**guide.md**

# Hvordan åbner jeg dette dokument?

Dette er en pdf-udskrift af iPython-Notebook-filen: `sympy-for-begyndere.ipynb`. For at åbne selve iPython-filen kan du enten:

## 1. Med Google Colab

Gå ind på colab.research.google.com og upload filen under fanen "Upload".

## 2. Med Python installeret på din computer

**Installér Python:** Det gør du nemmest ved at installere den såkaldte Anaconda-distribution:

1. Gå ind på anaconda.com og tryk Download.
2. Følg installationsguiden
3. Tjek, at du har installeret korrekt ved at åbne programmet Anaconda-Navigator (læs mere her: https://docs.anaconda.com/free/anaconda/install/verify-install/)
4. Hvis du oplever problemer, kan du få mere hjælp her: https://docs.anaconda.com/free/anaconda/install/

**Åbn filen**:

1. Åbn Anaconda-Navigator fra dine programmer/apps på din computer.
2. Åbn **Jupyter Notebook** (med "Launch"-knappen)
3. Du ser nu en oversigt over dit filsystem. Navigér hen til den mappe, hvor filen ligger og dobbeltklik på den for at åbne den.

# SymPy for begyndere

SymPy er et gratis, open-source Python-bibliotek, der mulliggør at man kan regne matematik symbolsk. Det betyder, at SymPy kan evaluere og simplificere matematiske udtryk eksakt, løse ligninger, tegne funktioner, differentiere, integerere og meget mere! SymPy er derfor et gratis alternativ til CAS-systemer som fx Maple, Ti-Nspire eller Mathematica.

**Forudsætninger i Python-programmering**

Da SymPy lever og ånder i Python-økosystemet, kræver denne introduktion, at du kender til det mest basale inden for Python-programmering. Specifikt skal du have lært om variable, funktioner og datatyper (tal, tekst, lister og dictionaries). Det er også en fordel at have arbejdet lidt med NumPy-biblioteket. Du kan fx lære det nødvendige i StudyNows kursus i Python for begyndere.

# Kom i gang

**Installering**

For at kunne bruge SymPy (og for at køre dette dokument) kræver det, at du har Python installeret og har installeret SymPy.

Begge dele burde være installeret korrekt, hvis bare du bruger Anaconda-distributionen.

Læs evt. mere om installation af SymPy her.

## Det interaktive "IPython Notebook" format

Der er mange forskellige redigeringsværktøjer (IDE'er) til at skrive Python-kode: Bl.a. Google Colab, Jupyter Notebook, Spyder og Visual Studio Code. Jeg foretrækker selv Visual Studio, men som begynder vil jeg anbefale dig enten at bruge Colab (kræver ikke installering, men åbnes via Google Drive) eller Jupyter (køres "lokalt" på din computer, efter du har installeret Anaconda-distributionen af Python). Både Colab og Jupyter kan åbne denne fil i et interaktivt "notebook"-format. Det betyder fx, at man kan skrive tekst som dette og vise grafer og resultatet af sine beregninger direkte i dokumentet.

### Genveje i Google Colab Notebook

| Kommando | Genvej |
|---|---|
| Kør celle | shift + enter |
| Kør celle og indsæt ny under | alt (option) + enter |
| Ændr celle til tekst | ctr + M, M |
| Ændr celle til kode | ctr + M, Y |
| Fjern celle | ctr + D, D |

## Indtasting af specielle tegn

| Tegn | Mac dansk tastatur | Windows dansk tastatur | US tastatur |
|---|---|---|---|
| { } | alt + shift + 8 og alt + shift + 9 | AltGR + 7 og Altgr + 0 | Shift + [ ] (right to P key) |
| [ ] | alt + 8 og alt + 9 | AltGR + 8 og Altgr + 9 | [ ] (right to P key) |
| \ | alt + shift + 7 | ctrl + alt + < eller AltGr + 9 | \ (left to enter) |
| _ | shift + - | shift + - | shift + - |

## Importering af SymPy og andre biblioteker

Nedenfor importerer jeg de biblioteker, som vi får brug for i dette dokument.

In [ ]:
```
import sympy as sp
import numpy as np
from IPython.display import display
```

Nu kan vi fremadrettet referere til SymPy bilbioteket som `sp` og til NumPy biblioteket som `np`.

### Slet alle variable

Man kan slette alle variable og importerede pakker i ens Notebook-session med kommandoen:

In [ ]:
```
%reset -f
```

Importér de pakker, vi skal bruge igen bagefter:

In [ ]:
```
import sympy as sp
import numpy as np
from IPython.display import display
```

## Genveje i Jupyter Notebook (se kun hvis relevant)

| Kommando | Genvej |
|---|---|
| Kør celle | shift + enter |
| Kør celle og indsæt ny under | alt (option) + enter |
| Ændr celle til tekst | esc + M |
| Ændr celle til kode | esc + Y |
| Fjern celle | esc + 2 x D |

# Basale Regnestykker

## Numerisk vs. symbolsk matematik

For at forstå, hvad SymPy kan, er det vigtigt at forstå forskellen på symbolsk og numerisk matematik. Numeriske beregninger kan du udføre med Python og NumPy-biblioteket, mens symbolsk matematik klares med Python og SymPy-biblioteket.

Antag, at vi gerne vil bestemme værdien af $(\sqrt{8})^2$. Dette kan vi gøre numerisk vha. NumPy:

In [ ]:
```
np.sqrt(8)**2
```

Out[ ]:
```
8.000000000000002
```

**Bemærk:**

1. I Python opløfter man et tal i en potens med en dobbelt-asterisk `**` i stedet for `^`-tegnet (`^`-tegnet betyder i Python logisk ekskluderende eller)
2. kvadratroden beregnes numerisk med NumPy's `sqrt`-funktion

Det numeriske resultat af $\sqrt{8}^2 = 8$ introducerer et ukorrekt 2-tal, som det 15. decimal. Det skyldes, at vi regner med et endeligt antal decimaler og derfor har en begrænset præcision. I langt de fleste tilfælde er præcisionen i numeriske beregninger som ovenfor selvfølgelig rigeligt, men det illustrerer alligevel at numerisk matematik kun er en tilnærmelse til det eksakte resultat.

Vi kunne i stedet bestemme det eksakte resultat som en symbolsk beregning:

In [ ]:
```
sp.sqrt(8)**2
```

Out[ ]: $8$

Når vi regner symbolsk, får vi det eksakte heltal $8$ som resultat i stedet for decimaltals-tilnærmelsen.

## Basale regnestykker

Der er nogle elementære funktioner som er gode at kende i SymPy

### Integer

Bruges til at angive at et tal er et heltal. Fx kan vi bestemme division af to heltal

In [ ]:
```
sp.Integer(2)/sp.Integer(6)
```

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

Bemærk, at det eksakte tal $\frac{1}{3}$ principielt er forskelligt fra den numeriske decimaltilnærmelse $0.3333333333333333$. Med Python uden SymPy får vi bare det numeriske resultat:

In [ ]:
```
2/6
```

Out[ ]:
```
0.3333333333333333
```

### Rational

En brøk mellem to heltal kan også repræsenteres med funktionen `Rational`:

In [ ]:
```
sp.Rational(2, 6)
```

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

### Matematiske konstanter

Matematiske konstanter som fx $\pi, e, i = \sqrt{-1}$ kan findes i sympy:

In [ ]:
```
sp.pi
```

Out[ ]: $\pi$

In [ ]:
```
sp.E
```

Out[ ]: $e$

In [ ]:
```
sp.I
```

Out[ ]: $i$

### Elementære matematiske funktioner

Elementære matematiske funktioner som fx $\sqrt{x}, \sin(x), \cos(x), \exp(x), \log(x)$ og $\log_{10}(x)$ har alle en sympy variant:

Fx. kan vi udregne:
$\sqrt{4} = 2, \quad \sin(\pi) = 0, \quad \cos(\pi) = -1, \quad \exp(2) = e^2, \quad \ln(e) = 1, \quad \log_{10}(1000) = 3$

In [ ]:
```
sp.sqrt(4), sp.sin(sp.pi), sp.cos(sp.pi), sp.exp(2), sp.log(sp.E), sp.log(1000
```

Out[ ]:
```
(2, 0, -1, exp(2), 1, 3)
```

### sympify

Med sympify funktionen kan du evaluere generelle udtryk skrevet i tekst. Fx:

$$\sqrt{(1+2)(3+5)+1} = 5$$

In [ ]:
```
sp.sympify('sqrt((1+2)*(3+5) + 1)')
```

Out[ ]: $5$

**Bemærk:**

1. Med `sympify` -funktionen slipper vi for at skrive " `sp.` " foran sympy-funktionerne.
2. `sympify` bruger en Python-funktion, der hedder `eval` . Det betyder, at man kan afvikle hvilken som helst kode, der proppes ind i teksstrengen til `sympify` . Det kan være et sikkerhedsproblem. Så længe, du kun bruger `sympify` i dine egne dokumenter, hvor du har fuld kontrol over, hvad du propper ind i funktionen, er der ikke noget problem dog.

Du kan også beregne flere udtryk i samme `sympify` -kald. Fx, hvis vi vil bestemme:

$$\sqrt{\frac{36}{2}} = 3\sqrt{2}, \quad \sin^{-1}(\sqrt{2}/2) = \frac{\pi}{4}, \quad e^{i\pi} = -1$$

In [ ]:
```
sp.sympify('sqrt(36/2), asin(sqrt(2)/2), exp(I*pi)')
```

Out[ ]:
```
(3*sqrt(2), pi/4, -1)
```

### IPython's display-funktion

Du kan bruge `display` -funktionen fra IPython-biblioteket til at vise resultaterne pænere (med $\LaTeX$):

In [ ]:
```
from IPython.display import display
display(*sp.sympify('sqrt(36/2), asin(sqrt(2)/2), exp(I*pi)'))
```

$3\sqrt{2}$

$\frac{\pi}{4}$

$-1$

Bemærk, at jeg skriver et asterisk-tegn `*` for at udpakke listen af resultater fra `sympify` .

### Eksplicit gangetegn mellem paranteser

Der skal være eksplicit gangetegn mellem paranteser for at `sympify` forstår det. Ellers får vi en fejl:

In [ ]:
```
# Denne linje fejler, hvis du forsøger at køre den.
sp.sympify('(3+4)(2+3)')
```

**Bestem numerisk værdi af symbolsk variabel**

`evalf` -funktionen muliggør, at man kan evaluere decimaltalsværdien af en symbolsk variabel. Fx er:

$$\sqrt{2} \approx 1.41421356237310$$

In [ ]:
```
sp.sqrt(2).evalf()
```

Out[ ]: $1.4142135623731$

# Udtryk med symbolske variable

## Symbolske variable og Python-variable

I Python kan vi ikke opskrive udtrykket $x + y$ uden først at fastsætte en værdi af $x$ og $y$.

In [ ]:
```
x + y
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/Users/rune.hoejlund/Development/StudyNow/SymPy Preperation/Introduction to SymPy.ipynb Cell 43 in <cell line: 1>()
----> <a href='vscode-notebook-cell:/Users/rune.hoejlund/Development/StudyNow/SymPy%20Preperation/Introduction%20to%20SymPy.ipynb#Y335sZmlsZQ%3D%3D?line=0'>1</a> x + y

NameError: name 'x' is not defined
```

Med SymPy, kan vi definere python-variablene `x` og `y` som de symbolske variable $x$ og $y$ :

In [ ]:
```
x, y = sp.symbols('x y')
x + y
```

Out[ ]: $x + y$

## Simplificering af udtryk

SymPy simplificerer automatisk nemme symbolske udtryk, fx er $x + y - x = y$:

In [ ]:
```
x+y-x
```

Out[ ]: $y$

Et andet eksempel kunne være simplificeringen:

$$\sin^2(x) + \cos^2(x) = 1$$

In [ ]:
```
expr = sp.sin(x)**2 + sp.cos(x)**2
expr
```

Out[ ]: $\sin^2(x) + \cos^2(x)$

Udtrykket er for kompliceret til, at SymPy ved, at den skal simplificere det automatisk, men vi kan tvinge det igennem med `simplify` -kommandoen:

In [ ]:
```
sp.simplify(expr)
```

Out[ ]: $1$

**expand**

Med `expand` -funktionen kan man gange paranteser ud:

$$x(x + y^2) = x^2 + xy^2$$

In [ ]:
```
sp.expand(x*(x+y**2))
```

Out[ ]: $x^2 + xy^2$

**factor**

Omvent kan man faktorisere et udtryk med `factor` :

In [ ]:
```
sp.factor(x**2 + x*y**2)
```

Out[ ]: $x\left(x + y^2\right)$

## Indsæt værdier af symbolske variable

Lad os definere udtrykket $x(x + y^2)$ som en variabel i Python:

In [ ]:
```
expr = x*(x+y**2)
expr
```

Out[ ]: $x\left(x + y^2\right)$

Med `subs` funktionen kan vi substituere en værdi ind for en symbolsk variabel:

Fx kan vi erstatte $x$ med $2$ i udtrykket, hvilket giver:

$$x(x + y^2)\Big|_{x=2} = 2y^2 + 4$$

In [ ]:
```
expr.subs(x, 2)
```

Out[ ]: $2y^2 + 4$

Vi kan også substituere både $x = 2$ og $y = 1$ ind i udtrykket på samme tid. Enten ved:

In [ ]:
```
expr.subs(x, 2).subs(y, 1)
```

Out[ ]: $6$

Eller ved at angive en liste af par med gamle og nye værdier `(før, efter)`:

In [ ]:
```
expr.subs([(x, 2), (y, 1)])
```

Out[ ]: $6$

**Bemærk:**

Python-variablen `expr` er stadig uændret:

In [ ]:
```
expr
```

Out[ ]: $x\left(x+y^2\right)$

Generelt vil funktioner som `subs` ovenfor ikke ændre SymPy-udtryk, men i stedet returnere nye udtryk. Fx kan vi definere resultatet af at indsætte $x=2, y=1$ som et nyt udtryk:

In [ ]:
```
expr2 = expr.subs([(x, 2), (y, 1)])
expr2
```

Out[ ]: $6$

## Lav SymPy-udtryk til Python-funktioner

Noget af det, der gør SymPy super stærkt, er at det er tæt integreret med Python og NumPy-biblioteket.

NumPy muliggør, at man lynhurtigt kan lave beregninger på tusindvis af datapunkter.

Fx kan vi bestemme $\sin^2(x)$ for 1000 værdier liggende mellem $x=0$ og $x=2\pi$

In [ ]:
```
x_np = np.linspace(0, 7, 1000)
out1 = np.sin(x_np)**2
```

Med SymPy, ville vi skulle klare dette med `for` loops, som er mindre elegante og unødigt ineffektive:

In [ ]:
```
out2 = []
for x in x_np:
    out2.append(sp.sin(x)**2)
```

**lambdify**

Heldigvis har SymPy den indbyggede funktion `lambdify`, der kan lave SymPy-udtryk om til Python-funktioner:

In [ ]:
```
x = sp.symbols('x')
expr = sp.sin(x)**2
f = sp.lambdify(x, expr, 'numpy')
```

Nu er Python-variablen `f` en NumPy-funktion, som hurtigt kan evalueres på NumPy arrays med tusindvis af datapunkter:

In [ ]:
```
out3 = f(x_np)
print('Does the SymPy lambdify result agree with np.sin(xs)**2?:', (np.sin(x_n
```

```
Does the SymPy lambdify result agree with np.sin(xs)**2?: True
```

Vi kan også indsætte vores egne udtryk for SymPy-funktioner.

Fx kunne vi tvinge vores funktion til at bruge tilnærmelsen $\sin(x) \approx x$ (som gælder for små $x$) i stedet for NumPy's sinus-funktion:

In [ ]:
```
def sin_small_x(x):
    return x

f = sp.lambdify(x, expr, {'sin': sin_small_x})
```

## Få $\LaTeX$ udtryk fra SymPy

SymPy's `latex` funktion omdanner et symbolsk udtryk til LaTeX-formattering, som du nemt kan indsætte som ligninger i LaTeX-dokumenter:

In [ ]:
```
expr = 3 * x**2/4
print(sp.latex(expr))
```

```
\frac{3 x^{2}}{4}
```

# Opgaver

1. Bestem $(9+1)(4+6)$ vha. SymPy.
2. Definér variablen $a$ til at være $\sqrt{3}$ og vis værdien af `a` i dokumentet her med $\LaTeX$ -formattering. Få LaTeX-formatterings-tekststrengen fra SymPy.
3. Bestem $e^3$ som decimaltal - først med NumPy og dernæst med SymPy
4. Tag 10-talslogartimen til 1420 med SymPy. Bestem eksakt- og decimaltalsværdi
5. Bestem $\frac{22}{7} - \pi$ som decimaltal med SymPy
6. Simplificér $\frac{1}{2}\left(e^{i2\pi x} + e^{-i2\pi x}\right)$.
7. Tjek, om $\sin^2(x) + 1/4(e^{i2x} + e^{-i2x} + 2) = 1$

## Løsninger til opgaverne

### Løsning til 1.

Kan klares med `sympify`:

In [ ]:
```
sp.sympify('(9+1)*(4+6)')
```

Out[ ]: $100$

Alternativt kan vi eksplicit angive, at tallene er heltal:

In [ ]:
```
(sp.Integer(9) + sp.Integer(1))*(sp.Integer(4) + sp.Integer(6))
```

Out[ ]: $100$

Faktisk er det tilstrækkeligt, bare at angive at ét af tallene er et heltal. Så snart, vi har introduceret en SymPy-kommando i en kodelinje, bliver hele beregningen lavet med SymPy.

In [ ]:
```
(sp.Integer(9) + 1)*(4 + 6)
```

Out[ ]: $100$

### Løsning til 2.

In [ ]:
```
a = sp.sqrt(3)
```

For at vise `a` kan vi bare skrive den på sidste linje i en kodeblok:

In [ ]:
```
a
```

Out[ ]: $\sqrt{3}$

Eller vi kan eksplicit bruge `display` -kommandoen:

In [ ]:
```
display(a)
```

$\sqrt{3}$

$\LaTeX$-tekststrengen fås med `latex` -kommandoen:

In [ ]:
```
print(sp.latex(a))
```

```
\sqrt{3}
```

### Løsning til 3.

In [ ]:
```
np.exp(3)
```

Out[ ]:
```
20.085536923187668
```

In [ ]:
```
sp.exp(3).evalf()
```

Out[ ]: $20.0855369231877$

### Løsning til 4.

In [ ]:
```
sp.log(1420, 10)
```

Out[ ]: $1 + \frac{\log(142)}{\log(10)}$

In [ ]:
```
sp.log(1420, 10).evalf()
```

Out[ ]: $3.15228834438306$

### Løsning til 5.

In [ ]:
```
(sp.Rational(22, 7) - sp.pi).evalf()
```

Out[ ]: $0.00126448926734962$

**Bemærk:**

$\pi$ fås som `sp.pi`.

### Løsning til 6.

In [ ]:
```
x = sp.symbols('x')
expr = sp.Rational(1/2) * (sp.exp(sp.I * 2 * sp.pi * x) + sp.exp(- sp.I * 2 *
sp.simplify(expr)
```

Out[ ]: $\cos(2\pi x)$

**Bemærk**

1. Det er vigtigt at bruge `Rational` frem for bare at skrive 1/2 for at få det eksakte resultat
2. $i$ fås som `sp.I`.

### Løsning til 7.

For at tjekke, om to udtryk er ens, kan vi enten trække venstre- og højresiden fra hinanden og tjekke, at det simplificerer til nul:

In [ ]:
```
LHS = sp.sin(x)**2 + sp.Rational(1, 4) * (sp.exp(sp.I * 2* x) + sp.exp(- sp.I
display(LHS)
RHS = 1
sp.simplify(LHS - RHS)
```

$$\frac{e^{2ix}}{4} + \sin^2(x) + \frac{1}{2} + \frac{e^{-2ix}}{4}$$

Out[ ]: $0$

Ligningen er altså sand.

Alternativt, kan vi bruge `.equals()`-funktionen:

In [ ]:
```
LHS.equals(RHS)
```

Out[ ]:
```
True
```

**Bemærk:**

Dette er et tilfælde, hvor `simplify` ikke automatisk kan se, at venstresiden bare reducerer til 1.

In [ ]:
```
sp.simplify(LHS)
```

Out[ ]:
$$\sin^2(x) + \frac{\cos(2x)}{2} + \frac{1}{2}$$

Generelt er vi ikke garanteret, at `simplify` finder den simpleste løsning (det er faktisk matematisk umuligt at garantere dette i alle tilfælde).

I sjældne tilfælde kan man dog forsøge, at kalde `simplify` flere gange for at se, om udtrykket kan reduceres yderligere:

In [ ]:
```
sp.simplify(sp.simplify(LHS))
```

Out[ ]:
$$1$$

# Tegn grafer og kurver

Lad os ryde alle variable og importere de nødvendige pakker igen:

In [ ]:
```
%reset -f
import sympy as sp
import numpy as np
from IPython.display import display
```

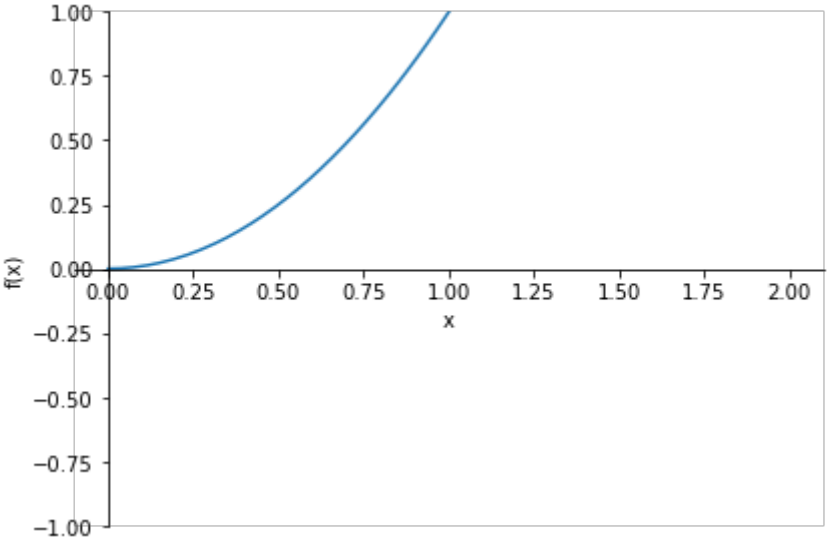
SymPy har en `plot` funktion, der gør det nemt hurtigt at tegne grafen for en funktion af symbolske variable.

## Grafer vha. SymPy

**Graf for en enkelt funktion**

Tegn grafen for funktionen $f(x) = x^2$ for $0 \leq x \leq 2$ og $-1 \leq y \leq 1$:

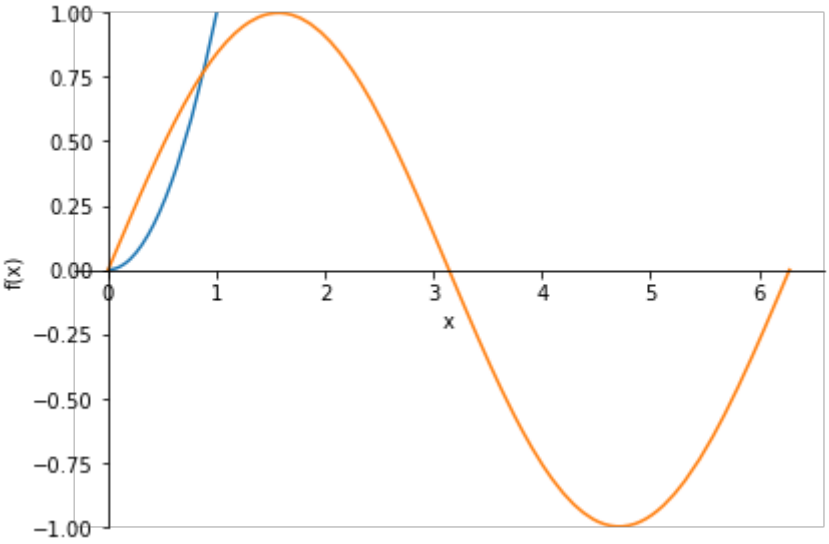
In [ ]:
```
x = sp.symbols('x')
f = x**2
p = sp.plot(f, (x, 0, 2), ylim=(-1, 1))
```




**Graf for flere funktioner**

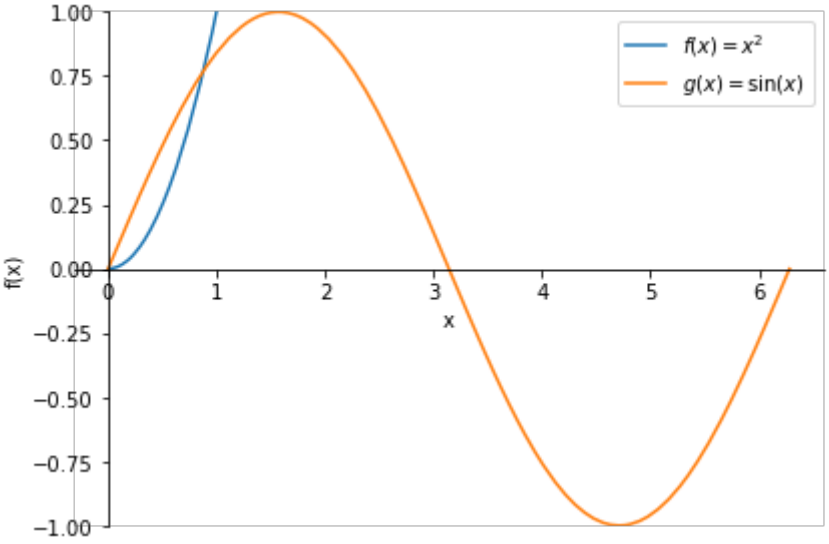
Tegn graferne for funktionerne $f(x) = x^2$ og $g(x) = \sin(x)$ i samme plot:

In [ ]:
```
g = sp.sin(x)
p = sp.plot(f, g, (x, 0, 2*sp.pi), ylim=(-1, 1))
```




En anden måde at tegne $f$ og $g$ i samme plot er vha. `extend` funktionen:

In [ ]:
```
p1 = sp.plot(f, (x, 0, 2*sp.pi), ylim=(-1, 1), show=False, label='$f(x) = x^2$
p2 = sp.plot(g, (x, 0, 2*sp.pi), ylim=(-1, 1), show=False, label='$g(x) = \sin
p1.extend(p2)
p1.show()
```




Bemærk, at indstillingen `show=False` gør at plottet først vises, når vi kalder `p1.show()`.

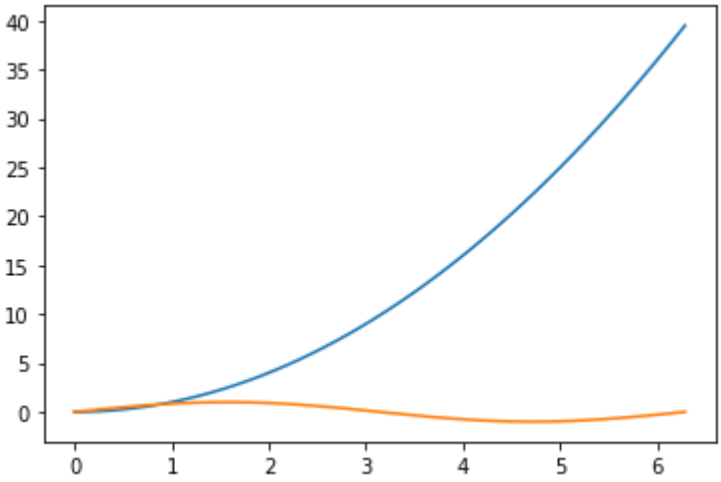
## Grafer vha. Matplotlib

Ønsker du pænere plots end SymPy kan give, anbefaler jeg at gå over til at tegne plotsene direkte i Matplotlib.

Dette kan du gøre med SymPy's `lambdify` funktion nævnt tidligere.

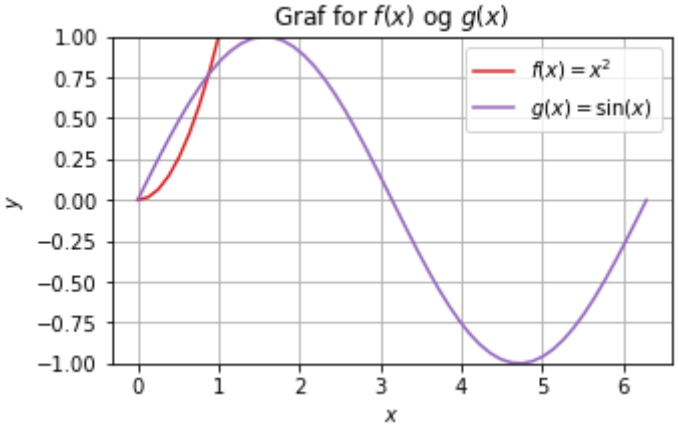
Fx kan vi tegne de to funktioner ovenfor vha. koden:

In [ ]:
```
import matplotlib.pyplot as plt
x_np = np.linspace(0, 2*np.pi)
f_np = sp.lambdify(x, f)
g_np = sp.lambdify(x, g)

plt.plot(x_np, f_np(x_np))
plt.plot(x_np, g_np(x_np))
plt.show()
```




Med Matplotlib kan man fx justere figurstørrelse, farver og tilføje gitterlinjer, legend, xlabel, ylabel og titel:

In [ ]:
```
import matplotlib.pyplot as plt
x_np = np.linspace(0, 2*np.pi)
f_np = sp.lambdify(x, f)
g_np = sp.lambdify(x, g)

plt.figure(figsize=(5, 3))
plt.plot(x_np, f_np(x_np), '-', color='tab:red', label=r'$f(x) = x^2$')
plt.plot(x_np, g_np(x_np), '-', color='tab:purple', label=r'$g(x) = \sin(x)$')
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.grid()
plt.ylim(-1, 1)
plt.legend()
plt.title('Graf for $f(x)$ og $g(x)$')
plt.show()
```




**Pæne plots med $\LaTeX$**

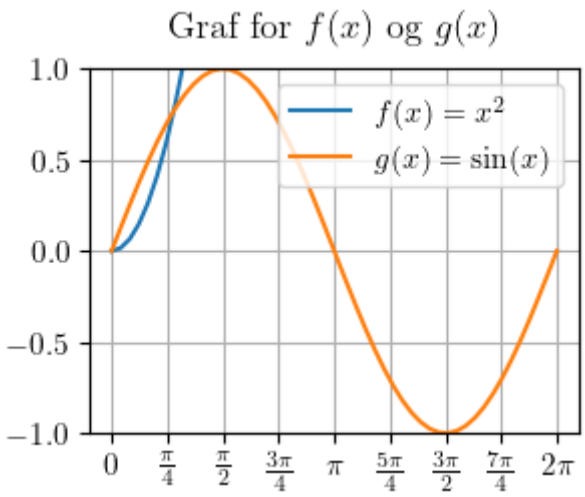
Man kan faktisk få Matplotlib til at skrive titler, labels m.m. med $\LaTeX$ for at få super pæne plots.

Du kan indsætte nedenstående kodecelle i starten af dine notebooks for at alle plots bliver vist med $\LaTeX$. Jeg har gjort kodecellen inaktiv ved at kommentere kodelinjerne, da indstillingen gør dokumentet lidt langsomere og kræver, at man har $\LaTeX$ installeret.

In [ ]:
```
# # Insert this in the start of your IPython notebook to render plots with LaT
# import matplotlib.pyplot as plt
# import matplotlib as mpl
# from matplotlib import rcParams
# rcParams.update(mpl.rcParamsDefault)
# rcParams.update({
#     "text.usetex": True,
#     "font.family": "serif",
#     "font.sans-serif": ["Computer Modern Roman"],
#     "font.size": 11})
# rcParams['axes.titlepad'] = 11

# cm = 1/2.54 # 1 cm = 1/2.54 inch
```

Eksemplet nedenfor illustrerer resultatet af vores plot med $\LaTeX$-rendering:

In [ ]:
```
# x_np = np.linspace(0, 2*np.pi)
# xticks = np.linspace(0, 2*np.pi, 9)
# xtick_labels = [r'$' + sp.latex(i * sp.pi/4) + '$' for i in range(len(xticks
# f_np = sp.lambdify(x, f)
# g_np = sp.lambdify(x, g)

# plt.figure(figsize=(8*cm, 6*cm))
# plt.plot(x_np, f_np(x_np), '-', label=r'$f(x) = x^2$')
# plt.plot(x_np, g_np(x_np), '-', label=r'$g(x) = \sin(x)$')
# plt.xlabel('$x$')
# plt.ylabel('$y$')
# plt.xticks(xticks, xtick_labels)
# plt.grid()
# plt.ylim(-1, 1)
# plt.legend()
# plt.title('Graf for $f(x)$ og $g(x)$')
# plt.show()
```




## Mere avancerede plots

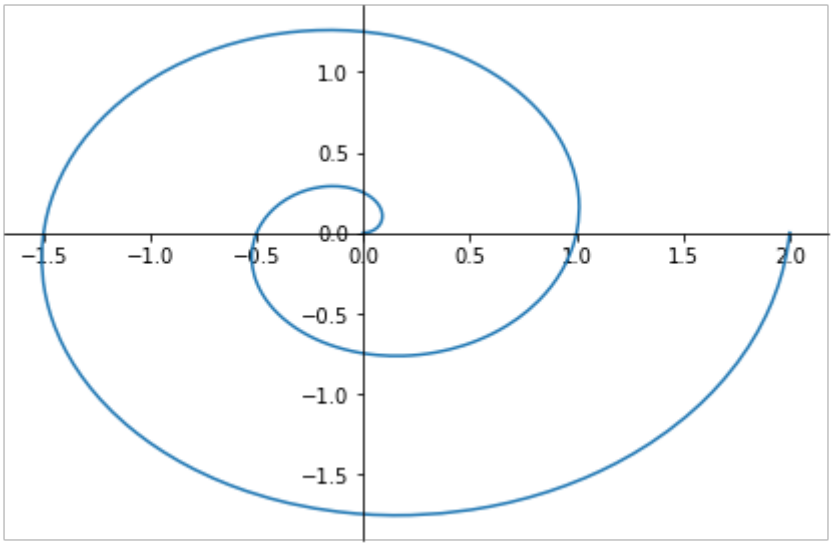
### Parametrisk plot

Tegn den parametriske kurve

$$(x(t), y(t)) = (t\cos(2\pi t), t\sin(2\pi t))$$

for $t \in [0, 2]$:

In [ ]:
```
t = sp.symbols('t')
f = (t * sp.cos(2*sp.pi*t), t * sp.sin(2*sp.pi*t))
p = sp.plot_parametric(f, (t, 0, 2))
```

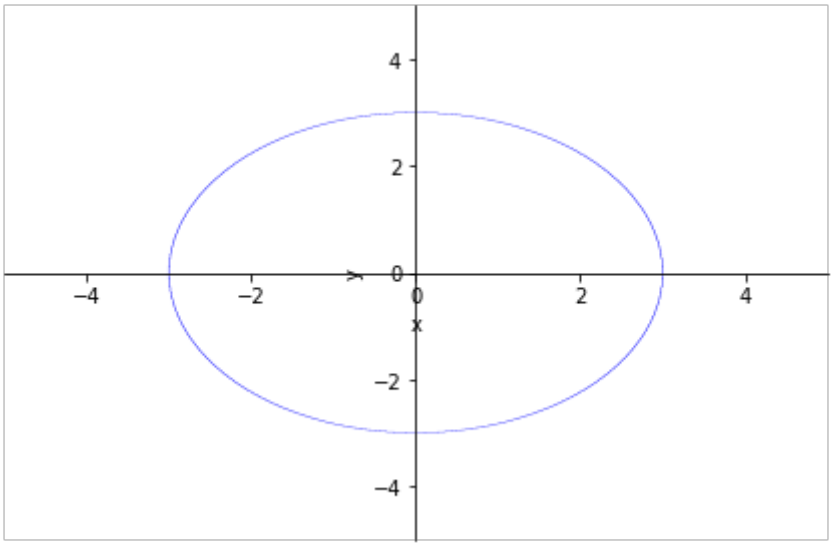



## Implicit plot

Tegn en cirkel med radius 3. Dette kan defineres som $(x, y)$-punkterne, der opfylder

$$x^2 + y^2 = 3^2$$

In [ ]:
```
x, y = sp.symbols('x y')
p = sp.plot_implicit(sp.Eq(x**2 + y**2, 9))
```



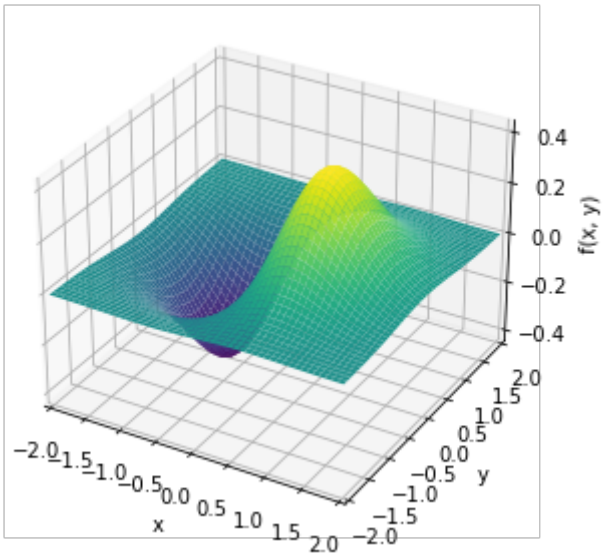

## 3D Plot

Tegn grafen for en funktion af to variable. Fx

$$f(x, y) = xe^{-x^2-y^2}$$

i intervallet $-2 \leq x \leq 2$ og $-2 \leq y \leq 2$:

In [ ]:
```
from sympy.plotting import plot3d
x, y = sp.symbols('x y')
p = plot3d(x*sp.exp(-x**2 - y**2), (x, -2, 2), (y, -2, 2))
```




# Løsning af ligninger

## Løsning af en enkelt algebraisk ligning

### Repræsentation af ligninger med `Eq`

I Python bruges lighedstegnet til at definere Python-variable.

Derfor, kan vi ikke bare bruge lighedstegnet for at repræsentere ligninger.

I stedet skal vi bruge `Eq` -funktionen. Fx kan ligningen

$$x^2 = y - 2$$

repræsenteres ved:

In [ ]:
```
x, y = sp.symbols('x y')
eq = sp.Eq(x**2, y-2)
eq
```

Out[ ]: $x^2 = y - 2$

### solve-kommandoen

Vi kan løse ligningen for $x$ med `solve` -kommandoen. Ligningen har løsningerne

$$x = \sqrt{y-2} \quad \text{og} \quad x = -\sqrt{y-2}$$

In [ ]:
```
sp.solve(eq, x)
```

Out[ ]:
```
[-sqrt(y - 2), sqrt(y - 2)]
```

Tilsvarende kan vi finde løsningen $y = x^2 + 2$ ved at isolere for $y$ i stedet:

In [ ]:
```
sp.solve(eq, y)
```

Out[ ]:
```
[x**2 + 2]
```

#### Udtræk løsninger fra `solve` -kommandoen

Bemærk, at vi med `solve` -kommandoen kan arbejde videre med resultatet. Fx kan vi evaluere den første af løsningerne ved $y = 5$

In [ ]:
```
sol = sp.solve(eq, x)
sol[0].subs(y, 5)
```

Out[ ]: $-\sqrt{3}$

Igen kan vi få resultatet som decimaltal ved at bruge `evalf` :

In [ ]:
```
sol[0].subs(y, 5).evalf()
```

Out[ ]: $-1.73205080756888$

Vi kan også gøre det samme for alle løsninger ved hjælp af en list comprehension (en måde at lave for loops i én linje i Python):

In [ ]:
```
[s.subs(y, 5).evalf() for s in sol]
```

Out[ ]:
```
[-1.73205080756888, 1.73205080756888]
```

#### solveset-kommandoen

Som et alternativ til `solve` kan man bruge `solveset` -kommandoen:

In [ ]:
```
sp.solveset(eq, x)
```

Out[ ]: $\left\{-\sqrt{y-2}, \sqrt{y-2}\right\}$

`solveset` angiver løsningen for $x$ som en løsningsmængde, hvilket er mere matematisk korrekt og mere læsbart.

Til gengæld kan vi så ikke generelt programmatisk udtrække løsningen, som vi kunne tidligere med `solve` -kommandoen.

## Løsningsdomænet

#### Antagelser om variable

Vi kan indkode visse antagelser om variablene $x$ og $y$. Betragt fx ligningen:

$$x^4 = 16$$

Uden antagelser har ligningen 4 komplekse løsninger:

In [ ]:
```
x = sp.symbols('x')
eq = sp.Eq(x**4, 16)
sp.solve(eq, x)
```

Out[ ]:
```
[-2, 2, -2*I, 2*I]
```

Kræver vi derimod, at $x$ er reel, så er der kun 2 mulige løsninger:

In [ ]:
```
x = sp.symbols('x', real=True)
eq = sp.Eq(x**4, 16)
sp.solve(eq, x)
```

Out[ ]:
```
[-2, 2]
```

**Begræns løsningsdomænet med `solveset`**

`solveset` -kommandoen giver flere muligheder for at kontrollere løsningsdomænet.

Fx kan vi igen begrænse $x \in \mathbb{R}$ ved at angive `domain` :

In [ ]:
```
x = sp.symbols('x')
eq = sp.Eq(x**4, 16)
sp.solveset(eq, x, domain=sp.S.Reals)
```

Out[ ]: $\{-2, 2\}$

**Løsninger i et interval**

Et andet eksempel kunne være, at vi vil finde løsninger til ligningen $\sin(x) = 0$ i intervallet $x \in [0, 2\pi]$:

In [ ]:
```
x = sp.symbols('x')
sp.solveset(sp.sin(x), x, domain=sp.Interval(0, 2*sp.pi))
```

Out[ ]: $\{0, \pi, 2\pi\}$

**Ligninger med uendelig mange løsninger**

Ligningen $\sin(x) = 0$ har faktisk uendeligt mange løsninger, fordi $\sin(x)$ er en periodisk funktion.

Vi kan finde alle løsninger vha. `solveset` :

In [ ]:
```
sp.solveset(sp.sin(x), x)
```

Out[ ]: $\{2n\pi \mid n \in \mathbb{Z}\} \cup \{2n\pi + \pi \mid n \in \mathbb{Z}\}$

$\mathbb{Z}$ er mængden af heltal og $\cup$ betegner foreningsmængden. Løsningerne er altså lige eller ulige multiplum af $\pi$. Dette kan skrives simplere som heltal af $\pi$:

$$\{n\pi \mid n \in \mathbb{Z}\}$$

**Bemærk**

Hvis vi bruger `solve` i stedet for `solveset` får vi kun løsninger inden for første periode ( $x \in [0, 2\pi[$ )af den periodiske funktion $\sin(x)$:

In [ ]:
```
sp.solve(sp.sin(x), x)
```

Out[ ]:
```
[0, pi]
```

## Ligningssystem med flere ligninger

Brug `solve` -kommandoen for at løse flere ligninger med flere ubekendte. Fx

$$\frac{1}{3}x = 2y + 4, \quad \frac{1}{3}x + y = 20$$

In [ ]:
```
x, y = sp.symbols('x y')
eq1 = sp.Eq(sp.Rational(1, 3) * x, 2 * y + 4)
eq2 = sp.Eq(sp.Rational(1, 3) * x + y, 20)
sol = sp.solve([eq1, eq2])
sol
```

Out[ ]:
```
{x: 44, y: 16/3}
```

Vi kan udtrække løsningen for $x$ via:

In [ ]:
```
sol[x]
```

Out[ ]:
$44$

Fx kunne vi evaluere funktionen $f(x, y) = \sqrt{x^2 + y^2}$ numerisk i løsningspunktet $(x, y) = \left(44, \frac{16}{3}\right)$:

In [ ]:
```
f = sp.sqrt(x**2 + y**2)
f.subs([(x, sol[x]), (y, sol[y])]).evalf()
```

Out[ ]:
$44.3220537029191$

## Løsning af uligheder

Med kommandoen `reduce_inequalities` kan SymPy løse simple uligheder af én variabel. Fx har uligheden

$$x(1 - x) > 0$$

løsningen $0 < x < 1$:

In [ ]:
```
sp.reduce_inequalities([x*(1-x) > 0], x)
```

Out[ ]:
$0 < x \wedge x < 1$

Vi kan også kombinere ligninger og uligheder. Fx kan vi finde løsninger større end $1$ for ligningen:

$$x^2 - 3x = -2, \quad 1 < x$$

In [ ]:
```
sp.reduce_inequalities([sp.Eq(x**2 - 3*x, - 2), 1 < x])
```

Out[ ]: $x = 2$

## Numerisk løsning af ligninger

Ikke alle ligninger giver et brugbart svar, når vi forsøger at løse dem eksakt. Fx er

$$\cos(x) = \frac{1}{7}$$

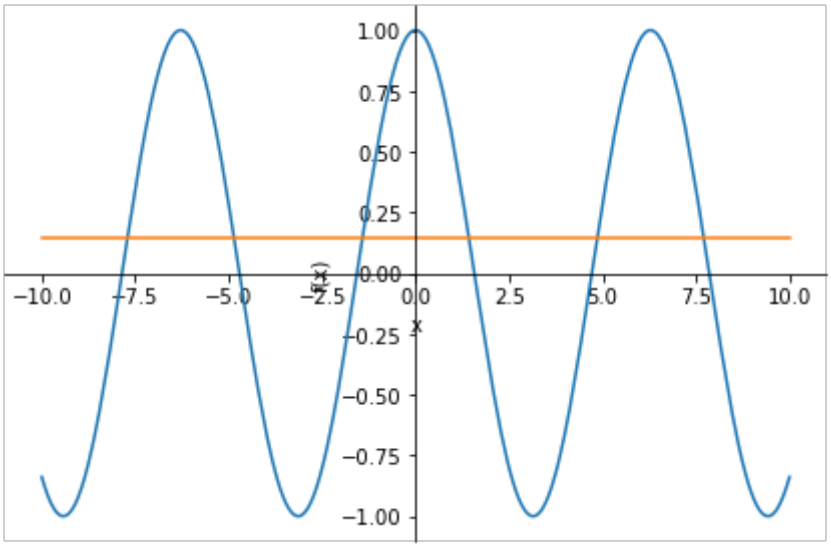
en transcendental ligning, hvis løsning ikke kan skrives mere eksakt end ovenfor. `solve`-kommandoen giver derfor ikke rigtig et brugbar svar:

In [ ]:
```
eq = sp.Eq(sp.cos(x), sp.Rational(1, 7))
sp.solve(eq)
```

Out[ ]:
```
[-acos(1/7) + 2*pi, acos(1/7)]
```

For at blive klogere, kan vi tegne de to funktioner, som skærer hinanden, der hvor løsningen er opfyldt:

In [ ]:
```
p = sp.plot(sp.cos(x), sp.Rational(1, 7))
```




Hvis vi ønsker at finde værdien af den første positive løsning, kan vi bruge SymPy's `nsolve`-kommando med $x = 2$ som et startgæt:

In [ ]:
```
sp.nsolve(eq, x, 2)
```

Out[ ]: $1.42744875788953$

**Bemærk:**

Den numeriske løsningsmetode er ret sensitiv til vores startgæt. Fx giver et startgæt på $x = 0$:

In [ ]:
```
sp.nsolve(eq, x, 0)
```

Out[ ]: $-6715.29764461709$

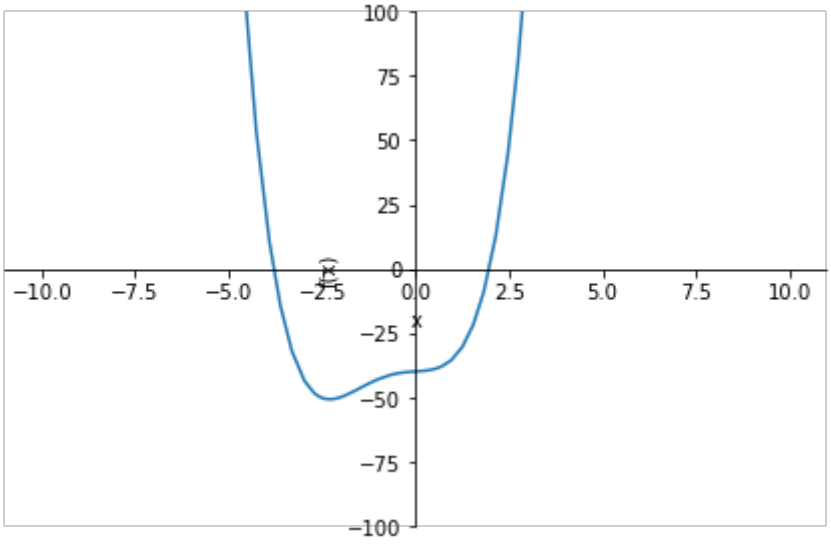
**Et andet eksempel**

Et andet eksempel kunne være ligningen:

$$x^4 + 3x^3 + x - 40 = 0$$

Igen kan vi tegne grafen for funktionen for ca. at finde nulpunkterne.

In [ ]:
```
eq = x**4 + 3*x**3 + x - 40
p = sp.plot(eq, ylim=(-100, 100))
```




Gode startgæt kunne derfor være $x \approx -3.5$ og $x \approx 2$:

In [ ]:
```
sp.nsolve(eq, x, -3.5), sp.nsolve(eq, x, 2)
```

Out[ ]:
```
(-3.79889680321069, 1.97048844688843)
```

Uden et godt startgæt, kan vi ikke finde en løsning:

In [ ]:
```
# Denne kodelinjer fejler, hvis du kører den
sp.nsolve(eq, x, 0)
```

# Grænseværdier

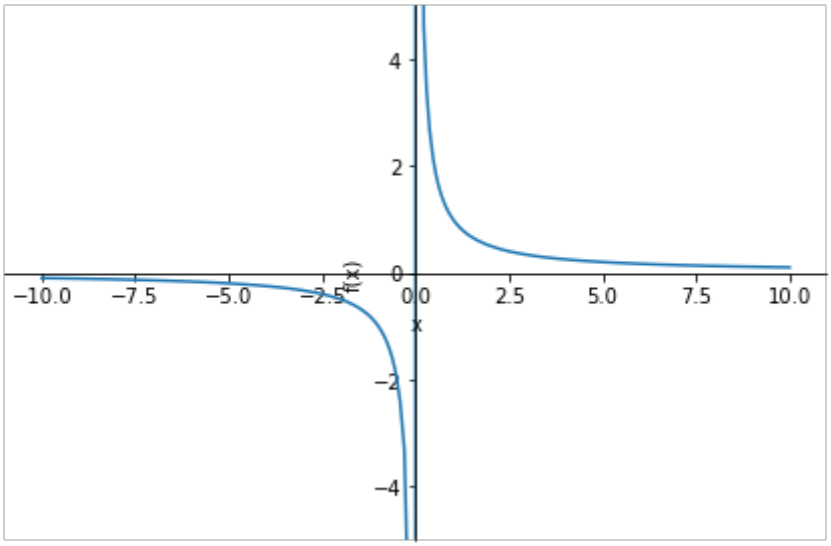
## Grænseværdi for endeligt tal

Betragt funktionen

$$f(x) = \frac{1}{x}$$

Vi kan opskrive funktionen med SymPy og tegne dens graf:

In [ ]:
```
f = 1/x
p = sp.plot(f, (x, -10, 10), ylim=(-5, 5))
```




Med SymPy kan vi bestemme grænseværdien af $f(x)$ for $x$ gående imod $2$.

Da $f(x)$ er kontinuert for $x > 0$ bør resultatet være:

$$\lim_{x \to 2} \frac{1}{x} = 1/2$$

In [ ]:
```
f = 1/x
sp.limit(f, x, 2)
```

Out[ ]: $\frac{1}{2}$

## Højre/Venstre grænseværdier

SymPy kan ikke automatisk spotte, når en grænseværdi ikke er veldefineret, som det er tilfældet for $1/x$ når $x \to 0$. I dette tilfælde er grænseværdien fra højre forskellig fra grænseværdien fra venstre. Vi kan dog eksplicit angive, at vi ønsker den ene frem for den anden grænseværdi:

In [ ]:
```
# Generel grænseværdi er i virkeligheden ikke
# veldefineret, men SymPy giver et bud alligevel:
sp.limit(f, x, 0)
```

Out[ ]: $\infty$

In [ ]:
```
sp.limit(f, x, 0, dir='+') # Grænseværdi fra højre
```

Out[ ]: $\infty$

In [ ]:
```
sp.limit(f, x, 0, dir='-') # Grænseværdi fra venstre
```

Out[ ]: $-\infty$

## Uendelige grænseværdier

Uendelig kan skrives som `oo` (to o'er) i SymPy. Dette er simpelthen bare fordi, at det er nemt at skrive og ligner tegnet $\infty$.

In [ ]:
```
sp.oo
```

Out[ ]: $\infty$

Vi kan også bestemme grænseværdier for $x \to \infty$ eller $x \to -\infty$. Fx:

In [ ]:
```
sp.limit(1/x, x, sp.oo)
```

Out[ ]: $0$

# Differentialregning

## Differentiering af en funktion af én variabel

Vi kan differentiere en funktion med SymPy's `diff` -kommando. Fx er

$$\frac{d}{dx}\left(\frac{1}{x}\right) = -\frac{1}{x^2}$$

In [ ]:
```
sp.diff(1/x, x)
```

Out[ ]: $-\frac{1}{x^2}$

**Højereordensafledede**

Vi kan bestemme den afledede til vilkårlig orden ved at angive ordnen som et tal.

Fx er den andenordensafledede:

$$\frac{d^2}{dx^2}\left(\frac{1}{x}\right) = \frac{2}{x^3}$$

In [ ]:
```
sp.diff(1/x, x, 2)
```

Out[ ]: $\frac{2}{x^3}$

## Differentiering af en funktion af flere variable

Vi kan bruge fuldstændig samme syntaks (programmatisk grammatik) som ovenfor til at bestemme partielle afledede. Som et eksempel har

$$h(x, y) = yx^2 + x\sin(y)$$

den partielt afledede mht. $x$:

$$\frac{\partial}{\partial x} h(x, y) = 2xy + \sin(y)$$

In [ ]:
```
x, y = sp.symbols('x y')
h = y*x**2 + x * sp.sin(y)
sp.diff(h, x)
```

Out[ ]: $2xy + \sin(y)$

**Blandede partielt afledede**

Vi kan også bestemme partielt afledede med blandede variable. Fx er

$$\frac{\partial^3}{\partial y \partial x^2} h(x, y) = 2$$

In [ ]:
```
sp.diff(h, x, x, y)
```

Out[ ]: $2$

# Integralregning

## Ubestemt Integration

Vi kan bestemme stamfunktionen til en funktion med `integrate`-kommandoen. Fx er:

$$\int \frac{1}{x}\, dx = \ln(x)$$

In [ ]:
```
sp.integrate(1/x)
```

Out[ ]: $\log(x)$

## Bestemt Integration

For at evaluere et bestemt integrale, kan vi angive variablen og integrationsgrænserne i en tuple. Fx er

$$\int_0^3 x^2\, dx = 9$$

In [ ]:
```
sp.integrate(x**2, (x, 0, 3))
```

Out[ ]: $9$

### Uendelige integraler

SymPy kan også bestemme integraler, hvor grænserne går mod $\pm\infty$. Fx er

$$\int_0^\infty e^{-x}\,dx = 1$$

In [ ]:
```
sp.integrate(sp.exp(-x), (x, 0, sp.oo))
```

Out[ ]: $1$

## Integration over flere variable

Vi kan også bestemme integraler over flere variable. Fx er

$$\int_{-\infty}^\infty \int_{-\infty}^\infty e^{-x^2-y^2}\,dx\,dy = \pi$$

In [ ]:
```
sp.integrate(sp.exp(-x**2-y**2), (x, -sp.oo, sp.oo), (y, -sp.oo, sp.oo))
```

Out[ ]: $\pi$

# Lineær Algebra

### Matricer

Vi kan opskrive matricer i SymPy med `Matrix`-kommandoen. Fx

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

Som input angiver vi en liste med en liste indeni

In [ ]:
```
A = sp.Matrix([
    [1, 2],
    [3, 4],
    [5, 6]])
A
```

Out[ ]: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$

En enkelt liste vil blive tolket som en kolonnevektor. Fx kan vi opskrive

$$v = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

ved

In [ ]:
```
v = sp.Matrix([0, 1, 0])
v
```

Out[ ]:
$$\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

## Matrixoperationer

### Matrixmultiplikation

Multiplikation mellem matricer klares nemt med `*` -tegnet:

In [ ]:
```
v.T*A
```

Out[ ]:
$$\begin{bmatrix} 3 & 4 \end{bmatrix}$$

### Skalering og addition

Skalering og addition fungerer, som man ville forvente:

In [ ]:
```
A + 2*A
```

Out[ ]:
$$\begin{bmatrix} 3 & 6 \\ 9 & 12 \\ 15 & 18 \end{bmatrix}$$

## Matrixinversion

Opløft en matrice i $-1$ for at bestemme den inverse:

In [ ]:
```
A = sp.Matrix([
    [1, 2],
    [3, 4]])
A**(-1)
```

Out[ ]:
$$\begin{bmatrix} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{bmatrix}$$

Vi kan tjekke, at ovenstående matrix faktisk er den inverse til $A$:

In [ ]:
```
A * A**(-1)
```

Out[ ]: $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

## Matrixtransponering

Transponér en matrix ved at skrive `.T` efter matricen:

In [ ]:
```
display(A, A.T)
```

$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$

## Kanoniske matricer

SymPy har indbyggede funktioner til at konstruere en række hyppigt brugte matricer.

### Identitetsmatricen

Identitetsmatricen fås med `eye` -kommandoen:

In [ ]:
```
sp.eye(4)
```

Out[ ]: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

### Matricer med 0 og 1

`zeros` og `ones` giver matricer, som er henholdsvis 0 og 1 overalt:

In [ ]:
```
sp.zeros(2, 3)
```

Out[ ]: $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

In [ ]:
```
sp.ones(2, 3)
```

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

## Determinant

Determinanten bestemmes ved at kalde metoden `.det()` på en matrice:

In [ ]:
```
A.det()
```

Out[ ]: $-2$

## Egenvektorer og egenværdier

Kommandoen `eigenvals` bestemmer egenværdierne af en matrice. Egenværdierne angives som `(egenværdi: algebraisk multiplicitet)`:

In [ ]:
```
A = sp.Matrix([[1, 2], [2, -1]])
A.eigenvals()
```

Out[ ]:
```
{-sqrt(5): 1, sqrt(5): 1}
```

Kommandoen `eigenvects` bestemmer egenvektorer med tilhørende egenværdier. Resultatet angives som `(egenværdi: algebraisk multiplicitet, [egenvektorer])`:

In [ ]:
```
A.eigenvects()
```

Out[ ]:
```
[(-sqrt(5),
  1,
  [Matrix([
   [1/2 - sqrt(5)/2],
   [              1]])]),
 (sqrt(5),
  1,
  [Matrix([
   [1/2 + sqrt(5)/2],
   [              1]])])]
```

Fx er den første egenvektor $v_1 = \begin{bmatrix} \frac{1}{2} - \frac{\sqrt{5}}{2} \\ 1 \end{bmatrix}$ med tilhørende egenværdi $\lambda_1 = -\sqrt{5}$. Vi kan tjekke, at det stemmer:

In [ ]:
```
lamb1 = A.eigenvects()[0][0] # Extract first eigenvalue
v1 = A.eigenvects()[0][2][0] # Extract first eigenvector
sp.simplify(A*v1 - lamb1*v1)
```

Out[ ]: $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$