Gausova eliminacija

Tema prva dva predavanja ovog kursa je rešavanje sistema linearnih jednačina.

Postoje mnogi realni problemi koji se svode ne rešavanje SLAJ.

Neke njih, kao što su rangiranje stranica na internetu pomoću algoritma *PageRank* koji koristi *Google* ili određivanje pozicije pomoću *GPS*, pokazaćemo u zavisnosti od vremena na kraju ovog ili sledećeg predavanja.

Algoritam Gausove eliminacije

Pre nego što stignemo do primena detaljno ćemo kroz kod objasniti kako funkcioniše algoritam Gausove eliminacije.

Gausova eliminacija je u stvari samo upoštenje načina na koji smo rešavali sisteme jedančina još od osnovne škole.

Recimo da imamo sledeći sistem:

$$\begin{gathered} 4x_1 - 3x_2 + x_3 = -8 \\ -2x_1 + x_2 - 3x_3 = -4 \\ x_1 - x_2 + 2x_3 = 3 \end{gathered}$$

Prisetimo se na koji način smo mi pokušavali da rešimo sisteme jednačina u osnovnoj i srednjoj školi.

Pokušavali samo na neki način da sistem svedemo na jednu jednačinu sa jednom nepoznatom, pa da onda tu nepoznatu odredimo deljenjem i zamenimo je u neku od prethodnih jednačina da dobijemo drugu nepoznatu i tako redom.

Gausova eliminacija nije ništa drugo nego samo sistematizacija tog postupka.

Krećemo tako što nam je cilj da eliminišemo promenljivu x_1 iz druge jednačine.

U prvom koraku prvu jednačinu množimo sa $\frac{2}{4} = \frac{1}{2}$ i onda je dodamo drugoj:

$$\begin{gathered} 4 \cdot \frac{1}{2} x_1 - 3 \cdot \frac{1}{2} x_2 + 1 \cdot \frac{1}{2} x_3 = -8 \cdot \frac{1}{2} \\ -2x_1 + x_2 - 3x_3 = -4 \end{gathered}$$

Prvu jednačinu prepisujemo bez promene jer će nam u tom obliku biti potrebna u narednim koracima.

Na taj način ne menjamo rešenje jer množenje jednačine konstanom ne utiče na rešenje.

$$\begin{gathered} 4x_1 - 3x_2 + x_3 = -8 \\ 0 - \frac{1}{2} x_2 - \frac{5}{2} x_3 = -8 \end{gathered}$$

Uklonili smo x_1 iz druge jednačine, sada ga uklanjamo i iz treće.

U sledećem koraku prvu jednačinu množimo sa $-\frac{1}{4}$ i onda je dodamo trećoj:

$$\begin{gathered} 4 \cdot -\frac{1}{4} x_1 - 3 \cdot -\frac{1}{4} x_2 + 1 \cdot -\frac{1}{4} x_3 = -8 \cdot -\frac{1}{4} \\ x_1 - x_2 + 2x_3 = 3 \end{gathered}$$

Prvu jednačinu prepisujemo bez promene jer će nam u tom obliku biti potrebna u narednim koracima.

$$\begin{gathered} 4x_1 - 3x_2 + x_3 = -8 \\ 0 - \frac{1}{4} x_2 + \frac{7}{4} x_3 = 5 \end{gathered}$$

Rezultat prethodnih operacija:

$$\begin{gathered} 4x_1 - 3x_2 + x_3 = -8 \\ 0 - \frac{1}{2} x_2 - \frac{5}{2} x_3 = -8 \\ 0 - \frac{1}{4} x_2 + \frac{7}{4} x_3 = 5 \end{gathered}$$

Još uvek nemamo jednu jednačinu sa jednom nepoznatom, pa nastavljamo tako što koristimo drugu jednačinu da izbacimo x_2 iz treće.

Drugu jednačinu da pomnožimo sa $-2 \cdot \frac{1}{4} = -\frac{1}{2}$ i onda je dodamo trećoj:

$$\begin{gathered} 4x_1 - 3x_2 + x_3 = -8 \\ 0 - \frac{1}{2} \cdot -\frac{1}{2} x_2 - \frac{5}{2} \cdot -\frac{1}{2} x_3 = -8 \cdot -\frac{1}{2} \\ 0 - \frac{1}{4} x_2 + \frac{7}{4} x_3 = 5 \end{gathered}$$

Drugu jednačinu prepisujemo bez množenja sa $-\frac{1}{2}$ jer će nam u tom obliku biti potrebna u narednim koracima.

$$\begin{gathered} 4x_1 - 3x_2 + x_3 = -8 \\ 0 - \frac{1}{2} x_2 - \frac{5}{2} x_3 = -8 \\ 0 - 0 + 3x_3 = 9 \end{gathered}$$

Sada smo sistem "konačno" dobili jednu jednačinu sa jednom nepoznatom.

$$\begin{gathered} 4x_1 - 3x_2 + x_3 = -8 \\ -\frac{1}{2} x_2 - \frac{5}{2} x_3 = -8 \\ 3x_3 = 9 \end{gathered}$$

Upravo smo završili prvu fazu Gausove eliminacije koja se zove *eliminacija unapred*.

Sada vrlo lako možemo da izračunamo x_3:

$$x_3 = \frac{9}{3} = 3$$

Zamenjujemo x_3 u drugu jednačinu i izračunavamo x_2:

$$\begin{gathered} -\frac{1}{2} x_2 - \frac{5}{2} \cdot 3 = -8 \\ -\frac{1}{2} x_2 = -8 + \frac{15}{2} \\ -\frac{1}{2} x_2 = -\frac{1}{2} \\ x_2 = 1 \end{gathered}$$

Kao poslednji korak zamenjujemo x_2 i x_3 u prvu jednačinu i izračunavamo x_1:

$$\begin{gathered} 4x_1 - 3 \cdot 1 + 3 = -8 \\ 4x_1 = -8 \\ x_1 = -2 \end{gathered}$$

Upravo smo završili drugu fazu Gausove eliminacije koja se zove *zamena unazad* i time rešili sistem.

Šta mislite da li je prethodni postupak primenjliv na sisteme veće od *3x3*?

Hajde da pokušamo da napišemo kod za metod koji smo koristili na prethodom primeru pa da vidimo da li može da se primeni na bilo koji sistem.

Kod ćemo razdvojiti na manje celine.

Prva celina je množenje prve vrste nekom vrednosti *p* i dodavanje drugoj vrsti.

In [1]:
```
import numpy as np

A=np.array([[4.,-3.,1.],[-2.,1.,-3.],[1.,-1.,2.]])
b=np.array([-8.,-4.,3.])

print(A)
print(b)
```
```
[[ 4. -3.  1.]
 [-2.  1. -3.]
 [ 1. -1.  2.]]
[-8. -4.  3.]
```

In [2]:
```
(n,m)=A.shape
p=2/4
for j in range(n): #imamo n elemenata u svakoj vrsti
    A[1,j]=A[1,j] + A[0,j]*p #indeksiranje ide od nule

print(A)
```
```
[[ 4.  -3.   1. ]
 [ 0.  -0.5 -2.5]
 [ 1.  -1.   2. ]]
```

Na koji način smo napisali prethodni kod:

- #### Imamo jednu promenljivu j koja je brojač za kolone.
- #### Menjamo sve elemente druge vrste, zato imamo A(2,j).
- #### Na svaki element druge vrste dodajemo odgovarajući element prve vrste koji je u istoj koloni (koloni j) "iznad njega".
- #### Pre nego što uradimo dodavanje, element prve vrste množimo sa p.

Na elemente treće vrste dodajemo elemente prve vrste pomnožene sa p.

In [3]:
```
(n,m)=A.shape
p=-1/4
for j in range(n):
    A[2,j]=A[2,j] + A[0,j]*p

print(A)
```
```
[[ 4.   -3.    1.  ]
 [ 0.   -0.5  -2.5 ]
 [ 0.   -0.25  1.75]]
```

Spajamo prethodna dva isečka koda u jedan.

In [4]:
```
A=np.array([[4.,-3.,1.],[-2.,1.,-3.],[1.,-1.,2.]])
(n,m)=A.shape
for i in range(1,3):
    p=2/4
    if i==2:
        p=-1/4
    for j in range(n):
        A[i,j]=A[i,j] + A[0,j]*p

print(A)
```
```
[[ 4.   -3.    1.  ]
 [ 0.   -0.5  -2.5 ]
 [ 0.   -0.25  1.75]]
```

Da li nam je potreban if ili možda postoji neki bolji način da odredimo p?

In [5]:
```
A=np.array([[4.,-3.,1.],[-2.,1.,-3.],[1.,-1.,2.]])
(n,m)=A.shape
for i in range(1,3):
    p=-A[i,0]/A[0,0]
    for j in range(n):
        A[i,j]=A[i,j] + A[0,j]*p

print(A)
```
```
[[ 4.   -3.    1.  ]
 [ 0.   -0.5  -2.5 ]
 [ 0.   -0.25  1.75]]
```

Upoštavamo sad kod tako da pored prve vrste množimo i drugu vrstu odgovrajućim brojem i dodajemo na treću.

In [6]:
```
(n,m)=A.shape
for k in range(2):
    for i in range(k+1,n):
        p = -A[i,k]/A[k,k]
        for j in range(n):
            A[i,j]=A[i,j] + A[k,j]*p
print(A)
```
```
[[ 4.  -3.   1. ]
 [ 0.  -0.5 -2.5]
 [ 0.   0.   3. ]]
```

Upoštavamo kod tako da možemo da radimo sa bilo kojom kvadratnom matricom *nxn*.

Pošto je vektor *b* deo sistema dodajemo i njega u kod.

In [7]:
```
A=np.array([[4.,-3.,1.],[-2.,1.,-3.],[1.,-1.,2.]])
b=np.array([-8.,-4.,3.])
```

In [8]:
```
(n,m)=A.shape
for k in range(n-1): #koristimo sve vrste da radimo eliminaciju osim poslednje zato petlja ide do n-1
    for i in range(k+1,n): #menjamo vrednosti 3 na n da bi omogućili rad sa proizvoljnim matricama
        p = -A[i,k]/A[k,k]
        for j in range(n):
            A[i,j]=A[i,j] + A[k,j]*p
        b[i]=b[i] + b[k]*p

print(A)
print(b)
```
```
[[ 4.  -3.   1. ]
 [ 0.  -0.5 -2.5]
 [ 0.   0.   3. ]]
[-8. -8.  9.]
```

Sada ste naučili kako funkcioniše prva faza Gausove eliminacije koja se zove *eliminacija unapred*.

Cilj ove faze je ono što smo upravo uradili, a to je svođenje matrice A na gornju trougaonu.

Prelazimo na drugu (i poslednju) fazu koja se naziva *zamena unazad*.

Kao prvi korak odredićemo poslednje x, tj. x_3.

Upoštićemo odmah kod i pišemo x_n jer je n broj kolona matrice *A*, tj. broj promenjljivih.

In [9]:
```
x=np.zeros(n)
x[n-1]=b[n-1]/A[n-1,n-1]
```

In [10]:
```
print(A)
print(b)
print(x)
```
```
[[ 4.  -3.   1. ]
 [ 0.  -0.5 -2.5]
 [ 0.   0.   3. ]]
[-8. -8.  9.]
[0. 0. 3.]
```

Određujemo sada x_2 tako što zamenjujemo x_3 u drugu jednačinu.

$$\begin{gathered} -\frac{1}{2} x_2 + \frac{5}{2} \cdot 3 = -8 \\ x_2 = (-8 + \frac{15}{2}) \cdot -\frac{2}{1} \end{gathered}$$

In [11]:
```
s=A[1,2]*x[2]
x[1]=(b[1]-s)/A[1,1]

print(x)
```
```
[0. 1. 3.]
```

Određujemo sada x_1 tako što zamenjujemo x_2 i x_3 u prvu jednačinu.

$$\begin{gathered} 4x_1 - 3 \cdot 1 + 3 = -8 \\ x_1 = (-8 - (-3 \cdot 1 + 3)) \cdot \frac{1}{4} \end{gathered}$$

In [12]:
```
s=A[0,1]*x[1]+A[0,2]*x[2]
x[0]=(b[0]-s)/A[0,0]

print(x)
```
```
[-2.  1.  3.]
```

Uopštavamo kod za x_1.

In [13]:
```
x=np.zeros(n)

s=0
for j in range(i+1,n):
    s = s + A[0,j]*x[j]
x[0]=(b[0]-s)/A[0,0]

print(x)
```
```
[-2.  0.  0.]
```

Upoštavamo kod za sve promenjlive.

In [14]:
```
x=np.zeros(n)
for i in range(n-1,-1,-1):
    s=0
    for j in range(i+1,n):
        s = s + A[i,j]*x[j]
    x[i]=(b[i]-s)/A[i,i]

print(x)
```
```
[-2.  1.  3.]
```

Spajamo kod za eliminaciju unapred i zamenu unazad i pišemo funkciju za Gausovu eliminaciju.

In [15]:
```
def gauss(A,b):
    (n,m)=A.shape
    for k in range(n-1):
        for i in range(k+1,n):
            p = -A[i,k]/A[k,k]
            for j in range(n):
                A[i,j]=A[i,j] + A[k,j]*p
            b[i]=b[i] + b[k]*p
    x=np.zeros(n)
    for i in range(n-1,-1,-1):
        s=0.
        for j in range(i+1,n):
            s = s + A[i,j]*x[j]
        x[i]=(b[i]-s)/A[i,i]
    return x
```

In [16]:
```
A=np.array([[4.,-3.,1.],[-2.,1.,-3.],[1.,-1.,2.]])
b=np.array([-8.,-4.,3.])

x=gauss(A,b)

print(x)
```
```
[-2.  1.  3.]
```

In [17]:
```
A1=np.random.rand(8,8)
b1=np.random.rand(8)
x1=gauss(A1,b1)

print(A1)
print(b1)
print(x1)
```
```
[[ 6.24175394e-03  6.51152444e-01  2.34375368e-01  4.80263442e-01
   5.32422108e-01  1.51160215e-01  8.69858742e-01  9.90003901e-01]
 [ 0.00000000e+00 -3.01925160e+01 -1.08052969e+01 -2.26734865e+01
  -2.51852842e+01 -7.01517551e+00 -4.08594730e+01 -4.58964452e+01]
 [ 0.00000000e+00  0.00000000e+00  2.78299554e-01  1.23305678e+00
   1.49245512e+00  6.02843936e-01  1.05672102e+00 -3.02700811e-01]
 [-1.11022302e-16  0.00000000e+00  0.00000000e+00 -5.44968065e-01
  -7.10772746e-01 -3.06395217e-01 -1.00888464e+00 -3.90628173e-01]
 [ 3.81023457e-16  0.00000000e+00  0.00000000e+00  0.00000000e+00
   5.82879035e-01  1.02388316e+00  2.06046584e+00  1.79716655e+00]
 [-1.44535264e-16  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  8.14213169e-01  2.48367007e-01  3.30616984e-01]
 [ 3.69629474e-17  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00 -1.26068348e+00 -1.03486683e+00]
 [-2.48153360e-16  7.10542736e-15  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00 -1.00315504e+00]]
[ 0.00546406  0.39783809 -0.98745348  0.27116574  0.54967321 -0.4533171
  1.12575058 -1.59896216]
[ 2.05449908  0.17746259 -2.44466484 -3.45496539  4.74573999 -0.53247139
 -2.20139255  1.59393325]
```

Pre nego što pređemo na sledeću celinu dat je malo optimizovaniji kod za Gausovu eliminaciju. Ovaj kod nije od značaja za vaše razumevanje algoritma već demonstrira kako možemo efikasno da koristimo vektorske operacije koje nude jezici kao što su MATLAB, Octave, Python itd. Kod nije deo obaveznog gradiva dat je samo informativno.

In [18]:
```
def gauss_vect(A,b):
    (n,m)=A.shape
    Aaug = np.zeros((n,n+1))
    Aaug[0:n,0:n]=A
    Aaug[:,n]=b

    for k in range(n-1):
        for i in range(k+1,n):
            m=-Aaug[i,k]/Aaug[k,k]
            Aaug[i,:]=Aaug[i,:]+m*Aaug[k,:]

    x=np.zeros(n)
    for i in range(n-1,-1,-1):
        x[i]=(Aaug[i,n]-np.dot(Aaug[i,0:n],x))/Aaug[i,i]
    return x
```

In [19]:
```
A=np.array([[4.,-3.,1.],[-2.,1.,-3.],[1.,-1.,2.]])
b=np.array([-8.,-4.,3.])

x=gauss_vect(A,b)
print(x)
```
```
[-2.  1.  3.]
```

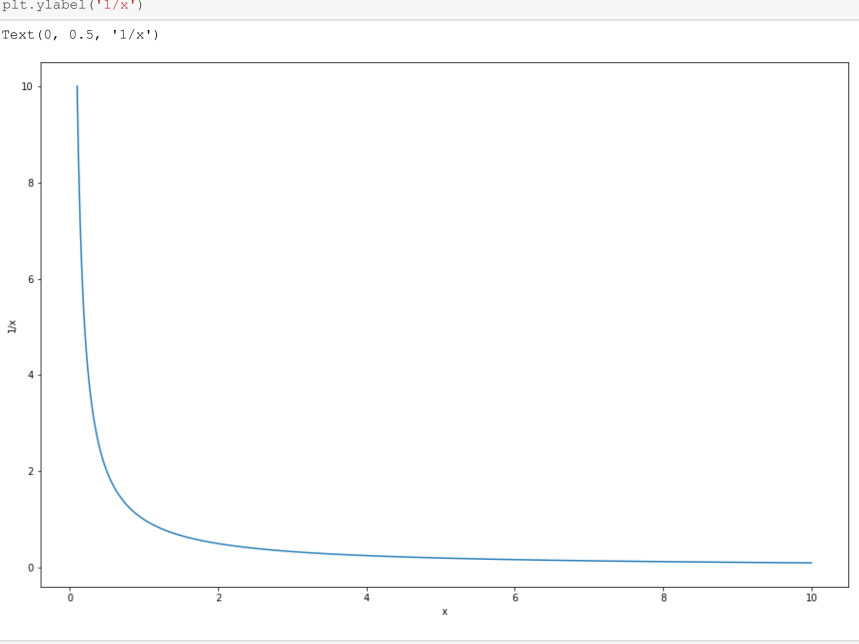
Napomena: u ispisu matrice Aug, nakon pokretanja funkcje gauss, poslednja kolona je vektor b pa zato deluje kao da rezultat nije gornja trougaona matrica.

Problemi sa orignialnim (naivnim) algoritmom Gausove eliminacije

Pored očiglednog problema deljenja nulom do koga dolazi kada određujemo vrednost $m = \frac{a_{ij}}{a_{ii}}$, a vrednost $a_{ii} = 0$ što se u navinoj GE ne proverava, postoje i malo suptilniji problemi o kojima ćemo se baviti u nastavku.

Ako pogledamo sledeći grafik funkcije $f(x) = \frac{1}{x}$, vidimo da kada su vrednosti x male, npr. manje od 1, za male promene x imamo velike promene $f(x)$ dok za vreće vrednosti x to ne važi. Ova osobina operacije deljenja nam je veoma značajna kao potencijalni izvor problema sa Gausovom eliminacijom.

In [20]:
```
import matplotlib.pyplot as plt

plt.figure(figsize=(15, 10))

x_corr=np.linspace(0.1,10,1000)
plt.plot(x_corr,1.0/x_corr)

plt.xlabel('x')
plt.ylabel('1/x')
```
Out[20]:
```
Text(0, 0.5, '1/x')
```



In [21]:
```
print(1/0.0049)
print(1/0.005)
```
```
204.08163265306123
200.0
```

In [22]:
```
print(1/2300)
print(1/2350)
```
```
0.0004347826086956522
0.000425531914893617
```

Povezaćemo sada operaciju deljenja i ograničen kapacitet računara za smeštanje brojeva i time pokazati šta su problemi sa postupkom GE.

Ograničen kapacitet računara za smeštanje brojeva može proizvesti gubitak informacija tokom računskih operacija. Na primer, ako pomnožimo sledeća dva broja koji mogu da se smeste na računar, rezultat je takav da ne može ceo biti smešten već moramo da ga zaokružimo.

12.123456789 * 14.123456789 = 171.225118092750190521

In [23]:
```
12.123456789 * 14.123456789
```
Out[23]:
```
171.2251180927502
```

Sličnu situaciju imamo i sa deljenjem. Na primer ako delimo 56.985/101.53 dobijamo sledeći rezultat pomoću digitrona sa beskonačnom preciznošću:

0.5612626809809908401457697232345119669063331035162021077514035260514133753570373288683147838077415542204274598640

Očigledno je da moramo da rezultat moramo da zaokružimo.

Gubitak informacija je naročito izražen kada su brojevi koji učestvuju u računskoj opreciji različitog reda veličine, kao u sledećem primeru.

1463.0345 + 0.000123456789 = 1463.034623456789

In [24]:
```
1463.0345 + 0.000123456789
```
Out[24]:
```
1463.034623456789
```

Ako bi prethodni zbir zbog ograničenog kapaciteta računara morali da zaokružimo na recimo 9 cifara izgubili bi sve osim jedne cifre drugog sabirka, tj. drugi sabirak bi skoro bio beznačajan.

U Gausovoj eliminaciji prilikom svođenja na gornju trougaonu matricu ponavljamo sledeća dva koraka:

1. $m = \frac{a_{i,j}}{a_{i,i}}$

2. $a_{i,j} = a_{i,j} - m * a_{k,j}$

Do gubitka informacija može doći ako se umanjenik i umanjilac u koraku 2. razlikuju po redu veličine, što se može dogoditi ako je m jako mali ili jako veliki broj. Vrednost m može biti jako velika u slučaju da je $a_{i,i}$ jako mali broj, tj. blizu 0. Iz tog razloga trebalo bi da pokušamo da izbegnemo deljenje jako malim brojevima tokom Gausove eliminacije. To postižemo upotrebom pivotinga.

Pivoting

- #### Iz do sada navedenog može se zaključiti da se problem u koraku 2. može rešiti ili ublažiti ako pivot element (sa kojim se u postupku eliminacije unapred vrši deljenje) ima što veću moguću vrednost.
- #### Iz oblasti algebre znamo da promena rasporeda jednačina i promenljivih (premeštanje vrsta i kolona matrice sistema) ne menja rešenje sistema.
- #### Ovu činjenicu možemo da iskoristimo da bi prilikom eliminacije svake promenljive na mesto pivot elementa postavli najveći mogući element po apsolutnoj vrednosti.
- #### Postupak postavljanja najvećeg mogućeg elementa po aspsulutnoj vrednosti na mesto pivota naziva se pivoting.
- #### Ako najveći mogući element tražimo u celoj matrici sistema, tačnije u delu koji nije anuliran u postupku elimancije unapred, onda izvršavamo kompletan pivoting.
- #### Pracijalni pivoting predstavlja efikasniju varijantu pivotinga pri kojoj novi pivot element tražimo samo u koloni u kojoj se nalazi promenljiva koja se trenutno eliminiše. U praksi se najčešće koristi parcijalni pivoting jer je efikasniji uz prihvatljiv pad kvaliteta rešenja.

In [25]:
```
A=np.array([[4.,-3.,1.],[-2.,1.,-3.],[1.,-1.,2.]])
print(A)
print(A[1:3,1])
```
```
[[ 4. -3.  1.]
 [-2.  1. -3.]
 [ 1. -1.  2.]]
[ 1. -1.]
```

In [26]:
```
print(np.argmax([12,54,7]))
```
```
1
```

In [27]:
```
def gauss_with_pivoting(A,b):
    (n,m)=A.shape

    for k in range(n-1):
        loc = np.argmax(np.abs(A[k:n,k]))
        loc = k+loc

        print(A)
        print("+++++++++++++++++++++++++++++++++++++++++++")

        temp = A[k,:].copy()
        A[k,:] = A[loc,:]
        A[loc,:] = temp

        print(A)
        print("-------------------------------------------")

        temp = b[k]
        b[k] = b[loc]
        b[loc] = temp
        for i in range(k+1,n):
            p = -A[i,k]/A[k,k]
            for j in range(n):
                A[i,j]=A[i,j] + A[k,j]*p
            b[i]=b[i] + b[k]*p

    x=np.zeros(n)
    for i in range(n-1,-1,-1):
        s=0.
        for j in range(i+1,n):
            s = s + A[i,j]*x[j]
        x[i]=(b[i]-s)/A[i,i]
    return x
```

In [28]:
```
A2=np.array([[5.,0.,2.,3.],[-2.,2.,2.,-3.],[0.,1.,1.,4.],[6.,2.,2.,4.]])
b2=np.array([1.,-1.,2.,1.])

x=gauss_with_pivoting(A2,b2)
```
```
[[ 5.  0.  2.  3.]
 [-2.  2.  2. -3.]
 [ 0.  1.  1.  4.]
 [ 6.  2.  2.  4.]]
+++++++++++++++++++++++++++++++++++++++++++
[[ 6.  2.  2.  4.]
 [-2.  2.  2. -3.]
 [ 0.  1.  1.  4.]
 [ 5.  0.  2.  3.]]
-------------------------------------------
[[ 6.          2.          2.          4.        ]
 [ 0.          2.66666667  2.66666667 -1.66666667]
 [ 0.          1.          1.          4.        ]
 [ 0.         -1.66666667  0.33333333 -0.33333333]]
+++++++++++++++++++++++++++++++++++++++++++
[[ 6.          2.          2.          4.        ]
 [ 0.          2.66666667  2.66666667 -1.66666667]
 [ 0.          1.          1.          4.        ]
 [ 0.         -1.66666667  0.33333333 -0.33333333]]
-------------------------------------------
[[ 6.00000000e+00  2.00000000e+00  2.00000000e+00  4.00000000e+00]
 [ 0.00000000e+00  2.66666667e+00  2.66666667e+00 -1.66666667e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  4.62500000e+00]
 [ 0.00000000e+00  2.22044605e-16  2.00000000e+00 -1.37500000e+00]]
+++++++++++++++++++++++++++++++++++++++++++
[[ 6.00000000e+00  2.00000000e+00  2.00000000e+00  4.00000000e+00]
 [ 0.00000000e+00  2.66666667e+00  2.66666667e+00 -1.66666667e+00]
 [ 0.00000000e+00  2.22044605e-16  2.00000000e+00 -1.37500000e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  4.62500000e+00]]
-------------------------------------------
```

Ako pogledamo poslednju matricu pred pivoting za naš primer vidimo da bi bez pivotinga u tom slučaju imali deljenje nulom.

Da li Gausova eliminacija može da rezultuje pogrešnim rešenjem?

- #### Sam postupak (algoritam) Gausove eliminacije ne sadrži grešku, ali GE može vratiti pogrešno rešenje ako vrednosti A ili b odstupaju od onih koje su originalno zadate.
- #### Zašto bi bilo odstupanja u vrednostima?
- #### Jedan od banalnih razloga može biti da je neko prilikom unosa podataka pogrešio.
- #### Drugi, mnogo realniji i češći razlog je da su vrednosti A i b takve da ne mogu da se potpuno tačno reprezentuju na računaru koji radi sa ograničenom preciznošću.

Logično pitanje je na koji način možemo da proverimo da li je GE vratio pogrešno rešenje.

- #### Prvi odgovor bio bi da jednostavno ubacimo dobijeno rešenje x u sistem i proverimo jednakost, tj. da pomnožimo x sa A i proverimo da li važi $Ax = b$.

U nastavku ćemo pokazati da ostatak $r = Ax - b$ nije pouzdana mera tačnosti kada brojeve reprezentujemo sa ograničenom preciznošću. Kod primera u nastavku računske operacije se izvršavaju na računaru koji može da smesti samo tri značajne cifre broja. To je namerno urađeno da bi se ilustrovao problem sa nepouzdanošću ostatka r. Savremeni računari mogu da smeste 16 značajnih cifara upotrebom tipa $double$, odnosno i njihov kapacitet je isto ograničen samo sa većim brojem cifara.

In [29]:
```
from sigfig import round

print(round(171.2251180927502, sigfigs = 4))
```
```
171.2
```

In [30]:
```
def gauss_low_precision(A,b):
    (n,m)=A.shape
    for k in range(n-1):
        for i in range(k+1,n):

            p_full_precision = -A[i,k]/A[k,k]
            p_rounded = round(p_full_precision, sigfigs = 3)

            print(p_full_precision)
            print(p_rounded)
            print()

            for j in range(n):
                a_mul_p_full_pr = A[k,j]*p_full_precision
                a_mul_p_rounded = round(a_mul_p_full_pr, sigfigs = 3)

                print(a_mul_p_full_pr)
                print(a_mul_p_rounded)
                print()

                A[i,j]=round(A[i,j] + a_mul_p_rounded,3)

            b_mul_p_full_pr = b[k]*p_full_precision
            b_mul_p_rounded = round(b_mul_p_full_pr, sigfigs = 3)

            print()
            print(b_mul_p_full_pr)
            print(b_mul_p_rounded)

            b[i]=round(b[i] + b_mul_p_rounded,3)
    print()
    print(A)
    print(b)
    print()
    x=np.zeros(n)
    for i in range(n-1,-1,-1):
        s=0.
        for j in range(i+1,n):
            s = round(s + round(A[i,j]*x[j],3),3)
        x[i]=round(round(b[i]-s,3)/A[i,i],3)
    return x
```

In [31]:
```
A=np.array([[0.641,0.242],[0.321,0.122]])
b=np.array([0.883,0.444])

print(A)
print()
print(b)
```
```
[[0.641 0.242]
 [0.321 0.122]]

[0.883 0.444]
```

In [32]:
```
x_low_pr = gauss_low_precision(A,b)
```

```
-0.500780031201248
-0.501

-0.321
-0.321

-0.12118876755070201
-0.121


-0.442188767550702
-0.442

[[0.641 0.242]
 [0.    0.001]]
[0.883 0.002]
```

```
C:\Users\kocha\Anaconda3\lib\site-packages\sigfig\sigfig.py:572: UserWarning: warning: 3 significant figures requested from number with only 1 significant figures
  warn("warning: %d significant figures requested from number with only %d significant figures" % (given['sigfigs'], len(num.map)))
C:\Users\kocha\Anaconda3\lib\site-packages\sigfig\sigfig.py:572: UserWarning: warning: 3 significant figures requested from number with only 2 significant figures
  warn("warning: %d significant figures requested from number with only %d significant figures" % (given['sigfigs'], len(num.map)))
```

Napomena oko prethodnog primera: zbog zaokruživanja vrednosti a_puta_p_full_prec = -0.12124 na tri značajne cifre, tj. na -0.121 dolazi do gubitka informacija. Prilikom sabiranja vrednosti -0.121 sa vrednošću 0.122 (kao deo postpuka eliminacije unapred), dobijamo vrednost 0.001 koja ima samo jednu zančajnu cifru. Taj gubitak se onda propagira dalje što onda, kao jedan od faktora, rezultuje pogrešnim rešenjem. Slična situacija je i sa vektorom b.

In [33]:
```
print(x_low_pr)
```

```
[0.622 2.   ]
```

In [34]:
```
x_low_pr_col=np.array([x_low_pr]).T
print(x_low_pr_col)
```

```
[[0.622]
 [2.   ]]
```

In [35]:
```
b_col=np.array([b]).T
residual_full_prec=np.matmul(A,x_low_pr_col)-b_col
print(residual_full_prec)
```

```
[[-0.000298]
 [ 0.      ]]
```

Ako radimo u ograničenoj preciznosti od 3 značajne cifre, ostatak je 0 što ukazuje na to da je naše rešenje tačno. Međutim tačno rešenje se već na prvoj značajnoj cifri razlikuje od našeg rešenja. Kao što ćete videti u nastavku.

In [36]:
```
A=np.array([[0.641,0.242],[0.321,0.122]])
b=np.array([0.883,0.444])

x_full_prec = gauss_with_pivoting(A,b)
```

```
[[0.641 0.242]
 [0.321 0.122]]
+++++++++++++++++++++++++++++++++++++++++++
[[0.641 0.242]
 [0.321 0.122]]
-------------------------------------------
```

In [37]:
```
print(x_full_prec)
```

```
[0.53461538 2.23269231]
```

In [38]:
```
print(x_low_pr)
```

```
[0.622 2.   ]
```

In [39]:
```
x_full_prec - x_low_pr
```

Out[39]:
```
array([-0.08738462,  0.23269231])
```

Prethodni primer pokazuje nepouzdanost ostatka za određivanje greške. Zato ćemo u nastavku predavanja pokazati na koji način možemo da procenimo grešku. Ne moramo znati tačnu grešku ako smo uspeli da je procenimo na malu vrednost tj. nije nam važno koliko je tačno greška mala, već da je mala. Za procenu nam je od velikog značaja kondicioni broj matrice.

Kondicioni broj funkcije

- #### Kondicioni broj funkcije $f(x)$ meri koliko promene ulaza x utiču na promene izlaza y.
- #### Veliki kondicioni broj znači da male promene ulaza daju velike na promene izlaza.
- #### Tada kažemo da je funkcija loše uslovljena.
- #### Ako ste ikad malo pomerili slavinu pod tušem, a voda je od hladne prešla na jako vrelu – to je loše uslovljena funkcija!

Kondicioni broj matrice

- #### U kontekstu rešavanja sistema linearnih jednačina kondicioni broj matrice A meri koliko male promene vektora b ili matrice A utiču na rešenje x.
- #### Kao što smo već pomenuli male promene mogu biti posledica lošeg unosa podataka ili ograničenog kapaciteta računara za smeštanje brojeva.
- #### Za izračunavanje kondicionog broja matrice značajan nam je koncept norme matrice koji ćemo ukratko objasniti u nastavku.

Norma vektora i matrice

- #### Norma matrice je relana vrednost koju dodeljujemo matrici.
- #### Postoje različite norme matrice koje se razlikuju po načinu na koji na osnovu matrice izračunavamo jednu relanu vrednosti.
- #### Detalji vezani za normu matrice predmet su linearne algebre, dok ćemo na ovom kursu pokazati samo neke od poznatih načina za određivanje norme matrice.

Maksimum zbira apsolutnih vrednosti vrsta matrice:

$\|A\|_\infty = \max\limits_i \sum_{j=1}^{n} |A_{i,j}|$

In [40]:
```
from numpy import linalg as la

B = np.array([[1.,2.],[1.,-3.9999]])
print(B)

la.norm(B,np.Inf)
```

```
[[ 1.      2.    ]
 [ 1.     -3.9999]]
```

Out[40]:
```
4.9999
```

Maksimum zbira apsolutnih vrednosti kolona matrice:

$\|A\|_1 = \max\limits_j \sum_{i=1}^{n} |A_{i,j}|$

In [41]:
```
la.norm(B,1)
```

Out[41]:
```
5.9999
```

Sada kada smo objasnili normu matrice, daćemo definiciju kondicionog broja matrice:

$cond(A) = \|A\| \|A^-1\|$

In [42]:
```
#po default koristi se najveca singularna vrednost kao norma
la.cond(B)
```

Out[42]:
```
3.3698448806330688
```

Kondicioni broj matrice i Gausova eliminacija

Kondicioni broj matrice i vektor b

- #### Ako u sistemu $Ax = b$, b nije tačno (zbog grešaka zaokruživanja npr.), koliko će se x razlikovati od tačnog rešenja?
- #### Ako umesto b imamo $b + \Delta b$ onda važi sledeća nejednakost:

$$\frac{\|\Delta x\|}{\|x\|} <= cond(A)\frac{\|\Delta b\|}{\|b\|}$$

- #### Dakle, relativna (procentualna) greška rešenja ograničena je ne samo sa relativnom greškom vektora b već i sa kondicionim brojem matrice A.
- #### Kondicioni broj se na neki način može smatrati "pojačivačem" greške vektora.
- #### Ako je kondicioni broj matrice A veliki, postoji mogućnost da će male promene vektora b rezultovati velikim promenama rešenja x, kao što ćete videti na sledećem primeru.
- #### Termin "postoji mogućnost" upotrebljen je namerno jer je relativna greška rešenja samo ograničena pomoću prethone formule (upotrebljen je znak nejednakosti, a ne znak jednakosti).

Kondicioni broj matrice i matrica A

- #### Ako umesto A imamo neku matricu E koja se razlikuje od A (zbog grešaka zaokruživanja npr.) onda važi sledeća nejednakost:

$$\frac{\|\Delta x\|}{\|x\|} <= cond(A)\frac{\|E\|}{\|A\|}$$

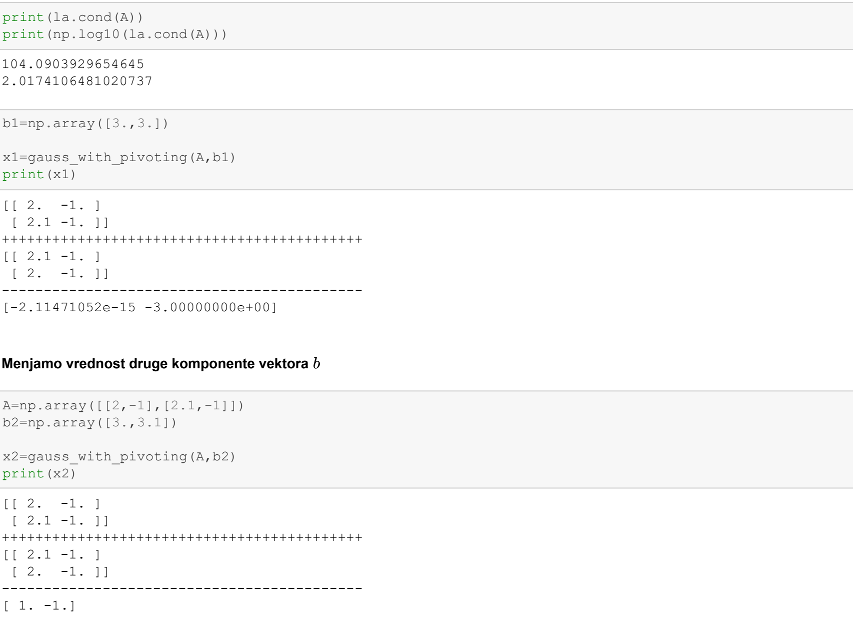
Primer rešavanja loše uslovljenog sistema pomoću Gausove eliminacije:

$$-x_1 + 2x_2 = 3$$
$$-x_1 + 2.1x_2 = 3$$

Grafik loše uslovljenog sistema. Da li primećujete nešto specifično na njemu?

In [43]:
```
plt.figure(figsize=(15, 10))

x1=np.linspace(-10,5,100)
x2=(3.+x1)/2.

plt.plot(x1,x2)

x2=(3.+x1)/2.1

plt.plot(x1,x2)

plt.xlabel('x1')
plt.ylabel('x2')
```

Out[43]:
```
Text(0, 0.5, 'x2')
```



In [49]:
```
A=np.array([[2,-1],[2.1,-1]])
print(A)
```

```
[[ 2.  -1. ]
 [ 2.1 -1. ]]
```

In [50]:
```
print(la.cond(A))
print(np.log10(la.cond(A)))
```

```
104.0903929654645
2.0174106481020737
```

In [51]:
```
b1=np.array([3.,3.])

x1=gauss_with_pivoting(A,b1)
print(x1)
```

```
[[ 2.  -1. ]
 [ 2.1 -1. ]]
+++++++++++++++++++++++++++++++++++++++++++
[[ 2.1 -1. ]
 [ 2.  -1. ]]
-------------------------------------------
[-2.11471052e-15 -3.00000000e+00]
```

Menjamo vrednost druge komponente vektora b

In [52]:
```
A=np.array([[2,-1],[2.1,-1]])
b2=np.array([3.,3.1])

x2=gauss_with_pivoting(A,b2)
print(x2)
```

```
[[ 2.  -1. ]
 [ 2.1 -1. ]]
+++++++++++++++++++++++++++++++++++++++++++
[[ 2.1 -1. ]
 [ 2.  -1. ]]
-------------------------------------------
[ 1. -1.]
```

Izračunavamo gornju granicu relativne greške rešenja

In [53]:
```
la.cond(A)*la.norm(b1-b2)/la.norm(b1)
```

Out[53]:
```
2.4876915511108306
```

Pošto znamo rešenje, izračunavamo relativnu grešku rešenja

In [54]:
```
la.norm(x1-x2)/la.norm(x1)
```

Out[54]:
```
0.7453559924999287
```

Relativna greska je preko ~75%, ali je opet drastično manja nego gornje ograničenje od ~245%.

Red veličine kondicionog broja i relativna greška

Postoji tvrđenje (koje nećemo dokazivati) pomoću koga možemo da povežemo relativnu grešku i red veličine kondicionog broja:

Ako imamo dve vrednosti x i $\hat{x}$, tada se one poklapaju u bar $m-1$ značajnih cifara ako važi:

$$\frac{\|x-\hat{x}\|}{\|x\|} \leq 5 * 10^{-m}$$

To znači da bi rešenje dobijeno pomoću GE imalo bar jednu značajnu cifru od tačnog rešenja, relativna greška mora da bude manja od $5 * 10^{-2} = 0.05$

Odnosno, ako želimo da budemo sigurni da se dve vrednosti poklapaju u bar jednoj značajnoj cifri, relativna greška mora biti manja ili jednaka od 5%

In [65]:
```
A=np.array([[2,-1],[2.1,-1]])

b1=np.array([3.,3.])
b2=np.array([3.,3.1])

print(la.cond(A))
print(np.log10(la.cond(A))) #red velicine kondicionog broja matrice A

b_err=b1-b2
print(b_err)

print(np.log10(la.norm(b1-b2)/la.norm(b1))) #red velicine promene vektora b
print(np.log10(la.cond(A)) + np.log10(la.norm(b1-b2)/la.norm(b1))) #red veclicine ogranicenja grekse tacnog resenja
```

```
104.0903929654645
2.0174106481020737
[ 0.  -0.1]
-1.6276362525516526
0.38977439555042115
```

Za naš primer vidimo da je kondicioni broj reda veličine takvog da odgovara približno broju 10 na stepen 2, što znači da naše rešenje potencijalno neće da se pokalapa sa tačnim rešenjem ni u jednoj cifri - kao što je istaknuto ranije, greška u rešenju zavisi i od promene vrednosti vektora b (ili matrice A), a ne samo od kondicionog broja matrice A.

Dakle, možemo da vidimo da nam vrednost $log10(cond(A))$ može poslužiti kao mera broja cifara tačnog rešenja koje možemo da izgubimo ako imamo greške u podacima prilikom rešavanja nekog sistema.

In [58]:
```
m=2 #provervavamo da li imamo bar jednu istu cifru sta tačnim rešenjem
(la.norm(x1-x2)/la.norm(x1))<5.*np.power(10.,-m)
```

Out[58]:
```
False
```

Pogledaćmo sada jedan primer dobro uslovljenog sistema, tj. sistema kod koga matrica A ima malu vrednosti kodicionog broja.

$$2x_1 - x_2 = 3$$
$$-2x_1 - x_2 = 3$$

In [59]:
```
plt.figure(figsize=(15, 10))

x1=np.linspace(-10,5,100)
x2=-(3.-2.*x1)

plt.plot(x1,x2)

x2=-(3.+2.*x1)

plt.plot(x1,x2)

plt.xlabel('x1')
plt.ylabel('x2')
```

Out[59]:
```
Text(0, 0.5, 'x2')
```

In [60]:
```
A=np.array([[2.,-1.],[-2.,-1.]])

print(A)
print(la.cond(A))
print(np.log10(la.cond(A)))
```

```
[[ 2. -1.]
 [-2. -1.]]
2.0000000000000004
0.3010299956639813
```

In [61]:
```
A=np.array([[2.,-1.],[-2.,-1.]])
b1=np.array([3.,3.])
b2=np.array([3.,3.1])

x1=gauss_with_pivoting(A,b1)

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

x2=gauss_with_pivoting(A,b2)
```

```
[[ 2. -1.]
 [-2. -1.]]
+++++++++++++++++++++++++++++++++++++++++++
[[ 2. -1.]
 [-2. -1.]]
-------------------------------------------
[[ 2. -1.]
 [-2. -1.]]
+++++++++++++++++++++++++++++++++++++++++++
[[ 2. -1.]
 [-2. -1.]]
-------------------------------------------
```

In [63]:
```
print(x1)
print(x2)
```

```
[ 0. -3.]
[-0.025 -3.05 ]
```

In [66]:
```
A=np.array([[2.,-1.],[-2.,-1.]])
b1=np.array([3.,3.])
b2=np.array([3.,3.1])

print(la.cond(A))
print(np.log10(la.cond(A))) #red velicine kondicionog broja matrice A

b_err=b1-b2
print(b_err)

print(np.log10(la.norm(b1-b2)/la.norm(b1))) #red velicine promene vektora b
print(np.log10(la.cond(A)) + np.log10(la.norm(b1-b2)/la.norm(b1))) #red veclicine ogranicenja grekse tacnog resenja
```

```
2.0000000000000004
0.3010299956639813
[ 0.  -0.1]
-1.6276362525516526
-1.3266062568876713
```

In [67]:
```
m=2 #provervavamo da li imamo bar jednu istu cifru sta tačnim rešenjem
(la.norm(x1-x2)/la.norm(x1))<5.*np.power(10.,-m)
```

Out[67]:
```
True
```

Ponekad je velika vrednost kondiciong broja rezultat veoma različitih opsega u kojima se nalaze vrednosti matrice sistema. U takvim slučajevima skaliranje vrednosti na isti opseg može da smanji kondicioni broj.

In [68]:
```
A=np.array([[0.6,135.4],[22.0,-17000]])
print(la.cond(A))
```

```
21930.586769589485
```

In [69]:
```
A=np.array([[0.6/135.4,135.4/135.4],[22.0/-17000,-17000/-17000]])
print(la.cond(A))
```

```
349.31946211007323
```

In [ ]:

In [ ]: