Vektorer och matriser i PYTHON

1 Något om vektorer och matriser i `NumPy`

En matris är ett rektangulärt talschema

$$\mathbf{A} = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}$$

Matrisen ovan har m rader och n kolonner, vi säger att den är av typ $m \times n$. Ett matriselement i rad nr i, kolonn nr j tecknas a_{ij}, där i är radindex och j är kolonnindex. I PYTHON skrivs detta `A[i-1,j-1]` och `shape(A)` ger matrisens typ.

En matris av typ $m \times 1$ kallas kolonnmatris (kolonnvektor) och en matris av typ $1 \times n$ kallas radmatris (radvektor):

$$\mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} c_1 & \dots & c_n \end{bmatrix}$$

Element nr i ges i Python av `b[i-1], c[i-1]` och antalet element ges av anropen `size(b), size(c)`. Som exempel tar vi

$$\mathbf{A} = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} 0 & 2 & 4 & 6 & 8 \end{bmatrix}$$

Vi skriver in detta i Python enligt (här markerar >>> att vi skriver raderna i Console):

```
>>> from numpy import array
>>> A=array([[1,4,7,10],[2,5,8,11],[3,6,9,12]])
>>> b=array([[1],[3],[5]])
>>> c=array([0,2,4,6,8])
```

Vi använder fält, klassen `array` i `NumPy`, för att bygga matriserna. Varje rad i matrisen eller vektorn omges med hakparenteser, liksom hela matrisen.

Vill vi se vektorernas och matrisens värden kan vi skriva variabelnamnen. PYTHON markerar att det är fråga om fält (`array`) genom att infoga namnet `array` framför värdena.

```
>>> b
array([[1],
       [3],
       [5]])
>>> c
```

```
array([0, 2, 4, 6, 8])
>>> A
array([[ 1,  4,  7, 10],
       [ 2,  5,  8, 11],
       [ 3,  6,  9, 12]])
```

Indexeringen i vektorer och matriser i PYTHON börjar på `0`. T.ex. `b[0]` är första elementet i vektorn `b`, `b[1]` är andra elementet. Sista elementet i `b` får vi med `b[2]` eller `b[-1]`. Elementet a_{23} på andra raden i 3:e kolonnen i `A` skrivs `A[1,2]`.

Uppgift 1 (a). Skriv in matriserna i PYTHON och skriv sedan ut matriselementen a_{23}, b_2, c_3. Prova `shape` och `size`. Ändra a_{23} till 15 genom att skriva `A[1,2]=15`.

(b) I PYTHON skapar `array` ett s.k. objekt av en klass som heter `ndarray`. Därför kan vi också skriva `A.shape` respektive `b.size` för att få reda på storlek och antal elemement. Testa även detta på matriserna och vektorerna ovan. Prova också anropet `c.shape=(5,1)`. Vad har hänt med `c` efter anropet?

Mer information om `array` i `NumPy` finns på sidan

`https://numpy.org/doc/stable/reference/arrays.ndarray.html`

1.1 Listor i PYTHON

Vi kan välja att lagra matrisvärden i listor.

Elementen i listor indexeras på samma sätt som i fälten, det som skiljer listor från fält är de funktioner som kan användas på dem. Om man vill göra beräkningar med elementen i matriserna, t.ex. multiplicera dem eller lösa linjära ekvationssystem brukar man använda fält från `NumPy` och inte listor.

Ibland vill vi se en tabell som en matris och då kan vi använda en lista istället för ett fält. Som exempel tar vi: Värmeförlusten hos den som vistas i kyla beror inte enbart på temperaturen, utan även på hur mycket det blåser. Tabellen visar vilken *effektiv temperatur* det blir vid olika temperaturer T (°C) och vindhastigheter v (m/s).

v \ T	10	6	0	-6	-10	-16	-26	-30	-36
2	9	5	-2	-9	-14	-21	-33	-37	-44
6	7	2	-5	-13	-18	-26	-38	-44	-51
10	6	1	-7	-15	-20	-28	-41	-47	-55
14	6	0	-8	-16	-22	-30	-44	-49	-57
18	5	-1	-9	-17	-23	-31	-45	-51	-59

Om vi ville göra något med dessa data i PYTHON så skulle vi kunna lagra temperaturer och vindhastigheter i rad- eller kolonnvektorer och effektiva temperaturerna i en matris som vi låter vara en lista i PYTHON.

Datatypen list finns inbyggd i PYTHON så vi behöver inte importera några paket för att skapa listor.

```
>>> T=[10,6,0,-6,-10,-16,-26,-30,-36]
>>> v=[[2],[6],[10],[14],[18]]
>>> Vind=[ [9,5,-2,-9,-14,-21,-33,-37,-44],
           [7,2,-5,-13,-18,-26,-38,-44,-51],
```

```
[6,1,-7,-15,-20,-28,-41,-47,-55],
[6,0,-8,-16,-22,-30,-44,-49,-57],
[5,-1,-9,-17,-23,-31,-45,-51,-59]]
```

Elementen i listan indexeras på samma sätt som för fält. Exempelvis ges den effektiva temperaturen vid vindhastigheten 10 m/s och temperaturen -6° av elementet

```
>>> Vind[2][3]
-15
```

Vill du läsa mer om listor i PYTHON hittar du det i avsnitten om listor (`Lists`) på sidan `https://docs.python.org/3/tutorial/index.html`.

(Vill du läsa mer om värmeförlust gå till SMHI:s hemsida och sök på vindavkylning.)

2 Funktioner i `NumPy`

2.1 Matematiska funktioner i `NumPy`

Vi tittar i dokumentationen för `NumPy` för att få en överblick över de matematiska funktioner som finns. Du hittar dokumentationen på sidan

`https://numpy.org/doc/stable/reference/routines.math.html`

Vi ser att funktionerna är grupperade, t.ex. en grupp med trigonometriska funktioner och längre ner en grupp med exponent- och logaritmfunktioner.

Funktioner som exempelvis sinus och cosinus, kan operera både på enskilda tal och på fält. Man får som resultat ett fält av samma storlek, vars element är funktionsvärdet av respektive element i argumentet.

Som exempel tar vi återigen

$$\mathbf{A} = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} 0 & 2 & 4 & 6 & 8 \end{bmatrix}$$

som vi skriver in i PYTHON enligt

```
>>> from numpy import array, sin
>>> A=array([ [1,4,7,10],[2,5,8,11],[3,6,9,12]])
>>> c=array([0,2,4,6,8])
```

Här skrev vi `from numpy import array, sin` för att importera klassen `array` och funktionen `sin`. Då importerar vi bara `array` och `sin` från paketet `numpy` och vi skriver sedan bara namnet på kommandot när vi anropar det.

Nu beräknar vi sinus av vektorn **c** och matrisen **A** med

```
>>> v=sin(c)
>>> print(v)
[ 0. 0.90929743 -0.7568025  -0.2794155   0.98935825]
>>> V=sin(A)
>>> print(V)
[[ 0.84147098 -0.7568025   0.6569866  -0.54402111]
 [ 0.90929743 -0.95892427  0.98935825 -0.99999021]
 [ 0.14112001 -0.2794155   0.41211849 -0.53657292]]
```

Man anger vektorn eller matrisen som argument till `sin` när man beräknar sinus för värdena. Funktionen beräknar elementvis sinus för alla värden i vektorn eller matrisen.

När vi använder `print` för att skriva ut elementen i vektorn och matrisen skrivs inte ordet `array` framför fältet i utskriften.

Kommandonas hjälptexter är inte alls nybörjaranpassade. De beskriver bara hur kommandona är tänkta att användas. Dessutom är beskrivningen väldigt teknisk. Men det brukar finnas exempel längst ner i hjälptexten som kan vara till hjälp.

Vi klickar på `tan` (gör det!), dvs. tangensfunktionen, på dokumentationssidan och ser på hjälptexten. Kommandot anropas med minst ett argument, ett tal, eller ett fält (vektor eller matris).

```
>>> from numpy import tan, pi
>>> t=array([-pi/5,pi/3])
>>> b=tan(t)
>>> print(b)
[-0.72654253  1.73205081]
```

Kommandot är ekvivalent med att anropa `sin(x)/cos(x)`.

Uppgift 2. Använd `plot` (från `Matplotlib.pyplot`) och rita upp tangensfunktionen på intervallet $0 \leq x \leq \pi$. Vad händer då $x = \frac{\pi}{2}$?

2.2 Fler kommandon i `NumPy`

I Python kan det finnas flera kommandon eller funktioner med samma namn. Då finns funktionerna i olika paket. För att säkerställa att man anropar en funktion eller konstant från ett visst paket kan man namnge paketet när man importerar det. Det är vanligt att man ger `NumPy` namnet `np` och skriver

```
import numpy as np
```

Observera att vi här importerar hela paketet `NumPy` och inte enskilda klasser eller funktioner som vi gjorde i avsnitt 2.1. Nu måste man infoga prefixet `np.` framför namnet så fort man använder någon funktion eller konstant från `NumPy`.

Vi använder samma vektor **c** och matris **A** från förra avsnittet och bildar

```
>>> import numpy as np
>>> A=np.array([ [1,4,7,10],[2,5,8,11],[3,6,9,12]])
>>> c=np.array([0,2,4,6,8])
```

Vi har redan sett funktionerna `size` och `shape`. Antal element i vektorn **c** ges av

```
>>> l=np.size(c)
>>> l
5
```

Med `np.size` säkerställer vi att det är funktionen `size` från `NumPy` som avses. Antalet rader och kolonner **A** fås med

```
>>> [m,n]=np.shape(A)
>>> m,n
(3,4)
```

Vi kan beräkna största och minsta element i fält med funktionerna `max` och `min`.

```
>>> print(np.max(c))
8
>>> print(np.max(A))
12
```

Summan och produkten av elementen i vektorn fås med `sum` och `prod`.

```
>>> s=np.sum(c)
>>> p=np.prod(A)
>>> print(p)
479001600
```

Vill vi sortera en vektor i stigande ordning kan vi göra det med `sort`. Om vi skriver

```
>>> c=np.array([2,4,1,-9,0])
>>> sc=np.sort(c)
>>> print(sc)
[-9  0  1  2  4]
>>> print(c)
[ 2  4  1 -9  0]
```

blir `sc` en sorterad vektor och `c` är kvar som förut. För en matris blir det varje rad som sorteras om i stigande ordning.

```
>>> A=np.array([[2,4,1,-9,0],[-8,0,2,-3,5]])
>>> sA=np.sort(A)
>>> print(sA)
[[-9  0  1  2  4]
 [-8 -3  0  2  5]]
```

3 Hantering av vektorer och matriser i NumPy

3.1 Vektorer i NumPy

Vektorn $\mathbf{c} = (0, 2, 4, 6, 8)$ från förra avsnittet kan bildas på flera sätt i `NumPy`. Dels med kommandot `array` men också med kommandot `arange`:

```
>>> from numpy import arange
>>> c=arange(0,9,2)
>>> c
array([0, 2, 4, 6, 8])
```

Om man använder funktionen `arange` anger man `start, slut, steg`. Man får *från och med* `start` *till* `slut` (men talet `slut` är inte inkluderat).

Med `start`=0, `slut`=8 och `steg`=2 får vi fältet $(0, 2, 4, 6)$.

```
>>> c=arange(0,8,2)
>>> print(c)
[0, 2, 4, 6]
```

För att få hela vektorn $(0, 2, 4, 6, 8)$ kan vi t.ex. skriva

```
>>> c=arange(0,10,2)
>>> print(c)
[0, 2, 4, 6, 8]
```

Det sista talet i fältet `c` blir det närmaste heltalet < 10 som passar. Vi kan också (som vi gjorde först) skriva

```
>>> c=arange(0,9,2)
>>> print(c)
[0, 2, 4, 6, 8]
```

Det sista talet blir heltalet närmst under 9 som passar.

Vi låter s få tredje värdet c_3 med `s=c[2]`. Det gäller att komma ihåg att index börjar med 0 i PYTHON.

```
>>> s=c[2]
>>> s
4
```

När man skapar nya vektorer eller delvektorer med hjälp av redan skapade vektorer måste man kopiera elementen till den nya vektorn. Vi kan bilda en ny vektor med samma värden som **c**

```
>>> from numpy import copy
>>> kopia=copy(c)
>>> kopia
array([0, 2, 4, 6, 8])
```

Elementen i **c** *kopieras* till `kopia`.

Vi kan bilda vektorn **v** av andra och femte värdet, dvs. $\mathbf{v} = (c_2, c_5)$, med

```
>>> v=copy(c[[1,4]])
>>> v
array([2, 8])
```

Andra och 5:e elementet i **c** *kopieras* till **v**. Observera att man måste skriva dubbla hakparenteser, `c[[1,4]]`.

Vi kan bilda vektorn **v** av de tre första elementen i **c**, dvs. $\mathbf{v} = (c_1, c_2, c_3)$, med

```
>>> v=copy(c[0:3])
>>> v
array([0, 2, 4])
```

Man använder egentligen `start:slut:steg` när man indexerar i vektorn `c`. Om man utelämnar `steg` får man steget 1. Man får indexen *från och med* `start` till `slut`. Så `c[0:3]` ger elementen på platserna $0, 1$ och 2 i `c`.

Vi kan ändra ett element i **v**, t.ex. låta $v_2 = 0$, med

```
>>> v[1]=0
>>> v
array([0, 0, 4])
```

3.2 Matriser i NumPy

Vi bildar samma matris **A** som i första avsnittet

```
>>> from numpy import array
>>> A=array([[1,4,7,10],[2,5,8,11],[3,6,9,12]])
>>> A
array([[ 1,  4,  7, 10],
       [ 2,  5,  8, 11],
       [ 3,  6,  9, 12]])
```

Vi låter s få värdet av elementet på rad 2, kolonn 3 i matrisen med `s=A[1,2]`

```
>>> s=A[1,2]
>>> s
8
```

och vi bildar en ny radvektor **v** av rad 3, alla kolumner med

```
>>> from numpy import copy
>>> v=copy(A[2,:])
>>> print(v)
[ 3,  6,  9, 12])
```

samt en vektor **u** av rad 2-3, kolonn 2 med

```
>>> u=copy(A[1:3,1])
>>> print(u)
[5, 6]
```

Vi använde `1:3` för att berätta vilka rader fältet som avsågs. Vi får raderna 1 och 2 i fältet, dvs. raderna 2 och 3 i matrisen. Kolonn 2 i matrisen har index 1 i fältet.

Python gör automatiskt om kolonnvektorn vi indexerar till en radvektor. Om vi vill att `u` ska vara en kolonnvektor kan vi använda `reshape`:

```
>>> from numpy import reshape
>>> u=reshape(u)
>>> print(u)
[[5]
 [6]]
```

Vi bildar en matris **V** av blocket i rad 1-2 och kolonn 2-3:

```
>>> V=copy(A[0:2,1:3])
>>> print(V)
[[4, 7],
 [5, 8]]
```

Vi kan transponera matrisen med `.T`:

```
>>> print(V.T)
[[4, 5],
 [7, 8]]
```

Uppgift 3 (a). Bilda en 2×2-matris $\mathbf{B}$ av det block av storlek 2×2 som står längst ned till höger (i de två sista raderna och de två sista kolonnerna) i matrisen $\mathbf{A}$. Ändra sedan elementet $b_{2,2}$ till 100.

(b) Transponera matrisen $\mathbf{B}$.

3.3 Operatorer på vektorer och matriser i `NumPy`

Om vi har två vektorer $\mathbf{u} = (2, 3, 5)$ och $\mathbf{v} = (1, 2, 3)$ av samma typ och vill bilda summan $\mathbf{a} = \mathbf{u} + \mathbf{v}$ och skillnaden $\mathbf{b} = \mathbf{u} - \mathbf{v}$, så gör vi det med `a=u+v` respektive `b=u-v`. Operationerna sker elementvis

$$\mathbf{a} = \mathbf{u} + \mathbf{v} = (2, 3, 5) + (1, 2, 3) = (2 + 1, 3 + 2, 5 + 3) = (3, 5, 8)$$
$$\mathbf{b} = \mathbf{u} - \mathbf{v} = (2, 3, 5) - (1, 2, 3) = (2 - 1, 3 - 2, 5 - 3) = (1, 1, 2)$$

eller med andra ord $a_i = u_i + v_i$ och $b_i = u_i - v_i$.

T.ex. vid grafritning behövs de elementvisa motsvarigheterna till multiplikation och division

$$\mathbf{u} \texttt{ * } \mathbf{v} = (2, 3, 5) \texttt{ * } (1, 2, 3) = (2 \cdot 1, 3 \cdot 2, 5 \cdot 3) = (2, 6, 15)$$
$$\mathbf{u} \texttt{ / } \mathbf{v} = (2, 3, 5) \texttt{ / } (1, 2, 3) = (2/1, 3/2, 5/3) = (2, 1.5, 1.666\ldots)$$

Här har vi lånat beteckningar från Python där vi skriver `u*v` respektive `u/v` för att utföra beräkningarna.

På samma sätt fungerar även elementvis upphöjt till. T.ex. kvadrering:

$$\mathbf{u}\texttt{**}2 = (2, 3, 5) \texttt{ **} 2 = (2^2, 3^2, 5^2) = (4, 9, 25)$$

Även här har vi lånat beteckningen ** från Python.

Det finns en funktion `linspace` i `NumPy` som är bra för att bygga upp vektorer och funktionerna `zeros` och `ones` för att bygga upp matriser fyllda med nollor respektive ettor samt funktionen `eye` för att göra enhetsmatriser. Kommandona finns beskrivna i dokumentationen på sidan

`https://numpy.org/doc/stable/reference/routines.array-creation.html`

Uppgift 4. Använd `linspace` (läs hjälptexten) för att bilda vektorn $\mathbf{d} = (2, 5, 8, 11, 14)$.

4 Linjära ekvationssystem

Linjära ekvationssystem kan vi lösa med Python om vi först skriver dem på matrisform (dvs. som matrisekvationer). Vi tar som exempel ekvationssystemet

$$\begin{cases} x_1 + 2x_2 + 3x_3 = 14 \\ 3x_1 + 2x_2 + x_3 = 10 \\ 7x_1 + 8x_2 = 23 \end{cases}$$

som skrivs på matrisform

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \\ 7 & 8 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 14 \\ 10 \\ 23 \end{bmatrix}$$

dvs.

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \quad \text{med } \boldsymbol{A} = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \\ 7 & 8 & 0 \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \text{och} \quad \boldsymbol{b} = \begin{bmatrix} 14 \\ 10 \\ 23 \end{bmatrix}.$$

I PYTHON använder vi kommandot `array` från paketet `NumPy` för att skapa matriserna. Eftersom detta ekvationssystem har entydig lösning kan vi använda kommandot `solve` från paketet `Numpy.linalg` för att beräkna lösningarna till ekvationssystemet. Vi bildar koefficientmatrisen $\boldsymbol{A}$ och högerledsvektorn $\boldsymbol{b}$ med

```
>>> import numpy as np
>>> A=np.array([[1,2,3],[3,2,1],[7,8,0]])
>>> b=np.array([[14],[10],[23]])
```

Med kommandot `solve` löser vi ekvationssystemet

```
>>> from numpy.linalg import solve
>>> x=solve(A,b)
>>> x
array([[1.],
       [2.],
       [3.]])
```

Lösningarna till ekvationssystemet finns i fältet `x`. Vi ser att $x_1 = 1, x_2 = 2, x_3 = 3$.

Här skrev vi `from numpy.linalg import solve` för att importera kommandot `solve`. Då importerar vi bara `solve` från paketet `numpy.linalg` och vi skriver sedan bara namnet på kommandot när vi anropar det.

Uppgift 5. Skriv följande ekvationssystem på matrisform och lös dem sedan med `solve`. (Skulle det finnas oändigt många lösningar eller om systemet saknar lösningar, måste vi använda andra kommandon är `solve`.)

(a) $\begin{cases} 2x_1 + 3x_2 = 8 \\ 5x_1 + 4x_2 = 13 \end{cases}$

(b) $\begin{cases} x_1 + 5x_2 + 9x_3 = 29 \\ 2x_1 + 5x_3 = 26 \\ 3x_1 + 7x_2 + 11x_3 = 39 \end{cases}$

(c) $\begin{cases} x_1 + \frac{1}{2}x_2 + \frac{1}{3}x_3 = \frac{11}{6} \\ \frac{1}{2}x_1 + \frac{1}{3}x_2 + \frac{1}{4}x_3 = \frac{13}{12} \\ \frac{1}{3}x_1 + \frac{1}{4}x_2 + \frac{1}{5}x_3 = \frac{47}{60} \end{cases}$

(d) $\begin{cases} x_1 + x_2 + 3x_3 + 4x_4 = 2 \\ -2x_1 + 2x_2 + 2x_3 = -4 \\ x_1 + x_2 + 2x_3 + 3x_4 = 1 \\ x_1 - x_2 - 2x_3 - x_4 = 1 \end{cases}$

Om det, som i (d)-uppgiften ovan, finns oändligt många lösningar måste vi använda andra kommandon än `solve`. Ett sätt är att använda klassen `Matrix` från det symboliska paketet `sympy`. Vi löser ekvationssystemet

$$\begin{cases} x_1 + 2x_2 + 3x_3 = 14 \\ 3x_1 + 2x_2 + x_3 = 10 \\ 7x_1 + 8x_2 = 23 \end{cases}$$

en gång till, denna gången med `sympy`:

```
>>> from sympy import Matrix
>>> A=Matrix([[1,2,3,14],[3,2,1,10],[7,8,0,23]])
>>> X=A.rref()
```

Först bildar vi totalmatrisen, sedan anropar vi `rref` från klassen `Matrix` som svarar med den reducerade trappstegsmatrisen och i vilka kolonner de ickefria variablerna finns (kom ihåg att kolonnerna numreras från 0).

```
(Matrix([
 [1, 0, 0, 1],
 [0, 1, 0, 2],
 [0, 0, 1, 3]]),
 (0, 1, 2))
```

Vi läser av lösningen och ser att $x_1 = 1, x_2 = 2, x_3 = 3$. Systemet har en unik lösning, så ingen av variablerna är fri.

Uppgift 5 (e). Lös ekvationssystemet i uppgift 5(d) en gång till, denna gången med `rref`. Formulera den allmänna lösningen. Vilka variabler är fria?

Uppgift 6. Den här uppgiften handlar om *Hilbertmatriser* och är lite mer avancerad. Hilbertmatriser ger typiskt upphov till (numeriskt) svårlösta ekvationssystem. Detta beror på att Hilbertmatriser är illa konditionerade vilket betyder att de förstorar fel i högerledet (som till exempel kan bero på avrundning) väldigt mycket.

(a) Bilda en 10×10-Hilbertmatris och beräkna dess determinant med hjälp av

```
>>> import numpy as np
>>> from scipy.linalg import hilbert
>>> A=hilbert(10)
>>> D=np.linalg.det(A)
```

Vad säger det här om lösbarhet av matrisekvationer på formen $\boldsymbol{Ax} = \boldsymbol{b}$? (Se kursbokens kapitel 3.) Hur säker är du på ditt svar på den första frågan?

(b) Skapa nu en vektor $\boldsymbol{x_{exakt}} \in \mathbb{R}^{10}$ med enbart ettor som koordinater (använd till exempel funktionen `ones` som nämns mot slutet av avsnitt 3 för att åstadkomma detta). Definiera sedan ytterligare en vektor $\boldsymbol{b} \in \mathbb{R}^{10}$ genom $\boldsymbol{b} = \boldsymbol{Ax_{exakt}}$:

```
>>> b=np.matmul(A,x_exakt)
```

Förklara hur det följer av diskussionen ovan att ekvationssystemet $\boldsymbol{Ax} = \boldsymbol{b}$ har den unika lösningen $\boldsymbol{x_{exakt}}$ som består av enbart ettor.

(c) Observera att så här långt har vi inte löst något ekvationssystem utan endast konstruerat ett ekvationssystem som vi vet lösningen till. Lös nu systemet $\boldsymbol{Ax} = \boldsymbol{b}$ med hjälp av kommandot `solve`. Vad kan du dra för slutsats av resultatet?

(d) Uppgifterna ovan behandlar 10×10-Hilbertmatrisen $\boldsymbol{A}$. Vi kunde lika gärna låtit $\boldsymbol{A}$ vara en $k \times k$-Hilbertmatris för något annat positivt heltal k och löst motsvarande uppgifter för denna matris (då hade vektorerna ovan varit vektorer i $\mathbb{R}^k$ i stället för vektorer i $\mathbb{R}^{10}$). Vilket är det minsta värdet på k sådant att fenomenet i deluppgift (c) förekommer?