Tehnici avansate

MATLAB furnizează un limbaj de nivel înalt care ne permite să gândim abstract despre matrice, vectori şi funcţii. Uneori, totuşi, trebuie să acordăm atenţie unor aspecte concrete, cum ar fi viteza de execuţie sau complexitatea programelor. În acest context câteva trucuri pot fi utile.

Mai devreme sau mai târziu, vom observa că MATLAB este uneori lent. El tinde să acorde prioritate timpului de dezvoltare a programelor în dauna timpului de execuţie. Dacă codul este prea lent şi nu vă satisfice, puteţi folosi profiler-ul pentru a detecta eventualele gâtuiri. Dacă se programează cu atenţie şi cu focalizare pe timpul de execuţie se pot obţine câştiguri importante în eficienţă. La limită, se pot scrie subrutinele consumatoare de timp în C sau Fortran şi să se lege codul compilat cu MATLAB. A se vedea în help la "external interfaces." Acest capitol sugerează câteva moduri de a obţine îmbunătăţiri substanţiale.

La fel, deși vectorii şi matricele sunt o paradigmă bună în multe situaţii, există probleme pentru care ele sunt consumatoare de resurse. Vom vedea şi alte metode de gestionare a datelor.

Cuprins

Prealocarea memoriei

MATLAB ascunde procesul complicat şi costisitor de alocare a memoriei pentru variabile. Această generozitate convenabilă poate cauza irosirea a mult timp de execuţie. Considerăm codul următor, care implementează metoda lui Euler pentru ecuaţia diferenţială vectorială $y' = Ay$ şi memorează valorile la fiecare moment de timp:

```
tic
A = rand(200);
y = ones(200,1);
dt = 0.001;
for n = 1:(1/dt)
    y(:,n+1) = y(:,n) + dt*A*y(:,n);
end
```

```
toc
```

```
Elapsed time is 0.124687 seconds.
```

Aceasta necesită aproximativ 0.109679 secunde pe un laptop fabricat în 2018. Aproape tot timpul se consumă pe o sarcină necomputaţională.

Când MATLAB întâlneşte instrucţiunea `y = ones(200,1)`, reprezentând condiţia iniţială, el cere sistemului de operare să-I aloce un bloc de memorie pentru 200 de numere în virgulă flotantă. La prima execuţie a ciclului, devine clar că avem nevoie de spaţiu pentru 400 de numere, deci va fi nevoie de un nou bloc de această dimensiune. La iteraţia următoare, dimensiunea nu mai corespunde şi este nevoie să fie alocată mai multă memorie. Acest program mic necesită 1001 de alocări de memorie individuale pentru a creşte dimensiunea!

Schimbând a doua linie în `y = ones(200,1001);` nu se schimbă nimic din matematica rezolvării problemei, ci se alocă toata memoria necesară o singură dată. Acest pas se numeşte **prealocare**. Cu prealocare programul necesită în jur de 0.092783 secunde. Pe acelaşi calculator ca mai sus viteza creşte de aproape 3 ori.

Vectorizarea

Vectorizarea se referă la evitarea ciclurilor `for` şi `while`. De exemplu, presupunem că `x` este un vector coloană şi că dorim să calculăm matricea D astfel încât $d_{ij} = x_i - x_j$. Implementarea standard necesită două cicluri imbricate:

```
x=(1:5)';
n = length(x);
D = zeros(n); % preallocation
for j = 1:n
    for i = 1:n
        D(i,j) = x(i) - x(j);
    end
end
```

Ciclurile se pot scrie aici în orice ordine. Ciclul interior poate fi înlocuit cu o operaţie vectorială:

```
n = length(x);
D = zeros(n); % preallocation
for j = 1:n
    D(:,j) = x - x(j);
end
```

Putem evita ciclul, trecând de la vectori la tablouri bidimensionale. Implementarea este un pic mai subtilă:

```
n = length(x);
X = x(:,ones(n,1)); % copy columns to make n by n
D = X - X.';
```

Cea de-a doua linie este un truc introdus în secţiunea referitoare la matrice (numit în cercurile MATLAB "Tony's trick"). De notat şi utilizarea lui .' în ultima linie pentru compatibilitate cu elemente complexe.

De ce este necesară vectorizarea? Sunt două motive: viteză şi stil. Niciunul nu este simplu. Până în jurul lui 2002, vectorizarea atentă ducea întotdeauna la îmbunătăţiri notabile de viteză. Dar, situaţia s-a schimbat într-o oarecare măsură datorată unei tehnici numite **accelerare JIT**. Accelerarea, care se aplică automat, poate înlătura penalizarea de viteză care apare traditional în ciclurile MATLAB. Nu orice ciclu poate fi optimizat, iar vectorizarea codului nu este critică pentru viteză în orice situaţie. De exemplu, pentru timpul în milisecunde pentru fiecare din cele trei metode de mai sus a fost 0.025, 0.010 şi respectiv 0.025. În acest caz, un nivel mediu de vectorizare s-a dovedit cel mai rapid.

Stilul este ceva subiectiv. Programatorilor vechi, dar care sunt noi în MATLAB, multiplicarea ciclurilor imbricate peste operaţii scalare li se pare naturală şi inevitabilă. Totuşi, MATLAB face uşor accesul la vectori şi matrice şi nu la elementele lor individuale, privite ca obiecte atomice. Superioritatea lui `A*B` faţă de un ciclu imbricat triplu pentru a calcula un produs de matrice este evidentă; multe probleme par să fie situate între aceste două extreme. În codurile de mai sus, de exemplu, fiecare versiune este mai scurtă de cât cea de dinaintea ei. În timp ce a doua nu este mai puţin clară decât prima, versiunea a treia ne cere să ne gândim un pic când o întâlnim prima dată. Spre deosebire de primele două versiuni, a treia nu este uşor de ajustat pentru a ţine cont de antisimetria evidentă din rezultat.

O altă ilustrare clasică a competiţiei implicată de vectorizare provine din eliminarea gaussiană. Iată implementarea clasică, fără nici o vectorizare:

```
n = length(A);
for k = 1:n-1
    for i = k+1:n
        s = A(i,k)/A(k,k);
        for j = k:n
            A(i,j) = A(i,j) - s*A(k,j);
        end
    end
end
```

Tabela 5. Timpii UC pentru EG cu trei niveluri de vectorizare

n	3 cicluri	2 cicluri	1 ciclu
100	0.0010	0.0166	0.0015
200	0.0073	0.0913	0.0111
300	0.0230	0.2266	0.0285
400	0.0565	0.4988	0.0844
500	0.1132	0.9127	0.2436
600	0.2810	1.3732	0.4510
700	0.3088	2.0859	0.7626
800	0.4787	3.1018	1.2415
900	0.6649	4.2185	2.1226
1000	1.8932	6.3728	2.6463

Din nou, putem începe vectorizarea cu ciclul cel mai interior, în `j`. Fiecare iteraţie a acestui ciclu este independentă de toate celelalte. Acest paralelism este o indicaţie serioasă că putem utiliza în loc o operaţie vectorială:

```
n = length(A);
for k = 1:n-1
    for i = k+1:n
        s = A(i,k)/A(k,k);
        cols = k:n;
        A(i,cols) = A(i,cols) - s*A(k,cols);
    end
end
```

Noua versiune face ideea algoritmică de operaţie pe linii mai clară şi mai de preferat. Totuşi, partea cea mai interioară a ciclului rămas este de asemenea vectorizabilă:

```
n = length(A);
for k = 1:n-1
    rows = k+1:n;
    cols = k:n;
    s = A(rows,k)/A(k,k);
    A(rows,cols) = A(rows,cols) - s*A(k,cols);
end
```

Trebuie să vă încordaţi un pic muşchii de algebra liniară pentru a vedea că produsul exterior vectorial din următoarea până la ultima linie este adecvat. Această observaţie este clarificatoare şi lămuritoare, dar nu conduce la o îmbunătăţire incontestabilă a stilului.

Pentru a compara viteza acestor trei versiuni, fiecare a fost rulată de 20 de ori pentru diferite valori ale dimensiunii `n` a matricei pe un laptop în MATLAB 2019a. Rezultatele în milisecunde pe factorizare se dau în tabela 6.1. Rezultatele acestui experiment argumentează impotriva unei recomandări uniforme a vectorizării bazate pe viteza de execuţie! Experimentul indică neajunsurile modului classic de gândire despre performanţa algoritmilor doar în termeni de număr de flops: nici una din liniile de mai sus nu este reprezentativă pentru complexitatea de $O\left(n^3\right)$.

O aplicaţie necontroversată a vectorizării este utilizarea operaţiilor pe tablouri şi a funcţiilor utilitare la evaluarea unei expresii matematice. În comparaţiile următoare, este dificil să susţinem că versiunile cu cicluri sunt superioare corespondentelor vectorizate, odată ce înţelegeţi funcţiile de utilizat:

```
y = t.*sin(t.^2);                y = zeros(size(t));
                                 for i = 1:length(t)
                                    y(i) = t(i)*sin(t(i)^2);
                                 end
%-------------------------------------------------------------
dz = diff( z([1:end 1]) );       dz = zeros(size(z));
                                 for j = 1:length(z)-1
                                    dz(j) = z(j+1)-z(j);
                                 end
                                 dz(end) = z(end)-z(1);
%-------------------------------------------------------------
e = 1+sum(1./cumprod(1:20));     e = 1; p = 1;
                                 for j = 1:20,
                                    p = p*j;
                                    e = e + 1/p;
                                 end
```

Concluzia este că ciclurile nu trebuie programate neglijent. Scrierea de cod care lucrează la nivel vectorial este uşoară şi naturală în cele mai multe cazuri. Dar dacă profiler-ul indică că se consumă mult timp într-un loc unde nivelul de vectorizare este selectabil, experimentarea este singurul mod de a vedea care nivel este cel mai bun. Pentru introducere în tipuri mai sofisticate de vectorizare, a se vedea help-urile online pentru `accumarray`, `arrayfun` şi `bsxfun`.

Mascare (Masking)

Un tip special de vectorizare este **mascarea**. Să considerăm un vector `x` de valori în care vrem să evaluăm o funcție definită pe porţiuni

$$f(x) = \begin{cases} 1+\cos(2\pi x), & |x| \le \frac{1}{2}, \\ 0, & |x| > \frac{1}{2}. \end{cases}$$

Metoda standard cu cicluri este:

```
x=-1:0.1:1;
f = zeros(size(x));
for j = 1:length(x)
    if abs(x(j)) <= 0.5
        f(j) = 1 + cos(2*pi*x(j));
    end
end
```

Modul mai scurt utilizează o mască:

```
f = zeros(size(x));
mask = (abs(x) < 0.5);
f(mask) = 1 + cos(2*pi*x(mask));
```

Masca este un index logic al lui `x` (vezi cursul referitor la matrice). Vă puteţi referi, la nevoie, la punctele nemascate utilizând `~mask` .

Considerăm o nouă versiune a sumei de numere prime de la pagina??. Iată cum putem număra numerele prime mai mici ca 100 şi cum le putem însuma:

```
isprm = isprime(1:100); % which are prime?
sum( isprm ) % how many primes
```

```
ans = 25
```

```
sum( find(isprm) ) % sum the primes
```

```
ans = 1060
```

Aici `find` converteşte un index logic într-unul absolut, i.e., furnizează un vector de numere prime.

Excepţii de domeniu

Este important să cunoștem regulile de domeniu și de vizibilitate când scriem cod format din mai multe funcții.

Cea mai puțin utilă și potențial cea mai vătămătoare violare a regulilor de domeniu provine de la **variabilele globale**. Orice variabilă poate fi declarată globală înainte de a i se atribui pentru prima dată o valoare în domeniul ei curent. După aceasta, orice alt spațiu de lucru poate declara variabila ca fiind globală și să-i acceseze sau să-i schimbe valoarea. Necazul cu variabilele globale este că nu se potrivesc cu proiectele mari și nici cu cele de mărime moderată; tind să apară rapid conflicte de nume și comportări neașteptate. În anumite momente valorile globale au fost mai mult sau mai puțin necesare și pe mai pot găsi încă exemple de coduri care le utilizează. La ora actuală există mecanisme

mai dezirabile și mai stabile. În particular, variabilele globale nu se vor utiliza pentru a transmite parametrii între funcții.

O excepție interesantă la regulile de domeniu o constituie variabila **persistentă**. Când o funcție declară o variabilă persistentă, valoarea variabilei este păstrată între apelurile acelei funcții particulare. În timp ce acest mecanism necesită puțină atenție pentru a asigura o utilizare corectă, este mult mai puțin general decât o variabilă globală, deoarece variabila persistentă este vizibilă doar în spațiul de lucru al funcției care o declară.

Variabilele persistente se pot utiliza pentru a înregistra informații despre starea internă a unei funcții, sau pentru a păstra rezultate preliminare costisitoare, care ar putea fi reutilizate ulterior. Considerăm exemplul care calculează numerele lui Fibonacci:

```
function y = fib(n)
persistent f
if length(f) < 2, f = [1 1]; end
for k = length(f)+1:n
   f(k) = f(k-2) + f(k-1);
end
y = f(1:n);
```

La primul apel al funcției, `f` va fi vid. Spre deosebire de alte variabile, variabilelor persistente li se atribuie o valoare inițială—concret, matricea nulă. Astfel, lui `f` i se vor atribui primele două valori din șir. După aceasta, `f` se extinde după cum este nevoie pentru a calcula primele n valori, iar șirul este returnat apelantului. La viitoarele apeluri ale lui `fib`, orice valoare calculată anterior este mai degrabă accesată decât calculată. Dacă funcția ar fi fost apelată cu secvența de lungime $n_1, n_2, \ldots, n_k$, efortul de calcul este proporțional cu $\max n_i$ nu cu $\sum n_i$. Același efect se poate realiza punând `f` ca argument de intrare, dar care face apelul de funcție reponsabil cu tratarea de date relevante doar în interiorul lui `fib`. Abordarea de aici este autoconținută.

Șiruri de caractere

Un șir de caractere MATLAB este delimitat între apostrofuri. Dacă apostroful este conținut în șir el trebuie dublat, ca în 'It"s Cleve"s fault'. Un șir de caractere este de fapt un vector linie de coduri de caractere și astfel șirurile de caractere pot fi concatenate utilizând sintaxa matricială:

```
str = 'Hello world';
str(1:5)
```

```
ans = 'Hello'
```

```
aa=double(str)
```

```
aa = 1x11
    72   101   108   108   111    32   119   111   114   108   100
```

```
char(aa)
```

```
ans = 'Hello world'
```

```
['Hello',' ','world']
```

```
ans = 'Hello world'
```

```
['Hello';'world']
```

```
ans =
    'Hello'
    'world'
```

Concatenarea verticală nu este atât de directă, deoarece toate liniile unui tablou trebuie să aibă aceeași lungime. Se poate utiliza `char` în locul completării șirurilor cu spații înainte de concatenare. Dacă, totuși, doriți să creați colecții de șiruri, atunci un mecanism mai bun este tabloul de celule. Sunt disponibile multe funcții pentru tratarea șirurilor de caractere; a se vedea help-ul pe `strfun`. Iată câteva exemple:

```
upper(str)
```

```
ans = 'HELLO WORLD'
```

```
strcmp(str,'Hello world')
```

```
ans =
   1
```

```
strfind(str,'world')
```

```
ans = 7
```

De un interes particular este conversia între numere și reprezentarea lor prin caractere. Se poate converti un șir, cum ar fi '3.14' în numărul corespunzător cu `str2num` sau `str2double` . De notat, diferența importantă între `double`, care convertește caracterele în codurile lor și `str2double`, care interpretează șirul ca un număr (îl convertește într-un număr)!

Pentru operația inversă de conversie a unui număr într-un șir de caractere, avem funcțiile `num2str` și `sprintf`. Ambele acceptă șiruri de conversie în stilul din C cu specificatori de conversie (%d, %f, etc.), dar din nefericire fără estensii pentru numere complexe. În `num2str`, se aplică un singur specificator de conversie tuturor elementelor unui tablou, în timp ce `sprintf` este mai flexibil, putând aplica specificatori de conversie multiplii, în mod ciclic, asupra datelor, procedând în ordinea uzuală pe linii la prelucrarea tablourilor. Pentru ieșiri pe ecran sau într-un fișier se utilizează `fprintf` .

O utilizare comună a funcției `sprintf` este de a crea șiruri, cum ar fi nume de fișiere, care conțin în ele întregi consecutivi, ca în exemplul:

```
for n = 1:12
   fn = sprintf('mydata_%.2i',n);
   load(fn), data(:,:,n) = A;
end
```

Acesta va încărca fișierele de date cu numele `mydata_01.mat` , ..., `mydata_12.mat`, memorând o matrice numită `A` ca nivele ale unui tablou tridimensional. De notat că

```
load mydata_01
load('mydata_01')
         |
```

sunt funcțional identice, dar al doilea mod este mai potrivit când numele de fișier este memorat într-o variabilă. Această „dualitate comandă/funcție" are loc pentru orice comandă MATLAB care admite argumente suplimentare separate prin spații. O altă utilizare comună este la formatarea unor tabele la ieșire. Script-ul următor afișează aproximațiile Taylor succesive ale lui $e^{1/4}$:

```
clear
T=zeros(7,1);
x=0.25; n=1:6; c=1./cumprod([1 n]);
for n=1:7
    T(n)=polyval(c(n:-1:1),x);
end
fprintf('\n T_n(x)           |T_n(x)-exp(x)|\n');
fprintf('-----------------------------------\n');
```

```
fprintf('%15.12f     %8.3e\n', [T;abs(T-exp(x))] )
```

```
 T_n(x)           |T_n(x)-exp(x)|
----------------------------------
 1.000000000000     1.250e+00
 1.281250000000     1.284e+00
 1.284016927083     1.284e+00
 1.284025404188     2.840e-01
 0.034025416688     2.775e-03
 0.000171250021     8.490e-06
 0.000000351584     1.250e-08
```

Tablouri de celule

Gruparea obiectelor de tipuri și dimensiuni diferite este o necesitate frecventă în programare. De exemplu, să presupunem că dorim să tabelăm polinoamele Cebîșev $T_0 = 1$, $T_1 = x$, $T_2 = 2x^2 - 1$, $T_3 = 4x^3 - 3x$, ș.a.m.d. În MATLAB, un polinom se poate reprezenta ca un vector de coeficienți, ordonați descrescător după puterile variabilei, deci lungimea depinde de gradul polinomului. Pentru o colecție de polinoame Cebîșev consecutive, începând cu T_0, am putea utiliza un tablou triunghiular, dar acest lucru nu este nici convenabil, nici general.

Tablourile de celule sunt un mecanism de grupare a obiectelor eterogene într-o singură variabilă. Ele sunt indexate ca tablourile numerice obișnuite, dar elementele lor pot fi orice, inclusiv alte tablouri de celule. Un tablou de celule se creaza sau se indexează prin acolade {} în loc de paranteze. Tablourile de celule pot avea orice mărime și dimensiune, iar elementele lor nu trebuie să fie de aceeași mărime sau tip. Datorită generalității lor, tablourile de celule sunt doar niște containere; ele nu suportă nici un fel de operații aritmetice.

```
strc = { 'Goodbye', 'cruel', 'world' }
```

```
strc = 1x3 cell
'Goodbye'    'cruel'      'world'
```

```
strc{2}
```

```
ans = 'cruel'
```

```
T = cell(1,9);
T(1:2) = { 1, [1, 0] };
for n=2:8
    T{n+1}=[2*T{n} 0] - [0 0 T{n-1}];
end
```

```
%T
T{4}
```

```
ans = 1x4
     4     0    -3     0
```

Al doilea exemplu evidențiază o subtilitate la extragerea elementelor dintr-un tablou de celule. Referirea într-o atribuire la `T(1:2)` se referă la un subtablou extras din tabloul de celule `T`; astfel membrul drept al atribuirii este un tablou de celule 1×2 compatibil. Construcția `T{n+1}` se referă la un singur element al tabloului, în acest caz un vector. În rezumat,

```
T{4}
```

```
ans = 1x4
     4     0    -3     0
```

```
T(4)
```

```
ans =
    {[4 0 -3 0]}
```

Astfel, membrul drept al atribuirii din ciclu nu trebuie inclus între acolade. Dacă facem aceasta nu se obține o eroare la primul pas al ciclului — elementul `T{3}` va fi el însuși un tablou de celule cu un element — dar va cauza o eroare la al doilea pas, când tabloul de celule este concatenat ilegal cu un vector. Cum se va comporta referința la `T{4:5}`, care se va referi la doi vectori? Răspunsul este că MATLAB acționează ca și cum cei doi vectori se introduc in linia de comandă:

```
T{4:5}
```

```
ans = 1x4
     4     0    -3     0

ans = 1x5
     8     0    -8     0     1
```

Această sintaxă se dovedește surprinzător de utilă, în particular idiomul `T{:}`, care este interpretat ca o listă de elemente ale lui `T,` separate prin virgulă. De exemplu, utilizând tabloul de celule `str` creat anterior, obținem

```
xstrc  = char('Goodbye','cruel','world')
```

```
xstrc =
    'Goodbye'
    'cruel  '
    'world  '
```

Combinată cu `cat` (comanda pentru concatenare de tablouri), această sintaxă se poate utiliza pentru a converti celule ce conțin date de dimensiune compatibilă în tablouri numerice:

```
c = { [3 4], [5 6] };
cat(1, [1 2], c{:} )
```

```
ans = 3x2
     1     2
     3     4
     5     6
```

```
cat(2, [1 2], c{:} )
```

```
ans = 1x6
     1     2     3     4     5     6
```

```
e = {}; cat(2, [1 2], e{:} )
```

```
ans = 1x2
     1     2
```

Se observă că pentru tabloul de celule vid `e`, expresia `e{:}` nu este numai vidă, dar înghite virgula din fața ei, care altfel ar cauza o eroare de sintaxă. Operația inversă de conversie a unui tablou în unul de celule se poate realiza prin `num2cell` :

```
Tc=num2cell(1:6);
%Tc
% num2cell(1:6,2)
```

Al doilea argument al lui `num2cell` specifică care dimensiune să se „împacheteze” într-o celulă; valoarea implicită este 1.

Tabloul de celule special `varargin` se utilizează pentru a transmite argumentele opționale funcțiilor. În acest mod, se pot scrie funcții care au o comportare care depinde de numărul de argumente transmise și care poate fi ignorată. De exemplu, presupunem că dorim să scriem o funcție care returnează specificația de culoare pentru albastru, atât în modelul de culoare RGB (implicit) sau în modelul HSV:

```
function b = blue(varargin)
if nargin < 1
    varargin = {'rgb'};
end
switch(varargin{1})
    case 'rgb'
        b = [0 0 1];
    case 'hsv'
        b = [2/3 1 1];
    otherwise
        error('Unrecognized color model.')
 end
```

Atât `blue` cât și `blue('rgb')` returnează prima opțiune, în timp ce `blue('hsv')` returnează cea de-a doua. Un al doilea mod de a utiliza `varargin` este când se poate da un număr variabil de parametrii de intrare. De exemplu, considerăm:

```
function s = add(s,varargin)
for n = 1:nargin-1
    s = s + varargin{n};
end
```

Aici, primul argument de intrare se atribuie numelui `s`, iar toate celelate tabloului de celule `varargin`. Cuvântul cheie `nargin` returnează numărul total de intrări, atât cele cu nume cât și cele opționale. Pentru a asigura aceeai funcționalitate și la ieșire, se poate utiliza `varargout` și `nargout` .

Structuri

Un tablou de tip structură este un tip de date care grupează date înrudite utilizând containere numite **câmpuri** (**fields**). Fiecare câmp poate conține orice tip de date. Accesul la datele dintr-un câmp se realizează cu notația punct de forma `structName.fieldName`.

Să presupunem că ținem evidența notelor studenților dintr-o grupă. Se poate crea o structură `student` astfel:

```
%student=struct()
student.name = 'Mumu';
student.homework = [10 10 7 9 10];
student.exam = [8 9];
student
```

```
student =
        name: 'Mumu'
    homework: [10 10 7 9 10]
        exam: [8 9]
```

Numele structurii este `student`. Datele sunt memorate în structură asociindu-le câmpuri (fields) cu nume, care sunt accesate utilizând notația punct.

Să adăugăm un student:

```
student(2).name = 'Ionescu';
student(2).homework = [4 6 7 3 0];
student(2).exam = [5 6];
student
```

Fields	name	homework	exam
1	'Mumu'	[10,10,7,9,10]	[8,9]
2	'Ionescu'	[4,6,7,3,0]	[5,6]

Avem acum un tablou de structuri. Tabloul poate avea orice mărime și oricâte dimensiuni. Toate elementele din tablou trebuie să aibă aceleași câmpuri, dar valorile conținute în acele câmpuri nu trebuie să fie compatibile între ele între diferitele elemente ale tabloului de structuri. De exemplu,

```
student(2).homework = 'missing'
```

Fields	name	homework	exam
1	'Mumu'	[10,10,7,9,10]	[8,9]
2	'Ionescu'	'missing'	[5,6]

este o continuare sintactic validă a instrucțiunilor de mai sus.

În paralel cu sintaxa tablourilor de celule `c{:}`, se pot utiliza tablourile și numele de câmpuri pentru a crea liste cu elementele separate prin virgulă ale tuturor elementelor tabloului. De exemplu:

```
student(2).homework = [4 6 7 3 0];
roster = {student.name}
```

```
roster = 1x2 cell
'Mumu'       'Ionescu'
```

```
hw = cat(1, student.homework )
```

```
hw = 2x5
    10    10     7     9    10
     4     6     7     3     0
```

În plus, ne putem referi la câmpuri al căror nume este memorat sub formă de șiruri de caractere:

```
score = 'exam';
cat(1, student.(score) )
```

```
ans = 2x2
     8     9
     5     6
```

Tabele (Tables)

O structura de tip `table` memorează date de tip tabelar sau orientate pe coloană - fiecare dată de tip coloană într-o variabilă. (Ca exemple am putea da coloanele dintr-un fișier text sau o foaie de lucru (Excel). Variabilele dintr-o tabelă pot avea tipuri diferite, dar toate variabilele trebuie să aibă același număr de linii. Variabilele din tabele au nume, la fel cum au câmpurile dintr-o structură. Funcția `summary` ne permite să obținem informații despre o tabelă.

Pentru a indexa într-o tabelă, se utilizează paranteze rotunde () pentru a returna o subtabelă sau acolade {} pentru a extrage conținutul. Variabilele și liniile se pot accesa utilizând numele lor.

Creăm o tabelă cu date despre pacienți. Generăm variabilele:

```
LastName = {'Sanchez';'Johnson';'Li';'Diaz';'Brown'};
Age = [38;43;38;40;49];
Smoker = logical([1;0;1;0;1]);
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];
```

Creăm o tabelă, `T`, privită ca un container pentru unele variabile din spațiul de lucru. Funcția table utilizează numele de variabile din spațiul de lucru ca nume de variabile din tabela `T`. O variabilă dintr-o tabelă poate avea mai multe coloane. De exemplu, `BloodPressure` din `T` este un tablou 5×2.

```
T = table(LastName,Age,Smoker,Height,Weight,BloodPressure)
```

	LastName	Age	Smoker	Height	Weight	BloodPressure	
1	'Sanchez'	38	1	71	176	124	93
2	'Johnson'	43	0	69	163	109	77
3	'Li'	38	1	64	131	125	83
4	'Diaz'	40	0	67	133	117	75
5	'Brown'	49	1	64	119	122	80

Se poate utiliza notația punct "." pentru a aceesa variabilele. De exemplu înălțimea media a pacienților se poate calcula cu:

```
meanHeight = mean(T.Height)
```

```
meanHeight = 67
```

Calculăm indicele de masă corporală (BMI) și îl adaugă la tabelă:

```
T.BMI = (T.Weight*0.453592)./(T.Height*0.0254).^2
```

	LastName	Age	Smoker	Height	Weight	BloodPressure		BMI
1	'Sanchez'	38	1	71	176	124	93	24.5467
2	'Johnson'	43	0	69	163	109	77	24.0706
3	'Li'	38	1	64	131	125	83	22.4858
4	'Diaz'	40	0	67	133	117	75	20.8305
5	'Brown'	49	1	64	119	122	80	20.4261

Adnotăm tabela cu o descriere a modului de calcul al `BMI`. Aceasta se poate face utilizând metadatele prin intermediul lui `T.Properties`.

```
T.Properties.Description = 'Patient data, including body mass index (BMI) calculated using
T.Properties
```

```
ans =
  TableProperties with properties:

            Description: 'Patient data, including body mass index (BMI) calculated
                  using Height and Weight'
                UserData: []
          DimensionNames: {'Row'  'Variables'}
           VariableNames: {'LastName'  'Age'  'Smoker'  'Height'  'Weight'
                'BloodPressure'  'BMI'}
     VariableDescriptions: {}
            VariableUnits: {}
      VariableContinuity: []
                RowNames: {}
```

```
CustomProperties: No custom properties are set.
Use addprop and rmprop to modify CustomProperties.
```

O întreagă tabela se poate accesa ca o matrice utilizând numele celei de-a doua dimensiuni ca variabilă. Creăm o tabelă cu date despre pacienți cu 5 linii.

```
Age = [38;43;38;40;49];
Smoker = logical([1;0;1;0;1]);
Height = [71;69;64;67;64];
Weight = [176;163;131;133;119];
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];

T2 = table(Age,Smoker,Height,Weight,BloodPressure)
```

	Age	Smoker	Height	Weight	BloodPressure	
1	38	1	71	176	124	93
2	43	0	69	163	109	77
3	38	1	64	131	125	83
4	40	0	67	133	117	75
5	49	1	64	119	122	80

Numele dimensiunilor tabelei se pot accesa prin intermediul proprietății DimensionNames. Numele implicit al celei de-a doua dimensiuni este `Variables`.

```
T2.Properties.DimensionNames
```

```
ans = 1x2 cell
'Row'        'Variables'
```

Datele din tabelă pot fi accesate ca o singură matrice utilizând sintaxa `T.Variables`. Ea este echivalenta cu `T{:,:}`.

```
T2.Variables
```

```
ans = 5x6
    38     1    71   176   124    93
    43     0    69   163   109    77
    38     1    64   131   125    83
    40     0    67   133   117    75
    49     1    64   119   122    80
```

Redenumim cea de-a doua dimensiune. Noul nume se poate utiliza pentru a accesa datele.

```
T2.Properties.DimensionNames{2} = 'PatientData';
T2.PatientData
```

```
ans = 5x6
    38     1    71   176   124    93
    43     0    69   163   109    77
    38     1    64   131   125    83
    40     0    67   133   117    75
    49     1    64   119   122    80
```

Dacă încercăm să accesăm T în același mod obținem o eroare de sintaxă

```
T.Variables
```

```
Error using  .
Unable to concatenate the table variables 'LastName' and 'Age',
because their types are cell and double.
```

Structura este complexă și există o gamă largă de funcții de acces. Pentru detalii a se vedea `doc table`.