for-ciklus

```
for ciklusvaltozo=vektor
  utasitasok
end
```

Példák

```
s=0;
for i=1:100
  s=s+i;
end
```

```
a=[4 2 -1 5];
s=0;
for i=a
  s=s+1/i;
end
```

```
s=100;
for i=98:-2:2
  s=s+i;
end
```

Megjegyzés

A Matlab vektorizált utasításai általában gyorsabbak, mint a for-ciklusként megírt kódok.

Feladat

Írjon egy kódot, melyben generál egy 1000000 elemű véletlen x vektort (használja a `rand` függvényt.) Számítsa ki azt az y vektort, melynek minden koordinátája 1-gyel nagyobb az x megfelelő koordinátájánál. Határozza meg az y vektort for-ciklussal is, illetve az $y = x + 1;$ parancs segítségével is, és mérje le mindkét kódrészlet futási idejét. (Használja a `tic` és `toc` függvényeket.) Vizsgálja meg azt is, hogy mit jelent futási időben, ha az első esetben inicializálja az y vektort (pl. egy, az x-szel megegyező méretű csupa 0 vektornak), illetve ha nem inicializálja. Ne feledkezzen meg a sorvégi pontosvesszőkről!!!
(Teljesen valós képet akkor kap, ha az adott nevű változók nem léteznek a workspace-ben. Ezért a kód minden futtatása előtt adja ki a `clear all` parancsot, illetve érdemes a két kódrészletben más-más változónevet használni `y` helyett.)

Függvények írása

A Matlab-függvények szerkezete:

```
function [kimenovaltozok]=fvneve(bemenovaltozok)
  utasitasok
end
```

- A függvény a `function` és `end` kulcsszavak között helyezkedik el
- a függvény bemenő változói a kerek zárójelek között vanank felsorolva, vesszővel elválasztva
- a függvény által visszaadott értékek a szögletes zárójelben vannak felsorolva, vesszővel elválasztva
- a bemenő értékekből az `utasitasok`-ban megadott módon határozza meg a visszaadott értékeket

Ha a fenti függvényt egy külön m-fájlban írtuk meg, akkor fvneve.m néven kell elmenteni. Ha a függvényt egy kód részeként hoztuk létre, akkor az m-fájl végén kell állnia (ekkor akár több ilyen függvény is lehet a fájl végén). A függvény ilyenkor csak az adott m-fájlon belül hívható (ú.n. lokális függvény).

Példák

```
function  y=fv1(x)
   y=x.^2-1;
end
```

Ekkor a `y=fv1(x)` utasítás eredménye a $x^2 - 1$ kifejezés értéke, ahol x akár vektor is lehet. Utóbbi esetben a függvény elemenként hajtódik végre és y is vektor (ezt az teszi lehetővé, hogy a függvényben minden művelet végrehajtható vektorokra is, mivel a négyzetreemelés jele elé kitettük a . jelet)
Például `y=fv1(-3)` esetén:

```
y =
     8
```

`y=fv1([1,2,3])` esetén:

```
y =
     0     3     8
```

Példák

```
function  [s,p]=fv2(x,y,z)
    s=x+y+z;
    p=x*y*z;
end
```

Ennek a függvénynek 3 bemenő paramétere van, ezeket vesszővel elválasztva soroljuk (akkor is, ha nem azonos típusúak, pl. egy skalár egy vektor és egy mátrix lenne).
A függvény 2 értékkel tér vissza: a bemenetként megadott 3 szám összegével és szorzatával.
Ha több visszatérési érték van, akkor ezeket a függvény létrehozásakor és hívásakor szögletes zárójelben soroljuk (akkor is, ha nem azonos típusúak).

```
>>[s,p]=fv2(-5,1,4)
s =
     0
p =
   -20
```

Ha a függvény hívásakor nem adunk meg kimenő változókat (vagy csak egyet adunk), akkor a két visszatérési érték közül az elsőt adja:

```
>>fv2(-5,1,4)
ans =
     0
```

Ha csak a második visszatérési értékre vagyunk kiváncsiak, akkor

```
>>[~,p]=fv2(-5,1,4)
p =
     -20
```

function handle

A function handle egy függvényekkel kapcsolatos adattípus.
Szükségünk lehet rá, pl. ha

- egy függvényt át akarunk adni egy másik függvénynek (pl a Matlab `integral` függvényének az integrálandó függvényt)
- parancssorban szeretnénk függvényt definiálni

Létrehozása: a @ szimbólum után következik

- egy már létező függvény neve, pl. az előbb létrehozott függvényekkel `af1=@fv1` vagy `af2=@fv2`
- egy ú.n. anoním függvény, pl. `af3=@(x) x.^3-x` vagy `af4=@(x,y) x+y-x.*y`

Hívása:

- `af1(-3)` vagy `af1([1,2,3])` vagy `[s,p]=af2(-5,1,4)`
- `af3(-3)` vagy `af3([1,2,3])` vagy `af4(6,1)`

Az előző példában miért

`af1([1,2,3])` és `af2(-5,1,4)`?

Az egyik esetben miért használtunk szögletes zárójelet, a másikban miért nem?

Az `af1` (illetve az `fv1`) egy egyváltozós függvény. Ez az egy változó lehet akár vektor is (`fv1`-ben elemenkénti műveleteket használtunk!), de a függvényt csak egyetlen változóval hívhatjuk. Ez most az `[1,2,3]` vektor. Ha elhagynánk a szögletes zárójelet, akkor 3 skalárral próbálnánk hívni a függvényt.

Az `af2` (illetve az `fv2`) egy háromváltozós függvény, 3 értékkel kell hívnunk. Ezek most a -5, 1 és 4. Ha szögletes zárójelbe tennénk a 3 értéket, akkor az egyetlen változó lenne (egy vektor).

Ha

`af3=@(x) x.^3-x` és `af4=@(x,y) x+y-x.*y`

akkor `af3` egy egyváltozós függvény (ahol a változó akár vektor is lehet, mert a hatványozás előtt használtuk a pontot). Hívhatjuk egyetlen számmal, ekkor egy számot ad vissza, vagy egy vektorral, ekkor a vektor minden elemére kiértékeli a függvényt (és az értékek vektorát adja vissza).

`af4` egy kétváltozós függvény. Hívhatjuk 2 számmal:

```
>> af4(6,1)
ans =
     1
```

de két (egyforma méretű) vektorral is, mert a szorzást elemenként definiáltuk. Ekkor a visszaadott érték is vektor:

```
>> af4([6,-1],[1,2])
ans =
     1     3
```

Kiértékelte a függvényt a $(6, 1)$ és $(-1, 2)$ párokra.

Összehasonlító operátorok

- $<$, $<=$, $>$, $>=$
  összehasonlítás, a szokásos jelentésekkel
- $==$
  az egyenlőség tesztelése
- $\sim=$
  ,,nem egyenlő”

Két mátrixra is alkalmazhatóak, ilyenkor elemenként történik az összehasonlítás

A visszaadott érték minden esetben egy logikai változó:

`logical 1` (igaz)

`logical 0` (hamis)

Példa.

```
>> 1<2
ans =
  logical
   1

>> 1>2
ans =
  logical
   0

>> [1,2,3]<[-1,1,4]
ans =
  1×3 logical array

   0   0   1
```

Logikai operátorok

- `A&B` A és B
- `A&&B` A és B (rövid kiértékelés)
- `A|B` A vagy B
- `A||B` A vagy B (rövid kiértékelés)
- `~A` nem A
- `xor(A,B)` A kizáró vagy B

A kiértékelések elemenként történnek.

rövid kiértékelés: ha az első kifejezés értéke egyértelműen eldönti a teljes kifejezés értékét, akkor a második kifejezést nem értékeli ki.

Logikai függvények

- `all(A)`
  egy sorvektorral tér vissza, oszloponként megvizsgálja, hogy minden elem 0-tól különböző-e (ugyanaz, mint `all(A,1)`)
- `all(A,2)`
  egy oszlopvektorral tér vissza, soronként megvizsgálja, hogy minden elem 0-tól különböző-e
- `any(A)`
  egy sorvektorral tér vissza, oszloponként megvizsgálja, hogy van-e 0-tól különböző elem (ugyanaz, mint `any(A,1)`)
- `any(A,2)`
  egy oszlopvektorral tér vissza, soronként megvizsgálja, hogy van-e 0-tól különböző elem

find

- `ind=find(A)`
  a nemnulla elemek sorszámaival tér vissza, ahol az elemek számozása oszlopfolytonosan történik. Ha A egy sorvektor, akkor a visszaadott érték is az, egyébként oszlopvektor
- `ind=find(A,n)`
  az A első n darab nemnulla elemének sorszámával tér vissza
- `ind=find(A,n,last)`
  az A utolsó n darab nemnulla elemének sorszámával tér vissza
- `[rowI,colI]=find(A)`
  a nemnulla elemek sor- és oszlopindexeivel tér vissza
- `[rowI,colI,elem]=find(A)`
  a nemnulla elemek sor- és oszlopindexeivel, illetve a nemnulla elemekkel tér vissza

Keresés tömbökben

Legyen `a=[1,-2,-1,0,5,4]`

- Számoljuk össze hány negatív eleme van a-nak!
  Az `a<0` kifejezés értéke egy logikai vektor: 1, ha a feltétel teljesül az adott elemre, 0, ha nem:

```
ans =
  1×6 logical array

   0   1   1   0   0   0
```

  Ha ennek a vektornak a koordinátáit összeadjuk, éppen a negatív elemek számát kapjuk, azaz az a vektor negatív elemeinek száma:

```
>> sum(a<0)
ans =

     2
```

Keresés tömbökben

Legyen `a=[1,-2,-1,0,5,4]`

- Soroljuk fel az a negatív elemeit!
  Ha az `a<0` logikai vektort beírjuk az a vektor indexargumentumába, akkor pontosan az 1-eseknek megfelelő helyen álló elemeket, azaz az a vektor negatív elemeit sorolja fel:

```
>> a(a<0)
ans =

    -2    -1
```

- Adjuk össze az a negatív elemeit!
  Az előbb eredményül kapott vektor elemeit kell csak összeadnunk:

```
>> sum(a(a<0))
ans =

    -3
```

Legyen

$$A = \begin{bmatrix} 1 & -2 & -1 \\ 0 & 5 & 4 \end{bmatrix}$$

- Számoljuk össze hány negatív eleme van `A`-nak!
  Az `A<0` kifejezés értéke:

```
ans =
  2×3 logical array

   0   1   1
   0   0   0
```

`sum(A<0)` vagy `sum(A<0,1)` adja a negatív elemek számát oszloponként:

```
>> sum(A<0)
ans =

     0     1     1
```

`sum(A<0,2)` adja a negatív elemek számát soronként:

```
>> sum(A<0,2)
ans =

     2
     0
```

Az összes negatív elem száma:

```
 >> sum(A<0,'all')
ans =

     2
```

Legyen

$$A = \begin{bmatrix} 1 & -2 & -1 \\ 0 & 5 & 4 \end{bmatrix}$$

- Soroljuk fel az `A` negatív elemeit!

```
>> A(A<0)
ans =

     -2
     -1
```

(Az elemeket mindig egy oszlopvektorban, oszlopfolytonosan sorolja fel.)

- Adjuk össze az `A` negatív elemeit!
  Az előbb eredményül kapott vektor elemeit kell csak összeadnunk:

```
>> sum(A(A<0))
ans =

     -3
```

Legyen `a=[1,-2,-1,0,5,4]`

- Számoljuk össze hány -1 és 2 közé eső eleme van a-nak! (A határok nélkül.)
  A `(a>-1)&(a<2)` kifejezés értéke egy logikai tömb: 1, ha egy elemre mindkét feltétel teljesül, egyébként 0:

```
ans =
  1×6 logical array

   1   0   0   1   0   0
```

  A megfelelő elemek számát megkapjuk, ha ezen utóbbi tömb elemeit összeadjuk:

```
>> sum((a>-1)&(a<2))
ans =

    2
```