge på, hvordan vi tager gradienter af funktioner. Dette er ikke helt så simpelt, og man kan nok nemt slippe afsted med blot at bruge funktionerne over. Jeg vil dog alligevel kort præsentere, hvordan man benytte sympy til at regne gradienter. Dette er dog en del af SymPy's vector pakke, og vi vil derfor vende tilbage til dette senere, hvor vi går mere i dybden.

Til at starte med er det dog vigtigt at vide, at gradienten er afhængig af vores type af koordinatsystem. Vi bliver derfor nødt til at generere et. Dette gør man ved at importere *CoordSys3D* fra sympy.vector og så generere det på samme måder som symboler og funktioner. Dette Coordinatsystem har nu nogle associeret scalarer som kan findes ved at kalde .base_scalars(). Som udgangspunkt vil coordinatsystemet være kartetisk med $x, y, z$ som scalarer og $\hat{i}, \hat{j}, \hat{k}$ som enhedsvektorer.

[73]:
```
from sympy.vector import CoordSys3D
S = CoordSys3D("S")
display(S.base_scalars())
display(S.base_vectors())
```

$$(\mathbf{x_S},\ \mathbf{y_S},\ \mathbf{z_S})$$

$$\left(\hat{\mathbf{i}}_\mathbf{S},\ \hat{\mathbf{j}}_\mathbf{S},\ \hat{\mathbf{k}}_\mathbf{S}\right)$$

Hvis vi nu ønsker skal vi referere til disse skalarer som S.x, S.y, S.z og ligeledes med S.i, S.j, S.k for enhedsvektorerne. Vi kan altså nu definere en funktion i dette kordinatsystem som følge:

[74]:
```
f = S.x ** 2 + S.y ** 2 + S.x * S.y
display(f)
```

$$\mathbf{x_S}^2 + \mathbf{x_S}\mathbf{y_S} + \mathbf{y_S}^2$$

Og vi kan nu tage gradienten, hvis vi ønkser det. Vi skal blot importere gradient() fra sympy.vector

[75]:
```
from sympy.vector import gradient
grad = gradient(f)
display(grad)
```

$$(2\mathbf{x_S} + \mathbf{y_S})\hat{\mathbf{i}}_\mathbf{S} + (\mathbf{x_S} + 2\mathbf{y_S})\hat{\mathbf{j}}_\mathbf{S}$$

Bemærk, at det ikke her er nødvendigt at angive retninger for gradienten, da den blot tager de tilgængelige retninger i vores koordinatsystem. Vores gradient er nu en vektor, og hvis vi ønsker kan vi prikke den med en anden vektor (eller benytte andre seje vektor-operationer). Her skal man blot importere dot fra sympy.vector:

[76]:
```
from sympy.vector import dot
a = 3 * S.i - 1 * S.j
dot(a, grad)
```

[76]:
$$5\mathbf{x_S} + \mathbf{y_S}$$

# 6 Lineær Algebra

I følgende sektion vil vi introducere linær algebra i SymPy. Vi vil først introducere vektor og matrice elementer for derefter at se på manipulation af disse. Dernæst vil vi kigge på løsningen af lineære ligningssystem med SymPY. Når først, man har fået styr på selve Matrix-klassen er det ret ligetil at udregne de fleste ønskede værdier såsom den inverse matrix, determinanten eller egenvektorer, da man blot kan gøre det med metoder, som bliver listet i bunden af dette afsnit.

## 6.1 Matrix-klassen

For at begynde på lineær algebra, skal vi først benytte de forskellige klasser. Til dette skal vi benytte Matrix-klassen, som vi både bruger til vektorer og matricer og vil altså være helt essentiel for at kunne lave Lineær Algebra i SymPy.

Kalder vi Matrix på en liste (af enten symboler eller givne værdier) får vi en kolonne vektor.

[77]:
```
from sympy import Matrix
from sympy.abc import a, b, c

v_col = Matrix([a, b, c])
display(v_col)
```

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

Vi kan istedet få en række vektor ved at pakke listen ind. Så ledes, at vores argument er en liste i en liste:

[78]:
```
v_row = Matrix([[a, b, c]])
display(v_row)
```

$$\begin{bmatrix} a & b & c \end{bmatrix}$$

Og hvis vi ønsker at danne en Matrix, så skal vi give den en liste med rækker på samme måde, som vi ville gøre i NumPy:

[79]:
```
A = Matrix([[a, b, c], [b, c, a], [c, a, b]])
display(A)
```

$$\begin{bmatrix} a & b & c \\ b & c & a \\ c & a & b \end{bmatrix}$$

Når vi først har defineret matricer kan vi regne på dem som vi forventer. Så vi kan lægge dem sammen og gange dem ved blot at skrive det. Vi kan altså gange en vektor/matrix med en koefficient:

[80]:
```
v_row * 5
```

[80]:

$$\begin{bmatrix} 5a & 5b & 5c \end{bmatrix}$$

Gange vektor med matrix:

[81]:
```
A * v_col
```

[81]:
$$\begin{bmatrix} a^2 + b^2 + c^2 \\ ab + ac + bc \\ ab + ac + bc \end{bmatrix}$$

Gange matricer sammen:

[82]:
```
A * A
```

[82]:
$$\begin{bmatrix} a^2 + b^2 + c^2 & ab + ac + bc & ab + ac + bc \\ ab + ac + bc & a^2 + b^2 + c^2 & ab + ac + bc \\ ab + ac + bc & ab + ac + bc & a^2 + b^2 + c^2 \end{bmatrix}$$

Gange vektorer sammen:

[83]:
```
v_row * v_col
```

[83]:
$$\begin{bmatrix} a^2 + b^2 + c^2 \end{bmatrix}$$

Og ligeldes for potenser, minus og lignende.

## 6.2 Indbyggede matricer

Ud over muligheden for selv at definere sin matrice gennem arrays, har SymPy nogle indbyggede funktioner, der gør det nemt at benytte de mest brugt matricer. Dem, der er værd at nævne er:

- Identitets-matrix (eye)
- Nuller eller etter matrice (zeros, ones)
- diagonal matricer (diag)

Vi kan altså hurtig definere en matrix med overstående:

[84]:
```
from sympy.matrices import eye, zeros, ones, diag

identity = eye(3)
ener3x3 = ones(3)
zeros3x4 = zeros(3, 4)
diagonal = diag(a, b, c)

identity, ener3x3, zeros3x4, diagonal
```

[84]:
$$\left( \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \right)$$

Vi kan også give diagonal matricen en under matrix, som den så sætter ind. Hvis vi nu vil have omrking x-aksen kan vi definere den ved en rotation i planet og så et 1-tal på indgang (1,1):

[85]:
```
theta = sp.symbols("\\theta")
plan_rot = Matrix([[sp.cos(theta), sp.sin(-theta)], [sp.sin(theta), sp.
 ↪cos(theta)]])
plan_rot
```

[85]:
$$\begin{bmatrix}\cos(\theta) & -\sin(\theta)\\ \sin(\theta) & \cos(\theta)\end{bmatrix}$$

[86]:
```
rum_rot = diag(1, plan_rot)
rum_rot
```

[86]:
$$\begin{bmatrix}1 & 0 & 0\\ 0 & \cos(\theta) & -\sin(\theta)\\ 0 & \sin(\theta) & \cos(\theta)\end{bmatrix}$$

### 6.3 Matrix generator

Slutteligt er det værd at nævne, at man kan definere matricer ved at skrive funktinoer. Dette gøres ved at definere en funktion, der tager to input (række, kolonne) og returnere en værdi. Vi kan eksempelvis lave et skakbræt ved at se, om summen af række- og kolonneindekset er lige:

[87]:
```
def chess(i, j):
    if (i + j)%2 == 0:
        return 1
    else:
        return -1

chess_board = Matrix(5, 5, chess)
chess_board
```

[87]:
$$\begin{bmatrix}1 & -1 & 1 & -1 & 1\\ -1 & 1 & -1 & 1 & -1\\ 1 & -1 & 1 & -1 & 1\\ -1 & 1 & -1 & 1 & -1\\ 1 & -1 & 1 & -1 & 1\end{bmatrix}$$

Hvis man ønsker, kan dette gøres lidt mere kompakt ved brugen af lambda-funktioner:

[88]:
```
Matrix(6, 10, lambda i, j: -1 + 2 * ((i + j) % 2 == 0))
```

[88]:

$$\begin{bmatrix} 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 \end{bmatrix}$$

### 6.4 Matrix-manipulation

Ud over at vide, hvordan man definere matricer er det en god idé at vide, hvordan man sætter dem sammen/splitter dem op og generalt bare manipulere denne klasse. Vi benytter her en eksempelmatrice med indgangene 1, 2, 3... osv:

[89]:
```
A = Matrix(4, 4, lambda i, j: 1 + 4 * i + j % 5)
A
```

[89]:
$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Vi kan nu kigge på et udsnit af denne matrice ved at "slice" på samme måde, som vi ville gøre med et numpy array:

[90]:
```
A[:2, 1:4], A[:, [0, 3]] # etc. Husk at Python nul-indekserer selvom man normalt␣
↪refererer til matrix-indgange med 1-indeksering
```

[90]:
$$\left( \begin{bmatrix} 2 & 3 & 4 \\ 6 & 7 & 8 \end{bmatrix}, \begin{bmatrix} 1 & 4 \\ 5 & 8 \\ 9 & 12 \\ 13 & 16 \end{bmatrix} \right)$$

Vi kan sætte flere matricer sammen med .row_join() og .col_join() som sætter to matricer sammen langs rækkerne og kolonnerne:

[91]:
```
B = eye(4)
A.row_join(B), A.col_join(B)
```

[91]:
$$\left( \begin{bmatrix} 1 & 2 & 3 & 4 & 1 & 0 & 0 & 0 \\ 5 & 6 & 7 & 8 & 0 & 1 & 0 & 0 \\ 9 & 10 & 11 & 12 & 0 & 0 & 1 & 0 \\ 13 & 14 & 15 & 16 & 0 & 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \right)$$

Men kan også benytte funktioner på alle matrix-indgangene. Dette gøres med .applyfunc()

[92]:
```
def f(x): return x ** 2 - 1
A.applyfunc(f)
```

[92]:
$$\begin{bmatrix} 0 & 3 & 8 & 15 \\ 24 & 35 & 48 & 63 \\ 80 & 99 & 120 & 143 \\ 168 & 195 & 224 & 255 \end{bmatrix}$$

## 6.5 Matrix-reduktion

### 6.5.1 Rækkeoperationer

Det første man støder på i Lineær Algebra er at benytte række-operationer til at sætte en funktion på echelonform. Sympy har desuden den fantastiske egenskab, at den giver os outputtet i Latex. Det er altså nemt at indsætte i en opgave, hvis man benytter SymPy til at udføre ens rækkeoperationer. Her benytter man .elemtentary_row_op() og given den som argument en af følgende:

- "n->kn" Dette ganger en række med en konstant:
- "n<->m" Bytter om på to rækker
- "n+km" Lægger en konstant gange en række m til rækken n

Desuden skal man alt efter hvilken operation, man ønsker at foretage give de nødvendige arguemnter. Man kan give følgende: \begin{itemize}

*row*: Den række man udfører operationen på

*k*: Den kostant der er involveret i rækkeoperationen (kan også være et symbol)

*row1*, *row2*: De rækker man ønsker at bytte om. Desuden er *row2* m i ‘‘n+km’’ \end{itemize} Husk! At man nulindeksere i python! Første række er altså række 0

Herunder har jeg forsøgt at vise et eksempel på hver af rækkeoperationerne. Man kan desuden også lave kolonne operationer på præcis samme måde, så skriver man bare .elementary_col_op() i stedet og erstatter ‘‘row’’ med ‘‘col’’ i argumenterne.

[93]:
```
A = A.elementary_row_op("n->kn", row = 1, k = 1/5)
A
```

[93]:
$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 1.0 & 1.2 & 1.4 & 1.6 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

[94]:
```
A = A.elementary_row_op("n->n+km", row = 1, k = -1, row2 = 0)
A
```

[94]:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & -0.8 & -1.6 & -2.4 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

[95]:
```
A = A.elementary_row_op("n<->m", row1 = 1, row2 = 3)
A
```

[95]:
$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 13 & 14 & 15 & 16 \\ 9 & 10 & 11 & 12 \\ 0 & -0.8 & -1.6 & -2.4 \end{bmatrix}$$

### 6.5.2 Echelonform

Det er altså forholdvis bøvlet at lave rækkeoperationer, heldigvis er det dog sådan, at man benytter disse for at sætte en matrice på reduceret echelon. Hvis vi eksempelvis redifinerer vores matrice A, gøres dette blot ve at skrive A.echelon_form()

[96]:
```
A = Matrix(4, 4, lambda i, j: 1 + 4 * i + j % 5)
A_echelon = A.echelon_form()
A, A_echelon
```

[96]:
$$\left( \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & -4 & -8 & -12 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \right)$$

Oftest er vi dog interreseret i den reducerede form. Denne fås let med funktionen .rref (står for reduced row echelon form):

[97]:
```
A, A.rref()
```

[97]:
$$\left( \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}, \left( \begin{bmatrix} 1 & 0 & -1 & -2 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, (0, 1) \right) \right)$$

.rref() returene to værdier, det første er matricen på reduceret echelonform og det andet element er en liste over de kolonner, som har et ledende 1-tal. Dette er altså de kolonner, hvis vektorer udgør en basis.

## 6.6 Ligningssystemer

En stor fordel ved lineær algebra er at vi kan søge løsninger til et lineært ligningssystem ved at opskrive den på matrix form. Vi kan altså omskrive:

$$x_1 + 3x_3 = 20$$
$$4x_2 + 6x_3 = 74$$
$$3x_1 + 6x_2 + 11x_3 = 136$$

til:

$$\begin{bmatrix} 1 & 0 & 3 \\ 0 & 4 & 6 \\ 3 & 6 & 11 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 20 \\ 74 \\ 136 \end{bmatrix}$$

Når vi har et ligningssystem på denne form kan vi uden videre beregne en løsning emd SymPy. Vi benytter blot LUsolve metoden på en matrice, og så giver den som input højre side af vores ligningssystem. Det ser altså sådan her ud:

[98]:
```
#Definere her matrix A og vektor b
A = Matrix([[1, 0, 3], [0, 4, 6], [3, 6, 11]])
b = Matrix([20, 74, 136])

#Løsningen findes nus blot ved:
sol = A.LUsolve(b)
display(sol)
```

$$\begin{bmatrix} 5 \\ 11 \\ 5 \end{bmatrix}$$

Vi kan nu tjekke om dette virkelig er en løsning ved blot at gange matricen på denne vektor:

[99]:
```
A * sol
```

[99]:
$$\begin{bmatrix} 20 \\ 74 \\ 136 \end{bmatrix}$$

Altså har vi fundet løsningen til dette ligningssystem.

Vi kan yderligere gøre det med symboler:

[100]:
```
from sympy.abc import a, b, c
v = Matrix([a, b, c])
A.LUsolve(v)
```

[100]:
$$\begin{bmatrix} -\frac{2a}{7} - \frac{9b}{14} + \frac{3c}{7} \\ -\frac{9a}{14} - \frac{b}{14} + \frac{3c}{14} \\ \frac{3a}{7} + \frac{3b}{14} - \frac{c}{7} \end{bmatrix}$$

Hvilket jo ligner, at vi blot har ganget en invers matrice på vektoren. (hvilket jo også er en fin måde at løse ligninger på)

## 6.7 Udregning af matrix-egenskaber

Meget af dette er forholdsvist trivielt, så jeg laver bare lige en kort liste over nyttige funktioner for nu:

- .det() Udregner determinant
- .trace() Udregner tracen (summen langs diagonalen)
- .inv Finder den inverse matrice
- .eigenvals() Udregner egenværdier
- .eigenvects() Udregner egenværdier og egenvektorer
- .nullspace() Beregner en basis for kernen/nulrummet
- .diagonalize() Diagonalisere matricen
- .T giver den transponerede matrice
- .adjoint giver den adjungerede matrice (komplekskonjugeret og transponeret)

Desuden kan man komme ud for at skulle bruge følgende:

- GramSchmidt() som kan importeres fra sympy.matrices