Introduktion till Python för tekniska och vetenskapliga beräkningar

Denna introduktion riktar sig till dig som redan kan en del grundläggande programmering i Python, till exempel genom att ha läst kursen DD1333. Vi går här igenom tre Python-paket som ofta används för tekniska och vetenskapliga beräkningar, och som vi kommer använda i denna kurs, nämligen NumPy, Matplotlib och SciPy. Introduktionen är upplagd så att du kan testa alla steg själv i en interaktiv Python-tolk.

Innehåll

1 Installera Python och tilläggspaket

Tips: Om du använder en av Ubuntu-datorerna i KTH:s datorsalar på plan 4 i E-huset kan du hoppa direkt till avsnitt 2, eftersom alla paket redan finns installerade där.

Tips #2: Om du redan har installerat Python på din dator tidigare kan du troligtvis fortsätta använda samma installation. Om något av paketen NumPy, Matplotlib eller SciPy saknas bör du dock lägga till dem, enligt instruktioner i avsnitt 1.1.

För att se om du redan har Python installerat på din dator kan du prova att köra kommandot `python3` i en terminal. Om Python-tolken öppnas kan du gå vidare till avsnitt 1.1. Annars måste du troligtvis installera Python, till exempel på något av följande sätt:

- Via python.org: `https://www.python.org/downloads/`

  Detta installerar en minimal Python-installation.

- Via Anaconda: `https://www.anaconda.com/download/success`

  Du kan antingen installera ”Anaconda Distribution” som inkluderar ett stort antal paket, eller Miniconda som bara inkluderar det nödvändigaste.

- Via ditt operativsystems pakethanterare (om det finns en)

När Python fungerar på din dator går du vidare till nästa avsnitt.

1.1 Nödvändiga tilläggspaket

De tre paket som vi behöver är:

- NumPy (`https://numpy.org/`)
- Matplotlib (`https://matplotlib.org/`)
- SciPy (`https://scipy.org/`)

Du kan börja med att se om paketen redan är installerade genom att öppna Python-tolken och därefter skriva in följande:

```
import numpy as np
print('NumPy version:', np.__version__)
import matplotlib as mpl
print('Matplotlib version:', mpl.__version__)
import scipy as sp
print('SciPy version:', sp.__version__)
```

Om allt fungerar skrivs versionerna på de tre paketen ut. Om något paket saknas får du ett `ModuleNotFoundError` som avbryter körningen. Då vet du att det paketet måste installeras.

Hur du installerar tilläggspaketen beror på hur du installerade Python:

- Via python.org: Använd `pip3`. I en terminal (inte i Python-tolken) kör du till exempel:

```
pip3 install numpy
pip3 install matplotlib
pip3 install scipy
```

- Via Anaconda: Använd `conda`. I en terminal (inte i Python-tolken) kör du till exempel:

```
conda install numpy
conda install matplotlib
conda install scipy
```

- Via ditt operativsystems pakethanterare: Använd pakethanteraren. Paketen heter oftast något i stil med `numpy`, `python-numpy` eller `python3-numpy` osv.

När alla tre paketen fungerar kan du gå vidare till nästa avsnitt.

1.2 Annan användbar programvara

Här är några tips på program som kan vara användbara men som inte är nödvändiga för kursen.

- IPython (`https://ipython.org/`): En extra interaktiv Python-tolk. Kan installeras med `pip3` eller `conda` enligt samma mönster som ovan (paketet heter `ipython`). Tolken startas sedan genom att köra `ipython3`.

  Tips: Om du använder IPython kan du lägga in följande i konfigurationsfilen för IPython:

  ```
  c.InteractiveShellApp.pylab = 'auto'
  c.InteractiveShellApp.extensions.append('autoreload')
  c.InteractiveShellApp.exec_lines.append('%autoreload 2')
  ```

  (Konfigurationsfilen heter som standard `profile_default/ipython_config.py` och finns i den mapp som skrivs ut när du kör `ipython3 profile list` i terminalen.)

  Den första raden gör så att Numpy och Matplotlib automatiskt importeras när du startar IPython (samma effekt som om man skulle köra `import numpy as np` och `import matplotlib.pyplot as plt`).

  De två andra raderna gör så att IPython automatiskt läser in moduler på nytt om de ändras, vilket är praktiskt när man utvecklar sin kod.

- Det går bra att skapa sina Python-skript i en vanlig texteditor och köra dem i terminalen eller i IPython, men vissa föredrar att använda en IDE (*integrated development environment*) där man både kan redigera koden och köra den. Det finns många IDE:er, här är några få exempel:
  - Spyder (`https://www.spyder-ide.org/`)
  - Visual Studio Code (`https://code.visualstudio.com/`)

- PyCharm (https://www.jetbrains.com/pycharm/)
- Jupyter (https://jupyter.org/)

Obs: Vissa IDE:er har inbyggd generativ AI, vilket i så fall måste **inaktiveras** för att inte bryta mot kursens **förbud** mot generativ AI under arbete med labbar och projekt.

2 NumPy: Vektorer och matriser

Innan du kan använda NumPy måste du importera paketet:

```
import numpy as np
```

Tips: Ovanstående rad kommer behövas i princip i alla Python-program du skriver i den här kursen. Gör det till en vana att inleda dina Python-filer med denna rad. När du testar NumPy-funktioner interaktivt i en Python-tolk, till exempel `python3` eller `ipython3`, behöver importeringen göras på nytt om du startar om tolken. För att undvika detta i IPython kan du använda tipset i avsnitt 1.2. (Du måste fortfarande ha raden i dina filer.)

Följande avsnitt är utformade för att du ska följa med steg för steg antingen i en Python-tolk eller i en skriptfil som du kör. Om du vill veta mer om en funktion kan du titta i NumPys dokumentation (https://numpy.org/doc/1.17/). Där finns också en sökfunktion.

(Version 1.17 är den NumPy-version som i skrivande stund är installerad på KTH:s datorer. Om du har en annan version på din dator bör du förstås läsa dokumentationen för den versionen istället.)

Du kan också använda den inbyggda hjälpen i Python-tolken. Prova till exempel:

```
help(np) # Visar hjälpen för NumPy-paketet
help(np.zeros) # Visar hjälpen för funktionen np.zeros
```

2.1 Skapa vektorer och matriser

NumPy använder en datatyp som kallas *array* för att representera både vektorer och matriser. En array har en *form* (”shape” på engelska) vilket är en tuple som visar hur många rader och kolumner arrayen har. Till exempel har en 5×3-matris formen `(5, 3)`. En vektor med n element har formen `(n,)`, dvs en tuple med ett element.

(Arrayer kan ha fler än två dimensioner, till exempel representerar en array med formen `(4, 5, 5)` ett matematiskt objekt med tre index, med element A_{ijk}.)

En array påminner om en Python-lista, men det finns några skillnader. Till exempel måste alla element i en array vara av samma typ, och en array har ett fixt antal element. NumPy-arrayer är oftast snabbare än Python-listor eftersom de är implementerade på en lägre nivå, närmare datorns hårdvara.

Exempel 2.1.1. För att skapa en vektor med *ekvidistanta* tal kan någon av funktionerna `np.linspace` och `np.arange` användas (testa!):

```
x1 = np.linspace(0, 1, 11)
x2 = np.arange(0, 1+0.1, 0.1)
print('x1 =', x1)
print('x2 =', x2) # x1 och x2 ska bli likadana
```

Indata till `np.linspace` är *start, slut, antal element*, medan indata till `np.arange` är *start, slut, steglängd*.

> **Obs:** För `linspace` är slutpunkten inkluderad i intervallet som genereras, medan för `arange` är slutpunkten **exkluderad** (precis som för Pythons inbyggda funktion `range`). Det är därför vi skriver `1+0.1` ovan.

Det rekommenderas att använda `np.arange` om steglängden är ett heltal, och `np.linspace` annars.

Exempel 2.1.2. För att skapa vektorer och matriser fyllda med nollor eller ettor, använd funktionerna `np.zeros` respektive `np.ones`:

```
A = np.zeros((3, 3))
B = np.ones((3, 4))
print('A =', A, sep='\n')
print('B =', B, sep='\n')
```

Indata till funktionen är formen på arrayen som ska skapas.

```
x3 = np.ones(4) # 4 betyder samma sak som (4,) i detta fall
print('x3 =', x3)
```

Uppgift 2.1.3. Funktionen `np.full` kan användas för att skapa arrayer fyllda med vilket värde som helst. Läs själv dokumentationen för `np.full` för att se hur den används. Skapa sedan en 2×5-matris `P` fylld med värdet `np.pi`. Vad är `np.pi` för värde?

Exempel 2.1.4. En arrays antal element och form kan fås på följande sätt:

```
print(A.size) # ger 9, antalet element
print(A.shape) # ger (3, 3), formen
print(x3.size) # ger 4
print(x3.shape) # ger (4,)
```

Antalet rader i en matris kan alltså fås som `A.shape[0]`, och antalet kolumner som `A.shape[1]`.

Exempel 2.1.5. Funktionen `np.array` kan skapa en array utifrån en Python-lista eller nästlad lista:

```
x4 = np.array([1., 2., 3., 4.])
C = np.array([[0, 1], [0.5, 0.5], [1, 0]])
print('x4 =', x4)
print('C =', C, sep='\n')
```

2.2 Operationer på vektorer och matriser

De flesta ”vanliga” matematiska operationer görs *elementvis* på vektorer och matriser. Om de två operanderna inte har samma form försöker NumPy matcha upp deras form så gott det går i en process som kallas *broadcasting* (mer info här). Om det inte går får du ett `ValueError` som säger att formerna inte matchar.

Uppgift 2.2.1. Givet arrayerna som skapades ovan samt

```
D = np.ones(C.shape)
x5 = np.array([0., 1., 2.])
```

testa följande operationer och beskriv vad de gör, eller om de ger ett felmeddelande:

A. `C + D`

B. `C * C`

C. `C ** 2`

D. `2 * C`

E. `1 / C`

F. `C - A`

G. `x4 % 2`

H. `x4 // 2`

I. `2 ** x4`

J. `B + x4`

K. `B + x5`

Tips: Operationerna ovan anropar implicit NumPy-funktioner såsom `np.add` (för `+`), `np.multiply` (för `*`), `np.power` (för `**`) osv. Du kan läsa om dessa funktioner i dokumentationen.

(Bra att veta: När flera operationer sätts ihop så bestäms ordningen de evalueras i av Pythons vanliga regler för evalueringsordning. Till exempel evalueras `**` innan `*` innan `+`, så att `2*x**2+3*x+5` betyder $2x^2 + 3x + 5$.)

Exempel 2.2.2. Matrismultiplikation så som vi lärt oss från linjär algebra kan utföras med operationen `@` (ekvivalent med funktionen `np.matmul`). Detta förutsätter att matrisernas former är kompatibla (annars fås ett `ValueError`). Till exempel:

```
E = np.array([[1., 1., 2.], [0., 1., 1.]]) # Ny (2, 3)-matris
F = E @ C # Resultatet blir en (2, 2)-matris
x6 = B @ x4 # Resultatet blir en (3,)-vektor
print('F =', F, sep='\n')
print('x6 =', x6)
```

Operationen `A = B @ C` motsvarar

$$A_{\clubsuit\heartsuit} = \sum_k B_{\clubsuit k} C_{k\heartsuit},$$

där ♣ och ♡ är platshållare som antingen kan representera ett index, eller avsaknad av ett index (om B eller C är en vektor).

Ovanstående innebär att även `x4 @ x4` fungerar, och motsvarar en vanlig skalärprodukt.

Exempel 2.2.3. För att lösa linjära ekvationssystem på formen $A\boldsymbol{x} = \boldsymbol{b}$ (där A är en kvadratisk matris) kan vi använda `x = np.linalg.solve(A, b)`:

```
b = np.array([1., 2.])
x7 = np.linalg.solve(F, b)
print('x7 =', x7)
```

För att kontrollera att lösningen är korrekt kan vi till exempel mäta normen $\|A\boldsymbol{x} - \boldsymbol{b}\|$ av residualen. Använd `np.linalg.norm` för vektor- och matrisnormer:

```
resnorm = np.linalg.norm(F @ x7 - b)
print('Residualens norm:', resnorm)
```

Residualens norm kanske inte är exakt noll. Detta beror på att datorn lagrar tal med ändlig precision, vilket leder till avrundningsfel. Jag fick residualens norm till 6.66×10^{-16}, och vi återkommer i avsnitt 2.4 till varför detta är en rimlig storleksordning att förvänta sig.

Uppgift 2.2.4. Testa följande funktionsanrop. Vad gör de? (Du kan förstås titta i dokumentationen eller använda `help` vid behov.)

A. `np.sum(x4)`

B. `np.sum(E)`

C. `np.sum(E, axis=0)`

D. `np.sum(E, axis=1)`

E. `np.prod(x4)`

Exempel 2.2.5. NumPy har ett stort antal matematiska funktioner som evalueras elementvis på arrayer. (Ännu fler funktioner finns i SciPy som vi kommer till i avsnitt 4.) Här är några få exempel:

```
    np.sin    np.cos    np.tan    np.arctan2    np.sqrt    np.exp
np.log    np.log10    np.maximum    np.floor    np.ceil    np.isfinite
```

En fullständig lista finns här. Titta gärna i dokumentationen eller använd `help` på några av dem för att se vad de gör.

De flesta av dessa funktioner används på liknande sätt och tar en array som indata, som funktionen evalueras elementvis på:

```
print('sqrt(x4) =', np.sqrt(x4))
print('sqrt(C) =', np.sqrt(C), sep='\n')
```

Tips: Använd array-operationer och elementvisa funktioner när det går! De är mycket snabbare än att loopa över arrayen och manuellt göra operationer på ett element i taget.

Exempel 2.2.6. För att testa om texten i tipsbubblan ovan stämmer, läs igenom nedanstående kod och kör den sedan (länk till koden: `ex_2_2_6_loop.py`):

```
'''
Jämför NumPys array-funktion med att anropa samma funktion på
varje element individuellt.
'''

import time
import numpy as np

N = 100000 # antal element
x = np.linspace(0, 1, N)

tic = time.time()
y1 = np.sin(x)
time1 = time.time() - tic
print(f'Array-funktion tog {time1} s.')

y2 = np.zeros(x.shape)
tic = time.time()
for i in range(x.size):
    y2[i] = np.sin(x[i])
time2 = time.time() - tic
print(f'Loop tog {time2} s.')

err = np.linalg.norm(y1 - y2)
print(f'Skillnad på funktionsvärden: {err}')
```

(Att kopiera längre kodblock ur ett PDF-dokument brukar fungera dåligt, eftersom indenteringen ofta inte bevaras. Kopiera istället koden från länken ovan.)

Uppgift 2.2.7. Läs om funktionerna `np.vstack`, `np.hstack` och `np.column_stack` i dokumentationen. Prova sedan följande funktionsanrop och beskriv vad de gör (eller om de ger ett felmeddelande):

A. `np.vstack((x5, x5))`

B. `np.hstack((x5, x5))`

C. `np.column_stack((x5, x5))`

D. `np.vstack((C, C))`

E. `np.hstack((C, C))`

F. `np.column_stack((C, C))`

G. `np.vstack((C, x7))`

H. `np.hstack((C, x5))`

I. `np.column_stack((C, x5))`

2.3 Indexering, omformning, transponat

Exempel 2.3.1. Vektorer indexeras precis som Python-listor (tänk på att index startar på 0):

```
print(x1[2]) # Tredje elementet i vektorn
print(x1[2:5]) # Element med index 2, 3, 4
print(x1[:-1]) # Alla element utom det sista
print(x1[1:]) # Alla element utom det första
```

Matriser indexeras med två index:

```
print(C[1,0]) # Element på rad 2, kolumn 1
print(C[1:,:]) # Plockar ut rektangulärt block från matrisen
print(C[(1,2,0),(0,0,1)]) # Element med index (1,0), (2,0) och (0,1)
```

Diagonalen till en matris kan plockas ut på något av följande sätt:

```
I = np.arange(min(A.shape))
print(A[I,I])
print(np.diag(A))
```

Om man indexerar en matris med endast ett index så får man en hel rad:

```
print(E[0]) # Första raden som en vektor
```

För att välja ut alla element som uppfyller ett visst villkor kan man göra så här:

```
I = (x1 > 0.5) # Ger en vektor med boolska värden (True/False)
print(x1[I])
x1[I] = 0.25 # Element kan skrivas över
print(x1)
```

Exempel 2.3.2. Metoden `.reshape` kan användas för att byta form på en array (antalet element får inte ändras). Den nya formen ges som indata. Till exempel:

```
g = np.arange(9)
print('g =', g)
G = g.reshape((3, 3))
print('G =', G, sep='\n')
```

Observera ordningen som elementen hamnar på i `G`: de fylls på rad för rad.

```
c = C.reshape(-1) # -1 betyder att storleken väljs automatiskt
print('c =', c)
```

Här blir c en vektor som innehåller elementen i matrisen C, rad för rad. Detta kan även åstadkommas med metoden `C.ravel()`.

Uppgift 2.3.3. Vad blir resulatet av `B + x5.reshape(-1,1)`?

Exempel 2.3.4. Transponatet till en matris `A` fås antingen som `A.T` eller `A.transpose()`:

```
print(C.T) # Matris av form (2, 3), transponat till C
print(C.reshape((2, 3))) # OBS: inte samma matris
```

Transponatet till en vektor är samma vektor.

Uppgift 2.3.5. Vad blir resultatet av följande beräkning, alltså `H`?

```
v = np.arange(5).reshape(1,-1)
H = v * v.T
```

2.4 Datatyper

Exempel 2.4.1. Arrayer i NumPy kan ha olika datatyper. Testa till exempel följande:

```
x = np.array([1, 2, 3])
y = np.array([1., 2., 3.])
print(x.dtype)
print(y.dtype)
```

Här kommer `x` få datatypen `np.int64` (dvs heltal) medan `y` får datatypen `np.float64` (dvs flyttal). Anledningen är att alla element som skickades in till `np.array` för `x` var heltal, medan alla element för `y` var flyttal. (Vi nämnde tidigare att alla element i en array måste vara av samma typ. Om en blandning av heltal och flyttal skickas in till `np.array` kommer resultatet vara en flyttalsarray.)

Ovanstående skillnad har några, kanske oväntade, konsekvenser. Testa till exempel följande:

```
print(y ** (-1))
print(x ** (-1))
```

Den första raden ger resultatet `[1.  0.5 0.33333333]`. Den andra raden ger ett felmeddelande, nämligen:

```
ValueError: Integers to negative integer powers are not allowed.
```

Även följande operation, som försöker dividera vektorerna med 2 och lagra resultatet i samma vektor, stöter på problem:

```
y /= 2
print(y)
x /= 2
print(x)
```

Den första utskriften blir `[0.5 1.  1.5]`. Rad tre ovan ger felmeddelandet:

```
UFuncTypeError: Cannot cast ufunc 'divide' output from dtype('float64')
to dtype('int64') with casting rule 'same_kind'
```

Felet är att resultatet av divisionen är en flyttalsvektor, som inte kan lagras i heltalsvektorn `x`.

Problemet kan lösas på något av följande sätt:

- `x = np.float64(x)` konverterar `x` till en flyttalsvektor.
- Ange nyckelordet `dtype` när arrayen skapas, till exempel så här:
  `x = np.array([1, 2, 3], dtype=float)`
  Detta gör att `x` skapas som en flyttalsvektor.
- Se till att åtminstone ett av talen som skickas till `np.array` är ett flyttal:
  `x = np.array([1., 2, 3])`

> **Tips:** Alla funktioner för att skapa arrayer som vi tog upp i avsnitt 2.1 accepterar nyckelordet `dtype`. Oftast behöver det inte anges, men ibland är det praktiskt om man vill att arrayen ska ha en viss typ (oftast någon av `float`, `int` eller `bool`).

(NumPy kommer automatiskt översätta datatypen `float` till `np.float64`, `int` till `np.int64` och `bool` till `np.bool` när de skickas in som `dtype`.)

Exempel 2.4.2. NumPys flyttalstyp `np.float64` implementerar IEEE:s standard 754 för flyttal i dubbelprecision, som beskrivs i avsnitt 0.3 i kursboken av Sauer [1]. Det går att skriva ut information om flyttalstypen med hjälp av funktionen `np.finfo`:

```
f = np.finfo(float)
print(f)
print(f.eps)
```

Den första utskriften ovan bör visa att flyttalstypen `np.float64` kan representera tal i intervallet $[-2^{1024}, 2^{1024}] \approx [-1.798 \times 10^{308}, 1.798 \times 10^{308}]$.

Den andra utskriften bör visa att talet ϵ_{mach}, kallat *maskinepsilon* eller *maskinprecision*, har värdet $2^{-52} \approx 2.22 \times 10^{-16}$. Detta är avståndet mellan talet 1 och det minsta tal större än 1 som

kan representeras. Detta hänger ihop med att flyttalen representeras med 52 binära värdesiffror, vilket blir ungefär 16 värdesiffror i bas 10.

Eftersom flyttal representeras med ett ändligt antal värdesiffror måste de avrundas, vilket leder till avrundningsfel ϵ som är i storleksordningen ϵ_{mach} relativt det sanna värdet. Vi talar alltså om ett relativt fel

$$\epsilon = \frac{|\tilde{x} - x|}{|x|}, \tag{1}$$

där $\tilde{x}$ är en approximation av det sanna värdet x. Detta är förklaringen till att den uppmätta residualens norm i exempel 2.2.3 var i storleksordningen $\epsilon_{\text{mach}} \times \|\boldsymbol{b}\|$. (Notera att residualnormen i exempel 2.2.3 var ett absolut fel och ska delas med $\|\boldsymbol{b}\|$ för att få ett relativt fel. I detta fall är $\|\boldsymbol{b}\| \approx 2.24$.)

2.5 Vyer och kopior

Exempel 2.5.1. Vissa indexeringsoperationer, omformning och transponering ger en *vy* (”view” på engelska) till den ursprungliga arrayen, inte en kopia. Detta betyder att ändringar som görs i den ursprungliga arrayen kommer synas även i vyn, och vice versa. En fördel med vyer är att de sparar minne (framförallt om arrayen är stor), men de kan leda till märkliga fel om man inte är medveten om att två variabler delar data. Testa följande exempel:

```
A = np.zeros((3, 4)) # Skapa matris med nollor
a = A[1,2] # Plocka ut enskilt element
B = A[1:,0] # Plocka ut delmatris
d = np.diag(A) # Plocka ut diagonalen
v = A[(0,1,2),(0,2,1)] # Plocka ut några element
At = A.T # Transponat
C = A.reshape((2, 6)) # Omforma matrisen
r = A.ravel() # Platta ut matrisen
print(A, a, B, d, v, At, C, r, sep='\n\n')
```

Allt som skrivs ut ovan består av nollor, eftersom `A` var en matris med nollor.

Vi ska nu testa att ändra på `A` och se vad som händer med de andra variablerna:

```
A[:] = np.arange(1, 13).reshape(A.shape) # Skriv talen 1, 2, ..., 12 till A
print('====')
print(A, a, B, d, v, At, C, r, sep='\n\n')
```

En jämförelse av utskrifterna kommer visa att inte bara `A` utan även flera andra av variablerna ändrades! Det är för att de är *vyer* till `A`. Vilka av variablerna som är vyer och vilka som inte är det kan bero på vilken version av NumPy du har.

Exempel 2.5.2. Om du vill tvinga NumPy att skapa en kopia av en array kan du använda metoden `.copy`. Exempel:

```
A = np.zeros((3, 4)) # Skapa matris med nollor
At = A.T.copy() # Kopia av transponat
```

```
A[:] = np.arange(1, 13).reshape(A.shape)
print(A, At, sep='\n\n')
```

Nu bör `At` fortsätta innehålla nollor även efter att `A` har ändrats eftersom det är en kopia.

3 Matplotlib: Visualisering och plottning

Vi importerar Matplotlib på följande sätt:

```
import matplotlib.pyplot as plt
```

Dokumentationen för Matplotlib finns på sidan `https://matplotlib.org/` och som vanligt kan du även använda `help`. Prova till exempel:

```
help(plt) # Visar hjälpen för PyPlot-modulen
help(plt.plot) # Visar hjälpen för funktionen plt.plot
```

Nedan följer fyra exempel på användning av Matplotlib. Fler exempel finns på Matplotlibs webbsida.

3.1 Plotta en funktionsgraf

Vi kommer använda en så kallad *lambda-funktion* för att definiera funktionen som ska plottas:

```
# Funktion att plotta
f = lambda x: np.sin(x) / x
```

> (För den som inte är bekant med lambda-funktioner är ovanstående i princip ekvivalent med
>
> ```
> def f(x):
>     return np.sin(x) / x
> ```
>
> .)

Nästa steg är att skapa ett nytt figurfönster att plotta i (detta är inte nödvändigt om du bara vill plotta i *en* figur, men om man vill skapa flera plottar kan man vilja ha dem i olika figurer):

```
plt.figure() # skapar en ny figur
```

(Funktionen `plt.figure` tar ett figurnummer som valfritt argument, vilket kan användas för att skapa en figur med ett valt nummer, eller växla till en redan existerande figur med givet nummer. Funktionen `plt.clf()` rensar den nuvarande figuren, men i detta fall behövs det inte.)

Vi ska nu evaluera funktionen i en mängd punkter i intervallet $[-20, 20]$ och plotta dess graf:

```
x = np.linspace(-20, 20, 200)
y = f(x)
plt.plot(x, y, label='$f(x)$') # Plotta
```

(Beroende på din programmiljö kan du eventuellt se plotten redan nu, till exempel om du skrivit in ovanstående i en IPython-tolk konfigurerad enligt avsnitt 1.2. I annat fall kommer plotten dyka upp på slutet av detta exempel.)

Det finns en mängd funktioner för att ställa in etiketter, axlar och liknande. Här ger vi några exempel:

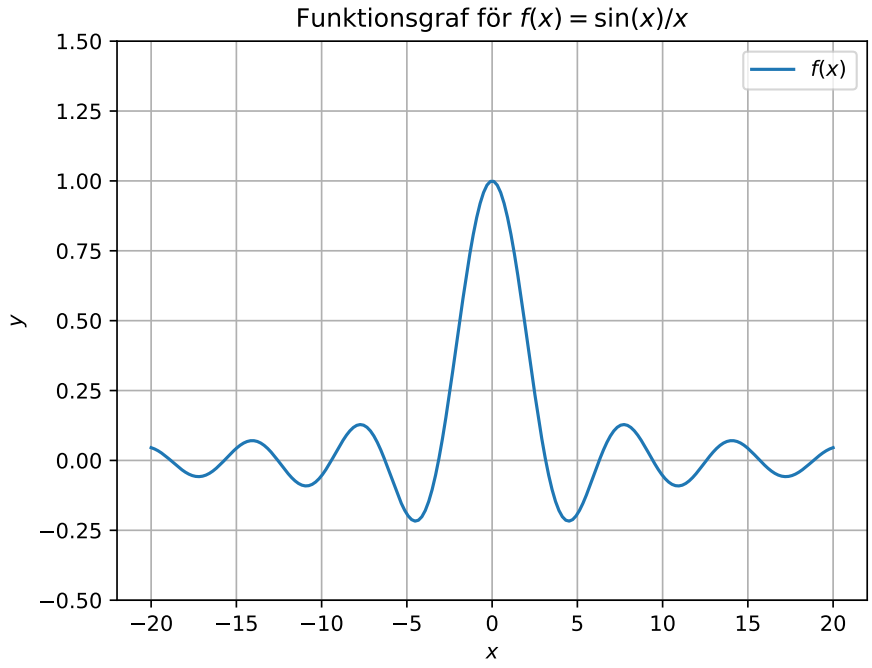
```
plt.grid() # Lägg till rutnät
plt.legend() # Lägg till förklaring (använder label från plt.plot)
plt.xlabel('$x$') # Lägg till etikett på x-axeln
plt.ylabel('$y$') # Lägg till etikett på y-axeln
plt.title(r'Funktionsgraf för $f(x) = \sin(x)/x$') # Lägg till rubrik
plt.ylim([-0.5, 1.5]) # Ändra y-axelns gränser
```

Till sist lägger vi in följande rader som behövs för att figuren ska synas om man kör skriptet icke-interaktivt:

```
if __name__ == '__main__':
    plt.show()
```

Hela skriptet sammansatt finns att ladda ned på länken `ex_3_1_graph.py`.

Figuren som skapas av programmet visas i figur 1.



Figur 1: Plot skapad av `ex_3_1_graph.py`.

3.2 Plotta datapunkter

Datapunkter kan också plottas med funktionen `plt.plot`, det är bara att ange en annan plottningsstil. Ett exempel finns i filen `ex_3_2_points.py`:

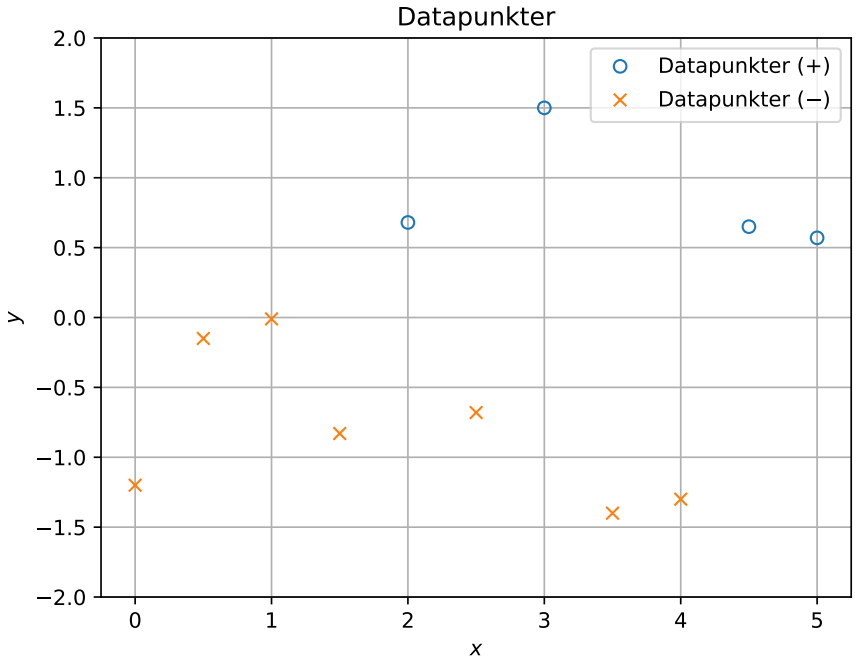
```
import numpy as np
import matplotlib.pyplot as plt

def main():
    # Datapunkter att plotta
    x = np.arange(0, 5+0.5, 0.5)
    y = np.array([-1.2, -0.15, -0.01, -0.83, 0.68, -0.68, 1.5, -1.4,
                  -1.3, 0.65, 0.57])

    fig = plt.figure(5) # skapar ny figur med givet nummer
    plt.clf() # rensar figuren innan plottning
    # Låt oss filtrera datapunkterna baserat på om deras y-värde
    # är positivt eller negativt, och plotta med olika stilar.
    neg = (y < 0)
    pos = ~neg
    plt.plot(x[pos], y[pos], 'o', markerfacecolor='none',
             label='Datapunkter ($+$)')
    plt.plot(x[neg], y[neg], 'x', label='Datapunkter ($-$)')

    plt.grid() # Lägg till rutnät
    plt.legend() # Lägg till förklaring (använder label från plt.plot)
    plt.xlabel('$x$') # Lägg till etikett på x-axeln
    plt.ylabel('$y$') # Lägg till etikett på y-axeln
    plt.title('Datapunkter') # Lägg till rubrik
    plt.ylim([-2, 2]) # Ändra y-axelns gränser

if __name__ == '__main__':
    main()
    plt.show()
```

(Vi har här lagt plottningen i en `main`-funktion. Det är en bra vana att strukturera sin kod i funktioner och undvika att lägga kod direkt på filens toppnivå. Detta underlättar om man till exempel vill importera koden från en annan fil.)

Figuren som skapas av programmet visas i figur 2.

3.3 Rita cirklar, ellipser och polygoner

Följande exempel finns i filen `ex_3_3_shapes.py`:

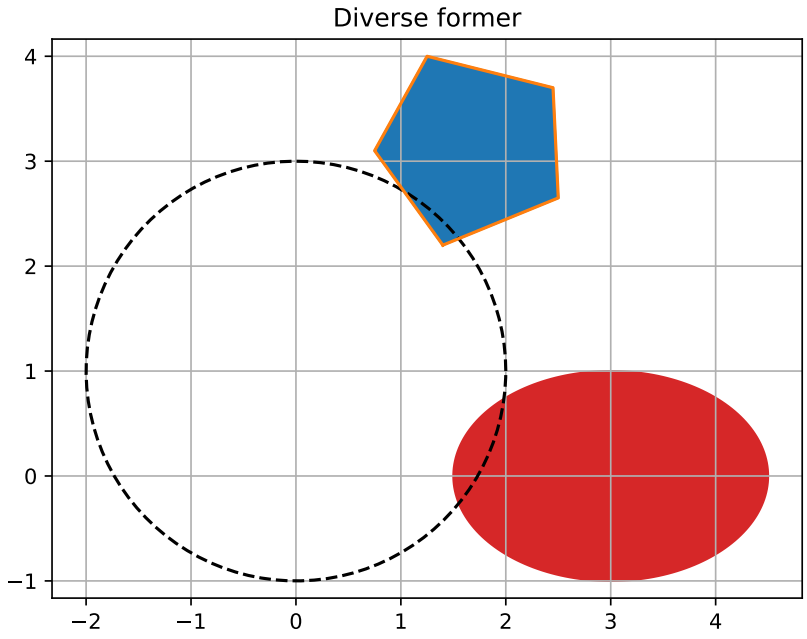


Figur 2: Plot skapad av `ex_3_2_points.py`.

```python
import numpy as np
import matplotlib.pyplot as plt

def main():
    fig = plt.figure(1)
    plt.clf()

    # Plotta en cirkel
    R = 2
    c = (0, 1)
    t = np.linspace(0, 2*np.pi, 200)
    x = c[0] + R*np.cos(t)
    y = c[1] + R*np.sin(t)
    plt.plot(x, y, '--', color='black')

    # Plotta en fylld ellips
    a = 1.5
    b = 1
    c = (3, 0)
    x = c[0] + a*np.cos(t)
    y = c[1] + b*np.sin(t)
    plt.fill(x, y, color='tab:red')
```

```
23
24      # Plotta en polygon
25      c = (2, 3)
26      x = c[0] + np.array([-0.6, 0.5, 0.45, -0.75, -1.25, -0.6])
27      y = c[1] + np.array([-0.8, -0.35, 0.7, 1.0, 0.1, -0.8])
28      plt.fill(x, y, color='tab:blue')
29      plt.plot(x, y, color='tab:orange')
30
31      plt.axis('equal') # riktiga x/y-proportioner
32      plt.grid()
33      plt.title('Diverse former')
34
35  if __name__ == '__main__':
36      main()
37      plt.show()
```

Figuren som skapas av programmet visas i figur 3.



Figur 3: Plot skapad av `ex_3_3_shapes.py`.

3.4 3D-visualisering

Här har vi ett exempel på en ytplot i 3D. Fler exempel på 3D-plottar finns på denna sida. Koden nedan finns i filen `ex_3_4_surf.py`:

```python
import numpy as np
import matplotlib.pyplot as plt

def main():
    # Skapa en figur med en 3D-axel
    fig = plt.figure(1)
    plt.clf()
    ax = fig.add_subplot(projection='3d') # viktigt

    # Plotta en funktionsgraf som en yta
    def f(x, y):
        r = np.sqrt(x**2 + y**2)
        return np.sin(r) / r

    x = np.linspace(-20, 20, 100)
    y = np.linspace(-20, 20, 100)
    X, Y = np.meshgrid(x, y)
    Z = f(X, Y)

    ax.plot_surface(X, Y, Z)
    plt.tight_layout() # utnyttja utrymmet bättre

if __name__ == '__main__':
    main()
    plt.show()
```

Figuren som skapas av programmet visas i figur 4.

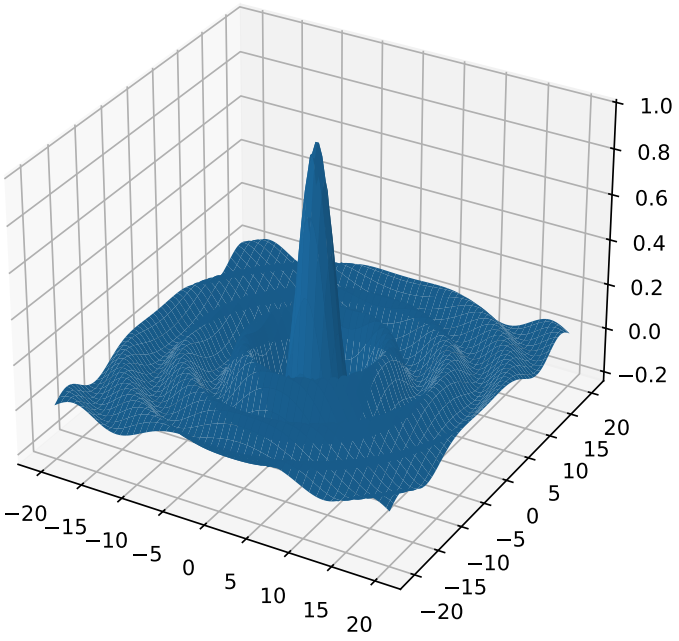
4 SciPy: Funktioner för numeriska beräkningar

SciPy innehåller en mängd funktioner för bland annat ekvationslösning, egenvärdesproblem, interpolation, integration och optimering, samt ett stort bibliotek av matematiska funktioner (utöver de som redan finns i NumPy). I detta dokument kommer vi endast att skrapa på ytan. Du kan läsa mer om SciPy i paketets dokumentation (https://docs.scipy.org/doc/scipy-1.3.3/reference/). Vi kommer även att återvända till SciPy i kursens labbar och projekt.

> (Version 1.3.3 är den SciPy-version som i skrivande stund är installerad på KTH:s datorer. Om du har en annan version på din dator bör du förstås läsa dokumentationen för den versionen istället.)

För att använda SciPy måste du först importera paketet:

```python
import scipy as sp
```



Figur 4: Plot skapad av `ex_3_4_surf.py`.

Funktionerna i SciPy är för det mesta designade för att interagera med NumPys arrayer.

Exempel 4.0.1. Undermodulen `sp.special` innehåller definitioner av matematiska funktioner. En lista över alla funktioner finns på denna sida. En funktion som finns definierad här är till exempel *felfunktionen*, $\operatorname{erf}(x)$, som definieras via integralen

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \mathrm{e}^{-t^2} \, \mathrm{d}t. \tag{2}$$

Integralens värde går inte att uttrycka i termer av elementära funktioner, men det kan beräknas numeriskt. Funktionen som beräknar $\operatorname{erf}(x)$ heter `sp.special.erf`.

Uppgift 4.0.2. (a) Vad är värdet av $\operatorname{erf}(10)$? $\operatorname{erf}(0)$? Du kan använda `sp.special.erf`, men det andra värdet kan också beräknas analytiskt.

(b) Vad ger `sp.special.erf` för värde när funktionen evalueras i `np.inf` (NumPys representation av ∞)? Vad är värdet i `-np.inf`?

(c) Vad har *derivatan* till funktionen $\operatorname{erf}(x)$ för värde i $x = 0$? Denna delfråga kan besvaras analytiskt, utan numeriska beräkningar.

Uppgift 4.0.3. (a) Plotta grafen till funktionen $\operatorname{erf}(x)$ på intervallet $x \in [-4, 4]$ genom att använda `sp.special.erf`. (Du kan använda exempelkoden från avsnitt 3.1 som utgångspunkt.)

Ser grafen ut som du förväntar dig med tanke på Uppgift 4.0.2?

(b) Rita även in tangentlinjen till grafen i punkten $x = 0$.

Referenser

[1] Timothy Sauer. *Numerical Analysis*. Pearson, 2012.

5 Lösningar till uppgifter

Uppgift 2.1.3. `P = np.full((2, 5), np.pi)`
`np.pi` är talet $\pi \approx 3.14159$.

Uppgift 2.2.1.

A. Elementvis addition:

```
[[1.  2. ]
 [1.5 1.5]
 [2.  1. ]]
```

B. Elementvis multiplikation:

```
[[0.   1.  ]
 [0.25 0.25]
 [1.   0.  ]]
```

C. Varje element i matrisen `C` kvadreras (elementvis), eftersom exponenten är en skalär. Resultatet blir detsamma som i B.

(Hade exponenten varit en matris av samma form som `C` så hade varje element i `C` höjts upp till motsvarande element i exponenten.)

D. Varje element i matrisen `C` multipliceras med 2, eftersom den ena faktorn är en skalär:

```
[[0. 2.]
 [1. 1.]
 [2. 0.]]
```

E. Resultatet blir en matris med element $E_{ij} = 1/C_{ij}$, dvs elementvis division, eftersom 1 är en skalär. Dessutom skrivs en `RuntimeWarning` ut, eftersom division med noll sker. Resultatet blir

```
[[inf  1.]
 [ 2.  2.]
 [ 1. inf]]
```

eftersom `x / 0` i NumPy blir `np.inf` (NumPys representation av ∞) förutsatt att `x > 0`. (Se dokumentationen för `np.divide` för mer information.)

F. Detta leder till ett `ValueError` eftersom `C` har form `(3, 2)` medan `A` har form `(3, 3)`.

G. Elementvis rest vid division med 2 (se även `np.remainder`):

```
[1. 0. 1. 0.]
```

H. Elementvis heltalsdivision med 2 (se även `np.floor_divide`):

```
[0. 1. 1. 2.]
```

I. Resultatet blir en vektor med 2 upphöjt till varje element i vektorn `x4`:

```
[ 2.  4.  8. 16.]
```

J. Resultatet blir

```
[[2. 3. 4. 5.]
 [2. 3. 4. 5.]
 [2. 3. 4. 5.]]
```

Enligt NumPys dokumentation kan man kontrollera om broadcastingen går igenom genom att matcha upp operandernas form *från höger till vänster*, dvs i detta fall:

```
B.shape:    (3, 4)
x4.shape:      (4,)
```

Om talen i samma kolumn antingen är samma eller ett av dem är 1 så går broadcastingen igenom (en tom plats räknas som 1). I detta fall är det alltså okej, och resultatet blir som om `x4` kopieras 3 gånger längs den första dimensionen (en matris med tre rader som alla är lika med `x4`) innan den adderas till `B`.

K. Detta leder till ett `ValueError` eftersom `B` har form `(3, 4)` medan `x5` har form `(3,)`. Om vi gör samma koll som i J så fås

```
B.shape:    (3, 4)
x5.shape:      (3,)
```

vilket innebär att broadcastingen inte går igenom.

(Det skulle gå om `x5` istället hade formen `(3, 1)`.)

Uppgift 2.2.4.

A. Summerar alla element i vektorn: `10.0`

B. Summerar alla element i matrisen: `6.0`

C. Summerar längs den **första** dimensionen (dvs radindex) och returnerar en vektor med summorna (lika många element som antal kolumner i matrisen): `[1. 2. 3.]`

Notera: ”axis” (axel) i NumPy syftar på en av arrayens dimensioner, och axlarna numreras från 0 och uppåt. Alltså är axel 0 radindex, axel 1 kolumnindex osv.

D. Summerar längs den **andra** dimensionen (dvs kolumnindex) och returnerar en vektor med summorna (lika många element som antalet rader i matrisen): `[4. 2.]`

E. Beräknar produkten av alla element i vektorn: `24.0`

Uppgift 2.2.7.

A. Lägger vektorn under sig själv som två rader:

```
[[0. 1. 2.]
 [0. 1. 2.]]
```

B. Lägger vektorn bredvid sig själv som en lång vektor:

```
[0. 1. 2. 0. 1. 2.]
```

C. Lägger vektorn bredvid sig själv som två kolumner:

```
[[0. 0.]
 [1. 1.]
 [2. 2.]]
```

D. Lägger matrisen under sig själv:

```
[[0.  1. ]
 [0.5 0.5]
 [1.  0. ]
 [0.  1. ]
 [0.5 0.5]
 [1.  0. ]]
```

E. Lägger matrisen bredvid sig själv:

```
[[0.  1.  0.  1. ]
 [0.5 0.5 0.5 0.5]
 [1.  0.  1.  0. ]]
```

F. Ger samma resultat som E.

G. Lägger vektorn under matrisen:

```
[[ 0.   1. ]
 [ 0.5  0.5]
 [ 1.   0. ]
 [ 2.5 -3.5]]
```

H. Ger `ValueError` eftersom vektorn med form `(3,)` inte går att lägga bredvid matrisen med form `(3, 2)` utan omformning.

I. Lägger vektorn som en kolumn bredvid matrisen:

```
[[0.  1.  0. ]
 [0.5 0.5 1. ]
 [1.  0.  2. ]]
```

Uppgift 2.3.3. Resultatet blir

```
[[1. 1. 1. 1.]
 [2. 2. 2. 2.]
 [3. 3. 3. 3.]]
```

Jämför med lösningen till uppgift 2.2.1 J. Notera först att `x5.reshape(-1,1)` kommer bli en matris med form `(3, 1)`. Därmed har operanderna till additionen former

```
B.shape:                 (3, 4)
x5.reshape(-1,1).shape: (3, 1)
```

vilket innebär att broadcastingen fungerar och `x5` kommer bli kopierad som en kolumn 4 gånger innan den adderas till `B`.

Uppgift 2.3.5. Notera att `np.arange(5)` ger en vektor `[0 1 2 3 4]`, som sedan omformas så att `v` blir en matris med en rad, nämligen `[[0 1 2 3 4]]`. Eftersom `v` är en matris och inte en vektor så kommer `v.T` inte vara samma som `v`; transponatet är en matris med en *kolumn*.

För att se om multiplikationen kan utföras kontrollerar vi om `v` och `v.T` kan broadcastas:

```
v.shape:    (1, 5)
v.T.shape:  (5, 1)
```

Eftersom ett tal i varje kolumn här är 1 så går det bra. För att genomföra multiplikationen kommer `v` kopieras som 5 rader, och `v.T` kopieras som 5 kolumner. Resultatet `H` blir alltså en 5×5-matris med alla kombinationer av produkter mellan element i `v`:

```
H =
[[ 0  0  0  0  0]
 [ 0  1  2  3  4]
 [ 0  2  4  6  8]
 [ 0  3  6  9 12]
 [ 0  4  8 12 16]]
```

Uppgift 4.0.2. (a) Den numeriska funktionen ger följande svar:

```
sp.special.erf(10):    1.0
sp.special.erf(0):     0.0
```

Värdet på erf(0) är noll eftersom en integral över ett tomt intervall ($\int_0^0$) är noll.

(b) Den numeriska funktionen ger följande svar:

```
sp.special.erf(np.inf):   1.0
sp.special.erf(-np.inf):  -1.0
```

(c) Derivatan till funktionen erf(x) ges av analysens huvudsats:

$$\text{erf}'(x) = \frac{2}{\sqrt{\pi}} e^{-x^2}. \tag{3}$$

Värdet i $x = 0$ är alltså $\text{erf}'(0) = \frac{2}{\sqrt{\pi}}$.

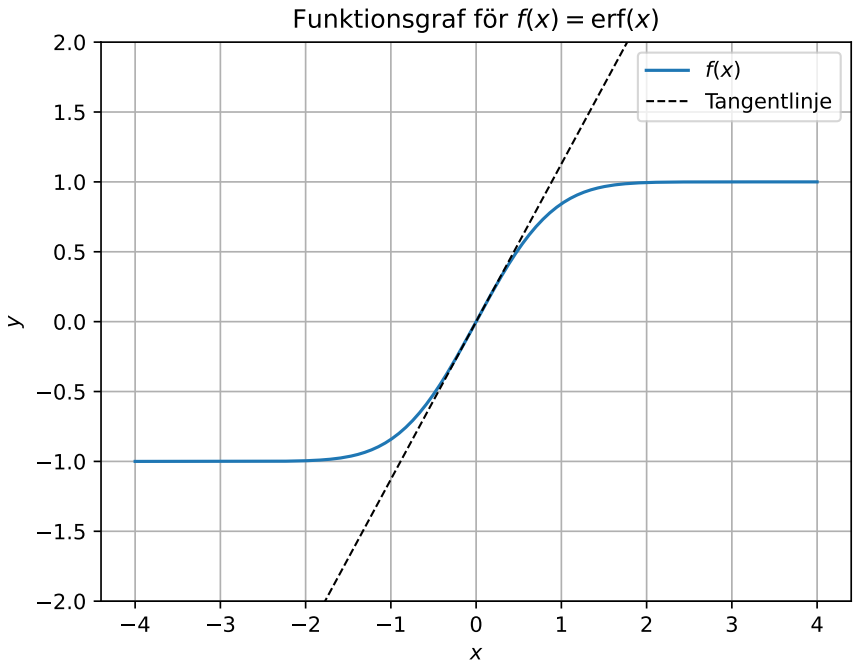
Uppgift 4.0.3. Koden till lösningen finns nedan och även i `sol_4_0_3.py`:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy as sp

def main():
    f = lambda x: sp.special.erf(x)
    plt.figure()
    x = np.linspace(-4, 4, 200)
    y = f(x)
    plt.plot(x, y, label='$f(x)$')
    plt.plot(x, (2/np.sqrt(np.pi))*x, 'k--', linewidth=1,
             label='Tangentlinje')
    plt.grid()
    plt.legend()
    plt.xlabel('$x$')
    plt.ylabel('$y$')
    plt.title(r'Funktionsgraf för $f(x) = \text{erf}(x)$')
    plt.ylim([-2, 2])

if __name__ == '__main__':
    main()
    plt.show()
```

Figuren som skapas av programmet finns i figur 5.

(a) Förväntat är att funktionen har värdet 0 i $x = 0$, och en positiv derivata i denna punkt. Funktionen förväntas vidare gå mot 1 då $x \to \infty$, och gå mot -1 då $x \to -\infty$. Detta ser ut att stämma in på grafen.

(b) Se figur 5.



Figur 5: Plot skapad av `sol_4_0_3.py`.