Σειρά εκτέλεσης των πράξεων για τον υπολογισμό του γινομένου $A^T \cdot A \cdot A^T$

Περιεχόμενα

1 Εισαγωγή

Σε αυτό το εργαστήριο, διαπιστώνουμε πειραματικά τη διαφορά στον χρόνο υπολογισμού του γινομένου $A^T \cdot A \cdot A^T$, ανάλογα με τη σειρά εκτέλεσης των πράξεων. Οι δύο μέθοδοι πραγματοποίησης του υπολογισμού που θα διερευνήσουμε είναι οι εξής:

1. Εκτέλεση πρώτα του γινομένου $A^T \cdot A$ και ύστερα πολλαπλασιασμός του αποτελέσματος με τον A^T, δηλαδή $(A^T \cdot A) \cdot A^T$.

2. Εκτέλεση πρώτα του γινομένου $A \cdot A^T$ και ύστερα πολλαπλασιασμός του αποτελέσματος από τα αριστερά με A^T, δηλαδή $A^T \cdot (A \cdot A^T)$.

Το κύριο κίνητρο πίσω από τη σύγκριση των συγκεκριμένων μεθόδων είναι η σε πολλές περιπτώσεις σημαντική διαφορά στην υπολογιστική πολυπλοκότητα, που προκύπτει από τη θεωρητική ανάλυσή τους.

1.1 Θεωρητική προσέγγιση του προβλήματος

Έστω ένας πίνακας A διαστάσεων $n_1 \times n_2$. Υπολογίζοντας το γινόμενο με τις δύο μεθόδους, προκύπτει η ακόλουθη πολυπλοκότητα:

A) **Μέθοδος Ι.** Με αυτή τη μέθοδο, πραγματοποιείται αρχικά το γινόμενο των πινάκων A^T (διαστάσεων $n_2 \times n_1$) και A (διαστάσεων $n_1 \times n_2$). Επομένως, προκύπτει ένας πίνακας $n_2 \times n_2$, ύστερα από πράξεις πολυπλοκότητας $n_2 \cdot n_1 \cdot n_2 = {n_2}^2 \cdot n_1 \ flops$. Έπειτα, ο τετραγωνικός πίνακας $n_2 \times n_2$ πολλαπλασιάζεται με τον A^T, διάστασης $n_2 \times n_1$. Επομένως, ύστερα από την πραγματοποίηση του πολλαπλασιασμού πολυπλοκότητας $n_2 \cdot n_2 \cdot n_1 = {n_2}^2 \cdot n_1 \ flops$, προκύπτει ο τελικός πίνακας, διαστάσεων $n_2 \times n_1$. Τελικά, η συνολική υπολογιστική πολυπλοκότητα της Μεθόδου 1 είναι:

$$2{n_2}^2 n_1 \ flops$$

B) **Μέθοδος ΙΙ.** Με αυτή τη μέθοδο, πραγματοποιείται αρχικά το γινόμενο των πινάκων A (διαστάσεων $n_1 \times n_2$) και A^T (διαστάσεων $n_2 \times n_1$). Επομένως, προκύπτει ένας πίνακας $n_1 \times n_1$, ύστερα από πράξεις πολυπλοκότητας $n_1 \cdot n_2 \cdot n_1 = {n_1}^2 \cdot n_2 \ flops$. Έπειτα, ο τετραγωνικός πίνακας $n_1 \times n_1$ πολλαπλασιάζεται από τα αριστερά με τον A^T, διάστασης $n_2 \times n_1$. Επομένως, προκύπτει πίνακας $n_2 \times n_1$, ύστερα από την πραγματοποίηση του πολλαπλασιασμού πολυπλοκότητας $n_2 \cdot n_1 \cdot n_1 = {n_1}^2 \cdot n_2 \ flops$. Τελικά, η συνολική υπολογιστική πολυπλοκότητα της Μεθόδου 1 είναι:

$$2{n_1}^2 n_2 \ flops$$

Συμπεραίνουμε ότι ο λόγος της υπολογιστικής πολυπλοκότητας της πρώτης μεθόδου προς της δεύτερης είναι ίσος με $\frac{n_2}{n_1}$. Επομένως, αναμένουμε για πίνακες με σημαντικά μεγαλύτερο πλήθος γραμμών απ΄ ό,τι στηλών ($n_1 \gg n_2$) να είναι γρηγορότερη η Μέθοδος 1, ενώ για σημαντικά μεγαλύτερο πλήθος στηλών απ ό,τι γραμμών ($n_2 \gg n_1$) να είναι γρηγορότερη η Μέθοδος 2.

1.2 Σκοπός του εργαστηρίου και παράμετροι που θα εξεταστούν

Έχοντας κατανοήσει τη διαφορά των δύο μεθόδων σε θεωρητικό επίπεδο, μπορούμε να διατυπώσουμε ακριβέστερα το βασικό σκοπό του εργαστηρίου: να αποτυπώσουμε με πειραματικά δοδομένα την επίπτωση που έχει η διαφορά ως προς την υπολογιστική πολυπλοκότητα στον χρόνο εκτέλεσης του πολλαπλασιασμού $A^T \cdot A \cdot A^T$ (σε προγραμματιστικό περιβάλλον Julia).

Ωστόσο, μόλις ξεκινήσουμε το σχεδιασμό του εργαστηρίου, ανακύπτει το γενικό ερώτημα με τι είδους πίνακες θα εκτελεστεί ο πολλαπλασιασμός. Πιο συγκεκριμένα:

($P1$) *Τι σχήμα θα έχουν οι πίνακες με τους οποίους θα πραγματοποιηθούν οι πράξεις.* Όπως ήδη αναλύσαμε, το σχήμα του πίνακα (δηλαδή το αν έχει περισσότερες γραμμές ή στήλες, ή αν είναι κατά προσέγγιση τετραγωνικός) αναμένουμε να επηρεάζει σημαντικά τους χρόνους εκτέλεσης των δύο μεθόδων. Επομένως, θα πρέπει να δημιουργήσουμε δεδομένα για διάφορα σχήματα του πίνακα A και να τα συγκρίνουμε μεταξύ τους. Όλα αυτά, βέβαια, τα λέμε σε καθαρά διαισθητικό επίπεδο. Πώς θα μπορούσαμε, όμως, να υλοποιήσουμε στο εργαστήριό μας την έννοια του σχήματος του πίνακα; Ένας φυσικός τρόπος θα ήταν να θεωρήσουμε πίνακες που έχουν μεταξύ τους ακριβώς τα ίδια στοιχεία (ή εγγραφές), ωστόσο αυτά είναι διατεταγμένα σε διαφορετικό αριθμό γραμμών και στηλών.[1] Ακολουθεί ένα παράδειγμα:

$$\begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{pmatrix}, \begin{pmatrix} \alpha_1 & \alpha_2 \\ \alpha_3 & \alpha_4 \end{pmatrix}, \begin{pmatrix} \alpha_1 & \alpha_2 & \alpha_3 & \alpha_4 \end{pmatrix}$$

- ($P1\_1$) Τέλος, ένα υποερώτημα που προκύπτει σχετικά με το σχήμα των πινάκων είναι *τι συμβαίνει στην περίπτωση των τετραγωνικών πινάκων.* Όταν η υπολογιστική πολυπλοκότητα είναι ίδια, μήπως για κάποιον λόγο η μία μέθοδος είναι και πάλι γρηγορότερη από την άλλη;

($P2$) *Τι τύπου θα είναι τα δεδομένα με τα οποία θα πραγματοποιηθούν οι πράξεις.* Παραδείγματος χάριν, αν έχουμε πίνακες με εγγραφές ακέραιους αριθμούς, οι χρόνοι εκτέλεσης των δύο μεθόδων θα επηρεαστούν αν οι εγγραφές, αντί για τύπου Float, γίνουν τύπου Int; Επίσης, ο λόγος των χρόνων υπολογισμού για τις δύο μεθόδους επηρεάζεται αν, αντί για εγγραφές τύπου Float64 (διπλής ακρίβειας), χρησιμοποιήσουμε εγγραφές τύπου Float32 (απλής ακρίβειας);[2]

($P3$) *Πώς εξαρτάται ο χρόνος εκτέλεσης των δύο μεθόδων από το πλήθος των γραμμών και των στηλών του πίνακα*; Πιο συγκεκριμένα:

- ($P3\_1$) *Για πίνακες με λίγες στήλες, πόσο πιο γρήγορη είναι η μία μέθοδος από την άλλη για διάφορες τιμές του πλήθους των γραμμών*; (όπως είδαμε, σε αυτήν την περίπτωση αναμένουμε ο χρόνος της πρώτης μεθόδου να είναι μικρότερος). Ιδιαίτερο κίνητρο για τη μελέτη της συγκεκριμένης κατηγορίας πινάκων αποτελεί το γεγονός ότι τα Big Data αποθηκεύονται σε πίνακες αυτής της μορφής.

[1] Στην υλοποίηση αυτών των πινάκων θα μας βοηθήσει ιδιαίτερα η εντολή reshape της Julia.

[2] Με την παράμετρο ($P2$) ουσιαστικά προσπαθούμε να πραγματοποιήσουμε μια γενίκευση των αρχικών μας παρατηρήσεων για περισσότερους τύπους δεδομένων, διερευνώντας παράλληλα και πιθανές διαφοροποιήσεις που μπορεί να υπεισέλθουν από την αλλαγή του τύπου των στοιχείων.

($P3\_2$) *Για πίνακες με λίγες γραμμές, πόσο πιο γρήγορη είναι η μία μέθοδος από την άλλη συναρτήσει του πλήθους των στηλών*; Ιδιαίτερο κίνητρο για τη μελέτη της συγκεκριμένης κατηγορίας πινάκων αποτελεί το γεγονός ότι τα High Dimensional Data αποθηκεύονται σε πίνακες αυτής της μορφής.

Τέλος, πέρα από τις παραπάνω παράμετρους, οι οποίες θα μπορούσαν να επηρεάζουν τους χρόνους εκτέλεσης της πράξης με τις δύο μεθόδους, υπάρχει και ένας ακόμα βασικότερος παράγοντας τον οποίο θα πρέπει να λάβουμε υπόψη:

($P0$) *Πώς θα υλοποιηθούν οι δύο μέθοδοι υπολογισμού.* Για παράδειγμα, ο πολλαπλασιασμός θα μπορούσε να πραγματοποιηθεί σε ένα βήμα (κατ' ευθείαν εκτέλεση της πράξης $A^T \cdot A \cdot A^T$ με τις δύο μεθόδους), ή σε δύο βήματα (εκχώρηση πρώτα του ενός γινομένου σε μία μεταβλητή και σε δεύτερο χρόνο πολλαπλασιασμός του αποτελέσματος με τον τρίτο πίνακα).

Προφανώς για να δημιουργήσουμε όλα τα δεδομένα θα πρέπει να επιλέξουμε έναν τρόπο υλοποίησης των δύο μεθόδων, γι' αυτό και θα ξεκινήσουμε διερευνώντας την παράμετρο ($P0$).

2 Βέλτιστη υλοποίηση των μεθόδων υπολογισμού

Συναρτήσεις Η υλοποίηση που θα επιλέξουμε για το εργαστήριο θα είναι αυτή που θα δίνει τους μικρότερους χρόνους. Έτσι, θα μπορέσουμε να συγκρίνουμε μεταξύ τους τους βέλτιστους χρόνους των δύο μεθόδων. Οι επιλογές που θα εξετάσουμε είναι οι εξής:

1. Η **αυτόματη** ή **σε ένα βήμα** υλοποίηση των μεθόδων, σύμφωνα με την οποία αφήνουμε την Julia να διαχειριστεί τον τρόπο εκτέλεσης της πράξης:

```
function M1(A)
B = (A'*A)*A'
return
end
```

```
function M2(A)
B = A'*(A*A')
return
end
```

2. Η υλοποίηση των μεθόδων **με αρχικοποίηση**, στην οποία γίνεται αρχικοποίηση του πίνακα στον οποίο αποθηκεύεται το αποτέλεσμα του πρώτου πολλαπλασιασμού πινάκων:

```
function M1_init(A)
D = size(A)
B = zeros(D[1], D[2]) #Initialization
B = (A'*A)
B = B*A'
return
end
```

(και όμοια η M2_init(A))

3. Η υλοποίηση των μεθόδων **σε δύο βήματα**, όπου το αποτέλεσμα του πρώτου πολλαπλασιασμού εκχωρείται σε μια μεταβλητή, προτού πολλαπλασιαστεί με τον τρίτο πίνακα:

```
function M1_two_step(A)
B = (A'*A)
B = B*A'
return
end
```

(και όμοια η M2_two_step(A))

Για τη μέτρηση του χρόνου εκτέλεσης του υπολογισμού με τις δύο μεθόδους, χρησιμοποιήθηκαν συναρτήσεις όμοιες με την ακόλουθη:[3]

```
function time_M1_init(A)
@btime M1_init($A)
end
```

Τέλος, για την παραγωγή των δεδομένων χρησιμοποιήθηκαν συναρτήσεις όμοιες με την ακόλουθη (με μοναδική διαφοροποίηση τη συνάρτηση που μετράει το χρόνο εκτέλεσης):

[3]Για τη μέτρηση του χρόνου προτιμήθηκε η εντολή @btime, επειδή ορισμένοι από τους χρόνους ήταν ιδιαίτερα μικροί (της τάξης 10^{-4}s), επομένως η ακρίβεια της μέτρησης έπρεπε να μεγιστοποιηθεί.

```
function P_0_init(n, m)
s = n*10^m
A = [i for i in range(-s/2+1, s/2)]
for i = 0:m-1
A = reshape(A, n*10^i, 10^(m-i))
println(n*10^i, 'x', 10^(m-i))
time_M1_init(A)
time_M2_init(A)
println("———-")
A = reshape(A, 10^(i+1), n*10^(m-i-1))
println(10^(i+1), 'x', n*10^(m-i-1))
time_M1_init(A)
time_M2_init(A)
println("———-")
end
end
```

Η παραπάνω συνάρτηση εκτυπώνει τους χρόνους εκτέλεσης της πράξης με τις δύο μεθόδους για πίνακες διαστάσεων $n \cdot 10^i \times 10^{m-i}$, καθώς και $10^{i+1} \times n \cdot 10^{m-i-1}$, όπου το i παίρνει τιμές από 0 ως $m-1$. Πιο διαισθητικά, εκτυπώνει τους χρόνους εκτέλεσης για πίνακες διαφόρων σχημάτων, με $s = n \cdot 10^m$ εγγραφές (οι οποίες είναι θετικοί και αρνητικοί ακέραιοι αριθμοί).

Δεδομένα Η προηγούμενη συνάρτηση, καθώς και οι όμοιές της, εκτελέστηκαν με ορίσματα $n = 3$, $m = 4$. Για την καλύτερη κατανόηση του τρόπου λειτουργίας τους, παραθέτουμε τα ακριβή δεδομένα που εκτύπωσε μία εξ αυτών:

```
3×10000
  496.231 ms (6 allocations: 763.40 MiB)
  115.500 μs (5 allocations: 468.97 KiB)
———-
10×3000
  31.927 ms (6 allocations: 69.12 MiB)
  104.500 μs (5 allocations: 469.72 KiB)
———-
30×1000
  2.847 ms (6 allocations: 8.09 MiB)
  133.600 μs (5 allocations: 476.03 KiB)
———-
100×00
  389.100 μs (6 allocations: 1.14 MiB)
  200.900 μs (6 allocations: 547.02 KiB)
———-
300×3100
  194.200 μs (6 allocations: 547.02 KiB)
  376.000 μs (6 allocations: 1.14 MiB)
———-
```

```
1000×30
        158.500 μs (5 allocations: 476.03 KiB)
        3.086 ms (6 allocations: 8.09 MiB)
————-
3000×10
        111.300 μs (5 allocations: 469.72 KiB)
        33.138 ms (6 allocations: 69.12 MiB)
————-
10000×3
        122.500 μs (5 allocations: 468.97 KiB)
        537.612 ms (6 allocations: 763.40 MiB)
————-
```

Τα πλήρη δεδομένα παρουσιάζονται στους ακόλουθους πίνακες:

1. Η αυτόματη υλοποίηση έδωσε τους εξής χρόνους εκτέλεσης[4]:

Διαστάσεις:	**3×10000**	**10×3000**	**30×1000**	**100×300**	**300×100**	**1000×30**	**3000×10**	**10000×3**
Μέθοδος 1	**490.909 ms**	**30.431 ms**	**2.851 ms**	**381.300 μs**	**181.850 μs**	**145.500 μs**	**97.800 μs**	**106.250 μs**
Μέθοδος 2	**99.650 μs**	**91.500 μs**	**119.800 μs**	**187.100 μs**	**361.850 μs**	**3.088 ms**	**33.027 ms**	**562.711 ms**

2. Η υλοποίηση με αρχικοποίηση, αντίστοιχα, έδωσε τους εξής χρόνους εκτέλεσης:

	3×10000	**10×3000**	**30×1000**	**100×300**	**300×100**	**1000×30**	**3000×10**	**10000×3**
Μέθοδος 1	**492.992 ms**	**31.865 ms**	**2.825 ms**	**388.650 μs**	**194.200 μs**	**160.100 μs**	**110.650 μs**	**123.000 μs**
Μέθοδος 2	**114.750 μs**	**105.600 μs**	**133.800 μs**	**201.400 μs**	**374.450 μs**	**3.053 ms**	**31.413 ms**	**551.072 ms**

3. Τέλος, η υλοποίηση σε δύο βήματα έδωσε τους εξής χρόνους εκτέλεσης:

	3×10000	**10×3000**	**30×1000**	**100×300**	**300×100**	**1000×30**	**3000×10**	**10000×3**
Μέθοδος 1	**505.267 ms**	**31.886 ms**	**2.817 ms**	**377.450 μs**	**179.700 μs**	**145.850 μs**	**97.950 μs**	**106.800 μs**
Μέθοδος 2	**99.850 μs**	**91.150 μs**	**119.750 μs**	**184.600 μs**	**363.950 μs**	**3.018 ms**	**31.478 ms**	**551.545 ms**

Προκειμένου να συγκρίνουμε τις τρεις υλοποιήσεις, συγκρίνουμε τα δεδομένα για κάθε μέθοδο ξεχωριστά:

Α. *Για τη Μέθοδο 1*:

Διαστάσεις:	**3×10000**	**10×3000**	**30×1000**	**100×300**	**300×100**	**1000×30**	**3000×10**	**10000×3**
Αυτόματη	**490.909 ms**	**30.431 ms**	**2.851 ms**	**381.300 μs**	**181.850 μs**	**145.500 μs**	**97.800 μs**	**106.250 μs**
Με αρχικοποίηση	**492.992 ms**	**31.865 ms**	**2.825 ms**	**388.650 μs**	**194.200 μs**	**160.100 μs**	**110.650 μs**	**123.000 μs**
Σε δύο βήματα	**505.267 ms**	**31.886 ms**	**2.817 ms**	**377.450 μs**	**179.700 μs**	**145.850 μs**	**97.950 μs**	**106.800 μs**

Παρατηρούμε ότι, αν και οι σχετικές αποκλίσεις των υλοποιήσεων ανά δύο[5] είναι πολύ μικρές (δηλαδή οι τιμές για κάθε διάσταση είναι παρόμοιες), η αυτόματη υλοποίηση είναι καλύτερη σε πίνακες με πολύ λιγότερες γραμμές απ' ό,τι στήλες, ή το αντίστροφο, ενώ στους κατά προσέγγιση τετραγωνικούς υπερτερεί η υλοποίηση σε δύο βήματα.

Β. *Για τη Μέθοδο 2*:

Διαστάσεις:	**3×10000**	**10×3000**	**30×1000**	**100×300**	**300×100**	**1000×30**	**3000×10**	**10000×3**
Αυτόματη	**99.650 μs**	**91.500 μs**	**119.800 μs**	**187.100 μs**	**361.850 μs**	**3.088 ms**	**33.027 ms**	**562.711 ms**
Με αρχικοποίηση	**114.750 μs**	**105.600 μs**	**133.800 μs**	**201.400 μs**	**374.450 μs**	**3.053 ms**	**31.413 ms**	**551.072 ms**
Σε δύο βήματα	**99.850 μs**	**91.150 μs**	**119.750 μs**	**184.600 μs**	**363.950 μs**	**3.018 ms**	**31.478 ms**	**551.545 ms**

Αυτή τη φορά, δεν μπορεί να βγει κάποιο τόσο καθαρό συμπέρασμα όσον αφορά την αυτόματη και τη σε δύο βήματα υλοποίηση. Φαίνεται να έχουν παρόμοιους χρόνους, αν και με τη

[4] Στην πραγματικότητα, οι χρόνοι που παρουσιάζονται είναι μέσοι όροι δύο εκτελέσεων.

[5] Μπορούμε να τις ορίσουμε ως η απόλυτη διαφορά των χρόνων δύο υλοποιήσεων προς τον έναν από τους αφαιρούμενους χρόνους.

σε δύο βήματα υλοποίηση για τις περισσότερες διαστάσεις ο υπολογισμός πραγματοποιείται ελάχιστα ταχύτερα. Από την άλλη, η υλοποίηση με αρχικοποίηση είναι πιο αργή (και πάλι η διαφορά δεν είναι πολύ μεγάλη, αλλά είναι σημαντικά μεγαλύτερη από αυτή των άλλων δύο τρόπων υλοποίησης).

Τελικά, τόσο η αυτόματη υλοποίηση όσο και η σε δύο βήματα, είναι εύλογες επιλογές. Τα benchmarks που δίνουν είναι παρόμοια και, μολονότι η σε δύο βήματα είναι ελαφρώς καλύτερη για περισσότερες τιμές των διαστάσεων απ' ό,τι η αυτόματη, η τελευταία έχει δύο άλλα πλεονεκτήματα: είναι πιο απλή στην υλοποίησή της (αφού είναι ο τρόπος με τον οποίο εκτελεί τον υπολογισμό η Julia, χωρίς κάποια παρέμβαση από το χρήστη) και φαίνεται να υπερτερεί στην περίπτωση των παραλληλόγραμμων πινάκων[6], τους οποίους εξετάζουμε στην παράμετρο ($P3$). Για τους λόγους αυτούς τελικά επιλέχθηκε η αυτόματη υλοποίηση. Τυπικά, ίσως μια καλύτερη επιλογή θα ήταν να εφαρμοστεί ειδικά για τους τετραγωνικούς πίνακες η υλοποίηση σε δύο βήματα, όμως, επειδή η διαφορά στους χρόνους των δύο υλοποιήσεων για αυτούς τους πίνακες είναι πολύ μικρή, θα κρατήσουμε και σε αυτήν την περίπτωση την αυτόματη υλοποίηση.

[6] **Παραλληλόγραμμους** θα ονομάζουμε τους πίνακες με πολύ λιγότερες γραμμές απ' ό,τι στήλες, ή πολύ λιγότερες στήλες απ' ό,τι γραμμές.

3 Εύρεση της γρηγορότερης μεθόδου συναρτήσει του σχήματος του πίνακα

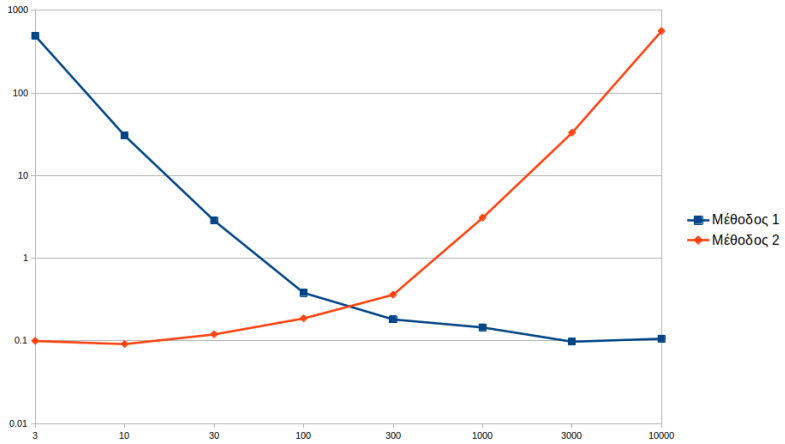
Έχοντας πλέον επιλέξει τον τρόπο υλοποίησης των δύο μεθόδων, μπορούμε να προχωρήσουμε στην ανάλυση της πρώτης παραμέτρου που αναμένουμε να επηρεάζει τον χρόνο εκτέλεσής τους. Αυτή είναι το σχήμα του πίνακα, το οποίο, όπως ήδη αναφέρθηκε, θα υλοποιηθεί μέσω πινάκων που έχουν τις ίδιες (θετικές και αρνητικές) ακέραιες εγγραφές, αλλά διαφορετικές διαστάσεις. Εάν το πλήθος των εγγραφών του πίνακα είναι s, τότε οι εγγραφές θα πάρουν τιμές από $-\frac{s}{2}+1$ ως $\frac{s}{2}$, σύμφωνα με την ακόλουθη συνάρτηση:

```
function P_1(n, m) s = n*10^m
A = [i for i in range(-s/2+1, s/2)]
for i = 0:m-1
A = reshape(A, n*10^i, 10^(m-i))
println(n*10^i, 'x', 10^(m-i))
time_M1(A)
time_M2(A)
println("———-")
A = reshape(A, 10^(i+1), n*10^(m-i-1))
println(10^(i+1), 'x', n*10^(m-i-1))
time_M1(A)
time_M2(A)
println("———-")
end
end
```

Η συνάρτηση αυτή είναι η ίδια που έδωσε τα δεδομένα για την αυτόματη υλοποίηση, στην προηγούμενη ενότητα. Μπορούμε, λοιπόν, να ανατρέξουμε στα ήδη υπάρχοντα αποτελέσματα για $n = 3$, $m = 4$:

Διαστάσεις:	**3×10000**	**10×3000**	**30×1000**	**100×300**	**300×100**	**1000×30**	**3000×10**	**10000×3**
Μέθοδος 1	**490.909 ms**	**30.431 ms**	**2.851 ms**	**381.300 μs**	**181.850 μs**	**145.500 μs**	**97.800 μs**	**106.250 μs**
Μέθοδος 2	**99.650 μs**	**91.500 μs**	**119.800 μs**	**187.100 μs**	**361.850 μs**	**3.088 ms**	**33.027 ms**	**562.711 ms**
Λόγος $\frac{M1}{M2}$	**4926.3322**	**332.5792**	**23.7980**	**2.0379**	**0.5026**	**0.0471**	**0.0030**	**0.0002**

Προκειμένου να οπτικοποιήσουμε τα παραπάνω δεδομένα, μπορούμε να δημιουργήσουμε το ακόλουθο διάγραμμα Χρόνου Υπολογισμού - Πλήθους γραμμών (η κλίμακα είναι λογαριθμική, καθώς τα δεδομένα του πίνακα μεταβάλλονται εκθετικά):



Παρατηρούμε ότι, σε συμφωνία με τη θεωρητική ανάλυση που πραγματοποιήσαμε, η Μέθοδος 2 έχει μικρότερους χρόνους για πίνακες με πολύ περισσότερες στήλες απ΄ ό,τι γραμμές, ενώ η Μέθοδος 1 υπερτερεί σε παραλληλόγραμμους πίνακες με πολλές γραμμές. Μάλιστα, όσο μεγαλώνει η διαφορά ανάμεσα στο πλήθος γραμμών και στηλών, η διαφορά στον χρόνο εκτέλεσης των δύο μεθόδων γίνεται εξαιρετικά σημαντική. Αντίθετα, σε κατά προσέγγιση τετραγωνικούς πίνακες, ο λόγος των χρόνων εκτέλεσης είναι κοντά στο 1.[7]

Ωστόσο, τα δεδομένα αυτά δεν επαρκούν για να μας οδηγήσουν σε ένα τόσο γενικό συμπέρασμα· αφορούν ορισμένες πολύ συγκεκριμένες περιπτώσεις πινάκων. Για να γενικεύσουμε το συμπέρασμά μας με μεγαλύτερο βαθμό βεβαιότητας, θα πρέπει να δοκιμάσουμε κι άλλες διαστάσεις πινάκων, δηλαδή κι άλλες τιμές των παραμέτρων n και m της συνάρτησης. Έτσι μόνο θα βεβαιωθούμε ότι οι παρατηρήσεις μας δεν στηρίζονται σε κάποια ιδιομορφία των συγκεκριμένων πινάκων.

Προσπαθώντας να μεταβάλουμε την τιμή του m (δηλαδή της τάξης του μέγιστου δυνατού πλήθους στηλών), αντιμετωπίζουμε ένα πρόβλημα: εάν το m πάρει τιμές μεγαλύτερες ίσες του 5, προκαλείται $OutOfMemoryError$. Δηλαδή τουλάχιστον μία από τις μεθόδους απαιτεί για τον υπολογισμό περισσότερη μνήμη απ΄ όση είναι δυνατό να διατεθεί.[8] Προκειμένου να εξηγήσουμε αυτήν την εξαιρετικά μεγάλη απαίτηση σε μνήμη, θα πρέπει να γυρίσουμε στη θεωρητική ανάλυση των αλγορίθμων. Εκεί, είχαμε παρατηρήσει (για έναν πίνακα A, διαστάσεων $n_1 \times n_2$) ότι, ύστερα από τον πρώτο πολλαπλασιασμό, και με τις δύο μεθόδους προκύπτει ένας τετραγωνικός πίνακας. Συγκεκριμένα, με τη Μέθοδο 1 προκύπτει ένας $n_2 \times n_2$ πίνακας, ενώ με τη Μέθοδο 2 προκύπτει ένας $n_1 \times n_1$ πίνακας. Στους παραλληλόγραμμους πίνακες, το n_1 ή το n_2 λαμβάνουν μεγάλες τιμές. Για παράδειγμα, αν χρησιμοποιήσουμε ως ορίσματα τις τιμές $n = 3$, $m = 5$, τότε ένας από τους πίνακες με τους οποίους θα εκτελεστεί η πράξη θα είναι διαστάσεων $n_1 = 100.000$ επί $n_2 = 3$. Εδώ, η μεγάλη τιμή του n_1 συνεπάγεται ότι η Μέθοδος 2 θα επιχειρίσει να δημιουργήσει έναν πίνακα διαστάσεων $10^5 \times 10^5$, ο οποίος έχει 10 δισεκατομμύρια εγγραφές! Αντίθετα, η Μέθοδος 1 θα δημιουργήσει, πέραν του τελικού γινομένου διαστάσεων $n_2 \times n_1$ (δηλαδή 100.000×3), και έναν τετραγωνικό πίνακα διαστάσεων $n_2 \times n_2$, δηλαδή 3×3 (ο οποίος έχει απαιτεί μόνο 9 θέσεις μνήμης). Γενικότερα, το πλήθος των θέσεων μνήμης που απαιτεί συνολικά η Μέθοδος 1 είναι $n_2 \cdot (n_2 + n_1)$, ενώ το πλήθος των θέσεων μνήμης που απαιτεί η Μέθοδος 2 είναι $n_1 \cdot (n_1 + n_2)$. Ο λόγος τους είναι $\frac{n_2}{n_1}$, δηλαδή ίδιος με το λόγο της υπολογιστικής πολυπλοκότητας. Επομένως, η πιο αποτελεσματική μέθοδος για κάθε πίνακα είναι και αυτή που απαιτεί τη λιγότερη μνήμη.

Προχωράμε, τώρα, στην παραγωγή δεδομένων που γενικεύουν τις αρχικές μας παρατηρήσεις, προσέχοντας να αποφύγουμε το $OutOfMemoryError$:

- Για $n = 3$, $m = 3$:

Διαστάσεις:	**3×1000**	**10×300**	**30×100**	**100×30**	**300×10**	**1000×3**
Μέθοδος 1	**2.315 ms**	**195.700 μs**	**91.200 μs**	**18.800 μs**	**10.100 μs**	**11.700 μs**
Μέθοδος 2	**10.900 μs**	**9.400 μs**	**18.500 μs**	**94.900 μs**	**199.100 μs**	**2.272 ms**
Λόγος	**212.3853**	**20.8191**	**4.9297**	**0.1981**	**0.0507**	**0.0051**

- Για $n = 5$, $m = 4$:

Διαστάσεις:	**5×10000**	**10×5000**	**50×1000**	**100×500**	**500×100**	**1000×50**	**5000×10**	**10000×5**
Μέθοδος 1	**470.605 ms**	**101.106 ms**	**3.129 ms**	**980.000 μs**	**245.600 μs**	**198.400 μs**	**133.000 μs**	**140.300 μs**
Μέθοδος 2	**132.500 μs**	**131.100 μs**	**176.800 μs**	**253.600 μs**	**939.400 μs**	**3.380 ms**	**119.737 ms**	**555.251 ms**
Λόγος	**3551.7358**	**771.2128**	**17.6980**	**3.8644**	**0.2614**	**0.0587**	**0.0011**	**0.0003**

[7] Όπως θα φανεί και από τους ακόλουθους πίνακες, όταν το πλήθος των γραμμών και των στηλών είναι της ίδιας τάξης (ως προς δυνάμεις του 10), ο λόγος παίρνει τιμές μεταξύ 0.1 και 10.

[8] Είναι πιθανό σε άλλους υπολογιστές το σφάλμα αυτό να προκύπτει και για $m = 4$, ειδικά αν το n είναι σχετικά μεγάλο. Στην περίπτωση αυτή, θα πρέπει να μειωθούν οι διαστάσεις των πινάκων.

- Για $n = 9$, $m = 4$:

Διαστάσεις:	9×10000	10×9000	90×1000	100×900	900×100	1000×90	9000×10	10000×9
Μέθοδος 1	502.472 ms	425.805 ms	3.701 ms	3.240 ms	378.200 μs	376.200 μs	240.700 μs	229.800 μs
Μέθοδος 2	214.100 μs	231.600 μs	362.400 μs	395.000 μs	3.105 ms	3.771 ms	505.987 ms	612.060 ms
Λόγος	2346.9033	1838.5363	10.2125	8.2025	0.1218	0.0998	0.0005	0.0004

Τα παραπάνω δεδομένα γενικεύουν όσα παρατηρήσαμε για την περίπτωση $n = 3$, $m = 4$. Μπορούμε πλέον να προχωρήσουμε στην εξέταση της οριακής περίπτωσης των τετραγωνικών πινάκων, όπου σε θεωρητικό επίπεδο δεν προκύπτει άμεσα κάποια διαφορά ανάμεσα στις δύο μεθόδους.

3.1 Σύγκριση για τετραγωνικούς πίνακες

Για τη σύγκριση των μεθόδων ως προς τετραγωνικούς πίνακες, επιλέχθηκαν διαστάσεις της μορφής 10^i και $3,16 \cdot 10^i$ (όπου το i λαμβάνει τιμές μεγαλύτερες ή ίσες του 2). Οι διαστάσεις αυτές προτιμούνται επειδή δημιουργούν τιμές που (κατά προσέγγιση) ισαπέχουν σε λογαριθμική κλίμακα. Αυτός είναι και ο λόγος που προηγουμένως δείχναμε μια προτίμηση για την τιμή $n = 3$.

Ακολουθεί η συνάρτηση που χρησιμοποιήθηκε για τη δημιουργία των δεδομένων. Οι παράμετροι n και m εδώ δηλώνουν την ελάχιστη και τη μέγιστη τιμή του εκθέτη i.

```
function P_1_1(n, m)
for i = n:m
for j = [1, 3.16]
temp = Int64(j*10^i)
s = (temp)^2
A = [i for i in range(-s/2+1, s/2)]
A = reshape(A, temp, temp)
println(temp)
time_M1(A)
time_M2(A)
println("———-")
end
end
end
```

Η παραπάνω συνάρτηση εκτελέστηκε για $n = 2, m = 4$ και έδωσε τους ακόλουθους χρόνους:

	100×100	316× 316	1.000× 1.000	3.160×3.160	10.000×10.000	31.600×31.600
Μέθοδος 1	116.400 μs	701.500 μs	14.557 ms	371.797 ms	10.445 s	407.903 s
Μέθοδος 2	114.700 μs	690.000 μs	14.860 ms	378.926 ms	10.428 s	531.285 s
Λόγος	1.0148	1.0167	0.9796	0.9812	1.0016	0.7678

Παρατηρούμε ότι, αν και για τις περισσότερες διαστάσεις οι μέθοδοι είναι πρακτικά ισοδύναμες, για τους μεγαλύτερους πίνακες η Μέθοδος 1 έχει μικρότερο χρόνο εκτέλεσης. Αυτή η διαφοροποίηση της τελευταίας τιμής των λόγων επαληθεύτηκε και με μία επανεκτέλεση του προγράμματος. Επομένως, για μεγάλους πίνακες, όπου ούτως ή άλλως ο χρόνος εκτέλεσης της πράξης και με τις δύο μεθόδους είναι αρκετά μεγάλος, είναι σημαντικό να προτιμάται η πρώτη μέθοδος.

4 Εύρεση της γρηγορότερης μεθόδου συναρτήσει του τύπου των εγγραφών

Έχοντας πλέον επιβεβαιώσει την αρχική μας εικασία (που διατυπώσαμε εξετάζοντας από θεωρητική σκοπιά τις δύο μεθόδους), προχωράμε στη γενίκευσή της και σε άλλους τύπους δεδομένων. Μέχρι τώρα, αφήναμε τον τύπο των εγγραφών του πίνακα να καθορίζεται από την Julia. Τροποποιώντας τη συνάρτηση P_1 ώστε να εκτυπώνει τον τύπο των εγγραφών, διαπιστώνουμε ότι ως τώρα όλα μας τα δεδομένα αφορούσαν πολλαπλασιασμό πινάκων με στοιχεία τύπου Float64. Προκειμένου να πραγματοποιήσουμε τη γενίκευση, θα πρέπει να τροποποιήσουμε την P_1 έτσι ώστε να μπορούμε να καθορίζουμε τον τύπο των εγγραφών:

```
function P_2(n, m, f=Float64)
s = n*10^m
A = [f(i) for i in range(-s/2+1, s/2)]
for i = 0:m-1
A = reshape(A, n*10^i, 10^(m-i))
println(n*10^i, 'x', 10^(m-i))
time_M1(A)
time_M2(A)
println("———-")
A = reshape(A, 10^(i+1), n*10^(m-i-1))
println(10^(i+1), 'x', n*10^(m-i-1))
time_M1(A)
time_M2(A)
println("———-")
end
end
```

Πρωτού προχωρήσουμε στη δημιουργία των δεδομένων, αξίζει να παρατηρήσουμε ότι, συγκρίνοντας τους χρόνους εκτέλεσης για διάφορους τύπους εγγραφών, θα μπορέσουμε να απαντήσουμε και στα εξής ενδιαφέροντα ερωτήματα σχετικά με τον τρόπο υπολογισμό του τριπλού γινομένου:

i) Εάν ο πίνακας έχει μόνο ακέραιες εγγραφές, η πράξη πραγματοποιείται γρηγορότερα όταν οι εγγραφές είναι τύπου Int ή τύπου Float;

ii) Εάν στην εκτέλεση της πράξης δεν μας ενδιαφέρει σε μεγάλο βαθμό η ακρίβεια, ο χρόνος εκτέλεσης θα μειωθεί σημαντικά αν χρησιμοποιήσουμε εγγραφές τύπου Float32 αντί για Float64;

Προχωράμε, τώρα, στην εκτέλεση της συνάρτησης P_2. Ως τιμές των πρώτων δύο ορισμάτων επιλέχθηκαν οι $n = 3$, $m = 4$ (οι τιμές αυτές ήταν σταθερές, έτσι ώστε οι πράξεις να πραγματοποιούνται επί της ουσίας μεταξύ των ίδιων πινάκων, απλά με διαφορετικούς τύπους εγγραφών). Το τρίτο όρισμα έλαβε τις τιμές Float32, Float16, Int64, Int32 και Int16. Δεδομένα τύπου Int8 δεν ήταν δυνατό να χρησιμοποιηθούν, καθώς δεν μπορούν να παρασταθούν όλες οι εγγραφές των πινάκων με 8 bits[9] (με 8 bits είναι δυνατό να παρασταθούν μόνο $2^8 = 256$ διαφορετικοί αριθμοί. Αξίζει να παρατηρήσουμε σε αυτό το σημείο την εντυπωσιακή αύξηση που παρουσιάζει το πλήθος των παραστήσιμων αριθμών αν διπλασιάσουμε τον αριθμό των bits: $2^{16} = 65.536$ αριθμοί).

[9] Η Julia έβγαλε το μήνυμα InexactError: Int8(-14999.0)

Float16	3×10000	10×3000	30×1000	100×300	300×100	1000×30	3000×10	10000×3
Μέθοδος 1	1.807 s	520.660 ms	197.155 ms	65.411 ms	22.039 ms	6.409 ms	1.937 ms	758.000 μs
Μέθοδος 2	543.100 μs	1.752 ms	5.898 ms	21.808 ms	66.130 ms	213.940 ms	567.678 ms	1.888 s
Λόγος $\frac{M1}{M2}$	3327.1957	297.1804	33.4274	2.9994	0.3333	0.0300	0.0034	0.0004

Float32	3×10000	10×3000	30×1000	100×300	300×100	1000×30	3000×10	10000×3
Μέθοδος 1	316.085 ms	18.573 ms	1.447 ms	259.800 μs	139.100 μs	107.700 μs	81.400 μs	102.100 μs
Μέθοδος 2	81.400 μs	75.900 μs	93.600 μs	141.300 μs	249.800 μs	1.534 ms	18.056 ms	362.237 ms
Λόγος $\frac{M1}{M2}$	3883.1081	244.7036	15.4594	1.8386	0.5568	0.0702	0.0045	0.0003

Float64	3×10000	10×3000	30×1000	100×300	300×100	1000×30	3000×10	10000×3
Μέθοδος 1	490.909 ms	30.431 ms	2.851 ms	381.300 μs	181.850 μs	145.500 μs	97.800 μs	106.250 μs
Μέθοδος 2	99.650 μs	91.500 μs	119.800 μs	187.100 μs	361.850 μs	3.088 ms	33.027 ms	562.711 ms
Λόγος $\frac{M1}{M2}$	4926.3322	332.5792	23.7980	2.0379	0.5026	0.0471	0.0030	0.0002

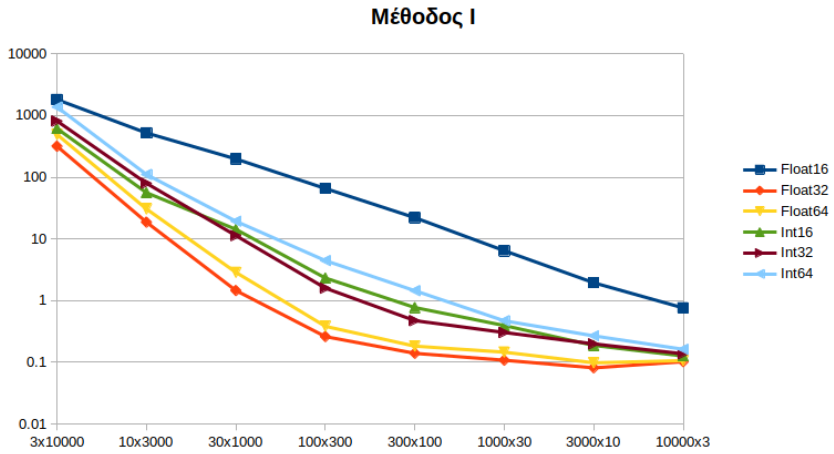
Int16	3×10000	10×3000	30×1000	100×300	300×100	1000×30	3000×10	10000×3
Μέθοδος 1	612.695 ms	55.960 ms	14.214 ms	2.308 ms	768.700 μs	392.400 μs	187.000 μs	126.900 μs
Μέθοδος 2	140.600 μs	205.800 μs	434.800 μs	789.900 μs	2.276 ms	12.721 ms	52.506 ms	282.851 ms
Λόγος $\frac{M1}{M2}$	4357.7169	271.9145	32.6909	2.9219	0.3377	0.0308	0.0036	0.0004

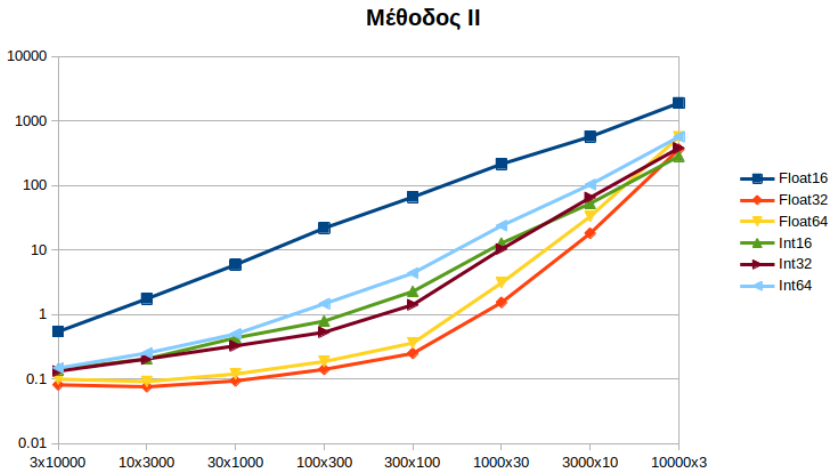
Int32	3×10000	10×3000	30×1000	100×300	300×100	1000×30	3000×10	10000×3
Μέθοδος 1	810.703 ms	79.422 ms	11.238 ms	1.585 ms	471.400 μs	302.700 μs	197.800 μs	136.700 μs
Μέθοδος 2	132.800 μs	205.200 μs	327.200 μs	531.600 μs	1.413 ms	10.320 ms	64.801 ms	379.642 ms
Λόγος $\frac{M1}{M2}$	6104.6913	387.0468	34.3460	2.9816	0.3336	0.0293	0.0031	0.0004

Int64	3×10000	10×3000	30×1000	100×300	300×100	1000×30	3000×10	10000×3
Μέθοδος 1	1.366 s	110.378 ms	19.265 ms	4.428 ms	1.441 ms	468.500 μs	267.200 μs	161.700 μs
Μέθοδος 2	148.200 μs	252.900 μs	499.100 μs	1.465 ms	4.369 ms	23.771 ms	102.761 ms	565.048 ms
Λόγος $\frac{M1}{M2}$	9217.2740	436.4492	38.5995	3.0225	0.3298	0.0197	0.0026	0.0003

Παρατηρούμε ότι, ανεξάρτητα από τον τύπο των εγγραφών, στους παραλληλόγραμμους πίνακες με πολύ περισσότερες στήλες απ΄ ό,τι γραμμές γρηγορότερη είναι η Μέθοδος 2, ενώ στους παραλληλόγραμμους πίνακες με πολύ περισσότερες γραμμές απ΄ ό,τι στήλες ισχύει το αντίθετο. Μάλιστα, όταν χρησιμοποιούνται εγγραφές τύπου Int32 ή Int64, η διαφορά ανάμεσα στις δύο μεθόδους είναι εντονότερη απ΄ ό,τι όταν χργσιμοποιούνται εγγραφές τύπου Float.

Επιπλέον, όσον αφορά τη σύγκριση των τύπων εγγραφών μεταξύ τους, οι Int16 , Int32, Int64 και Float16 είναι σημαντικά πιο αργοί από τους Float32 ή Float64. Επομένως, σε γενικές γραμμές οι εγγραφές τύπου Float θα πρέπει να προτιμούνται σε σχέση με τις εγγραφές τύπου Int (εξαίρεση αποτελεί το ζεύγος Int16 - Float16, το οποίο όμως, όπως φαίνεται από τους ιδιαίτερα μεγάλους χρόνους εκτέλεσης, θα πρέπει ούτως ή άλλως να αποφεύγεται). Τέλος, αξίζει να παρατηρήσουμε και ότι οι πράξεις με εγγραφές τύπου Float32 είναι ελαφρώς γρηγορότερες από τις πράξεις με εγγραφές τύπου Float64. Μάλιστα, η ποσοστιαία βελτίωση του χρόνου που υπεισέρχεται από τη χρήση δεδομένων Float32 αντί για Float64 γίνεται σημαντική (αγγίζοντας το 50%) όταν έχει επιλεχθεί η πιο αργή μέθοδος υπολογισμού. Ακολουθεί η γραφική σύγκριση των χρόνων ως προς τον τύπο των εγγραφών του πίνακα, για τις Μεθόδους 1 και 2:


Μέθοδος I
10000
1000
100
10
1
0.1
0.01
3x10000
10x3000
30x1000
100x300
300x100
1000x30
3000x10
10000x3
Float16
Float32
Float64
Int16
Int32
Int64


Μέθοδος II
10000
1000
100
10
1
0.1
0.01
3x10000
10x3000
30x1000
100x300
300x100
1000x30
3000x10
10000x3
Float16
Float32
Float64
Int16
Int32
Int64

5 Σύγκριση για πίνακες με πολλές γραμμές ή πολλές στήλες

Θα ξεκινήσουμε διερευνώντας τους πίνακες με πολύ περισσότερες γραμμές απ΄ ό,τι στήλες.[10] Ήδη γνωρίζουμε ότι σε αυτήν την κατηγορία πινάκων η Μέθοδος 1 είναι γρηγορότερη από τη Μέθοδο 2. Σκοπός μας είναι να διαπιστώσουμε πώς μεταβάλλεται ο λόγος των δύο μεθόδων, καθώς το πλήθος των γραμμών αυξάνεται. Τα πλήθη των στηλών που θα χρησιμοποιήσουμε είναι 3 (που αντιστοιχεί στην επιλογή 1 στην παρακάτω συνάρτηση), 5 (που αντιστοιχεί στην επιλογή 2) και 10 (που αντισοιχεί στην επιλογή 3). Το πλήθος των γραμμών, από την άλλη, λαμβάνει ως τιμές διαδοχικές δυνάμεις του 10, από 10^n μέχρι 10^m.

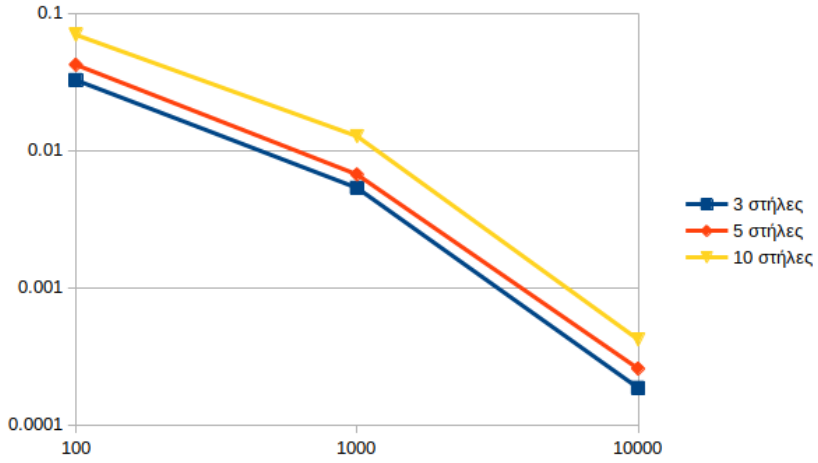
```
function P_3_Big_Data(n, m, mode)
L = [3, 5, 10]
c = L[mode]
for i = n:m
s = c*10^i
A = [Float64(i) for i in range(-s/2+1, s/2)]
A = reshape(A, 10^i, c)
println(10^i, 'x', c)
time_M1(A)
#println("1 done!") #Determines which method throws OutOfMemoryError
#If i≥5, M2(A) throws exception. Therefore, time_M2(A) needs to be commented out
time_M2(A)
println("———-")
end
```

Όπως δηλώνουν και τα σχόλια στη συνάρτηση, εάν το n πάρει τιμές μεγαλύτερες ή ίσες του 5, η Μέθοδος 2 προκαλεί $OutOfMemoryError$. Εάν θέλουμε να εκτελέσουμε παρόλα αυτά τη Μέθοδο 1, μπορούμε να μετατρέψουμε την εντολή που καλεί τη συνάρτηση time_M2 σε σχόλιο, ώστε να μην εκτελείται.

Ακολουθεί το διάγραμμα του Λόγου Χρόνου Εκτέλεσης των Μεθόδων 1 και 2 προς το Πλήθος των Γραμμών:[11]

[10] Υπενθυμίζουμε ότι τέτοιοι είναι οι πίνακες που χρησιμοποιούνται για την αποθήκευση των Big Data.

[11] Εδώ επιλέγουμε να παρουσιάσουμε τα δεδομένα διαγραμματικά, επειδή αυτό διευκολύνει την ταυτόχρονη σύγκριση των λόγων ως προς το πλήθος των γραμμών του πίνακα (επιλέγοντας μία από τις γραμμές του διαγράμματος και διατρέχοντάς την από τα αριστερά προς τα δεξιά) και ως προς το πλήθος των στηλών (συγκρίνοντας τις γραμμές του διαγράμματος μεταξύ τους).



Παρατηρούμε ότι, ανεξάρτητα από το πλήθος των στηλών, καθώς το πλήθος των γραμμών αυξάνεται, η Μέθοδος 1 γίνεται ολοένα ταχύτερη από τη μέθοδο 2 (δηλαδή ο λόγος μειώνεται). Επιπλέον, οι τιμές του λόγου δεν παρουσιάζουν σημαντικές διαφοροποιήσεις για τις διάφορες τιμές του πλήθους των στηλών, δηλαδή μια μικρή μεταβολή του πλήθους των στηλών δεν επηρεάζει σε μεγάλο βαθμό το πόσο ταχύτερη είναι η Μέθοδος 1 από τη Μέθοδο 2.

Προτού προχωρήσουμε στους παραλληλόγραμμους πίνακες με λίγες γραμμές, θα είχε ενδιαφέρον να δούμε τους χρόνους εκτέλεσης της πράξης από τη Μέθοδο 1, για πίνακες με περισσότερες από 10^4 στήλες. Για τον λόγο αυτό, τροποποιούμε, όπως προείπαμε, τη συνάρτηση ώστε να μην καλεί τη Μέθοδο 2, οπότε παράγονται τα ακόλουθα δεδομένα:

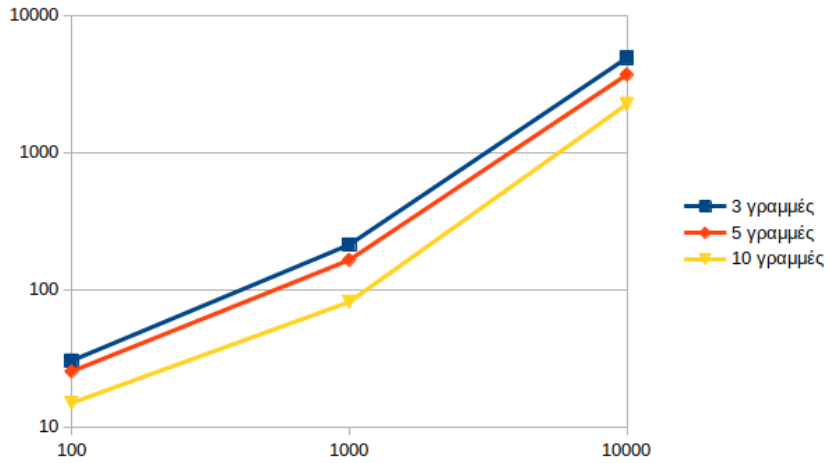
Γραμμές:	10^2	10^3	10^4	10^5	10^6	10^7	10^8
3 στήλες	**1.720 μs**	**11.500 μs**	**107.200 μs**	**946.600 μs**	**12.144 ms**	**123.828 ms**	**1.232 s**
5 στήλες	**2.211 μs**	**15.000 μs**	**141.500 μs**	**1.383 ms**	**21.338 ms**	**213.499 ms**	**2.251 s**
7 στήλες	**3.900 μs**	**31.200 μs**	**254.500 μs**	**4.606 ms**	**57.967 ms**	**594.739 ms**	**14.894 s**

Για αριθμό γραμμών μεγαλύτερο από 10^9, ο χρόνος υπολογισμού γίνεται πολύ μεγάλος (η εκτέλεση της συνάρτησης χρειάστηκε να διακοπεί). Επομένως, η Μέθοδος 1 μπορεί να χρησιμοποιηθεί για πίνακες με λίγες στήλες που έχουν πλήθος γραμμών το πολύ της τάξης του 10^8.

Επαναλαμβάνουμε, τώρα, την ίδια διαδικασία για πίνακες με πολύ περισσότερες στήλες απ΄ ό,τι γραμμές. Αρχικά, δημιουργούμε μια συνάρτηση:

```
function P_3_High_Dimensional_Data(n, m, mode)
L = [3, 5, 10]
r = L[mode]
for i = n:m
s = r*10^i
A = [Float64(i) for i in range(-s/2+1, s/2)]
A = reshape(A, r, 10^i)
println(r, 'x', 10^i)
#If i≥5, M1(A) throws exception. Therefore, time_M1(A) needs to be commented out
#time_M1(A)
#print("1 done!") #Determines which method throws OutOfMemoryError
time_M2(A)
println("———-")
end
end
```

Ύστερα, την εκτελούμε με ορίσματα 2 και 4 και παρουσιάζουμε γραφικά τους λόγους των δύο μεθόδων:



Αντίστοιχα με προηγουμένως βλέπουμε ότι, καθώς ο αριθμός των στηλών αυξάνεται, η Μέθοδος 2 γίνεται ολοένα πιο γρήγορη από τη Μέθοδο 1. Επιπλέον, ο λόγος των χρόνων των δύο μεθόδων δεν φαίνεται να επηρεάζεται ιδιαίτερα έντονα από την αλλαγή του αριθμού των γραμμών.

Τέλος, εστιάζουμε στη Μέθοδο 2 και εξετάζουμε τους χρόνους εκτέλεσης του πολλαπλασιασμού για μεγαλύτερες τιμές του πλήθους των στηλών. Έτσι, προκύπτουν τα ακόλουθα δεδομένα:

Στήλες:	10^2	10^3	10^4	10^5	10^6	10^7	10^8
3 γραμμές	**1.640 μs**	**10.800 μs**	**98.800 μs**	**972.000 μs**	**10.976 ms**	**108.077 ms**	**1.103 s**
5 γραμμές	**1.980 μs**	**14.100 μs μs**	**131.800 μs**	**1.354 ms**	**14.963 ms**	**147.417 ms**	**1.589 s**
7 γραμμές	**3.638 μs**	**29.000 μs**	**219.600 μs**	**3.074 ms**	**30.649 ms**	**299.361 ms**	**15.066 s**

6 Συμπεράσματα - Προτάσεις

Στις προηγούμενες παραγράφους καταλήξαμε στα εξής συμπεράσματα, τα οποία οδηγούν σε προτάσεις για τον τρόπο υλοποίησης:

- Σε όλες τις περιπτώσεις, η αυτόματη υλοποίηση είναι μια καλή επιλογή, αν και η υλοποίηση σε δύο βήματα είναι ελάχιστα γρηγορότερη για τους κατά προσέγγιση τετραγωνικούς πίνακες. Επομένως, σε γενικές γραμμές προτείνεται ο τρόπος πραγματοποίησης του γινομένου να αφήνεται στη Julia, μέσω των εντολών (A′*A)*A′ και A′*(A*A′). Μόνο στην περίπτωση των πινάκων που είναι κατά προσέγγιση τετραγωνικοί, ειδικά εάν είναι σημαντική η οποιαδήποτε (έστω και μικρή) μείωση του χρόνου υπολογισμού, προτείνεται η σε δύο βήματα υλοποίηση.

- Όπως περιμέναμε και από τη θεωρητική ανάλυση των Μεθόδων, η Μέθοδος 1 έχει μικρότερους χρόνους για πίνακες με περισσότερες γραμμές απ΄ ό,τι στήλες, ενώ η Μέθοδος 2 θα πρέπει να προτιμάται σε πίνακες με περισσότερες στήλες απ΄ ό,τι γραμμές. Για μεγάλους τετραγωνικούς πίνακες, τέλος, φαίνεται να υπερισχύει η Μέθοδος 1. Επομένως, αν και για τους περισσότερους τετραγωνικούς πίνακες με τους οποίους εκτελέστηκε το γινόμενο οι δύο Μέθοδοι ήταν πρακτικά ισοδύναμες, για τη συγκεκριμένη κατηγορία πινάκων προτείνεται η πρώτη μέθοδος.

- Για πίνακες τύπου Big Data, δηλαδή πίνακες με πολύ λιγότερες στήλες απ΄ ό,τι γραμμές, όσο μεγαλύτερο γίνεται το πλήθος των γραμμών, τόσο γρηγορότερη γίνεται η Μέθοδος 1 από τη Μέθοδο 2. Μάλιστα, η Μέθοδος 2 προκάλεσε $OutOfMemoryError$ για πίνακες με πλήθος γραμμών της τάξης του 10^5, ενώ με τη Μέθοδο 1 ο υπολογισμός πραγματοποιούνταν σε σχετικά μικρό χρόνο (λιγότερο από $20s$) για πίνακες με έως και 10^8 γραμμές.

- Από την άλλη, για πίνακες τύπου High Dimensional Data, δηλαδή πίνακες με πολύ περισσότερες στήλες απ΄ ό,τι γραμμές, ισχύει το αντίθετο: η Μέθοδος 2 είναι πολύ ταχύτερη (με τον λόγο των χρόνων των δύο μεθόδων να παίρνει ολοένα μεγαλύτερες τιμές καθώς αυξάνεται το πλήθος των στηλών) και πραγματοποιεί τον υπολογισμό σε μικρό χρόνο για πίνακες με 10^8 ή λιγότερες στήλες, ενώ η Μέθοδος 1 προκαλεί $OutOfMemoryError$ για πίνακες με 10^5 στήλες. Συνεπώς, αν και για πίνακες με Big Data είναι ιδιαίτερα σημαντικό να προτιμάται η Μέθοδος 1, για πίνακες με High Dimensional Data είναι εξίσου σημαντικό να προτιμάται η Μέθοδος 2.

- Όσον αφορά των τύπο των εγγραφών του πίνακα, συνιστάται να χρησιμοποιούνται δεδομένα τύπου Float, ακόμα και αν όλες οι τιμές είναι ακέραιοι αριθμοί (όπως είδαμε, η εκτέλεση της πράξης με εγγραφές τύπου Int είναι πιο αργή). Επιπλέον, τα δεδομένα τύπου Float32 θα πρέπει να προτιμώνται επί των Float64 μόνο στις περιπτώσεις όπου η ακρίβεια του αποτελλέσματος της πράξης δεν είναι ιδιαίτερα σημαντική και είναι εξαιρετικά επιθυμητή η μείωση του χρόνου εκτέλεσης του γινομένου. Τέλος, αξίζει να σημειώσουμε ότι τα προηγούμενα συμπεράσματα ισχύουν ανεξάρτητα από τον τύπο των εγγραφών του πίνακα με τον οποίο πραγματοποιείται ο υπολογισμός.

7 Σημειώσεις

1. Η έκδοση της Julia που χρησιμοποιήθηκε ήταν η 1.7.1.
2. Το παρόν έγγραφο δημιουργήθηκε με το λογισμικό LaTeX.
3. Για τη δημιουργία των διαγραμμάτων χρησιμοποιήθηκε το λογισμικό Libre Office Calc.
4. Για τη μορφοποίηση των δεδομένων, ώστε να είναι συμβατά με το Libre Office Calc και το περιβάλλον tabular του LaTeX, χρησιμοποιήθηκαν προγράμματα που γράφτηκαν σε γλώσσα Python (version 3.9.7).