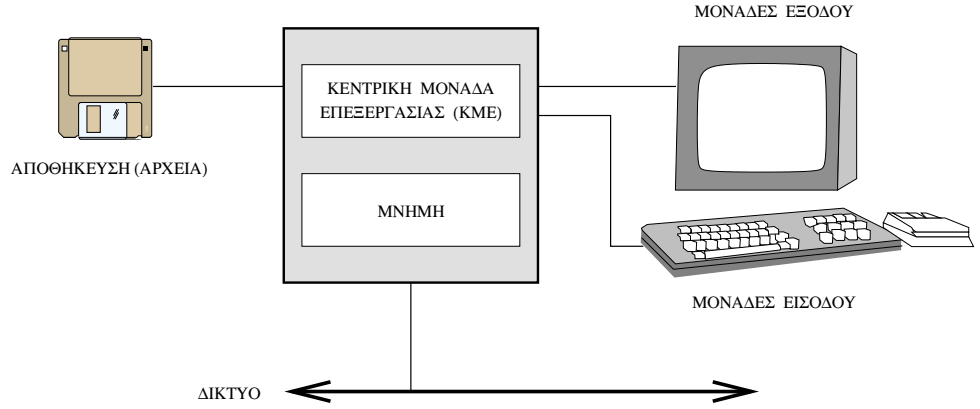
Εισαγωγή

141. Πληροφορική Ι

- Εισαγωγικές έννοιες αρχιτεκτονικής και αριθμητικής υπολογιστών.
- Αλγοριθμική επίλυση προβλημάτων.
- Βασικές δομές και αποτελεσματικότητα αλγορίθμων.
- Προγραμματισμός με Matlab / Octave.
- Εφαρμογές σε μαθηματικά, και όχι μόνο, προβλήματα.

Τα βασικά μέρη ενός υπολογιστή



Η επιστήμη των υπολογιστών

Ο υπολογιστής είναι *εργαλείο* επίλυσης προβλημάτων λόγω:

- *ταχύτητας* υπολογισμού και μεγέθους *μνήμης*·
- γενικής χρησιμότητας μέσω της έννοιας του *προγραμματισμού*.
  Ενα *πρόγραμμα*:
  - Είναι μια αλληλουχία κατάλληλων οδηγιών (εντολών) που εκτελεί ο υπολογιστής για την επίλυση κάποιου προβλήματος. Η λειτουργία του υπολογιστή επομένως, *προσαρμόζεται* στο επιλυόμενο πρόβλημα.
  - Υλοποιεί έναν *αλγόριθμο*.

Αλγόριθμος

Μαθηματική μεθοδολογία υπολογιστικής επίλυσης κάποιου προβλήματος. Κάθε αλγόριθμος ικανοποιεί τα παρακάτω κριτήρια:

- Είσοδος (≥ 0 εξωτερικά δεδομένα).
- Εξοδος (≥ 1 αποτέλεσμα/τα).
- Ορισμένος (περιέχει σαφείς και ακριβείς οδηγίες).
- Τερματισμός (μετά από πεπερασμένο αριθμό βημάτων ή πεπερασμένο υπολογιστικό χρόνο).
- Κάθε οδηγία, μεμονωμένα, είναι εξαιρετικά απλή.

Βήματα για την υπολογιστική επίλυση προβλήματος

1. Ανάλυση δεδομένων του προβλήματος.
2. Μαθηματική διατύπωση του προβλήματος.
3. Ανάπτυξη τεχνικών επίλυσης: *αλγόριθμος*. *Σχεδιασμός* ή *επιλογή* αλγορίθμου. Συνήθως: ένα πρόβλημα—πολλοί αλγόριθμοι.
4. Διατύπωση του αλγορίθμου σε γλώσσα προγραμματισμού: *πρόγραμμα*.
5. Εκτέλεση προγράμματος για συγκεκριμένα δεδομένα.
6. Ερμηνεία αποτελεσμάτων.

Βασικές εντολές

Αλγόριθμοι και Matlab

Οι 5 βασικές αλγοριθμικές ενέργειες και οι αντίστοιχες εντολές Matlab:

είσοδος δεδομένων	**input**
έξοδος αποτελεσμάτων	**disp**
πράξεις και αναθέσεις τιμών σε μεταβλητές	+ − ∗ / =
έλεγχος ποσοτήτων - επιλογή ανάλογης δράσης	**if** -**else**
επαναληπτική εκτέλεση	**while**, **for**

Κάθε αλγόριθμος περιγράφεται *τελικά* με κάποιες από τις παραπάνω οδηγίες και μόνο αυτές.

Τα βήματα ενός αλγορίθμου/προγράμματος εκτελούνται *σειριακά* από πάνω προς τα κάτω.

Ενας αλγόριθμος τερματίζει σε *πεπερασμένο* αριθμό βημάτων.

Μετατροπή βαθμών Fahrenheit σε βαθμούς Celsius

Fahrenheit σε Celsius: $c = (5/9)(f - 32)$

```
% Ενα πρώτο πρόγραμμα
f = 451;              % Τι συμβαίνει στους 451 βαθμούς F
c = (f − 32)∗5/9;
disp(c);
```

Τα *σχόλια* εισάγονται με το χαρακτήρα **%** και αναπτύσσονται μέχρι το τέλος της γραμμής. Δίνουν επεξηγήσεις στον αναγνώστη του προγράμματος. Αγνοούνται κατά την εκτέλεση.

Σταθερές: 451, 32, 5, 9

Μεταβλητές: f, c

Πράξεις με όρους σταθερές ή μεταβλητές: $- * /$

Ανάθεση τιμών σε μεταβλητές: =

Εξοδος αποτελεσμάτων: **disp**

Τερματισμός εντολών με το χαρακτήρα ';'

Αριθμητικές σταθερές

- Χρησιμοποιούνται για αριθμητικές ποσότητες που δεν αλλάζουν

  451 32 −5 +9.2 2.7183 0 1.0 0.

  αλλά και η *ειδική σταθερά* **pi** για το π.

- Σε *εκθετική μορφή* συνήθως για πολύ μικρούς ή πολύ μεγάλους αριθμούς (ο αριθμός $a \times 10^{\beta}$ γράφεται *a*e*β* ή *a*E*β*)

  −6.023e+23 1.7E−308 9e−2 1.e10

Μεταβλητές

- Παραμετρική διατύπωση αλγορίθμων/προγραμμάτων.
- Αναφορά σε τιμές που *μεταβάλλονται* σ' ένα πρόγραμμα, ή τιμές που δεν είναι γνωστές πριν την εκτέλεση (σε αντίθεση με τις *σταθερές*).
- Για να χρησιμοποιηθεί μια μεταβλητή στο πρόγραμμα πρέπει να έχει πάρει κάποια τιμή.
- Επιτρεπτά ονόματα μεταβλητών είναι συνδυασμοί *λατινικών* γραμμάτων, αριθμών και του χαρακτήρα '_'. Ο πρώτος χαρακτήρας όμως δεν μπορεί να είναι αριθμός! Παραδείγματα:

  ΝΑΙ: f c r2d2 mesos_oros

  ΟΧΙ: 456 **disp** 3po mesos-oros

- Σ' ένα πρόγραμμα οι μεταβλητές είναι *θέσεις μνήμης*.

Εκφράσεις

Οι εκφράσεις έχουν *τιμές* και συνήθως *εκφράζουν* κάποιο υπολογισμό.

Μια *μεταβλητή* από μόνη της αποτελεί έκφραση[α']: f.

Μια *σταθερά* από μόνη της αποτελεί έκφραση: 3.14

Είναι γενικά συνδυασμοί μεταβλητών, σταθερών και τελεστών[β']

```
area = 3.14 * radius * radius;
```

Μερικοί αριθμητικοί τελεστές:

μονομελείς: π.χ. πρόσημα + και −

διμελείς: πρόσθεση +, αφαίρεση −, πολ/σμός ∗, διαίρεση /, κ.α.

[α'] με την προυπόθεση να έχει αρχικοποιηθεί

[β'] και κλήσεις συναρτήσεων, όπως θα δούμε αργότερα.

Υπολογισμός εκφράσεων - σειρά υπολογισμού

Ανάγκη για ακριβείς κανόνες που καθορίζουν ακριβώς τη σημασία μια έκφρασης: Η *προτεραιότητα* υπολογισμών καθορίζει τη σειρά εκτέλεσης των τελεστών σε μια έκφραση.

Η έκφραση $\boxed{a+b*a-b}$	4 + 3 * 4 - 3
ισούται με $\boxed{(a+b)*(a-b)}$	(4 + 3) * (4 - 3) = 7
ή με $\boxed{a+(b*a)-b}$??	4 + (3*4) - 3 = 13

Προτεραιότητα τελεστών:

1. πρώτα οι παρενθέσεις (), αρχίζοντας από τις πιο εσωτερικές
2. μετά οι 'πολλαπλασιασμοί': $*, /$
3. τέλος οι 'προσθέσεις': $+, -$

Υπολογισμός εκφράσεων - συσχετισμός τελεστών

Η προτεραιότητα δεν ξεκαθαρίζει την κατάσταση όταν όλοι οι τελεστές είναι στο ίδιο επίπεδο προτεραιότητας.

Η έκφραση $\boxed{a/b*c}$ ισούται με $\boxed{a/(b*c)}$ ή με $\boxed{(a/b)*c}$??

Οι *κανόνες συσχετισμού* καθορίζουν τη σειρά μεταξύ συνεχόμενων τελεστών της ίδιας προτεραιότητας. Υπάρχει διαφορά? $16/4*2$

Κανόνες συσχετισμού:

Οι περισσότεροι αριθμητικοί τελεστές στο Matlab, υπολογίζονται *από αριστερά προς τα δεξιά*, για το ίδιο επίπεδο προτεραιότητας:

Η έκφραση $\boxed{a/b*c}$ ισούται με $\boxed{(a/b)*c)}$

Η έκφραση $\boxed{a+b-c+d}$ ισούται με $\boxed{((a+b)-c)+d}$

Εντολές ανάθεσης

Οι μεταβλητές παίρνουν συνήθως τιμές με τις *εντολές ανάθεσης*

μεταβλητή = έκφραση

Η εκτέλεση μιας εντολής ανάθεσης υλοποιείται σε δυο διακριτά βήματα

1. Υπολογισμός της έκφρασης στο δεξί μέλος.
2. Αποθήκευση της *τιμής* της έκφρασης στη μεταβλητή που καθορίζεται αριστερά του τελεστή ανάθεσης =.

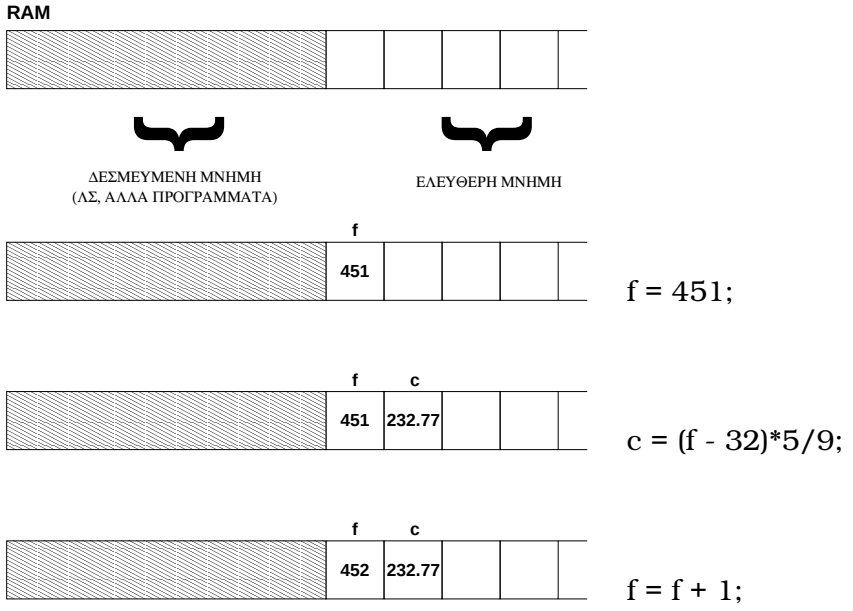
Επομένως η

```
x = x + 1;
```

εντολή ανάθεσης στο Matlab, και όχι μια αλγεβρική εξίσωση!

Μια μεταβλητή μπορεί να έχει μόνο μία τιμή· όταν η μεταβλητή πάρει κάποια νέα τιμή με εντολή ανάθεσης, η προηγούμενη τιμή της χάνεται.

Η εικόνα της μνήμης



Εξοδος

- Με την εντολή **disp**() που έχει 2 παραλλαγές:

  disp(*έκφραση*)
  disp(*'μήνυμα'*)

  Για εκτύπωση:
  - της τιμής της μεταβλητής x: **disp**(x)
  - της τιμής της έκφρασης $1 + 1$: **disp**(1+1)
  - του μηνύματος «Matlab is cool!»: **disp**('Matlab is cool')!
  - του αριθμου 13: **disp**(13) αλλά και **disp**('13')

  ΛΑΘΟΣ: **disp**(f , c) (μόνο ένα όρισμα).

Εξοδος (συνεχ.)

- Αν δεν τερματίσουμε μιά εντολή με ';', τότε εκτυπώνεται η τίμη της έκφρασης της εντολής. Π.χ.

```
f = 451              % Εκτυπώνεται ο αριθμός 451
c = (f − 32)∗5/9 % Εκτυπώνεται η τιμή της c
c                    % Εκτυπώνεται η τιμή της c
f , c                % Εκτυπώνεται η τιμή των f , c σε 2 γραμμές
```

- Με την εντολή **fprintf** (), για πλήρη έλεγχο της μορφής της εκτύπωσης.

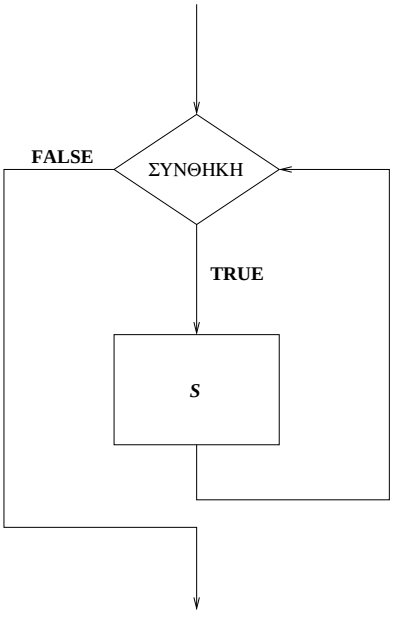
Επανάληψη υπό συνθήκη

Μετατροπή βαθμών Fahrenheit σε Celsius για $0^oF, 1^oF, 2^0F, \ldots, 212^oF$

```
% Μεταβλητές
% f — βαθμοί Fahrenheit
% c — βαθμοί Celsius
f = 0;
c = (f − 32)∗5/9;
disp(f); disp(c);
f = 1;
c = (f − 32)∗5/9;
disp(f); disp(c);
 ...
f = 212;
c = (f − 32)∗5/9;
disp(f); disp(c);
```

```
% Μεταβλητές
% f — βαθμοί Fahrenheit
% c — βαθμοί Celsius
f = 0;
while f <= 212,
   c = (f − 32)∗5/9;
   disp(f); disp(c);
   f = f + 1;
end
```


Η εντολή while



while *λογική συνθήκη* ,
εντολές S
end

Η εκτέλεση των εντολών *S* *επαναλαμβάνεται* όσο αληθεύει η ΛΣ (λογική συνθήκη). Η τιμή (αληθής ή ψευδής) της ΛΣ πρέπει:

- να να μπορεί να υπολογιστεί πριν την εκτέλεση της **while** (δηλ. η ΛΣ να είναι ορισμένη)
- να μεταβάλλεται με κάποια από τις εντολές *S*, διαφορετικά η αληθεια της ΛΣ δεν θα αλλάξει.

Λογικές (boolean) εκφράσεις

Στην απλούστερη μορφή τους συγκρίσεις, που σχηματίζονται από:

- μεταβλητές,
- σταθερές,
- αριθμητικές εκφράσεις, και
- *τελεστές συσχέτισης*:

Στα μαθηματικά:	$<$	$\leq$	$>$	$\geq$	$=$	$\neq$
Στο Matlab:	<	<=	>	>=	==	~=

Παραδείγματα:

```
air_temperature > 30.0
25 <= sea_temperature
f <= 212
divisor ~= 0
```


Τιμή λογικών εκφράσεων

Κάθε έκφραση έχει μια τιμή. Ποια είναι η τιμή μιας λογικής έκφρασης?

Απάντηση: την θεωρούμε ΑΛΗΘΗ ή ΨΕΥΔΗ.

Στο Matlab, όμως, είναι ένας ακέραιος.

- ΨΕΥΔΗΣ είναι 0 (και 0 είναι ΨΕΥΔΗΣ)
- ΑΛΗΘΗΣ είναι 1 (αλλά και οποιαδήποτε μη μηδενική τιμή).
- Αλλά το αποτέλεσμα μιας αληθούς έκφρασης ισούται με 1 (μονάδα)

```
disp (4 < 7);  % εκτυπώνει 1
```

Είσοδος δεδομένων

Μετατροπή 7 θερμοκρασιών C σε βαθμούς F[α΄] (με **while**). Χρειάζεται:

- επανάληψη του υπολογισμού 7 φορές: χρήση μεταβλητής μετρητή.
- μηχανισμός (εντολή) εισόδου δεδομένων: **input**.

```
% Μετατρέπει 7 θερμοκρασίες C σε βαθμούς F
% f, c — θερμοκρασίες F, C
% m   — μετρητής
m = 1;
while m <=7,
    c = input('θερμοκρασία Celsius ? ');
    f = (9/5)*c + 32;
    disp('ισοδύναμη θερμοκρασία Fahrenheit '); disp(f);
    m = m + 1;
end
```

[α΄] $f = (9/5)c + 32$

Η εντολή input

μεταβλητή = **input**(*'μήνυμα '*);

Κατά την εκτέλεση της, η **input**:

- Εμφανίζει το *μήνυμα* στην οθόνη και περιμένει από το χρήστη να πληκτορολογήσει ένα αριθμό, έστω x.
- Η *μεταβλητή* παίρνει την τιμή x.

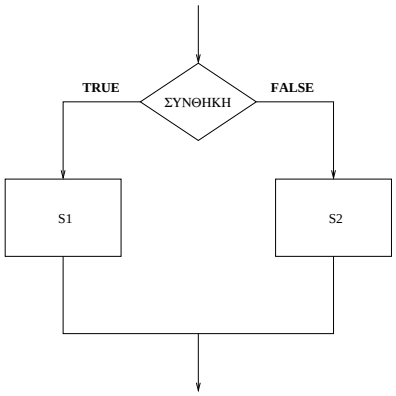
Με κάθε εντολή **input** δίνουμε τιμή σε *μία μόνο* μεταβλητή.

Επιλογή εναλακτικής πορείας εκτέλεσης

Έλεγχος n θερμοκρασιών F για το αν κυμαίνονται μεταξύ ανεκτού εύρους βαθμών C:

```
clo = input('Ελάχιστη ανεκτή θερμοκρασία Celsius ? ');
chi = input('Μέγιστη ανεκτή θερμοκρασία Celsius ? ');
flo = 9*clo /5 + 32;    fhi = 9*chi /5 + 32;
n = input('Δώσε το πλήθος των μετρήσεων ? ');
m = 1;
while m <= n,
    f = input('Δώσε θερμοκρασία Fahrenheit ? ');
    if f>=flo & f<=fhi,
        disp('ΑΝΕΚΤΗ ΘΕΡΜΟΚΡΑΣΙΑ ');
    else
        disp('ΜΗ ΑΝΕΚΤΗ ΘΕΡΜΟΚΡΑΣΙΑ ');
    end
    m = m + 1;
end
```


Η εντολή if




```
if λογική συνθήκη,
    εντολές S1
else,
    εντολές S2
end
```

Οι *εντολές S1* εκτελούνται όταν αληθεύει η ΛΣ (λογική συνθήκη). Στην αντίθετη περίπτωση εκτελούνται οι *εντολές S2*. Οταν δεν υπάρχει κλάδος *S2*, παίρνει τη μορφή:

```
if λογική συνθήκη,
    εντολές S
end
```

Λογικοί (boolean) τελεστές

Σύζευξη (και) με τελεστή & (διμελής)

Αν η θερμοκρασία είναι κάτω απο 0^oC και βρέχει, τότε χιονίζει.

Διάζευξη (ή) με τελεστή | (διμελής)

Αν η ομάδα μου τερματισει στις πρώτες θέσεις του πρωταθλήματος ή κατακτήσει το κύπελλο, τότε θ' αγωνιστεί του χρόνου στην Ευρώπη.

΄Αρνηση (δεν) με τελεστή ~ (μονομελής)

Αν δεν είναι Σάββατο ή Κυριακή, τότε είναι εργάσιμη μέρα.

Πίνακας αλήθειας:

p	q	~p	p & q	p | q
1	1	0	1	1
1	0		0	1
0	1	1	0	1
0	0		0	0

Σύνθετες λογικές εκφράσεις

```
f>=flo & f<=fhi
~(2>3)
~2>3
8*2 - 1 >= 15 | 8/2*2 == 8
```

Προτεραιότητα τελεστών:

- πρώτα οι *αριθμητικοί* τελεστές,
- μετά οι τελεστές *συσχέτισης* και η άρνηση
- και τέλος οι διμελείς *λογικοί* τελεστές.

Προσοχή, στον υπολογισμό εκφράσεων της μορφής: $a \odot \chi \odot \beta$
όπου $\odot$ κάποιος από τους τελεστές συσχέτισης (π.χ. $1 < 5 < 3$)

```
x = 5;
x > 1 & x < 3        % FALSE
1 < x < 3            % TRUE
```

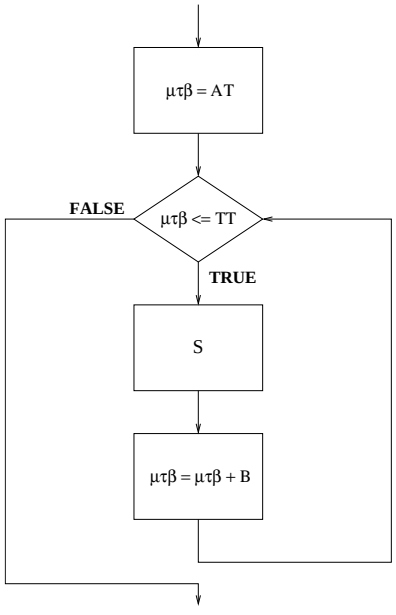
Οι εκφράσεις αυτές γράφονται πάντα σαν: $a \odot \chi \;\&\; \chi \odot \beta$

Μια άλλη εντολή επανάληψης

Μετατροπή 7 θερμοκρασιών C σε βαθμούς F (με χρήση της εντολής **for**).

```
% Μετατρέπει 7 θερμοκρασίες C σε βαθμούς F
% f , c — θερμοκρασίες F , C
% m   — μετρητής
for m = 1:7,
    c = input('θερμοκρασία Celsius ? ');
    f = (9/5)*c + 32;
    disp('ισοδύναμη θερμοκρασία Fahrenheit ');
    disp(f);
end
```


Η εντολή for




```
for μτβ = ΑΤ :Β :ΤΤ ,
    εντολές S
end
```

για τις εκφράσεις
ΑΤ: αρχική τιμή, *Β*: βήμα, *ΤΤ*: τελική τιμή.
Δεν γίνεται *καμμία* επανάληψη όταν

- ΑΤ>ΤΤ & Β>0
- ΑΤ<ΤΤ & Β<0

Για μοναδιαίο *Β*, η **for** γράφεται:

```
for μτβ = ΑΤ :ΤΤ ,
    εντολές S
end
```

Παραδείγματα for

```
for i = 1:10, disp(i); end
for j = 1:1:10, disp(j); end
for m = 2:3:10, disp(m); end
for odd = 1:2:100, disp(odd); end
for even = 0:2:100, disp(even); end
for nothing = 100:10:1, disp(nothing); end
for nada = 100:-500:100000, disp(nada); end
for countdown = 10:-1:1, disp(countdown); end
```


Σχέση for και while

```
for μτβ = ΑΤ:Β:ΤΤ,
   εντολές;
end
```

ισοδ.

```
μτβ = ΑΤ;
while μτβ <= ΤΤ,
   εντολές;
   μτβ = μτβ + Β;
end
```

Η **for** προτιμάται όταν ο αριθμός των επαναλήψεων είναι γνωστός.

Αγνωστο πλήθος επαναλήψεων

Σε πόσα χρόνια θα διπλασιαστει αποταμιευμένο κεφάλαιο 1000 €, που τοκίζεται με 5% το χρόνο?

```
my_money = 1000;
n = 0;
while my_money < 2000,
  my_money = my_money * 1.05;
  n = n + 1;
end
disp('Χρόνια για διπλασιασμό χρημάτων: ');
disp(n);
```

Προτιμάται η **while** αντί της **for**.

Χρήσιμες 'εντολές'

... για οποιεσδήποτε *εκφράσεις A* και *B*

Σύνταξη	Τι υπολογίζει	Παραδείγματα
A^*B*	ύψωση σε δύναμη	2^3, x^(2+y)
sqrt(*A*)	τετραγωνική ρίζα	**sqrt**(2)
abs(*A*)	απόλυτη τιμή	**abs**(-13.6)
floor(*A*)	στρογγυλοποίηση πρός $-\infty$	**floor**(2.3), **floor**(-2.3)
ceil(*A*)	στρογγυλοποίηση πρός $+\infty$	**ceil**(2.3), **ceil**(-2.3)
fix(*A*)	ακέραιο μέρος	**fix**(2.3), **fix**(-2.3)
rem(*A*,*B*)	*A* - *B****fix**(*A*/*B*)	**rem**(3,2), **rem**(2.2,2)

Λιγότερα κέρματα

Ελάχιστος αριθμός κερμάτων για δεδομένο χρηματικό ποσόν $< €1$

```
poson = input('Δώσε ποσόν σε λεπτά (100) ? ');
c50 = fix (poson/50);                    % Κέρματα των 50λ
poson = rem(poson, 50);
c20 = fix (poson/20);                    % Κέρματα των 20λ
poson = rem(poson, 20);
c10 = fix (poson/10);                    % Κέρματα των 10λ
poson = rem(poson, 10);
c5 = fix (poson/5);                      % Κέρματα των 5λ
poson = rem(poson, 5);
c2 = fix (poson/2);                      % Κέρματα των 2λ
c1 = rem(poson, 2);                      % Κέρματα του 1λ
disp('Συνολικός αριθμός κερμάτων: ');
disp(c50 + c20 + c10 + c5 + c2 + c1);
```


Μετατροπή cm σε ft και in

```
% 1in = 2.54cm, 1ft = 12in
cm = input('Δώσε μήκος σε cm? ');
total_inches = cm/2.54;
feet = fix (total_inches /12);
inches = total_inches − feet∗12; % ή inches = rem(total_inches , 12)
disp(' ft : ' ); disp(feet);
disp(' in : ' ); disp(inches);
```

Αλγοριθμικές τεχνικές

Αθροισμα 10 Αριθμών

1ος Τρόπος (μη αλγοριθμικός)

$$s = n_1 + n_2 + n_3 + n_4 + \cdots + n_{10}$$

Ανάγκη για 11 μεταβλητές και 10 αναγνώσεις

```
n1 = input('n1? ');
n2 = input('n2? ');
n3 = input('n3? ');
 ...
n10 = input('n10? ');
s = n1+n2+n3+n4+n5+n6+n7+n8+n9+n10;
disp(s);
```

Τι γίνεται με πρόσθεση 100 αριθμών?

$\Longleftrightarrow$

```
s = 0;
n1 = input('n1? ');
s = s + n1;
n2 = input('n2? ');
s = s + n2;
 ...
n10 = input('n10? ');
s = s + n10;
disp(s);
```

Χρειάζονται στην πράξη και οι 10 μεταβλητές n1, n2, ..., n10??

Αθροισμα 10 Αριθμών

2ος Τρόπος (αλγοριθμικός)

ΠΑΡΑΤΗΡΗΣΗ: Εφικτή η πρόσθεση με χρήση ένος μόνο προσθετέου.

Χρειάζονται 2 *μόνο* μεταβλητες: μία για τον εκάστοτε προσθετέο (value) και μία για το αθροισμα (total).

- Ο αθροιστής total αρχικά παίρνει την τιμή 0.
- Για *κάθε* προσθετέο εκτελούνται τα βήματα:
  1. Ανάγνωση μιας ακέραιας τιμής και αποθήκευση στην value
  2. Πρόσθεση της value στο τρέχον άθροισμα total

```
total = 0;
for i=1:10,
    value = input('Επόμενος όρος ? ');
    total = total + value;
end
disp( total );
```


Επεξεργασία αγνώστου πλήθους δεδομένων - I

Το πλήθος των δεδομένων δεν είναι πάντα το ίδιο. Καταμέτρηση δεδομένων.

Γενικός αλγόριθμος

```
n = input('Πλήθος δεδομένων ? ');
% Αρχική επεξεργασία
for i=1:n,
    % Επεξεργασία δεδομένου i
end
% Τελική επεξεργασία
```

π.χ. αθροισμα αριθμών

```
n = input('Πλήθος προσθετέων ? ');
sum = 0;
for i=1:n,
    value = input('Επόμενος όρος ? ');
    sum = sum + value;
end
disp(sum);
```

ΠΡΟΒΛΗΜΑ με πολύ μεγάλες λίστες αριθμών.

Επεξεργασία αγνώστου πλήθους δεδομένων - ΙΙ

ΑΝΤΙΜΕΤΩΠΙΣΗ: ανάγνωση και επεξεργασία δεδομένων μεχρι να εισαχθεί κάποια *ειδικό* δεδομένο (π.χ. 0 ή -1), που δεν είναι επεξεργάσιμο, απλώς σηματοδοτεί το τέλος των δεδομένων.

Γενικός αλγόριθμος

```
% Αρχική επεξεργασία
end_of_data = % Ειδική τιμή τέλους
% Διάβασε τον πρώτο όρο
value = input('Value? ');
while value ~= end_of_data,
    % Επεξεργασία τρέχοντος δεδομένου
    % Διάβασε το επόμενο δεδομένο
    value = input('Value? ');
end
```

π.χ. γινόμενο αριθμών

```
end_of_data = 0;
prod = 1;
value = input('Πρώτος όρος ? ');
while value ~= end_of_data,
    prod = prod*value;
    value = input('Επόμενος όρος ? ');
end
disp(prod);
```


Αθροισμα θετικών όρων λίστας

Για λίστα αγνώστου πλήθους όρων.

Χρειάζεται ο έλεγχος του προσήμου κάθε αριθμού στη λίστα.

```
end_of_data = 0;
sum = 0;
disp('Αθροισμα θετικών όρων λίστας ');
disp('Δωσε 0 για τερματισμό ');
value = input('Πρώτος όρος ? ');
while value ~= end_of_data,
    if value > 0,
        sum = sum + value;
    end
    value = input('Επόμενος όρος ? ');
end
disp(sum);
```

Αθροισμα ψηφίων ακεραίου

Για τον ακέραιο n:

- Εύρεση τελευταίου ψηφίου: **rem**(n, 10)
- Αποκοπή τελευταίου ψηφίου: **fix** (n/10)

Π.χ. για το 1985:

rem(1985,10) $\longrightarrow$ 5

fix (1985/10) $\longrightarrow$ 198

```
n = input('Δώσε θετικό ακέραιο ? ');
dsum = 0;
while n>0,
    dsum = dsum + rem(n, 10);
    n =  fix (n/10);
end
disp(dsum);
```


Ατέρμονες επαναλήψεις

Υπάρχει περίπτωση μια επαναληπτική εντολή να μην τερματίζει.

Παράδειγμα:

```
while n >= 0,
    dsum = dsum + rem(n, 10);
    n =  fix (n/10);
end
```

Οταν εξαντληθούν όλα τα ψηφία, το n θα παίρνει *διαρκώς* την τιμή 0 και η παραπάνω **while** δεν τερματίζει.

Πρώτοι Αριθμοί

Βασικό πρόβλημα στη *Θεωρία Αριθμών* είναι ο έλεγχος άν κάποιος ακέραιος είναι πρώτος.

> *Ενας ακέραιος* > 1 *είναι πρώτος αν έχει ακριβώς δύο διαιρέτες: τη μονάδα και τον εαυτό του.*

Οι πρώτοι αριθμοί σημαντικοί στην *κρυπτογραφία* (= μελέτη κωδίκων): στις ηλεκτρονικές επικοινωνίες, οι υπολογιστές χρησιμοποιούνται για κωδικοποίηση και αποκωδικοποίηση· πολλές τεχνικές κωδικοποίησης βασίζονται στη θεωρία πρώτων αριθμών

Αλγόριθμος Π1: έλεγχος πρώτων αριθμών

Είναι ο n πρώτος?

- Καταμέτρηση όλων των διαιρετών του n (χρειάζεται να υπολογισθούν). Αν είναι ακριβώς 2 τότε ο n είναι πρώτος.
- Οι πιθανοί διαιρέτες του n είναι $\leq n$, άρα αρκεί να ελεγχθεί ποιοί από τους $1, 2, \ldots, n$ είναι διαιρέτες του n.

Πρόγραμμα για τον αλγόριθμο Π1

```
% Ελέγχει αν ένας ακέραιος είναι πρώτος αριθμός
% n        — αριθμός για έλεγχο
% num_div — πλήθος διαιρετών
n = input('Δώσε ακέραιο αριθμό ? ');
num_div = 0;
for  i  = 1:n,
    if  rem(n,i ) == 0,   num_div = num_div + 1;  end
end
 if  num_div == 2,
   disp('Ο αριθμός είναι πρώτος ');
else,
   disp('Ο αριθμός είναι σύνθετος ');
end
```


Βελτίωση της αποτελεσματικότητας του Π1: αλγόριθμος Π2

- Ο Π1 δεν είναι πρακτικός για μεγάλους αριθμούς: ο 1000000 φαίνεται οτι ΔΕΝ ειναι πρώτος στον έλεγχο με το 2. Αρα ο αλγόριθμος μπορεί να τερματίζει μόλις βρεθεί διαιρέτης > 1.
- Περιτεύει η εξέταση πιθανών διαιρετών που είναι *άρτιοι* και > 2: αν ο n δεν διαιρείται από το 2, τότε δεν διαιρείται από κανένα άρτιο.
- Αριθμοί $> \sqrt{n}$ δεν χρειάζεται να ελεγχθούν σαν πιθανοί διαιρέτες: Αν $\delta_1 \neq \delta_2$ είναι δυο διαιρέτες του n τέτοιοι ώστε $\delta_1 \times \delta_2 = n$, τότε ένας από τους δ_1, δ_2 θα είναι *οπωσδήποτε* $< \sqrt{n}$ (απόδειξη με απαγωγή σε άτοπο).

Πρόγραμμα για τον αλγόριθμο Π2

```
% Ελέγχει αν ένας ακέραιος είναι πρώτος αριθμός
% n          — αριθμός για έλεγχο
% divisor    — πιθανοί διαιρέτες
% is_prime — 'λογική' μεταβλητή
%              = 1, ο αριθμός είναι πρώτος
%              = 0, ο αριθμός δεν είναι πρώτος
%
n = input('Δώσε ακέραιο αριθμό ? ');
% έστω ότι ο αριθμός είναι πρώτος
is_prime  = 1;
%
 if  rem(n,2)==0,
    is_prime  = 0;
else
```



```
    divisor  = 3;
   while  divisor <=sqrt(n) & is_prime ,
       if  rem(n, divisor ) == 0,
           is_prime  = 0;
       else
           divisor  =  divisor  + 2;
       end
   end
end
 if  is_prime ,
   disp('Ο αριθμός είναι πρώτος ');
else ,
   disp('Ο αριθμός είναι σύνθετος ');
end
```

Προβλήματα με τον αλγόριθμο Π2

- Λάθος για $n = 1$ (όχι πρώτος) και $n = 2$ (ο μόνος πρώτος άρτιος)
- Ο Π2 μπορεί να είναι *αργότερος* από τον Π1 γιατί υπολογίζει την **sqrt**(n) σε ΚΑΘΕ επανάληψη:

```
while  divisor <=sqrt(n) & is_prime ,
```

  Αντιμετώπιση:

```
limit  =  sqrt(n);
while  divisor <= limit  &  is_prime ,
```

- Ο Π2 μπορεί να δώσει λάθος αποτελέσματα ανάλογα με τον υπολογιστή. Π.χ: για το 49 με διαιρέτες τους 1, 7, 49, η **sqrt**(49) μπορεί να υπολογίζεται σαν 6.9999999 σε κάποιο μηχάνημα. Το 7 δεν θα ελεγχθεί σαν πιθανός διαιρέτης και ο Π2 θα δώσει οτι *ο 49 είναι πρώτος*. Αρα χρειάζεται ελεγχος μέχρι **sqrt**(n) + 1.

Πρόγραμμα για τον αλγόριθμο Π3

```
% ... όπως και σε Π2
if  (rem(n,2)==0 & n>2) | n==1,
    is_prime  = 0;
else
    divisor  = 3;
    limit  =  sqrt(n)+1;
    while  divisor <= limit  &  is_prime ,
        if  rem(n, divisor ) == 0,
            is_prime  = 0;
        else
            divisor  =  divisor  + 2;
        end
    end
end                 % ... όπως και σε Π2
```

Επιλογή αλγορίθμου

Μεταξύ των συντακτικά *ορθών* Π1 και Π3:

- Ο Π3 είναι ταχύτερος, ειδικά όταν αποτελεί μέρος μεγαλύτερης εφαρμογής όπου καλείται πολλές φορές.
- Ο Π1 είναι απλούστερος και πιο ευανάγνωστος.

Σε εφαρμογές όπου ο χρόνος είναι κρίσιμος προτιμάται ο αποτελεσματικότερος αλγόριθμος.

Για εύκολη συντήρηση προγραμμάτων δυνατή και η χρησιμοποίηση απλούστερων αλγορίθμων σε βάρος της αποτελεσματικότητας.

Πίνακες

Δομές δεδομένων

Σε πολλά (κυρίως μαθηματικά) προβλήματα (π.χ. ανάλυση πειραμάτων, στατιστική επεξεργασία, γραφικές παραστάσεις, επίλυση γραμμικών συστημάτων, κ.α.):

- Ανάγκη για αποθήκευση/διαχείριση *πολλών* δεδομένων.
- Για την αποτελεσματικότερη (κυρίως από πλευράς ταχύτητας επεξεργασίας, αλλά και προγραμματιστικής ευκολίας), διαχείριση τους απαιτείται η οργάνωση των δεδομένων ανάλογα με:
  - τις μεταξύ τους σχέσεις·
  - τη σειρά επεξεργασίας τους·
  - τον τρόπο και το μέσον αποθήκευσης τους.
- Δομές δεδομένων.

Μονοδιάστατοι πίνακες (array)

- Η απλούστερη δομή (συλλογή) δεδομένων είναι ο 1-διάστατος πίνακας (array) που είναι:

  Διατεταγμένη Η σειρά των στοιχείων είναι *σημαντική*.

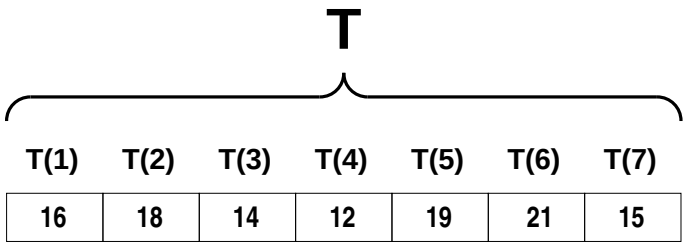
  Ομογενής Ολα τα δεδομένα είναι του *ιδίου* τύπου.
- Αναλογία με πίνακες στα μαθηματικά.
- Χρησιμοποιούνται όταν θέλουμε να φυλάξουμε τα δεδομένα μας για περαιτέρω υπολογισμούς στο προγραμμά μας. Π.χ.
  - Οταν θέλω απλώς να προσθέσω 10 αριθμούς χρησιμοποιώ μια μόνο μεταβλητή για όλα τα δεδομένα μου.
  - Οταν θέλω να συγκρίνω καθένα από τους 10 αριθμούς με τον μέσο όρο τους, χρησιμοποιώ ένα array 10 θέσεων.

Παράδειγμα

Για να διαχειριστούμε τις ημερήσιες θερμοκρασίες μιας εβδομάδας

ΚΥ	ΔΕ	ΤΡ	ΤΕ	ΠΕ	ΠΑ	ΣΑ
16	18	14	12	19	21	15

Χρησιμοποιείται ο 1-διάστατος πίνακας T



όπου η θερμοκρασία της Κυριακής είναι η T(1), της Δευτέρας η T(2), ...

Τα στοιχεία του πίνακα αποθηκεύονται σε ...
... *διαδοχικές και συνεχόμενες* θέσεις μνήμης.

Είσοδος/έξοδος μονοδιάστατων πινάκων

Ενα-ενα τα στοιχεία

```
for i=1:7
   t(i)=input('Δώσε στοιχείο ');
end
```

Ο χρήστης περιμένει το μήνυμα και εισάγει ένα-ένα τα στοιχεία του t

Ολόκληρος ο πίνακας

```
t = input('Δώσε τον πίνακα t ');
```

Ο χρήστης περιμένει το μήνυμα και εισάγει ολόκληρο τον πίνακα:

(16 18, 14, 12 19 21 15)

Τα στοιχεία του πίνακα διαχωρίζονται με κενά ή κόμμα.

Εξοδος του στοιχείου t(i):
disp(t(i))

Εξοδος ολόκληρου του πίνακα:
disp(t)

Επεξεργασία με μονοδιάστατους πίνακες

```
% Αποκλίσεις από μέση ημερήσια θερμοκρασία για μια εβδομάδα
% T   — πίνακας θερμοκρασιών
% mt  — μέση ημερήσια θερμοκρασία
% DT  — διαφορές θερμοκρασιών από μέση τιμή
T = input('Δώσε τον πίνακα θερμοκρασιών ');
sum = 0;
for i=1:7,
   sum = sum + T(i);
end
mt = sum/7;
disp('Μέση ημερήσια θερμοκρασία '); disp(mt);
for i=1:7,
   DT(i) = T(i) — mt;
end
disp('Θερμοκρασιακές διαφορές '); disp(DT);
```


Μέγιστο και ελάχιστο στοιχείο πίνακα 100 στοιχείων

```
for i=1:100,
    list (i) = input('Δώσε στοιχείο της λίστας ');
end
small = list (1);          % το ελάχιστο στοιχείο της λίστας
large = list (1);          % το μέγιστο στοιχείο της λίστας
for i=2:100,
   if list (i) < small,
      small = list (i);
   else,
      if list (i) > large,
         large = list (i);
      end
   end
end
```

Δισδιάστατοι πίνακες

	Μαθήματα Πληροφορικής		
ΑΜ Φοιτητή	Πληροφορική Ι	Πληροφορική ΙΙ	Γλώσσες Προγ/σμού
201000	5	9	2
222222	10	8	0
250001	7	2	5
...	...	...	...
230712	6	6	6

- Αν n είναι το πλήθος των φοιτητών που εξετάστηκαν στα 3 αυτά μαθήματα πληροφορικής, τότε τα παραπάνω δεδομένα παριστάνονται σαν ένας πίνακας $n \times 3$ (n γραμμών και 3 στηλών).
- Κάθε γραμμή του πίνακα έχει τον ίδιο αριθμό στηλών: ο φοιτητής 222222 ή έχει πάρει 0 στο μάθημα 'Γλώσσες Προγ/σμού' δεν έχει εξεταστεί σε αυτό.

Αναθέσεις τιμών σε μεταβλητές πινάκων

- Μονοδιάστατοι πίνακες

```
x  = (1 2 3);
y    = (5, 5, 6, 7, 8, 9);
```

- Δισδιάστατοι πίνακες

```
A = (1 2 3
       4 5 6);
B   = (1 2 3; 4, 5 6);
```

- Αναθέσεις τιμών σε μεμονωμένα στοιχεία ενός πίνακα:

```
x (1) = 8.5;
A(2,1) = 11
```

Είσοδος/έξοδος δισδιάστατων πινάκων

Ενα-ενα τα στοιχεία

```
for i=1:100
    for j=1:3
        M(i,j)=input('Δώσε στοιχείο ');
    end
end
```

Ο χρήστης περιμένει το μήνυμα και εισάγει ένα-ένα τα στοιχεία του M

Ολόκληρος ο πίνακας

```
M = input('Δώσε τον πίνακα M');
```

Ο χρήστης περιμένει το μήνυμα και εισάγει ολόκληρο τον πίνακα.

Επεξεργασία με δισδιάστατους πίνακες

Δίνονται οι βαθμολογίες 100 φοιτητών που εξετάστηκαν σε 6 μαθήματα πληροφορικής. Να βρεθει:

1. Ο μέσος όρος του κάθε φοιτητή.
2. Το μάθημα στο οποίο πέρασαν οι περισσότεροι φοιτητές.

Επεξεργασία με δισδιάστατους πίνακες (συνεχ.)

```
% M                 — πίνακας βαθμολογιών
% numpass           — αριθμός φοιτητών που πέρασαν στο μάθημα i
% student_friendly  — μάθημα με τους περισσότερους επιτυχόντες
M = input('Πίνακας βαθμολογιών ');
for i =1:100,                % Για κάθε φοιτητή
    % Υπολόγισε το μέσο όρο
    sum = 0;
    for j =1:6,
        sum = sum + M(i,j );
    end
    disp(sum/6);
end
%
```



```
% Αριθμός επιτυχόντων σε κάθε μάθημα
for j =1:6,
    numpass(j) = 0;
    for i =1:100,
        if M(i , j ) >= 5,
            numpass(j) = numpass(j) + 1;
        end
    end
end
 student_friendly  = 1;
for i =2:6,
    if numpass(i) > numpass(student_friendly),
        student_friendly  =  i ;
    end
end
disp( student_friendly );
```

Η εντολή size

Χρησιμοποιείται για την εύρεση του αριθμού των στηλών ή των γραμμών ενός πίνακα.

Αν A είναι ένας δισδιάστατος πίνακας $p \times q$ τότε:

- **size**(A,1) δίνει τον αριθμό των γραμμών p του A, και
- **size**(A,2) δίνει τον αριθμό των στηλών q του A.

Ενας μονοδιάστατος πίνακας x είναι ουσιαστικά ένας πίνακας $1 \times n$ (με μια γραμμή και n στήλες. Αρα το πλήθος των στοιχείων του υπολογίζεται σαν: **size**(x,2)

Μέγιστο και ελάχιστο στοιχείο μονοδιάστατου πίνακα

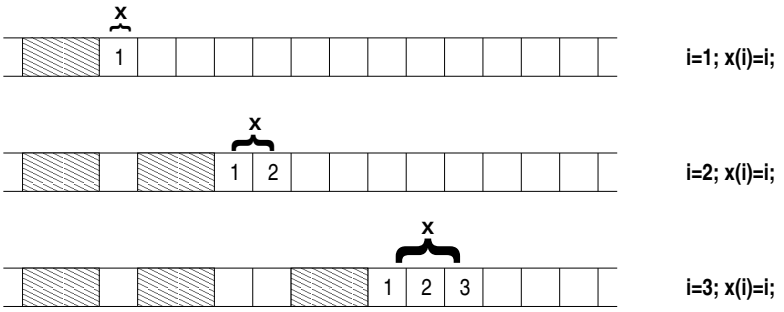
Ο αλγόριθμος στη γενικότερη μορφή του:

```
 list  = input('Δώσε τη λίστα ');
small =  list (1);          % το ελάχιστο στοιχείο της λίστας
large =  list (1);          % το μέγιστο στοιχείο της λίστας
for  i=2:size( list ,2),    % πλήθος στοιχείων με την εντολή size
    if   list (i) < small,
        small =  list (i);
    else,
        if   list (i) > large,
            large =  list (i);
        end
    end
end
```

Αρχικοποίηση πινάκων

```
for i=1:100,
   x(i) = i;
   % εντολές που δεσμεύουν μνήμη
end
```

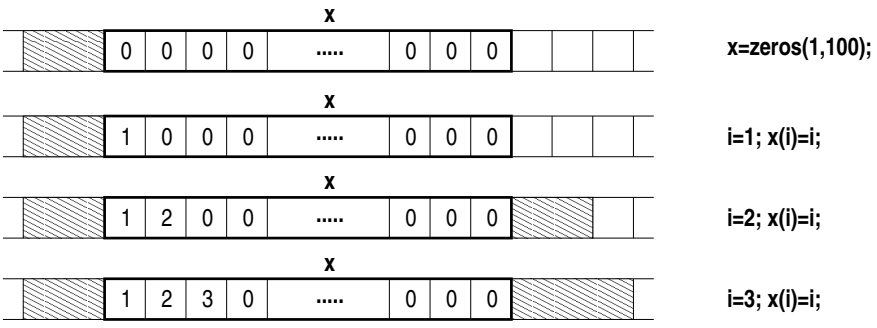
Η εικόνα της μνήμης:



Αρχικοποίηση πινάκων (συνεχ.)

```
x = zeros(1,100);  % δημιουργεί μηδενικό διάνυσμα 100 θέσεων
for i=1:100,
   x(i) = i;
   % εντολές που δεσμεύουν μνήμη
end
```

Η εικόνα της μνήμης:



Γραφικές παραστάσεις: η εντολή plot

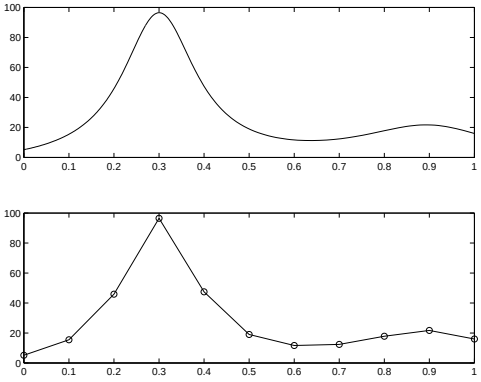
Αν οι συντεταγμένες των σημείων $(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)$ είναι αποθηκευμένες στους μονοδιάστατους πίνακες x και y αντίστοιχα, τότε η εντολή

plot(x, y)

ενώνει γραφικά, με ευθύγραμμα τμήματα το σημείο (x_1, y_1) με το (x_2, y_2), το σημείο (x_2, y_2) με το (x_3, y_3), ..., το σημείο (x_{n-1}, y_{n-1}) με το (x_n, y_n)

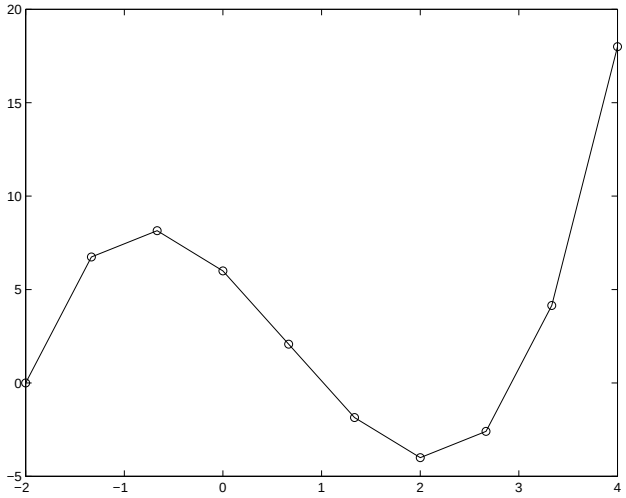
Π.χ για τη συνάρτηση:

$$\frac{1}{(x-0.3)^2+0.1} + \frac{1}{(x-0.9)^2+0.4} - 6$$



Γραφική παράσταση πολυωνύμου

Εστω η: $y = x^3 - 2x^2 - 5x + 6, \qquad x \in [-2, +4].$



... χρειάζεται ο υπολογισμός τιμών (x_i, y_i) σε *διακριτά* σημεία.

Υπολογισμός τιμών πολυωνύμου I

Για δεδομένο n και $x_1 = -2,\ x_2,\ x_3,\ \ldots,\ x_n = +4$.

```
n = input('αριθμός σημείων ? ');
xleft = -2;
xright = 4;
h = ( xright -xleft)/(n-1);        % βήμα x
x = zeros(1,n);                    % αρχικοποίηση x
y = x;                             % αρχικοποίηση y
i = 1;
for t= xleft :h: xright ,
   x(i) = t;
   y(i) = t^3 - 2*t^2 - 5*t + 6;
   i = i + 1;
end
plot(x,y);
```


Υπολογισμός τιμών πολυωνύμου II

Για δεδομένο h και $x_1 = -2,\ x_2 = x_1 + h,\ x_3 = x_2 + h,\ \ldots$

```
h = input('βήμα αύξησης του x ? ');
xleft = -2;
xright = 4;
n = fix (( xright -xleft)/h ) + 1;   % αριθμός σημείων
x = zeros(1,n);                      % αρχικοποίηση x
y = x;                               % αρχικοποίηση y
i = 1;
for t= xleft :h: xright ,            % δεν υπολογίζεται πάντα το xright
   x(i) = t;
   y(i) = t^3 - 2*t^2 - 5*t + 6;
   i = i + 1;
end
plot(x,y);
```

Ο κανόνας του Horner

Σε κάθε επανάληψη (για κάθε σημείο (x_i, y_i)), για τον υπολογισμό της τιμής της έκφρασης:

```
y(i) = t^3 − 2*t^2 − 5*t + 6;
```

γίνονται 5 πολλαπλασιασμοί (2 για t^3, 2 για 2*t^2 και 1 για 5*t).

Αν η εκφραση υπολογιστεί ισοδύναμα σαν:

```
y(i) = ((t−2)*t − 5)*t + 6;
```

τότε εκτελούνται *μόνο* 2 πολλαπλασιασμοί.

Προφανώς: μεγαλύτερο υπολογιστικό όφελος για πολυώνυμα μεγάλου βαθμού και/ή πολλές επαναλήψεις (σημεία).

Υπολογισμός ρητών συναρτήσεων

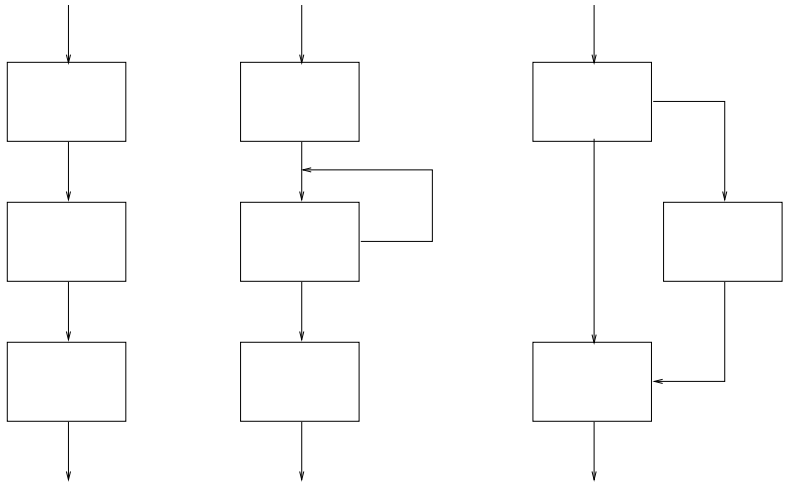
Για $h = 0.25$, υπολογισμός: $y = (x^3 + 2x^2 - 5x + 6)/(x^2 - x - 2)$, στο $[-4, 3]$

```
h = 0.25;   xleft = −4;   xright = 3;
n = fix ((xright −xleft)/h ) + 1;   % αριθμός σημείων
x = zeros(1,n);   y = x;   i = 1;
for t=xleft :h: xright ,
   x(i) = t;
   denom = t^2 − t − 2;          % υπολογισμός παρανομαστή
   if denom == 0,
      disp('η συνάρτηση δεν ορίζεται στο: '); disp(t);
   else,
      y(i) = ( t^3 − 2*t^2 − 5*t + 6) / denom;
      disp(x(i)); disp(y(i));
   end
   i = i + 1;
end
```

Συναρτήσεις

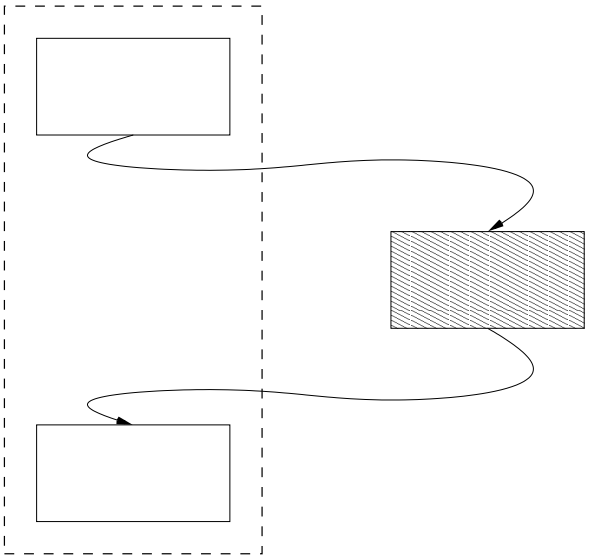
Ροή ελέγχου

Είναι η σειρά με την οποία εκτελούνται οι εντολές. Μέχρι τώρα, «σειριακή», «επαναληπτική» και εκτέλεση «υπό συνθήκη».



Μια άλλη μορφή ροής ελέγχου

Ενα πρόγραμμα *καλεί* μια *συνάρτηση* (ή *υπο-πρόγραμμα*), ο έλεγχος μεταφέρεται προσωρινά στις εντολές της συνάρτησης και, αφού εκετελεστούν *επιστρέφει* στον αρχικό κώδικα.



Η (μεγάλη) ιδέα πίσω από τις συναρτήσεις

- Εντόπισε ένα «υπο-πρόβλημα» που χρειάζεται να επιλυθεί σαν μέρος του προγράμματος (της συνολικής αλγοριθμικής επίλυσης για το πρόβλημα που αντιμετωπίζει το πρόγραμμα).
- Επίλυσε αλγοριθμικά το υπο-πρόβλημα και γράψε τον κώδικα μόνο μια φορά.
- Δώσε στο κωδικα του υπο-προβλήματος ένα όνομα: αυτό τον μετατρέπει σε συνάρτηση.
- Οταν συναντήσεις ξανά το ίδιο υπο-πρόβλημα, χρησιμοποίησε απλώς το όνομα της συνάρτησης για να ζητήσεις να εκτελεστεί *εμβόλιμα* ο κώδικας της συνάρτησης προτού ο έλεγχος επιστρέψει πίσω.

Συναρτήσεις

Σύνολο συγκεντρωμένων εντολών με χαρακτηριστικό όνομα. Αυτοτελής κώδικας για συγκεκριμένη υπολογιστική εργασία, που «αποκρύπτει» προγραμματιστικές λεπτομέρειες.

Γιατί συναρτήσεις?

- Λογική σχεδίαση προγραμμάτος.
- «Μαύρα κουτιά».
- Επαναχρησιμοποίηση κώδικα σε άλλα προγράμματα.
- Αποφυγή επανάληψης κώδικα στο ίδιο πρόγραμμα.

Είδη συναρτήσεων:

1. Βιβλιοθήκης (μέρος του Matlab)·
2. Ορισμένες από προγραμματιστή.

Σχεδιασμός συναρτήσεων, απόκρυψη υλοποίησης

Μια συνάρτηση:

- υλοποιεί κάποιον αλγόριθμο (χρειάζεται δεδομένα εισόδου/εξόδου)
- παίρνει δεδομένα εισόδου από την συνάρτηση που την κάλεσε, και
- δίνει δεδομένα εξόδου πίσω στην συνάρτηση που την κάλεσε,

και πρέπει να έχει:

- μοναδικό όνομα·
- καλά σχεδιασμένο interface (= πως επικοινωνει με το περιβάλλον της = τι είδους *παραμέτρους* χρειάζεται για να λειτουργήσει σαν «μαύρο κουτί»)

Μια καλά σχεδιασμένη συνάρτηση λειτουργεί σαν «μαύρο κουτί»

Προγραμματισμός συναρτήσεων

- Ορισμός συνάρτησης

```
function (παράμετροι εξόδου ) = όνομα (παράμετροι εισόδου)
    εντολές
return
```

- Κλήση συνάρτησης

```
(ορίσματα εξόδου ) = όνομα (ορίσματα εισόδου );
```


Υπολογισμός $N! = 1 \times 2 \times 3 \times \cdots \times N$

```
function p = Factorial (n)
%FACTORIAL(N) Υπολογίζει το N!
p = 1;
for i=1:n , p = p * i ; end
return
```

Κλήση από αρχείο προγράμματος η το περιβάλλον Matlab:

```
a = Factorial (5);
```

Κλήση από άλλη συνάρτηση:

```
function c = Combinations(n, k)
%COMBINATIONS(N,K) Συνδυασμοί N αντικειμένων ανά K
c = Factorial (n ) / ( Factorial (k ) * Factorial (n−k));
return
```

Εμβαδόν κύκλου και δακτυλίου

```
function E = circle_area(r)
E = pi*r*r;
return
```

```
function E = ring_area(inner , outer)
%RING_AREA Εμβαδόν κυκλικού δαλτυλίου
% inner , outer — Εσωτερική/εξωτερική ακτίνα
inner_area = circle_area(inner );   % με χρήση της  circle_area
outer_area = circle_area(outer);
E = outer_area — inner_area;
return
```


Τοπικές μεταβλητές

- Μια συνάρτηση μπορεί να ορίσει δικές της *τοπικές μεταβλητές*.
- Οι τοπικές μεταβλητές έχουν νόημα ΜΟΝΟ μέσα στη συνάρτηση
  - *Δημιουργούνται* όταν καλείται η συνάρτηση.
  - *Παύουν να υπάρχουν* όταν η συνάρτηση επιστρέψει.
- Οι παράμετροι μιας συνάρτησης είναι επίσης τοπικές.
- Οι παράμετροι *αρχικοποιούνται αντιγράφοντας* την τιμή του ορίσματος.

Ενα πρόγραμμα με συναρτήσεις

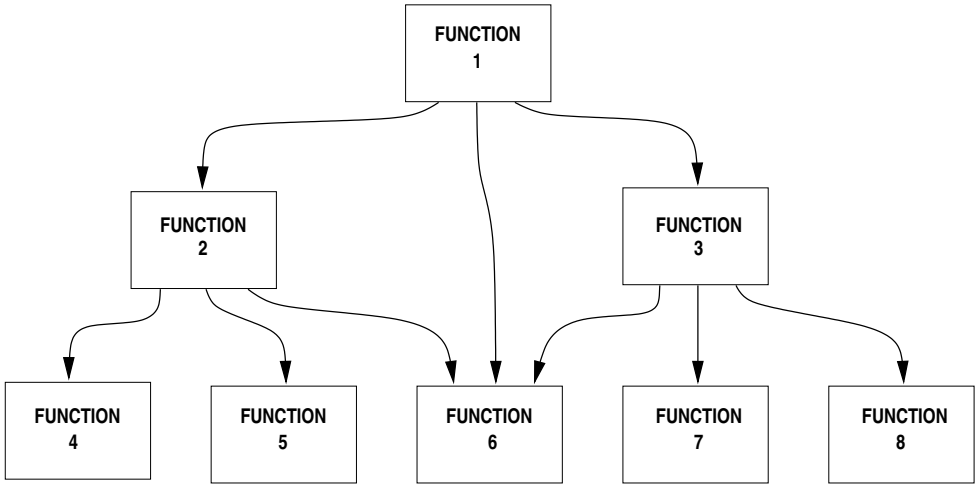
```
r1 = input('Εσωτερική ακτίνα δακτυλίου ');
r2 = input('Εξωτερική ακτίνα δακτυλίου ');
disp(ring_area(r1 ,r2 ));
```

Κάθε συνάρτηση στο Matlab γράφεται σε δικό της αρχείο

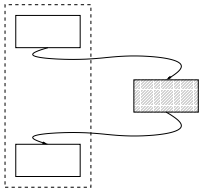
- Το παραπάνω γράφεται εκτελείται άμεσα στο περιβάλλον Matlab, ή γράφεται σε αρχείο με κατάληξη .m
- Η συνάρτηση circle_area ΠΡΕΠΕΙ να γραφτεί σε αρχείο circle_area.m
- Η συνάρτηση ring_area ΠΡΕΠΕΙ να γραφτεί σε αρχείο ring_area.m

Το Matlab εντοπίζει τον κώδικα μιάς συνάρτησης ΜΟΝΟ σε αρχεία με το ίδιο όνομα και κατάληξη .m

Δομή προγραμμάτων: ιεραρχία συναρτήσεων



Ροή ελέγχου συνάρτησης



Οταν καλείται μια συνάρτηση:

1. *Δεσμεύεται* χώρος στη μνήμη για τις παραμέτρους και τις τοπικές μεταβλητές της συνάρτησης.
2. Οι τιμές των ορισμάτων *αντιγράφονται* στις αντίστοιχες μεταβλητές των παραμέτρων.
3. Ο έλεγχος *μεταφέρεται* στο σώμα της συνάρτησης.
4. *Εκτελούνται* οι εντολές της συνάρτησης
5. Ο έλεγχος και τα δεδομένα εξόδου *επιστρέφονται* στην καλούσα συνάρτηση.
6. *Αποδεσμεύεται* ο χώρος στη μνήμη για τις παραμέτρους και τις τοπικές μεταβλητές της συνάρτησης.

Συνάρτηση για έλεγχο πρώτων αριθμών

```
function is_prime = isPrime(n)
%ISPRIME(N) Επιστρέφει 1 αν ο N είναι πρώτος, 0 αν είναι σύνθετος
is_prime = 1;
if (rem(n,2)==0 & n>2) | n==1,
   is_prime = 0;
else
    divisor = 3;
    limit = sqrt(n)+1;
   while divisor <= limit & is_prime,
       if rem(n, divisor ) == 0, is_prime = 0; else, divisor = divisor + 2; end
   end
end
return
```

Ζεύγη πρώτων αριθμών

Πρόβλημα Να βρεθούν όλα τα ζεύγη πρώτων αριθμών p και q στο διάστημα $[3, 200]$ για τα οποία $q = 2p + 1$.

Ανάλυση Αρκεί ο έλεγχος περιττών αριθμών *στο διάστημα* $[3, 99]$. Για κάθε περιττό p ελέγχεται άν ο $2p + 1$ είναι πρώτος. Χρησιμοποιείται η συνάρτηση isPrime.

```
for p = 3:2:99,
    if isPrime(p),
        if isPrime(2*p+1),
            disp(p); disp(2*p+1);
        end
    end
end
```


Διάταξη 3 αριθμών

Πρόβλημα

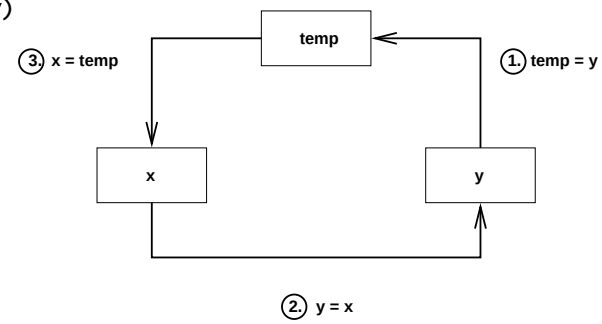
Να γραφεί συνάρτηση που δέχεται 3 αριθμούς a, b και c και τους επιστρέφει διατεταγμένους έτσι ώστε $a \leq b \leq c$.

Ανάλυση Ενας μή προφανής (αλλά κομψός) αλγόριθμος:

1. Εναλλαγή των a και b, αν χρειάζεται, έτσι ώστε $a \leq b$.
2. Εναλλαγή των b και c, αν χρειάζεται, έτσι ώστε $b \leq c$. (Τώρα η μεταβλητή c έχει την μεγαλύτερη τιμή, αλλά τα a, b δεν είναι κατ' ανάγκη διατεταγμένα).
3. Εναλλαγή των a και b, αν χρειάζεται, έτσι ώστε $a \leq b$.

Διάταξη 3 αριθμών

```
function (x, y) = sort2(x, y)
%Διάταξη 2 αριθμών
if y<x,
   temp = y;
   y = x;
   x = temp;
end
return
```




```
function (a, b, c) = sort3(a, b, c)
%Διάταξη 3 αριθμών
(a, b) = sort2(a, b);
(b, c) = sort2(b, c);
(a, b) = sort2(a, b);
return
```


Μαθηματικές εφαρμογές

Μέγιστος Κοινός Διαιρέτης (ΜΚΔ) - Ι

Εξαντλητικός αλγόριθμος Δ1

1. Εστω MKD(x,y) == x (δεν μπορεί να είναι >x και να διαιρεί ταυτόχρονα το x).
2. Αν ο x διαιρεί τον y, ΤΕΛΟΣ: ΜΚΔ είναι ο x.
3. Αν ο x ΔΕΝ διαιρεί τον y, έστω οτι MKD(x,y) == (x−1). ΒΗΜΑ 1.

```
function g = MKD(x, y)
g = x;
while rem(x,g)~=0 | rem(y,g)~=0,
   g = g − 1;
end
return
```


Παρατηρήσεις στον αλγόριθμο Δ1

- Το **while** εκτελείται όσο η συνθήκη: **rem**(x,g)~=0 | **rem**(y,g)~=0 είναι true δηλ. σταματάει όταν: **rem**(x,g)==0 & **rem**(y,g)==0
- Ο αλγόριθμος Δ1 τερματίζει πάντα (με MKD(x,y) == 1).

Νόμοι De Morgan
Για τις λογικές προτάσεις (εκφράσεις), p και q ισχύει:

$\neg(p \,\&\, q) \iff \neg p \mid \neg q$

και

$\neg(p \mid q) \iff \neg p \,\&\, \neg q$

Πρόβλημα με Δ1: χρειάζεται πολλά βήματα, π.χ.
1000000 βήματα για τον υπολογισμό MKD(1000005, 1000000) == 5.

Μέγιστος Κοινός Διαιρέτης (ΜΚΔ) - II

Αλγόριθμος του Ευκλείδη[α΄] Δ2

Βήμα 1: Υπολόγισε r = **rem**(x, y).

Βήμα 2: Αν r==0, τότε MKD(x,y)==y.

Βήμα 3: Αν r~=0, εκτέλεσε το **Βήμα 1** με x=y και y=r.

Ο Δ2 υπολογίζει τον MKD(1000005, 1000000) σε *δύο μόνο βήματα*.

```
function y = MKD(x, y)
r = rem(x, y);
while r ~= 0,
   x = y;  y = r;
   r = rem(x, y);
end
return
```

[α΄] *Στοιχεία*, Βιβλίο 7, Πρόταση ΙΙ.

Αριθμητικοί Αλγόριθμοι

Παράδειγμα για τον υπολογισμό μαθηματικών συναρτήσεων όπως η **sqrt** για την τετραγωνική ρίζα.

Ενδεικτικές κατηγορίες αριθμητικών αλγορίθμων:

- Διαδοχικές προσεγγίσεις.
- Ανάπτυγμα σειράς.

Διαδοχικές προσεγγίσεις

1. Εστω οτι η αυθαίρετη τιμή x_0 είναι η απάντηση.
2. Υπολογισμός καλύτερης 'προσεγγισης' x_1 χρησιμοποιώντας το x_0 (π.χ. αν x_0 πολύ μεγάλο επιλέγεται μικρότερο x_1 και αντίστροφα).
3. Αν εξασφαλίζεται οτι κάθε νέα προσέγγιση με τη μεθοδολογία του ΒΗΜΑΤΟΣ 2 πλησιάζει περισσότερο την απάντηση τότε η επαναληπτική διαδικασία 1-2-3 οδηγεί διαδοχικά σε καλύτερες προσεγγίσεις. ΤΕΛΟΣ όταν επιτευχθεί κάποια ικανοποιητική ακρίβεια.

Ελεγχος 'ισότητας' δυο πραγματικών αριθμών

Πρόβλημα ακρίβειας στις συγκρίσεις πραγματικών.

Υπολογιστικά, 2 πραγματικοί x, y, θεωρούνται ίσοι όταν ικανοποιούν κάποια προσεγγιστική σχέση όπως η:

$$\frac{|x-y|}{\min(|x|,|y|)} < \varepsilon$$

όπου ε μια σταθερά για την επιθυμητή ακρίβεια.

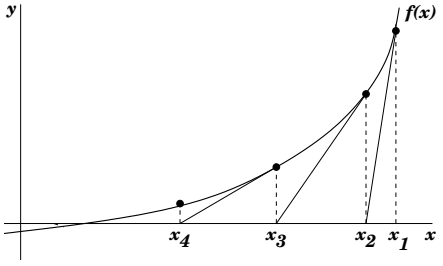
Βοηθητικές συναρτήσεις

```
function bool = approxEqual(x, y)
epsilon = 0.000001;
num = abs(x − y);
den = minF(abs(x), abs(y));
if (num+den == num),
    bool = (x==y);
else,
    bool = (num/den < epsilon);
end
return
```

```
function y = minF(x , y)
if x<y , y = x ; end
return
```


Η μέθοδος Newton για τετραγωνική ρίζα

$x = \sqrt{a}$ ισοδ. με εύρεση της θετικής ρίζας της συνάρτησης $f(x) = x^2 - a$.



$$\frac{f(x_0)}{x_0 - x_1} = f'(x0) \Rightarrow x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

και για την $x^2 - a$:

$$x_{i+1} = x_i - \frac{x_i^2 - a}{2x_i}$$

Χρειάζονται 2 μεταβλητές για τα x_{i+1} και x_i ?

Τετραγωνική ρίζα με αλγόριθμο Newton

```
function x = NewtonSqrt(a)
 if a == 0,  x = 0;  return ;  end
 if a < 0,  disp("ERROR!");  x = −1;  return ;  end
x = a;
while ~approximatelyEqual(a, x∗x)
    % x = x − (x∗x − a)/(2∗x);
    x = (x + a/x)/2;
end
return
```


Ανάπτυγμα σειράς

Το παράδοξο του Ζήνωνα:

$$s = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \dots$$

δηλ. άπειρο πλήθος όρων με πεπερασμένο όμως άθροισμα:

$$2s = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \dots$$

ή

$$2s = 1 + s$$

Αρα η σειρά συγκλίνει στο $s = 1$.

Γενικός όρος της σειράς: $t_i = 1/2^i$ (δυναμοσειρά)

Οι δυναμοσειρές σημαντικές για τα υπολογιστικά μαθηματικά.

Σε αλγόριθμους υπολογισμού δυναμοσειρών αναζητείται σχέση που δίνει τον i-στο όρο από τον όρο $i - 1$.

Υπολογισμός της $s = \sum \frac{1}{2^i}$

Κάθε νέος όρος προκύπτει από τον προηγούμενο:

$$t_i = \frac{1}{2^i} = \frac{1}{2^{i-1}}\frac{1}{2} = t_{i-1}\frac{1}{2}$$

```
sum = 0;
term  = 0.5;
while 1,
    sum = sum + term;
    term = term/2;
end
```

Αλλά από κάποιο σημείο και μετά το άθροισμα για τον υπολογιστή παραμένει σταθερό (ακρίβεια αναπαράστασης δεκαδικών ψηφίων). Τερματισμός των επαναλήψεων με τον έλεγχο:

```
sum == sum + term
```


Το πρόγραμμα για τη σειρά του Ζήνωνα

```
sum = 0;
term  = 0.5;
while sum ~= sum + term,
    sum = sum + term;
    term = term/2;
end
disp("Αθροισμα της σειράς του Ζήνωνα ");
disp(sum);
```

Υπολογισμός τετραγωνικής ρίζας - Σειρά Taylor

Διαφορίσιμες συναρτήσεις προσεγγίζονται στην περιοχή του ∂ από το *ανάπτυγμα Taylor*

$$f(x) = f(\partial) + f'(\partial)(x-\partial) + f''(\partial)\frac{(x-\partial)^2}{2!} + f'''(\partial)\frac{(x-\partial)^3}{3!} + \cdots + f^{(n)}(\partial)\frac{(x-\partial)^n}{n!}$$

Για την τετραγωνική ρίζα: $f(x) = x^{1/2}$ και:

$f'(x)$	$f''(x)$	$f'''(x)$	$\ldots$
$\frac{1}{2}x^{-1/2}$	$-\frac{1}{4}x^{-3/2}$	$\frac{3}{8}x^{-5/2}$	$\ldots$

Ο τύπος Taylor υπολογίζει παραγώγους στο ∂. Για ευκολία έστω $\partial = 1$, τότε

$f(1)$	$f'(1)$	$f''(1)$	$f'''(1)$	$\ldots$
1	$\frac{1}{2}$	$-\frac{1}{4}$	$\frac{3}{8}$	$\ldots$

οπότε:

$$\sqrt{x} = 1 + \frac{1}{2}(x-1) - \frac{1}{4}\frac{(x-1)^2}{2!} + \frac{3}{8}\frac{(x-1)^3}{3!} + \cdots = \sum_0^\infty t_i$$

Προγραμματισμός της μεθόδου Taylor για $\sqrt{x}$

Οι διαδοχικοί όροι της σειράς είναι:

t_0	t_1	t_2	t_3	$\ldots$
1	$\frac{1}{2}(x-1)$	$-\frac{1}{4}\frac{(x-1)^2}{2!}$	$\frac{3}{8}\frac{(x-1)^3}{3!}$	$\ldots$

η γενικά της μορφής:

```
coeff * (xpower / factorial)
```

Η σχέσεις που δίνουν τα νέα coeff, xpower, factorial για τον όρο $i+1$ από τις αντίστοιχες προηγούμενες τιμές για τον όρο i ειναι:

```
coeff     = coeff*(0.5 - i);
xpower    = xpower*(x - 1);
factorial = factorial*(i + 1);
```

Τετραγωνική ρίζα με αλγόριθμο Taylor - I

```
function sum = TSqrt(x)
term = factorial = coeff = xpower = 1;
sum = 0;
i = 0;
while sum ~= sum + term,
    sum = sum + term;
    % προετοιμασία του όρου i + 1
    coeff = coeff*(0.5 - i);
    xpower = xpower*(x - 1);
    factorial = factorial*(i + 1);
    term = coeff*xpower/factorial;
    i = i + 1;
end
return
```


Ορια σύγκλισης σειράς Taylor

Προβλημα

Για $\vartheta = 1$, η $\sqrt{x}$ από τον τύπο του Taylor στην περιοχή του ϑ *συγκλίνει MONO* για:

$$0 < x < 2$$

Λυση

Αναγωγή του προβλήματος για οποιοδήποτε x στην περιοχή σύγκλισης.

Παρατηρηση

$$\sqrt{4x} = 2\sqrt{x}$$

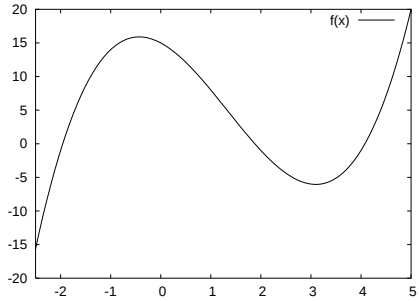
δηλ. μετά από διαδοχικές διαιρέσεις του x με το 4, υπολογίζεται η $\sqrt{x}$ στο διάστημα $(0, 2)$ και ακολουθεί διόρθωση του αποτελέσματος με αντίστοιχο αριθμό πολλαπλασιασμών $\times 2$.

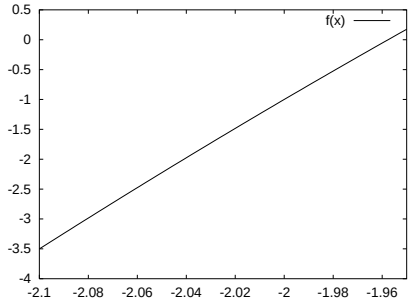
Τετραγωνική ρίζα με αλγόριθμο Taylor - II

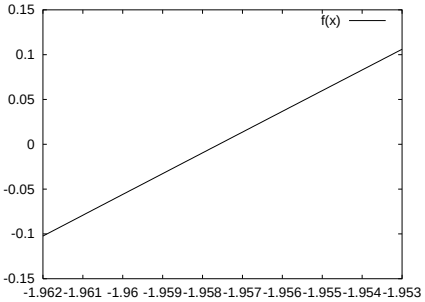
```
function y = TaylorSqrt (x)
 if x == 0,  y = 0;  return ;  end
 if x < 0,  disp("ERROR!");  y = −1;  return ;  end
correction = 1;
while x >= 2,
    x = x / 4;
    correction = correction ∗ 2;
end
y = TSqrt(x)∗correction;
return
```


Εύρεση ριζών συνάρτησης: γραφική μέθοδος

1. γραφ. παράσταση $f(x)$ στο $[a, \beta]$.
2. εκτίμηση «καλύτερου» $[a', \beta']$.
3. γραφ. παράσταση $f(x)$ στο $[a', \beta']$.
4. επανάληψη 2 και 3 μέχρι ...



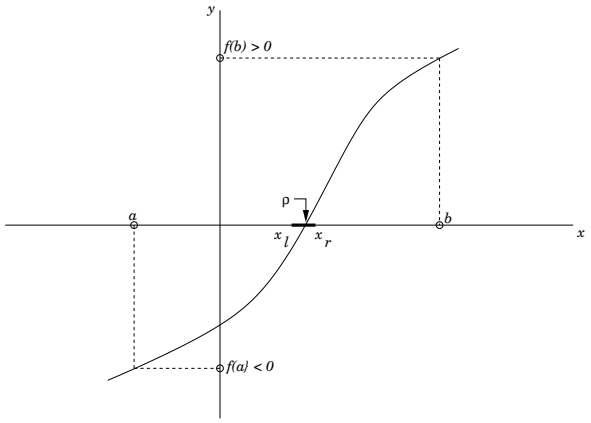




Αυτοματοποιημένη αναζήτηση ριζών

Αναζήτηση ριζών συνάρτησης f στο διάστημα $[a, b]$

1. Αναζήτηση *υποδιαστημάτων* που που περιέχουν ρίζες.
2. Περιορισμός του υποδιαστήματος $[x_l, x_r]$ που περιέχει τη ρίζα ρ, ετσι ώστε $|x_l - x_r| < \epsilon$ η καλύτερα approxEqual(xl,xr), δηλ. $x_l \simeq x_r \simeq \rho$.



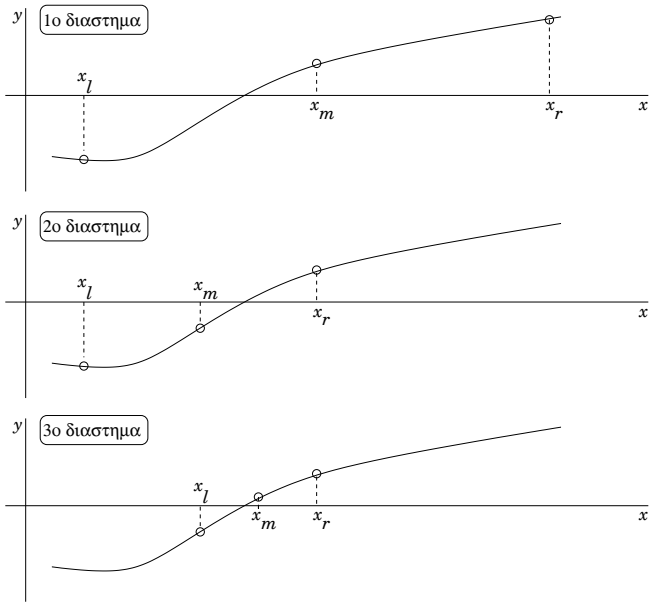
Αναζήτηση υποδιαστημάτων με ρίζες

```
 xfirst  = input('Αριστερό άκρο αρχικού διαστήματος ');
 xlast  = input('Δεξί άκρο αρχικού διαστήματος ');
nsteps = input('Αριθμός υποδιαστημάτων ');
% Ελεγξε αν το αρχικό διάστημα περιέχει κάποια ρίζα
 if  f( xfirst )*f( xlast ) > 0,
    disp('Δεν υπάρχει αλλαγή προσήμου στο διάστημα ');
else,
    xstep = ( xlast - xfirst )/nsteps;
    xl =  xfirst  ;    xr = xl + xstep;
    while f(xl)*f(xr ) > 0
        xl = xr ;    xr = xr + step;
    end
    disp('Υπάρχει ρίζα μεταξύ των ');    disp(xl );    disp(xr );
end
```

Συστηματικός περιορισμός του διαστήματος με τη ρίζα

```
xfirst = input('Αριστερό άκρο αρχικού διαστήματος ?');
xlast = input('Δεξί άκρο αρχικού διαστήματος ?');
if f(xfirst)*f(xlast) > 0,
   disp('Δεν υπάρχει αλλαγή προσήμου στο διάστημα ');
else,
   while ~approxEqual(xlast, xfirst),
      % διαιρώ συνεχώς με 25, μέχρι το διάστημα να είναι αρκετά μικρό
      xstep = (xlast - xfirst)/25;
      xl = xfirst ; xr = xl + xstep;
      while f(xl)*f(xr) > 0
         xl = xr ; xr = xr + step;
      end
      xfirst = xl ; xlast = xr;
   end
   disp('Υπάρχει ρίζα μεταξύ των '); disp(xl); disp(xr);
end
```


Η μέθοδος της διχοτόμησης



Αλγόριθμος διχοτόμησης

```
% xl , xr — άκρα διαστημάτων
% fl , fr — οι τιμές της συνάρτησης στα άκρα
% xm — μέσον διαστημάτων
% fm — η τιμή της συνάρτησης στο μέσον κάθε διαστήματος
%
% αρχικό διάστημα
xl = input('Αριστερό άκρο ? ');
xr = input('Δεξί άκρο ? ');
% υπολογισμός f(x) στα άκρα του διαστήματος
fl = f(xl);
fr = f(xr);
if fl*fr > 0,
   disp('Δεν υπάρχει αλλαγή προσήμου στο διάστημα ');
else ,                          % αλλαγή προσήμου, υπάρχει ρίζα
```



```
   xm = (xl + xr)/2;
   while ~approxEqual(xl, xm),
      fm = f(xm);
      if fl*fm <= 0,            % επιλογή αριστερού υποδιαστήματος
         xr = xm;
      else ,                    % επιλογή δεξιού υποδιαστήματος
         xl = xm;
         fl = fm;
      end
      xm = (xl + xr)/2;
   end
   disp('Η ρίζα είναι: ');
   disp(xm);
   disp('Η τιμή της συνάρτησης στη ρίζα είναι: ');
   disp(f(xm));
end
```

Παρατηρήσεις

Η μ. της διχοτόμησης ανεξάρτητη της f (εφαρμόζεται σε *οποιαδήποτε* συνάρτηση).

Π.χ. για να βρούμε τις ρίζες της $f(x) = x^3 - 4x^2 - 4x + 12$ αρκεί να γράψουμε την παρακάτω συνάρτηση f του Matlab

```
function y = f(x)
y = x^3 - 4*x^2 - 4*x + 12;
```

Για τις ρίζες μιάς *άλλης* συνάρτησης $g(x)$ θα πρεπει:

1. ή να γράψουμε συνάρτηση g του Matlab και ταυτόχρονα να τροποποιήσουμε τον αλγόριθμο της διχοτόμησης ώστε να καλεί τη συνάρτηση g αντί της f
2. ή να γραψουμε συνάρτηση f του Matlab που υπολογίζει την $g(x)$ και να αφήσουμε το πρόγραμμα της διχοτόμησης ως έχει.

Ο μηχανισμός feval

Ιδανικά, θα θέλαμε η μ. της διχοτόμησης να γραφεί σαν συνάρτηση και να βρίσκει τις ρίζες μιας *οποιασδήποτε συνάρτησης ανεξάρτητα από το όνομά της*.

Αν matlab_fun είναι το όνομα μιας συνάρτησης Matlab με μια παράμετρο εισόδου και μια εξόδου τότε η εντολή

```
a = feval('matlab_fun', z);
```

είναι ισοδύναμε με την κλήση της συνάρτησης

```
a = matlab_fun(z);
```

Π.χ. τόσο η **sin**(**pi**/4) όσο και η **feval**('sin', **pi**/4) υπολογίζουν το $\sin(\pi/4)$.

Συνάρτηση για μ. διχοτόμησης

```
function (root, err) = bisect(f, xl, xr)
% err      — κώδικας λάθους
%            err == 0, βρέθηκε ρίζα
%            err == 1, δεν βρέθηκε ρίζα
err = 0;
fl = feval(f, xl);   fr = feval(f, xr);
if fl*fr > 0,
   err = 1;   root = 0;
else,                          % αλλαγή προσήμου, υπάρχει ρίζα
   xm = (xl + xr)/2;
   while ~approxEqual(xl, xm),
      fm = feval(f, xm);
      if fl*fm <= 0,           % επιλογή αριστερού υποδιαστήματος
         xr = xm;
```

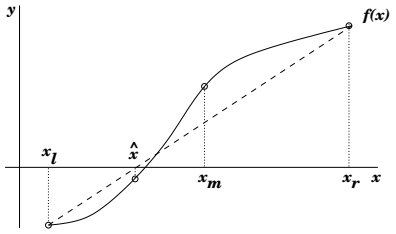


```
      else,                    % επιλογή δεξιού υποδιαστήματος
         xl = xm;   fl = fm;
      end
      xm = (xl + xr)/2;
   end
   root = xm;
end
```

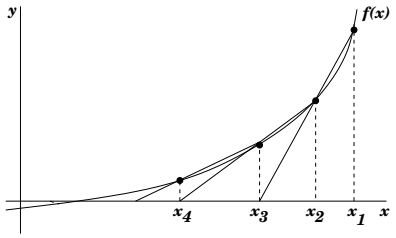
και καλείται π.χ. για τη συνάρτηση **sin**

```
(r, not_found) = bisect('sin', 2*pi-0.5, 2*pi+0.5);
if (not_found),
   disp('Δεν υπάρχει αλλαγή προσήμου στο διάστημα');
else
   disp('Η ρίζα είναι:');
   disp(r);
end
```

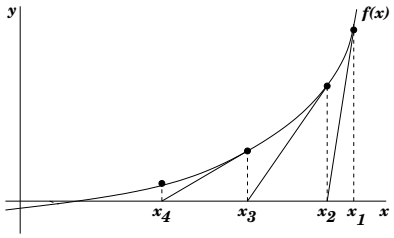
Αλλες μέθοδοι εύρεσης ριζών



μ. της χορδής



μ. της τέμνουσας



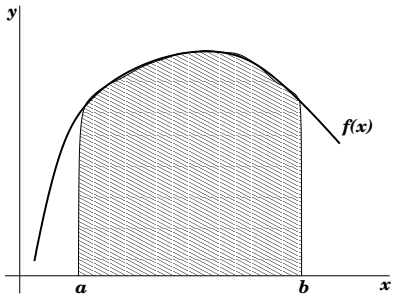
μ. Newton

Αριθμητική ολοκλήρωση

Σημαντικό υποπρόβλημα σε πολλές εφαρμογές σε διάφορα επιστημονικά πεδία:

$$S = \int_a^b f(x)\,dx = \lim_{n\to+\infty} S_n$$

Εξ' ορισμού είναι το άθροισμα των εμβαδών απείρων ορθογωνίων μηδενικού πλάτους.



$$S_n = \sum_{i=1}^{n} f(x_i') \times (x_{i+1} - x_i)$$

Αρνητικές τιμές της $f(x)$ αφαιρούνται από το άθροισμα.

Ο κανόνας του ορθογωνίου

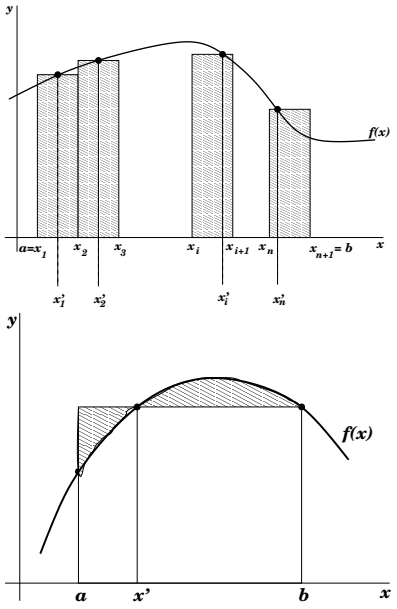
Προσέγγιση με ικανό αριθμό ορθογωνίων, ή καλύτερα από θ. μέσης τιμής ολοκληρωτικού λογισμού:

$$\exists\, x' : S_1 = (b-a) \times f(x') \equiv S$$

Και επειδή το x' δεν είναι γνωστό θεωρούμε

$$f(x') \simeq \sum_{i=1}^{n} w_i \times f(x_i), \quad \text{με} \sum_{i=1}^{n} w_i = 1$$

για διάφορα $x_i \in [a, b]$.



Ο κανόνας του ορθογωνίου (συνεχ.)

Στην απλούστερη μορφή:

- επιλέγονται ίσα βάρη: $w_i = 1/n$
- τα x_i είναι n ισαπέχοντα σημεία στο $[a, b]$

Ανάλογα με ποιό άκρο των υποδιαστημάτων χρησιμοποιείται:

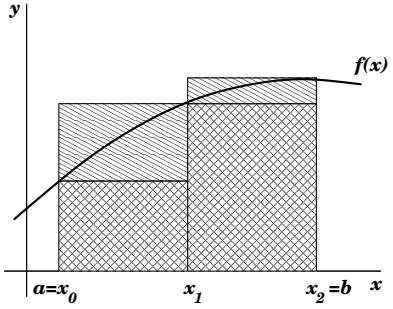
$$x_i = a + i \times (b-a)/n, \quad i = 0, 1, 2, \ldots, n-1,$$
$$x_i = a + i \times (b-a)/n, \quad i = 1, 2, \ldots, n$$

και οι αντίστοιχοι τύποι του ορθογωνίου:

$$R' = \frac{b-a}{n} \times \sum_{i=0}^{n-1} f(x_i),$$

$$R'' = \frac{b-a}{n} \times \sum_{i=1}^{n} f(x_i)$$

Για f αύξουσα στο $[a, b]$: $R' < S < R''$

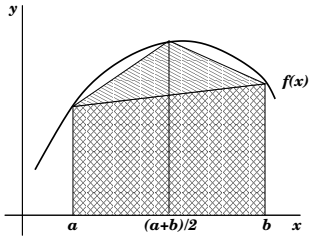


Ο κανόνας του τραπεζίου

$$T = \frac{R' + R''}{2} = \frac{1}{2}\left[\frac{b-a}{n} \times \sum_{i=0}^{n-1} f(x_i) + \frac{b-a}{n} \times \sum_{i=1}^{n} f(x_i)\right]$$

$$= \frac{b-a}{2 \times n} \times \left[f(x_0) + 2 \times \sum_{i=1}^{n-1} f(x_i) + f(x_n)\right]$$

$$= \frac{h}{2} \times \left[f(a) + 2 \times \sum_{i=1}^{n-1} f(a + i \times h) + f(b)\right], \quad \text{με } h = \frac{b-a}{n}$$

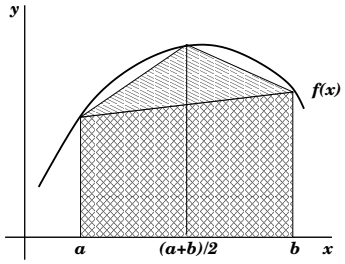


Συνάρτηση για ολοκλήρωση με κανόνα του τραπεζίου

```
function area = trapezoid(f , a , b , n)
% a, b − όρια ολοκλήρωσης
% n     − αριθμός υποδιαστημάτων
h = (b−a)/n;
sum = 0;
for i=1:n−1
    x = a + i∗h;
    sum = sum + feval(f , x);
end
fa = feval(f , a);    fb = feval(f , b);
area = h∗((fa+fb)/2 + sum);
return
```

Εκτίμηση σφάλματος στον κανόνα του τραπεζίου

Για τον υπολογισμό: $\int_0^1 7 - 7x^6\,dx$ (ακριβής τιμή: 6).



Για μικρά διαστήματα και ομαλές συναρτήσεις, μια εκτίμηση (άνω φράγμα) του σφάλματος ολοκλήρωσης δίνεται από $T_{2n} - T_n$.

n	T_n	σφάλμα e	$T_n - T_{n/2}$
2	5.195313	0.804687	
4	5.785767	0.214233	0.599454
8	5.945597	0.054403	0.159828
16	5.986347	0.013653	0.040750
32	5.996584	0.003416	0.010237

Για διπλάσιο n το σφάλμα διαιρείται με το 4, άρα: $e \simeq c/n^2$

Μιά ακριβέστερη *εκτίμηση* του σφάλματος ολοκλήρωσης δίνεται από $(T_n - T_{n/2})/3$

Αποτελεσματικότητα αλγορίθμων

Αποτελεσματικότητα αλγορίθμων

- Ενας *σωστός* αλγόριθμος δεν είναι κατ' ανάγκη και *αποτελεσματικός*.
- Παρά τη συνεχή αύξηση των υπολογιστικών δυνατοτήτων, το θέμα της αποτελεσματικότητας παραμένει πολύ σημαντικό: ανάγκη για επίλυση ακόμα μεγαλύτερων προβλημάτων.
- Ενα πρόβλημα μπορεί στην πράξη να μην επιλύεται από ένα μη αποτελεσματικό αλγόριθμο.
- Η αποτελεσματικότητα ανάγεται στην ελαχιστοποίηση του αριθμού των πράξεων που απαιτούνται για τον υπολογισμό ενός αποτελέσματος.
- Οταν τεκμηριωθεί η αποτελεσματικότητα ενός αλγορίθμου για μια κατηγορία προβλημάτων, τότε ο αλγόριθμος μπορεί να εφαρμοστεί με *προβλέψιμη* συμπεριφορά για οποιοδήποτε πρόβλημα της ίδιας κατηγορίας.
- Το θέμα της αποτελεσματικότητας των αλγορίθμων είναι είναι από τα σημαντικότερα στην Πληροφορική. Εδώ δίνεται μόνο μια βασική εισαγωγή.

Μέτρηση της αποτελεσματικότητας

Η αποτελεσματικότητα εκφράζεται από τον ρυθμό αύξησης των απαιτήσεων του αλγορίθμου σε *χρόνο* (πράξεις) και *χώρο* (μνήμη) σαν συνάρτηση του μεγέθους του προβλήματος.

- Αν ο αλγόριθμος εφαρμοστεί σε πρόβλημα διπλάσιου μεγέθους, πόσες περισσότερες πράξεις θα εκτελεστούν και πόσα περισσότερα δεδομένα θα πρέπει να αποθηκευτούν?
- Απαντώντας στην ερώτηση αυτή μπορούμε να προβλέψουμε αλλά και να συγκρίνουμε την αποτελεσματικότητα αλγορίθμων.

Το μέγεθος του προβλήματος n αντιπροσωπεύει το πλήθος των δεδομένων που επεξεργάζεται ένα αλγόριθμος. Π.χ. έλεγχος αν ο αριθμός n είναι πρώτος, άθροισμα n αριθμών, εύρεση πραγματικών ριζών πολυωνύμου βαθμού n.

Πολυπλοκότητα αλγορίθμου O

Γραμμικός αλγόριθμος $O(n)$

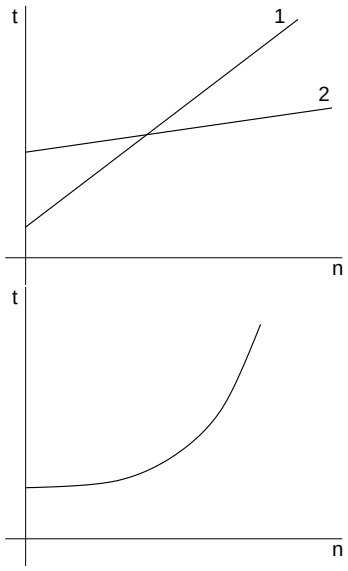
$$t = a + b \times n$$

για διπλάσιο μέγεθος προβλήματος διπλασιάζεται και ο χρόνος επίλυσης.

Τετραγωνικός αλγόριθμος $O(n^2)$

$$t = a + b \times n + c \times n^2$$

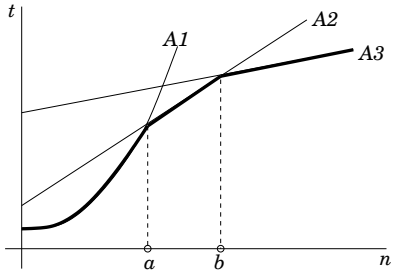
για διπλάσιο μέγεθος προβλήματος τετραπλασιάζεται και ο χρόνος επίλυσης.



Παρατηρήσεις

Ο ταχύτερος αλγόριθμος συχνά εξαρτάται από το μέγεθος του προβλήματος.

Αλλοι παράγοντες, π.χ. η είσοδος δεδομένων, μπορεί να επιβαρύνουν ένα αλγόριθμο καθώς αυξάνεται το μέγεθος του.



Η αποτελεσματικότητα ενός αλγορίθμου σε χρόνο/χώρο εξετάζεται σε 2 επίπεδα:

μαθηματικής δομής: καθορίζει την πολυπλοκότητα του αλγορίθμου.

υλοποίησης: ενας «καλός» αλγόριθμος μπορεί να υλοποιηθεί αναποτελεσματικά.

Παρατηρήσεις (συνεχ.)

Συχνά οι βελτιώσεις σε χρόνο και χώρο είναι *ασυμβίβαστες*. Π.χ. για τον έλεγχο ενός πρώτου αριθμού μπορούμε να ελέγξουμε σαν πιθανούς διαιρέτες:

1. ή *όλους* τους περιτούς αριθμούς $\leq \sqrt{n}$,
2. ή μόνο τους *πρώτους* αριθμούς $\leq \sqrt{n}$, από μια λίστα *αποθηκευμένων* πρώτων αριθμών.

Ο 2ος αλγόριθμος είναι ενδεχομένως αποτελεσματικότερος σε *χρόνο* αλλά χρειάζεται μια μεγάλη *αποθηκευμένη* λίστα πρώτων αριθμών, που μπορεί να δεσμεύει περισσότερο *χώρο* από εκείνο που διαθέτουμε.

Ανάλυση παραγόντων

Αναζήτηση των παραγόντων (όχι κατ' ανάγκη πρώτων) ακεραίου n.

```
% Αλγόριθμος  A
n = input(' ? ' );          % Q1
m = 2;                      % Q2
while m <= n−1,
    if  rem(n, m) == 0,     % P1
        disp(m);            % P2
    end
    m = m + 1;              % P3
end
```

Για κάθε n ($n \geq 2$) εκτελούνται $n-2$ επαναλήψεις.
Αν $P = P_1 + P_2 + P_3$, $Q = Q_1 + Q_2$ είναι ο χρόνος που απαιτείται για την εκτέλεση των εντός και εκτός του **for** εντολών αντίστοιχα, τότε ο συνολικός χρόνος είναι

$$P \times (n-2) + Q = P \times n + (Q - 2 \times P)$$

και επειδή, τα P, Q ανεξάρτητα του n συμπεραίνουμε ότι πρόκειται για *γραμμικό* αλγόριθμο $O(n)$.

Ανάλυση παραγόντων (συνεχ.)

Για περαιτέρω βελτίωση, χρειάζεται μείωση των επαναλήψεων.

Για κάθε παράγοντα m του n και ο $c = n/m$ είναι επίσης παράγοντας και χρειάζονται μόνο $\approx \sqrt{n}$ επαναλήψεις.

Υπάρχει επιπλέον ένα αμελητέο κόστος Q_3, ενώ το κόστος κάθε επανάληψης αυξάνεται (στη χειρότερη περίπτωση) κατά P_2+P_4 που είναι ανεξάρτητα του n.

```
% Αλγόριθμος  B
n = input( ' ? ' );          % Q1
m = 2;                       % Q2
k = floor ( sqrt (n ));      % Q3
while m <= k,
    if  rem(n, m) == 0,      % P1
        c = n/m;             % P4
        disp(m); disp(c );   % 2xP2
    end
    m = m + 1;               % P3
end
```

Στη χειρότερη περίπτωση:
$(P_1 + 2 \times P_2 + P_3 + P_4) \times \lfloor\sqrt{n}\rfloor + Q_1 + Q_2 + Q_3$ και επομένως ο νέος αλγόριθμος είναι $O(\sqrt{n})$ ως προς το χρόνο.

Κανονικοποιημένες μονάδες χρόνου

Το κόστος των αριθμητικών πράξεων ενός αλγορίθμου εκφράζεται παίρνοντας σαν μονάδα μέτρησης το χρόνο που χρειάζεται ο υπολογιστής για να κάνει μια πρόσθεση. Ενδεικτικές τιμές (εξαρτώνται από την αρχιτεκτονική του επεξεργαστή):

$+, -, \times$	$/$	$\sqrt{\cdot}$	floor	rem	λογικές πράξεις	input	disp
1	4	4	1	6	1	g	p

όπου το κόστος για **rem**(m,n) υπολογίζεται από $m - (m/n) \times n$.

Αναλυτική μέτρηση πράξεων

```
% Αλγόριθμος A
n = input(' ? ');          % g
m = 2;                     % 1
while m <= n−1,            % 2∗(n−2)+2
    if rem(n, m) == 0,     % 7∗(n−2)
        disp(m);           % p∗(n−2)
    end
    m = m + 1;             % 2∗(n−2)
end
```

$(11 + p) \times n - 2 \times p + g - 19$

```
% Αλγόριθμος B
n = input(' ? ');          % g
m = 2;                     % 1
k = floor(sqrt(n));        % 6
while m <= k,              % 1∗(k−1)+1
    if rem(n, m) == 0,     % 7∗(k−1)
        c = n/m;           % 5∗(k−1)
        disp(m);           % p∗(k−1)
        disp(c);           % p∗(k−1)
    end
    m = m + 1;             % 2∗(k−1)
end
```

$(15 + 2 \times p) \times \lfloor\sqrt{n}\rfloor - 7 - 2 \times p + g$

Παρατηρήσεις

Ο αλγόριθμος που προκύπτει από τον Β αντικαθιστώντας τις γραμμές

```
k = floor(sqrt(n));
while m <= k,
```

από την

```
while m <= floor(sqrt(n))
```

εκτελεί 6 επιπλέον μονάδες κόστους ανά επανάληψη και συνολικά εκτελεί

$$(21 + 2 \times p) \times \lfloor\sqrt{n}\rfloor - 7 - 2 \times p + g$$

«προσθέσεις» (πράξεις).

Μέτρηση πράξεων για διάφορα n

	Αλγόριθμος Α	Αλγόριθμος Β
n	$11n + p(n-2) + g - 19$	$15\lfloor\sqrt{n}\rfloor + 2p(\lfloor\sqrt{n}\rfloor - 1) + g - 7$
4	$25 + 2p + g$	$23 + 2p + g$
10	$91 + 8p + g$	$38 + 4p + g$
100	$1081 + 98p + g$	$143 + 18p + g$
500	$5481 + 498p + g$	$323 + 42p + g$
1000	$10981 + 998p + g$	$458 + 60p + g$

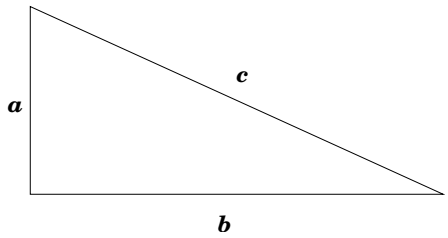
«Πυθαγόρειες τριάδες»

Να βρεθούν όλοι οι ακέραιοι a, b, c για τους οποίους ισχύει

$$a^2 + b^2 = c^2$$

και

$$a + b + c \leq p$$



ή ισοδύναμα, να βρεθούν όλα τα ορθογώνια τρίγωνα με ακέραια μήκη πλευρών, των οποίων η περίμετρος δεν υπερβαίνει μια μέγιστη τιμή p.

«Πυθαγόρειες τριάδες»- Αλγόριθμος 1

```
maxperimeter = input('Μέγιστη περίμετρος ? ');
for side1 = 1: maxperimeter,
   for side2 = 1: maxperimeter,
      for hyp = 1:maxperimeter,
         if side1^2+side2^2==hyp^2 & side1+side2+hyp<=maxperimeter,
            disp(side1 ); disp(side2 ); disp(hyp);
         end
      end
   end
end
```

Το μέγεθος του προβλήματος p είναι η τιμή της maxperimeter.

Η εντολή **if** εκτελείται maxperimeter^3 φορές και επομένως ο αλγόριθμος 1 είναι $O(p^3)$.

Παρατηρήσεις στον αλγόριθμο 1

- Εστω c ο μεγαλύτερος από τους 3 ακεραίους. Από τριγωνική ανισότητα:

$$c \leq a + b \Rightarrow 2c \leq a + b + c \Rightarrow c \leq p/2$$

- Ο αλγόριθμος 1 θα υπολογίσει π.χ. για τους αριθμούς $3 : 4 : 5$ όλους τους δυνατούς συνδυασμούς τους. Αυτό αποφεύγεται αν θεωρήσουμε ότι οι πλευρές είναι διατεταγμένες έτσι ώστε $a \leq b \leq c$.
- Η πιο εσωτερική εντολή επανάληψης μπορεί να τερματιστεί όταν το άθροισμα των πλευρών ξεπεράσει την περίμετρο.

«Πυθαγόρειες τριάδες»- Αλγόριθμος 2

```
maxperimeter = input('Μέγιστη περίμετρος ? ');
for side1 = 1: maxperimeter/2,
    for side2 = 1: maxperimeter/2,
        sumside = side1 + side2;
        hyp = side2;
        while hyp<=maxperimeter/2 & sumside+hyp<=maxperimeter,
            if side1^2+side2^2==hyp^2,
                disp(side1 ); disp(side2 ); disp(hyp);
            end
            hyp = hyp + 1;
        end
    end
end
```


Παρατηρήσεις στον αλγόριθμο 2

Μια λεπτομερής καταμέτρηση των πράξεων του αλγορίθμου 2, δίνει

$$\frac{p^3}{48} + \frac{p^2}{8} + \frac{p}{6}$$

δηλ. ο αλγόριθμος 2 είναι και πάλι $O(p^3)$.

Για περαιτέρω βελτίωση μπορούμε να εξαλείψουμε την πιο εσωτερική εντολή επανάληψεις παρατηρώντας ότι:

- Η hyp μπορεί αν υπολογιστεί από τις side1 και side2

  ```
  hyp = fix ( sqrt (side1^2+side2^2));
  ```

- Αφού υπολογίσουμε την hyp θα πρέπει να ελέγξουμε αν είναι ακέραιος αριθμός

  ```
  if hyp^2==side1^2+side2^2
  ```

«Πυθαγόρειες τριάδες»- Αλγόριθμος 3

```
maxperimeter = input('Μέγιστη περίμετρος ? ');
for side1 = 1: maxperimeter/2,
    side2 = side1;
    sidesq = 2*side1^2;
    while side2<=maxperimeter/2 & sidesq<=maxperimeter^2/4,
        sidesq = side1^2 + side2^2;
        hyp = fix ( sqrt (sidesq ));
        if hyp^2==sidesq,
            disp(side1 ); disp(side2 ); disp(hyp);
        end
        side2 = side2 + 1;
    end
end
```

Εκτελούνται $p^2/8 + p/4$ πράξεις, δηλ. ο αλγόριθμος 3 είναι $O(p^2)$.

Αναζήτηση και ταξινόμηση

Αναζήτηση (search)

Πρόβλημα: αναζήτηση της καταχώρησης key στη λίστα LIST

Μορφές αναζήτησης:

- Ύπαρξη συγκεκριμένης καταχώρησης στη λίστα.
- Θέση καταχώρησης στη λίστα.
- Συχνότητα εμφάνισης κάποιας καταχώρησης στη λίστα.

Η επιλογή αλγορίθμου αναζήτησης εξαρτάται από:

- Μέγεθος της λίστας.
- Διάταξη της λίστας.
- Πλήθος αναζητήσεων.

Η αποτελεσματικότητα της αναζήτησης καθορίζεται από τον αριθμό των *συγκρίσεων* που απαιτούνται σαν συνάρτηση του μεγέθους n της λίστας.

Γραμμική (linear) αναζήτηση

Τα στοιχεία της λίστας ελέγχονται το ένα μετά το άλλο στη σειρά.

- Αν η λίστα ΔΕΝ είναι διατεταγμένη: απαιτούνται $n/2$ συγκρίσεις *κατά μέσο όρο*, και n συγκρίσεις στη *χειρότερη περίπτωση* (όταν η καταχώρηση είναι το τελευταίο στοιχείο της λίστας).
- Σε διατεταγμένη λίστα απαιτούνται πάντα $n/2$ συγκρίσεις.

Αλγόριθμος γραμμικής αναζήτησης

```
function (pos, hit) = linear(key, LIST)
% hit — TRUE (1) αν υπάρχει key στη LIST
% pos — θέση του key στη LIST
n = max(size(LIST,1), size(LIST,2));   % πλήθος στοιχείων LIST
hit = 0;
loc = 1;
while loc<=n & ~hit,
    if key == LIST(loc),
        pos = loc;
        hit = 1;
    else
        loc = loc + 1;
    end
end
```


Δυαδική binary αναζήτηση

Προυπόθεση η *διάταξη* της λίστας. Η σύγκριση του key με το μεσαίο στοιχείο της λίστας κατατάσσει την αναζητούμενη καταχώρηση σε ένα από τα δύο (αριστερό, δεξιό) τμήματα (υπολίστες) της LIST. Η διαδικασία επαναλαμβάνεται για το νέο τμήμα, που περιέχει τα *μισά* στοιχεία του προηγούμενου.

Ο μέγιστος αριθμός συγκρίσεων για τη δυαδική αναζήτηση εκφράζει ουσιαστικά *πόσες φορές το μέγεθος της λίστας n διαιρείται με το 2*:

$$\lfloor \log_2 n \rfloor + 1$$

Αριθμός συγκρίσεων σε γραμμική και δυαδική αναζήτηση:

n	4	8	16	128	512	1024	1048576
γραμμική (μ.ο.)	2	4	8	64	256	512	524288
δυαδική (max)	3	4	5	8	10	11	21

Αλγόριθμος δυαδικής αναζήτησης

```
function (pos, hit) = binary(key, LIST)
n = max(size(LIST,1), size(LIST,2));
hit = 0;
left = 1; right = n;
while ~hit & left<=right,
   mid = floor((left+right)/2);
   if key == LIST(mid),
      pos = mid; hit = 1;
   else
      if key < LIST(mid), right = mid - 1; else left = mid + 1; end
   end
end
return
```


Ταξινόμηση (sort)

- Διάταξη των στοιχείων μιας λίστας με βάση κάποια σχέση μεγέθους μεταξύ τους.
- Οι τεχνικές ταξινόμησης είναι ανεξάρτητες από το είδος διάταξης. Εδώ ζητείται η διάταξη κάποιας λίστας σε *αύξουσα* σειρά.
- *Και* η αποτελεσματικότητα της ταξινόμησης καθορίζεται από τον αριθμό των *συγκρίσεων* που απαιτούνται, σαν συνάρτηση του μεγέθους n της λίστας.

Ταξινόμηση με επιλογή (selection)

Έστω OLD η αρχική λίστα.

1. Δημιουργείται μια 2η λίστα NEW ίσου μεγέθους, αρχικά κενή.
2. Επιλέγεται το μικρότερο στοιχείο small της OLD.
3. Το small μεταφέρεται στην πρώτη διαθέσιμη θέση της NEW.
4. Η θέση του small στην OLD 'αδειάζει' (αντικαθίσταται από ένα πολύ μεγάλο αριθμό).
5. Επανάληψη των βημάτων 2, 3, και 4 έως ότου 'αδειάσει' η OLD.

Αλγόριθμος selection sort

Η ταξινόμηση με επιλογή πραγματοποιείται σε n βήματα (περάσματα) με $n-1$ συγκρίσεις σε *κάθε* πέρασμα.

```
function NEW = select(OLD)
n = max(size(OLD,1), size(OLD,2));
NEW = zeros(1,n);
Inf = 1e9;                          % τιμή άδειας θέσης
for i = 1:n
   [small , pos] = min(OLD);
   NEW(i) = small;
   OLD(pos) = Inf;
end
return
```

Αλγόριθμος selection sort (συνεχ.)

Χρειαζόμαστε επίσης συνάρτηση **min** που επιστρέφει το μικρότερο στοιχείο small μιας λίστας LIST καθώς και τη θέση του pos στη λίστα.

```
function (small, pos) = min(LIST)
n = max(size(LIST,1), size(LIST,2));
small = LIST(1);
pos = 1;
for i = 2:n
    if LIST(i) < small,
        small = LIST(i);
        pos = i;
    end
end
return
```


Ταξινόμηση με εναλλαγή (exchange)

Κατηγορία αλγορίθμων ταξινόμησης: συγκρίσεις *ανά ζεύγη* και εναλλαγή των τιμών τους για *μερική* διάταξη. Η *ολική* διάταξη επιτυγχάνεται μετά από μια σειρά κατάλληλων *μερικών* διατάξεων. Διαφοροποιούνται στη συστηματοποίηση του ελέγχου των ζευγών.

Ο αλγόριθμος της φυσαλίδας (bubble sort). Σε κάθε πέρασμα το max στοιχείο που απομένει στη λίστα 'αναδύεται' στη θέση $n - i + 1$.

Π.χ. το πρώτο πέρασμα για τη λίστα: 7, 9, 8, 3, 4 είναι:

βήμα 1ο: 7 9 8 3 4

βήμα 2ο: 7 9 8 3 4

βήμα 3ο: 7 8 9 3 4

βήμα 4ο: 7 8 3 9 4

βήμα 5ο: 7 8 3 4 9

Ο αλγόριθμος της φυσαλίδας (bubble sort) - I

```
function LIST = bubble(LIST)
n = max(size(LIST ,1), size( LIST ,2));
% maxz — μέγιστος αριθμός ζευγών σε κάθε πέρασμα
for maxz = n−1:−1:1,
   for z = 1: maxz
      if LIST (z ) > LIST (z+1),
         temp = LIST(z );
         LIST (z ) = LIST (z+1);
         LIST (z+1) = temp;
      end
   end
end
```


Πρόβλημα με τον αλγόριθμο I

... είναι δυνατό να έχει επιτευχθεί διάταξη *προτού* ολοκληρωθούν όλα τα περάσματα. STOP μετά από το πρώτο πέρασμα στο οποίο ΔΕΝ έγινε καμμία εναλλαγή...

- Χρησιμοποιούμε μια *λογική* μεταβλητή, έστω xflag.
- *Αρχικά* σε κάθε πέρασμα i η xflag είναι 0 (FALSE). Αν γίνει έστω και μια εναλλαγή στη διάρκεια του i τότε xflag = 1.
- Αν μετά το πέρασμα i, η τιμή της xflag παραμείνει 0, δεν έχουν γίνει εναλλαγές στοιχείων στο i και επομένως η λίστα είναι ήδη ταξινομημένη από το πέρασμα $i-1$.

Ο αλγόριθμος της φυσαλίδας (bubble sort) - II

```
function LIST = bubble(LIST)
n = max(size(LIST ,1), size( LIST ,2));
maxz = n−1;
xflag = 1;      % =TRUE υποθέτουμε ότι θα γίνουν εναλλαγές
while xflag
   xflag = 0; % =FALSE ΔΕΝ έχουν γίνει ακόμα εναλλαγές στο πέρασμα
   for z = 1: maxz
      if LIST (z ) > LIST (z+1),
         temp = LIST(z ); LIST (z ) = LIST (z +1); LIST (z+1) = temp;
         xflag = 1;
      end
   end
   maxz = maxz − 1;
end
```


Ανάλυση του αλγορίθμου II

- Σε κάθε πέρασμα απαιτείται μια σύγκριση *λιγότερη* από το προηγούμενο πέρασμα.
- *Χειρότερη* περίπτωση όταν το ελάχιστο στοιχείο της λίστας βρίσκεται στην τελευταία θέση. Μέγιστος αριθμός συγκρίσεων:

$$(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}$$

- Όταν η λίστα είναι ήδη διατεταγμένη γίνονται $n-1$ συγκρίσεις.

Ταξινόμηση με εισαγωγή (insertion)

Έστω OLD η αρχική λίστα.

1. Δημιουργείται μια 2η λίστα NEW ίσου μεγέθους, αρχικά κενή.
2. Διατρέχονται τα στοιχεία της OLD με τη σειρά και εισάγονται κατάλληλα στη NEW, αφού ενδεχομένως μετακινηθούν κάποια από τα στοιχεία της NEW για να δημιουργήσουν θέση για την *εισαγωγή* του.

Ταξινόμηση με εισαγωγή (insertion) (συνεχ.)

Παράδειγμα:

Βήμα	OLD	NEW
1ο	7 9 8 3 4 7	7 - - - - -
2ο	7 9 8 3 4 7	7 9 - - - -
3ο	7 9 8 3 4 7	7 8 9 - - -
4ο	7 9 8 3 4 7	3 7 8 9 - -
5ο	7 9 8 3 4 7	3 4 7 8 9 -
6ο	7 9 8 3 4 7	3 4 7 7 8 9

Προγραμματιστικά, η OLD διατρέχεται από αριστερά προς τα δεξιά (δείκτης i), ενώ η NEW από δεξιά προς αριστερά (δείκτης j).

Αλγόριθμος insertion sort

```
function NEW = insert(OLD)
n = max(size(OLD,1),size(OLD,2));
NEW = zeros(1,n);
for i = 1:n
    hit = 0;          % FALSE
    j = i − 1;
    while j>0 & ~hit,
        if NEW(j)>OLD(i),
            NEW(j+1) = NEW(j);
            j = j − 1;
        else
            hit = 1;  % TRUE
        end
    end
    NEW(j+1) = OLD(i);
end
```


Ανάλυση αλγορίθμου εισαγωγής

- Σε κάθε βήμα i απαιτούνται κατά μ.ο. $(i-1)/2$ συγκρίσεις κατά τη *γραμμική αναζήτηση* στη NEW. Επομένως ο μ.ο. του αριθμού συγκρίσεων για *όλα* τα βήματα είναι:

$$(0+1+\cdots+(n-1))/2 = \frac{n(n-1)}{4}$$

- Χειρότερη περίπτωση όταν η λίστα είναι διατεταγμένη ανάποδα, με μέγιστο αριθμό συγκρίσεων:

$$\frac{n(n-1)}{2}$$

- Όταν η λίστα είναι ήδη διατεταγμένη απαιτούνται $n-1$ συγκρίσεις.

Αλγόριθμοι γραμμικής άλγεβρας

Εσωτερικό γινόμενο

Υπολογισμός $\sigma = x^T y$ με $x, y \in \mathbb{R}^n$

```
function (sigma, err) = inner(x, y)
err = 0;
n = max(size(x,1), size(x,2));
if n ~= max(size(y,1), size(y,2)),
   err = 1;
   return;
end
sigma = 0;
for i = 1:n
   sigma = sigma + x(i)*y(i);
end
```

Ο αλγόριθμος είναι $O(n)$.

Πολλαπλασιασμός πίνακα με διάνυσμα

Υπολογισμός $y = Ax$ με $A \in \mathbb{R}^{m \times n}, x \in \mathbb{R}^n, y \in \mathbb{R}^m$

```
function (y, err) = matvec(A, x)
m = size(A,1);  n = size(A,2);
err = 0;
if size(x,1) ~= n, err = 1; return; end
y = zeros(m, 1);
for i = 1:m
   sum = 0;
   for j = 1:n, sum = sum + A(i,j)*x(j); end
   y(i) = sum;
end
```

Ο αλγόριθμος είναι $O(m \times n)$.

Παράδειγμα πολ/σμού πίνακα με διάνυσμα

Κάθε μήνα 2 ανταγωνιστικές επιχειρήσεις Α και Β, διατηρούν ποσοστό από τους πελάτες τους t_{AA} και t_{BB} αντίστοιχα, ενώ μετακινούνται t_{AB} από Β προς Α και t_{BA} από Α προς Β. Ο συνολικός αριθμός πελατών είναι σταθερός. Αν και οι ρυθμοί μεταβολής παραμένουν σταθεροί, και σε κάποια χρονική στιγμή οι 2 εταιρείες έχουν x_A και x_B πελάτες αντίστοιχα, μετά από 1 μήνα οι πελάτες είναι:

$$\begin{bmatrix} t_{AA} & t_{AB} \\ t_{BA} & t_{BB} \end{bmatrix} \begin{bmatrix} x_A \\ x_B \end{bmatrix} = Tx$$

και ο αλγόριθμος για τον αριθμό πελατών μετά από p μήνες:

```
p = input('Μήνες ?');
for i = 1:p, (x, err) = matvec(T, x); end
if ~err, disp(x); end
```

(Τι μπορείτε να συμπεράνεται για τα στοιχεία του T?)

Γινόμενο πινάκων

Υπολογισμός: $C = AB$ με $A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times k}, C \in \mathbb{R}^{m\times k}$

```
function [C, err] = matmat(A, B)
m = size(A,1); n = size(A,2);  k = size(B,2);
err = 0;
if size(B,1)~=n, err=1; return; end
C = zeros(m,k);
for i = 1:m
   for j = 1:k
      sum = 0;
      for l = 1:n,  sum = sum + A(i,l)*B(l,j);  end
      C(i,j) = sum;
   end
end
```


Παράδειγμα γινομένου πινάκων

Ο πίνακας μετασχηματισμού T12 για ένα έτος, για τις επιχειρήσεις A και B του προηγούμενου παραδείγματος υπολογίζεται από

$$\text{T12} = T^{12}$$

```
T12 = matmat(T,T);
for p = 3:12
   T12 = matmat(T12,T);
end
```

Προσομοίωση

Προσομοίωση

Κατάστρωση μοντέλων συστημάτων και διεξαγωγή υπολογιστικών πειραμάτων με τα μοντέλα αυτά για:

- πρόβλεψη μελλοντικής συμπεριφοράς του συστήματος (π.χ. μετεωρολογικά μοντέλα)·
- διερεύνηση εναλλακτικών σεναρίων λειτουργίας του συστήματος (π.χ. οικονομικά μοντέλα).

Η προσωμοίωση περιγράφει τη μεταβολή του συστήματος με το χρόνο. Η *ποσοτική* περιγραφή του συστήματος σε κάποια χρονική στιγμή δίνει την *κατάσταση* του συστήματος, και οι μεταβλητές που την περιγράφουν ονομάζονται *μεταβλητές κατάστασης*.

Προσομοίωση (συνεχ.)

Βασικό πρόβλημα προσομοίωσης: πως οι μεταβλητές κατάστασεις μεταβάλλονται με το χρόνο.

Συνεχή μοντέλα: ζητείται η κατάσταση για κάθε t.

Διακριτά μοντέλα: κατάσταση σε διακριτές χρονικές στιγμές $(t + \delta t)$.

Τα συνεχή μοντέλα προσεγγίζονται από διακριτά.

Οι κανόνες μεταβολής της κατάστασης μπορεί να δίνουν:

- προκαθορισμένη (νομοτελειακή) εξέλιξη του συστήματος (η νέα κατάσταση καθορίζεται πλήρως από τις προηγούμενες).
- πιθανοτική (στοχαστική) εξέλιξη (από διαφορετικές δυνατές νέες καταστάσεις επιλέγεται μια, πιθανοτικά).

Δομή αλγορίθμων προσομοίωσης

1. **input** παραμέτρους προσομοίωσης και αρχικές καταστάσσεις
2. **while** δεν έχει επιτευχθεί το κριτήριο τερματισμού
   (α) $t \leftarrow t + \delta t$
   (β) Αλλαγή στις μεταβλητές κατάστασης — νέα κατάσταση
   end
3. Συμπεράσματα για συμπεριφορά του συστήματος

Το βήμα 2(β) είναι το σημαντικότερο και συνήθως απαιτεί πολλούς και πολύπλοκους υπολογισμούς.

Στο 3ο βήμα η συμπεριφορά του συστήματος περιγράφεται συνοψίζοντας σημαντικές λειτπυργίες είτε με κάποιο πίνακα είτε γραφικά.

Παράδειγμα επένδυσης

Μισθωτός με ετήσιο εισόδημα s και ετήσια αύξηση μισθού $100a\%$ επενδύει κάθε χρόνο το $100\beta\%$ του προηγούμενου μισθού του με ετήσιο επιτόκιο $100\epsilon\%$. Ποιά η κατάσταση της επένδυσης σε 5, 10, 15, ..., 30 χρόνια?

Ο μισθός με το χρόνο:

$$s(t+1) = s(t) + as(t)$$

και η αποταμίευση p:

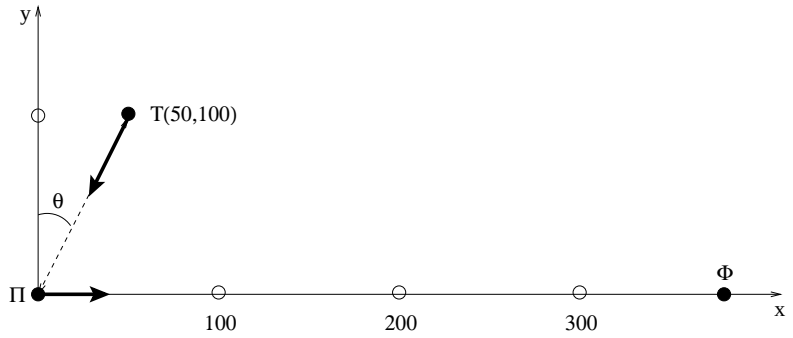
$$p(t+1) = p(t) + \epsilon p(t) + \beta s(t)$$

Επένδυση (συνεχ.)

```
s = input('Αρχικός μισθός ?');
p = input('Αρχική αποταμίευση ?');
alpha = input('Ετήσια αύξηση ?');
beta = input('Ποσοστό επένδυσης ?');
epsilon = input('Ετήσιο επιτόκιο ?');
for t = 0:30
    p = p + epsilon*p + beta*s;
    s = s + alpha*s;
    if rem(t,5)==0,
        disp(t); disp(s); disp(p);
    end
end
```

Παράδειγμα καταδίωξης

Πεζοπόρος (Π) κινείται προς φράχτη (Φ) σε απόσταση 400m και καταδιώκεται από ταύρο (Τ) με αρχική κατάσταση όπως στο σχήμα:



Αν οι ταχύτητες των Π και Τ είναι αντίστοιχα v_Π=8m/sec και v_T=8.5m/sec, ποιά η κατάληξη της καταδίωξης?

Καταδίωξη (συνεχ.)

Η θέση του Τ εξαρτάται από τη θέση του Π. Για μικρά δt θεωρούμε ότι ο Τ κινείται σε ευθεία προς τον Π. Οι κινήσεις τους καθορίζονται από τις συντεταγμένες τους. Σε κάθε χρονική στιγμή η μεταξύ τους απόσταση είναι:

$$s(t) = \sqrt{(x_T(t) - x_\Pi(t))^2 + y_T(t)^2}$$

η κίνηση του Π:

$$x_\Pi(t + \delta t) = x_\Pi(t) + v_\Pi * \delta t$$

και η κίνηση του Τ:

$$x_T(t + \delta t) = x_T(t) + \delta t * v_T * \sin\theta = x_T(t) + \delta t * v_T * \frac{x_\Pi(t) - x_T(t)}{s(t)}$$

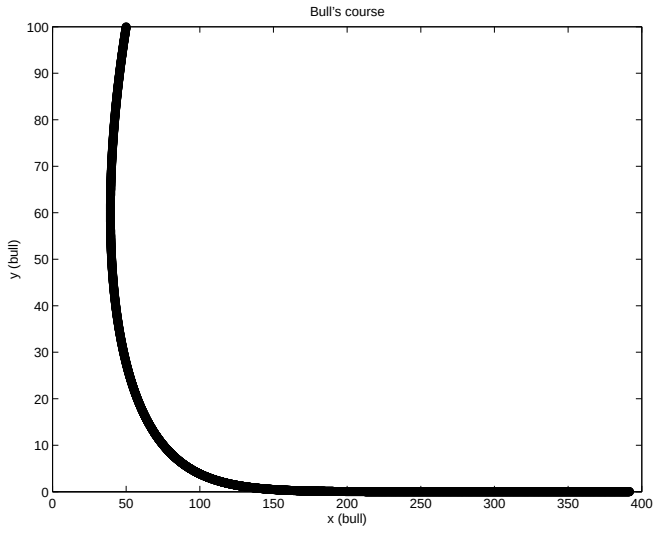
$$y_T(t + \delta t) = y_T(t) + \delta t * v_T * \cos\theta = y_T(t) + \delta t * v_T * \frac{-y_T(t)}{s(t)}$$

Αλγόριθμος καταδίωξης

```
% Αρχικές θέσεις ταύρου, οδοιπόρου και φράχτη
xb = input('x ταύρου ?'); yb = input('y ταύρου ?');
xh = input('x πεζοπόρου ?'); xf = input('x φράχτη ?');
dt = input('dt?' );   vb = input('v ταύρου ?');   vh = input('v πεζοπόρου ?');
t = 0;   s = sqrt((xb−xh)^2+yb^2);
while xh<xf & s>1
    t = t + dt;
    xb = xb + (xh−xb)*vb*dt/s; yb = yb − yb*vb*dt/s;
    xh = xh + vh*dt;
    s = sqrt((xb−xh)^2+yb^2);
end
 if xh >= xf ,
    disp('Ο Π σώθηκε '); disp(s);
else
    disp('Ο Τ έφτασε τον Π μετά από '); disp(xh); disp('m')
end
```

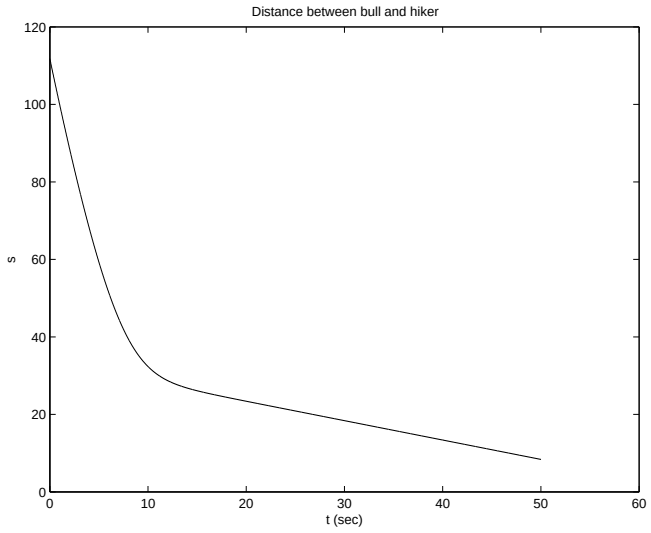

Η πορεία του ταύρου

Για ταχύτητες και αρχικές θέσεις όπως προηγουμένως:



Η απόσταση ταύρου - πεζοπόρου

Για ταχύτητες και αρχικές θέσεις όπως προηγουμένως:



Παράδειγμα δυναμικής

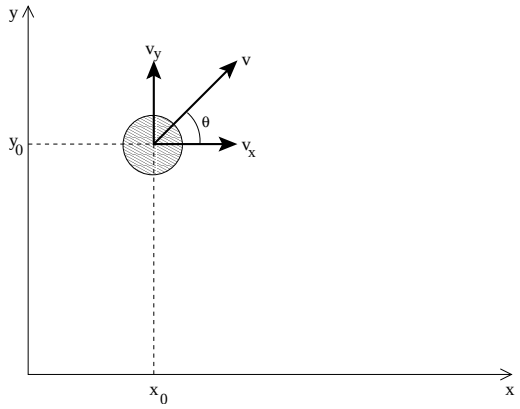
Κίνηση αντικειμένων υπό την επίδραση εξωτερικών δράσεων. Επιλέγεται σύστημα συντεταγμένων, χαράσεται το *διάγραμμα ελευθέρου σώματος* που έχει μάζα m, ταχύτητα v, επιτάχυνση γ και πάνω του ασκείται συνολική δύναμη F. Για την περιγραφή του μοντέλου χρησιμοποιούνται *για κάθε κατεύθυνση* x, y οι σχέσεις:

$$F_x = m\gamma_x, \qquad F_y = m\gamma_y$$
$$dv_x = \bar{\gamma}_x dt, \qquad dv_y = \bar{\gamma}_y dt$$
$$dx = \bar{v}_x dt, \qquad dy = \bar{v}_y dt.$$

όπου $\bar{v}, \bar{\gamma}$ οι *μέσες τιμές* για ταχύτητα και επιτάχυνση αντίστοιχα.

Δυναμική (συνεχ.)

Αναπήδηση μπάλλας σε επίπεδο δάπεδο, με αρχική ταχύτητα υ στη θέση (x_0, y_0) και επιτάχυνση βαρύτητας $g = 9.81\mathrm{m/sec}^2$.



Σε κάθε αναπήδηση η κατακόρυφη συνιστώσα της ταχύτητας ελαττώνεται κατά 10%.

Δυναμική (συνεχ.)

Εξισώσεις κίνησης:

$$\upsilon_x(t) = \sigma\tau\alpha\theta.$$

$$\upsilon_y(t + \delta t) = \upsilon_y(t) - g * \delta t$$

$$x(t + \delta t) = x(t) + \upsilon_x(t) * \delta t$$

$$y(t + \delta t) = y(t) + \frac{\upsilon_y(t) + \upsilon_y(t + \delta t)}{2} * \delta t$$

Για την αναπήδηση:

if $y(t + \delta t) \leq 0$,
$\upsilon_y(t + \delta t) \leftarrow |0.9 * \upsilon_y(t + \delta t)|$
end

Αλγόριθμος αναπήδησης

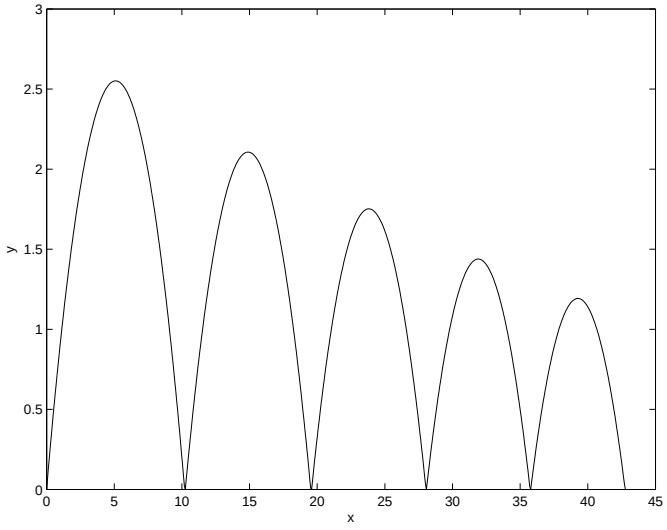
```
x = input('αρχικό  x ? ');
y = input('αρχικό  y ? ');
v = input('αρχική ταχύτητα (μέτρο) ? ');
theta = input('γωνία βολής ? ');
dt = input('dt?');
nmax = input('μέγιστος αριθμός αναπηδήσεων ? ');
g = −9.8;
vx = v∗cos(theta);
vy_old = v∗sin(theta);
```


Αλγόριθμος αναπήδησης (συνεχ.)

```
n = 0;
t = 0;
while n<nmax,
    t = t + dt;
    x = x + vx∗dt;
    vy_new = vy_old + g∗dt;
    y = y + (vy_old + vy_new)∗dt/2;
    vy_old = vy_new;
    if y<=0,
        vy_old = abs(0.9∗vy_old);
        y = 0;
        n = n + 1;
    end
end
```

Γραφική παράσταση αναπήδησης

Για $x_0 = y_0 = 0, \upsilon = 10\text{m/sec}, \vartheta = \pi/4, \text{nmax} = 5$ και $\delta t = 0.01$:



Δυναμική πληθυσμών

Αλληλεπίδραση πληθυσμών μεταξύ τους και με το περιβάλλον.

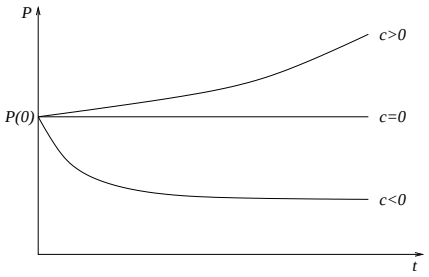
Ενα είδος - σταθερός ρυθμός ανάπτυξης

Ο πληθυσμός P σε χρόνο t:

$$P(t + \delta t) = P(t) + c * P(t) * \delta t = (1 + c * \delta t) * (P(t))$$

και για *ισαπέχοντα* χρονικά διαστήματα:

$$P(t) = P(0) * (1 + c * \delta t)^n, \qquad t = n * \delta t$$



Πληθυσμός με περιορισμένα αποθέματα τροφής

Το περιβάλλον μπορεί να υποστηρίξει μέχρι $P_{\max}$ άτομα. Ο ρυθμός ανάπτυξης προσεγγίζεται από:

$$c' = c * \frac{P_{\max} - P(t)}{P_{\max}}$$

που για $P(t) \to 0, \quad P(t) \to P_{\max}$ δίνει αντίστοιχα $c' = c$ και $c' = 0$.

Ο πληθυσμός στο χρόνο:

$$P(t + \delta t) = P(t) + c' * P(t) * \delta t$$

και με κανονικοποίηση $p(t) = P(t)/P_{\max}$ προκύπτει η *θεμελιώδης εξίσωση της οικολογίας*:

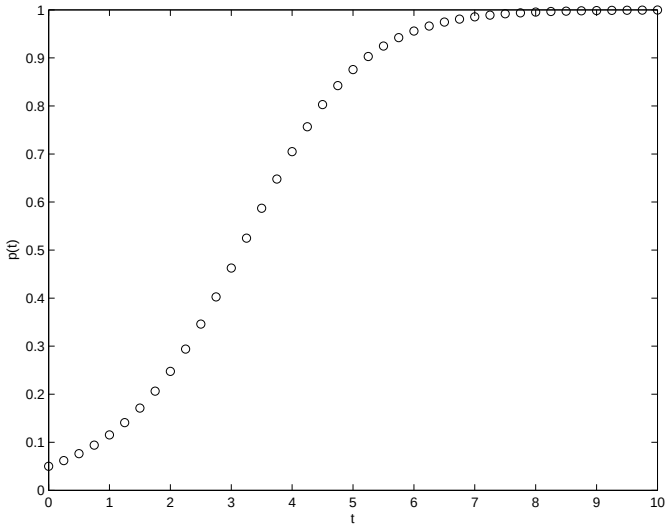
$$p(t + \delta t) = p(t) + c * (1 - p(t)) * p(t) * \delta t$$

Αλγόριθμος για θεμελιώδη εξ. οικολογίας

```
function (p, t) = eco(c, dt, p0, tlimit)
nsteps = floor(tlimit/dt)+1;
t = zeros(1,nsteps);
p = zeros(1,nsteps);
p(1) = p0;
for i=2:nsteps
   t(i) = t(i-1) + dt;
   p(i) = p(i-1) + c*(1-p(i-1))*p(i-1)*dt;
end
```

Γραφική παράσταση

Για $c = 1, p_0 = 0.05, \delta t = 0.25, t_{\text{limit}} = 10$:



Αρπακτικά και λεία

Ενας πληθυσμός (λεία) αποτελεί τροφή για έναν άλλο (αρπακτικά):

Λαγοί $R(t)$**:** αναπαράγονται 4 λαγοί το μήνα από κάθε ζεύγος ($c = 2$).

$$R(t + \delta t) = R(t) + 2 * \delta t * \frac{R_{\max} - R(t)}{R_{\max}} * R(t) \qquad (1)$$

Αλεπούδες $F(t)$**:** αυξάνουν κατά 4 το χρόνο ανά ζεύγος (για ένα μήνα $c = 1/12 = 0.167$) *όταν* $R(t) = R_{\max}$ *και* $F(t) <<<$. Ελαττώνονται ανάλογα. Όταν αντιστοιχούν 15 λαγοί ανά αλεπού, τότε οι αλεπούδες δεν αυξάνονται πλέον ($c' = 0$):

$$c' = \frac{R(t) - 15 * F(t)}{R_{\max}}$$

και για τον πληθυσμό των αλεπούδων:

$$F(t + \delta t) = F(t) + 0.167 * \delta t * \frac{R(t) - 15 * F(t)}{R_{\max}} * F(t)$$

Αρπακτικά και λεία (συνεχ.)

Αλλά οι λαγοί μειώνονται από τις αλεπούδες με ρυθμό ανάλογο του διαθέσιμου πληθυσμού τους, που αντιστοιχεί σε 15 λαγούς ανά αλεπού όταν $R(t) = R_{\max}$. Οπότε η (1) γίνεται:

$$R(t+\delta t) = R(t) + 2 * \delta t * \frac{R_{\max} - R(t)}{R_{\max}} * R(t) - 15 * \delta t * \frac{R(t)}{R_{\max}} * F(t)$$

Με κανονικοποίηση ως πρός $R_{\max}$, δηλ. $r(t) = R(t)/R_{\max}$ και $f(t) = F(t)/R_{\max}$ οι λαγοί:

$$r(t+\delta t) = r(t) + 2 * \delta t * r(t) * (1 - (r(t)) - 15 * \delta t * r(t) * f(t)$$

και οι αλεπούδες:

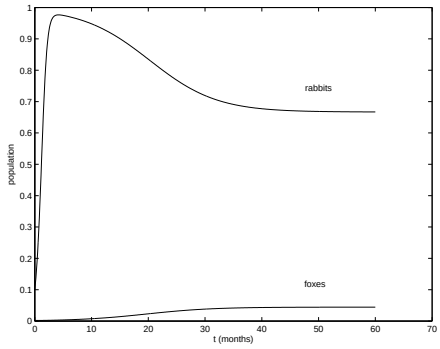
$$f((t+\delta t) = f(t) + 0.167 * \delta t * f(t) * (r(t) - 15 * f(t))$$

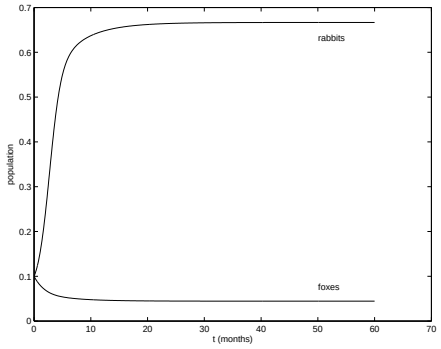
Αλγόριθμος αρπακτικών-λείας

```
function (r, f, t) = foxrabbit(tmax, nsteps, r0, f0)
% tmax   − διάρκεια παρατήρησης (σε μήνες)
% nsteps − βήματα χρόνου ανά μήνα
% r0, f0 − αρχικοί πληθυσμοί
t = zeros(1, tmax*nsteps+1);
f = zeros(1, tmax*nsteps+1); r = zeros(1, tmax*nsteps+1);
f(1) = f0; r(1) = r0; dt = 1/nsteps; i = 1;
for m = 1:tmax
   for n = 1:nsteps
      i = i + 1;
      t(i) = t(i−1) + dt;
      f(i) = f(i−1)+0.167*dt*f(i−1)*(r(i−1)−15*f(i−1));
      r(i) = r(i−1)+2*dt*r(i−1)*(1−r(i−1))−15*dt*r(i−1)*f(i−1);
      if r(i)<0, r(i)=0; end
   end
end
```

Γραφικές παραστάσεις

Για tmax = 60 μήνες και nsteps = 10:





Το συγκεκριμένο οικοσύστημα ισορροπεί ανέξάρτητα από τους αρχικούς πληθυσμούς.