Επιστημονικός Υπολογισμός Ι

Εργαστηριακή Άσκηση

Ονοματεπώνυμο: Μενύχτας Ευθύμιος **ΑΜ:** 5585 **Έτος Εισαγωγής:** 2012-2013

Χαρακτηριστικά Συστήματος:

Ο επεξεργαστής που χρησιμοποιήθηκε για την υλοποίηση της άσκησης είναι ο Intel Core i7 3770K. Είναι αρχιτεκτονικής Ivy Bridge, διαθέτει 4 φυσικούς πυρήνες και 8 λογικά νήματα με HyperThreading. Ο συγκεκριμένος επεξεργαστής έχει υπερχρονιστεί στα 4.8GHz. Η RAM του υπολογιστή είναι 8GB DDR3 στα 2133MHz. Η ιεραρχία της κρυφής μνήμης του επεξεργαστή φαίνεται στον παρακάτω πίνακα:

Level	Capacity	Associativity	Latency	Bandwidth (Read)	Bandwidth (Write)	Update Policy [1]
L1 Data	32KB/Core	8-way	0.9 ns	598.38 GB/s	299.62 GB/s	Write-back
L1 Code	32KB/Core	8-way	N/A	N/A	N/A	N/A
L2	256KB/Core	8-way	2.5 ns	311.42 GB/s	194.57 GB/s	Write-back
L3	8MB Shared	16-way	7.5 ns	225.58 GB/s	180.84 GB/s	Write-back

Συγκριτικά, ο χρόνος προσπέλασης της RAM μετρήθηκε στα 40.8 ns ενώ οι ταχύτητες ανάγνωσης και εγγραφής στα 32062 MB/s και 32643 MB/s αντίστοιχα, δηλαδή αρκετές τάξεις μεγέθους πιο κάτω σε σχέση με την CPU Cache.

Το FMA instruction set απουσιάζει καθώς υλοποιήθηκε στην επόμενη γενεά επεξεργαστών της Intel (Haswell).

Η συλλογή πληροφοριών για τον παραπάνω πίνακα έγινε με τα προγράμματα CPU-Z και AIDA64, ενώ το Update Policy αναφέρεται σε έναν οδηγό για Code Optimization της Intel. (βλ. [1])

Η έκδοση του MATLAB που χρησιμοποιήθηκε για την άσκηση είναι η R2016b.

Μέρος Β:

Για την υλοποίηση αυτού του μέρους σε όλους τους κώδικες δημιουργείται ένα τυχαίο διάνυσμα real_x (με χρήση της randn) που χρησιμοποιείται για την δημιουργία του διανύσματος b. Στην συνέχεια οι κώδικες προσπαθούν να προσεγγίσουν το real_x (μεταβλητές x) με αποδεκτό απόλυτο σφάλμα (μεταβλητές abs_err) μικρότερο από 10^-6.

Στην πρώτη υλοποίηση (iterative_refinement.m) σαν αρχική προσέγγιση θέτω το x{1} που αποτελείται από 1 παντού, και στην συνέχεια με κάθε επανάληψη αυτό πλησιάζει το real_x.

iterative_refinement.m

```
function [ k,x,abs_err ] = iterative_refinement( A,W,H,real_x,b )
    E = W * transpose(H);
    inv_A = inv(A);

    k = 1;
    x{k} = ones(size(real_x,1),1);
    abs_err{k} = norm(real_x - x{k}, Inf);

    while (abs_err{k} >= 10^-6)
        x{k+1} = inv_A * (-E * x{k} + b);
        abs_err{k+1} = norm(real_x - x{k+1}, Inf);

        k = k + 1;
    end

    if isnan(abs_err{k})
        fprintf('No convergence.\n');
    else
        fprintf('Convergence achieved.\n');
    end
end
```

Για να συγκλίνει η μέθοδος αυτή στο real_x πρέπει ο δείκτης κατάστασης του μητρώου A να μην είναι πολύ μεγάλος. Πιο συγκεκριμένα παρατηρήθηκε ότι για δ.κ. μέχρι και 5 η σύγκλιση είναι σίγουρη, ενώ για δ.κ. από 48 και πάνω η σύγκλιση δεν είναι σίγουρη, με τις πιθανότητες να μικραίνουν όσο ο δ.κ. μεγαλώνει.

Χρησιμοποιώντας τα μητρώα A,W και H που δίνονται στην εκφώνηση της άσκησης παρατηρήθηκαν τα εξής (βλέπε πίνακα παρακάτω) μετά από πεπερασμένο αριθμό εκτελέσεων. Λόγω της φύσης του προβλήματος οι χρόνοι εκτέλεσης και οι απαιτούμενες επαναλήψεις παίρνουν ένα μεγάλο εύρος τιμών. Τα περιεχόμενα του πίνακα είναι καθαρά ενδεικτικά.

Για h=2 ο δ.κ. του A είναι άσχημος σε κάθε περίπτωση, με αποτέλεσμα να μην έχει σίγουρη σύγκλιση η μέθοδος για κανένα n.

Στις υπόλοιπες περιπτώσεις η σύγκλιση είναι σίγουρη και ο αριθμός των απαιτούμενων επαναλήψεων μικραίνει καθώς μεγαλώνει το n.

	cond(A)	Επαναλήψεις	Χρόνος Εκτέλεσης	Πιθανότητα Σύγκλισης
h=2; n=10	48.3742	<10 έως χιλιάδες	~10^-4 έως ~0.14	~50%
h=2; n=100	4.1336e+03	<10 έως χιλιάδες	~10^-4 έως ~0.20	Σπάνια
h=2; n=1000	4.0610e+05	<10 έως ~500	~0.05 έως ~1.27	Σχεδόν Ποτέ
h=3; n=10	4.5503	4 έως 22	~10^-4	100%
h=3; n=100	4.9942	3 έως 9	~10^-4 έως ~10^-5	100%
h=3; n=1000	4.9999	3 έως 5	~0.06 έως ~0.07	100%
h=4; n=10	2.8443	4 έως 11	~10^-4 έως ~10^-6	100%
h=4; n=100	2.9981	3 έως 7	~10^-4 έως ~10^-5	100%
h=4; n=1000	3.0000	3 έως 4	~0.06	100%

Στη συνέχεια υλοποιήθηκε η μέθοδος επίλυσης με χρήση της φόρμουλας SMW σε διπλή ακρίβεια.

SMW_dp.m

```
function [ x,abs_err ] = SMW_dp( A,W,H,real_x,b )
    inv_A = inv(A);
    inv_AE = inv_A - (inv_A*W*transpose(H)*inv_A) / (1+transpose(H)*inv_A*W);

    x = inv_AE * b;
    abs_err = norm(real_x - x, Inf);
end
```

Οι αντίστοιχες παρατηρήσεις φαίνονται στον ακόλουθο πίνακα. Είναι φανερό πως η αριθμητική διπλής ακρίβειας οδηγεί σε πολύ μικρά σφάλματα αν και ο απαιτούμενος χρόνος εκτέλεσης για μεγάλα μητρώα είναι αισθητά μεγάλος.

	cond(A)	Απόλυτο Σφάλμα	Χρόνος Εκτέλεσης
h=2; n=10	48.3742	~10^-15	~3.4000e-05
h=2; n=100	4.1336e+03	~10^-13	~5.0000e-04
h=2; n=1000	4.0610e+05	~10^-11	~0.0550
h=3; n=10	4.5503	~10^-16	~3.4000e-05
h=3; n=100	4.9942	~10^-15	~5.0000e-04
h=3; n=1000	4.9999	~10^-14	~0.1300
h=4; n=10	2.8443	~10^-16	~3.4000e-05
h=4; n=100	2.9981	~10^-15	~5.0000e-04
h=4; n=1000	3.0000	~10^-14	~0.1450

Ακολουθεί ο κώδικας που συνδιάζει τη χρήση της φόρμουλας SMW σε μονή ακρίβεια με την επαναληπτική εκλέπτυνση.

SMW_sp_iter_ref.m

```
function [ k,x,abs_err ] = SMW_sp_iter_ref( A,W,H,real_x,b )
    AE = A + W * transpose(H);
    inv_A = inv(single(A));
    inv_AE = inv_A - (inv_A*W*transpose(H)*inv_A) / (1+transpose(H)*inv_A*W);

    k = 1;
    x{k} = double(inv_AE * b);
    abs_err{k} = norm(real_x - x{k}, Inf);

    while (abs_err{k} >= 10^-6)
       r{k+1} = b - AE*x{k};
       z{k+1} = double(inv_AE * r{k+1});
       x{k+1} = x{k} + z{k+1};
       abs_err{k+1} = norm(real_x - x{k+1}, Inf);

       k = k + 1;
    end

    if isnan(abs_err{k})
        fprintf('No convergence.\n');
    else
        fprintf('Convergence achieved.\n');
    end
end
```

Εκτελώντας τον παραπάνω κώδικα με κατώφλι σύγκλισης το 10^-6 για τους ίδιους συνδιασμούς των h και n όπως και πριν προκύπτει ο ακόλουθος πίνακας:

	cond(A)	Επαναλήψεις	Χρόνος Εκτέλεσης	Ακρίβεια Σύγκλισης
h=2; n=10	48.3742	0 έως 1	~1.0000e-04	~10^-7 έως ~10^-13
h=2; n=100	4.1336e+03	1	~5.5000e-04	~10^-7 έως ~10^-9
h=2; n=1000	4.0610e+05	2 έως 3	~0.0430	~10^-7 έως ~10^-9
h=3; n=10	4.5503	0	~8.0000e-05	~10^-7 έως ~10^-8
h=3; n=100	4.9942	0 έως 1	~7.0000e-04	~10^-7 έως ~10^-13
h=3; n=1000	4.9999	1	~0.1050	~10^-12
h=4; n=10	2.8443	0	~8.5000e-05	~10^-7 έως ~10^-8
h=4; n=100	2.9981	0 έως 1	~0.0010	~10^-7 έως ~10^-13
h=4; n=1000	3.0000	1	~0.0950	~10^-12

Σε σχέση με την SMW διπλής ακρίβειας ο χρόνος εκτέλεσης για μεγάλα μητρώα είναι αισθητά μικρότερος με ικανοποιητική ακρίβεια. Στα μικρά μητρώα παρατηρείται μια αύξηση στον χρόνο εκτέλεσης που πιθανώς οφείλεται στο overhead της όλης μεθόδου που είναι ασύμφορο.

Το script που χρησιμοποιήθηκε για την εκτέλεση και την χρονομέτρηση των τριών μεθόδων είναι το ακόλουθο:

meros2.m

```
clear;

h = 4;    % 2,3,4
n = 1000; % 10,100,1000

A = toeplitz([h,-1,zeros(1,n-2)]);
W = randn(n,1);
W = W/norm(W);
H = W(n:-1:1);
real_x = randn(n,1);
b = (A+W*transpose(H)) * real_x;

cond_A = cond(A);

f1 = @() iterative_refinement(A,W,H,real_x,b);
f2 = @() SMW_dp(A,W,H,real_x,b);
f3 = @() SMW_sp_iter_ref(A,W,H,real_x,b);

%t1 = timeit(f1);
%t2 = timeit(f2);
%t3 = timeit(f3);

%[k,x,abs_err] = iterative_refinement(A,W,H,real_x,b);
%[x,abs_err] = SMW_dp(A,W,H,real_x,b);
%[k,x,abs_err] = SMW_sp_iter_ref(A,W,H,real_x,b);
```

Μέρος Γ:

Το SHA-1 hash του ΑΜ μου (5585) είναι 9bcd6c8c398327684bae8be3c6df07ef9db45b6d. Τα τέσσερα πρώτα αριθμητικά ψηφία είναι τα 9683 επομένως χρησιμοποιείται το myHash = 9683.

Η συνάρτηση arrowNW υλοποιήθηκε ως εξής:

arrowNW.m

```
function [ A ] = arrowNW( T, n )
    m = size(T,1);
    D = T + m * eye(m);

    B = ones(n,1);
    B(1) = 0;
    B = ones(n) - B*transpose(B);
    B(1,1) = 0;

    A = kron(eye(n),D) + kron(B,T);
end
```

Αρχικά υπολογίζεται το μητρώο D σύμφωνα με τον τύπο που δίνεται. Στη συνέχεια δημιουργείται ένα μητρώο B που έχει 1 σε ολόκληρη την πρώτη γραμμή και πρώτη στήλη εκτός από την θέση (1,1) όπου έχει 0 όπως και παντού αλλού, δηλαδή όπου θέλουμε τα μπλοκ μητρώα T. Στη συνέχεια χρησιμοποιώντας την συνάρτηση kron δημιουργείται το μητρώο μορφής ΒΔ βέλους Α.

Ακολουθεί μια απεικόνιση του μητρώου A που προκύπτει από την παραπάνω συνάρτηση για m = 4 και n = 20 χρησιμοποιώντας την εντολή spy, καθώς και των μητρώων L και U που προκύπτουν από την εντολή [L,U] = lu(A) της MATLAB.

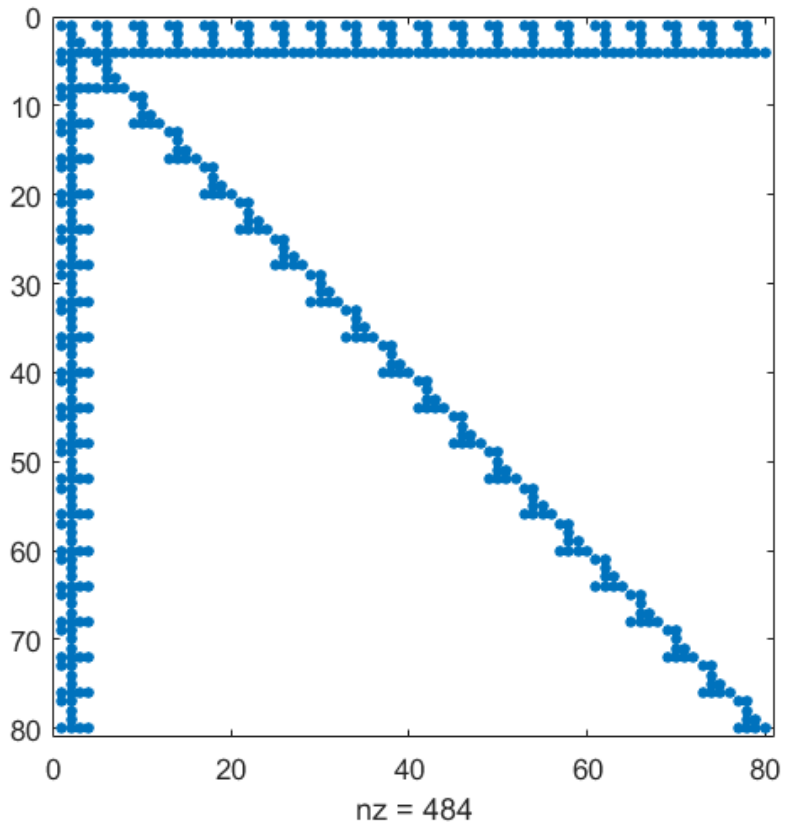

Illustration 3.1: spy(A)

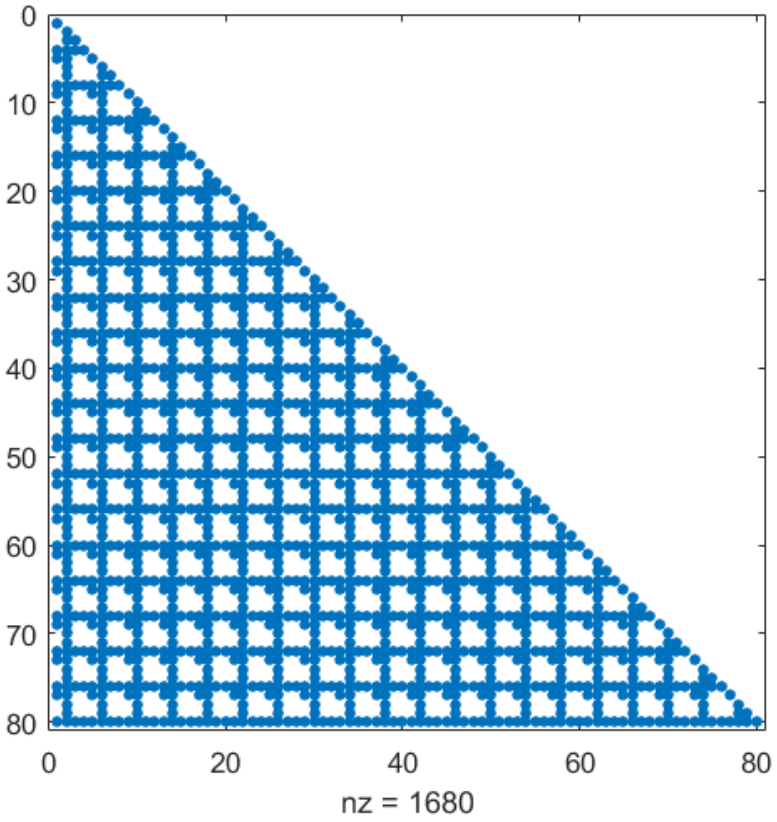

Illustration 3.2: spy(L)

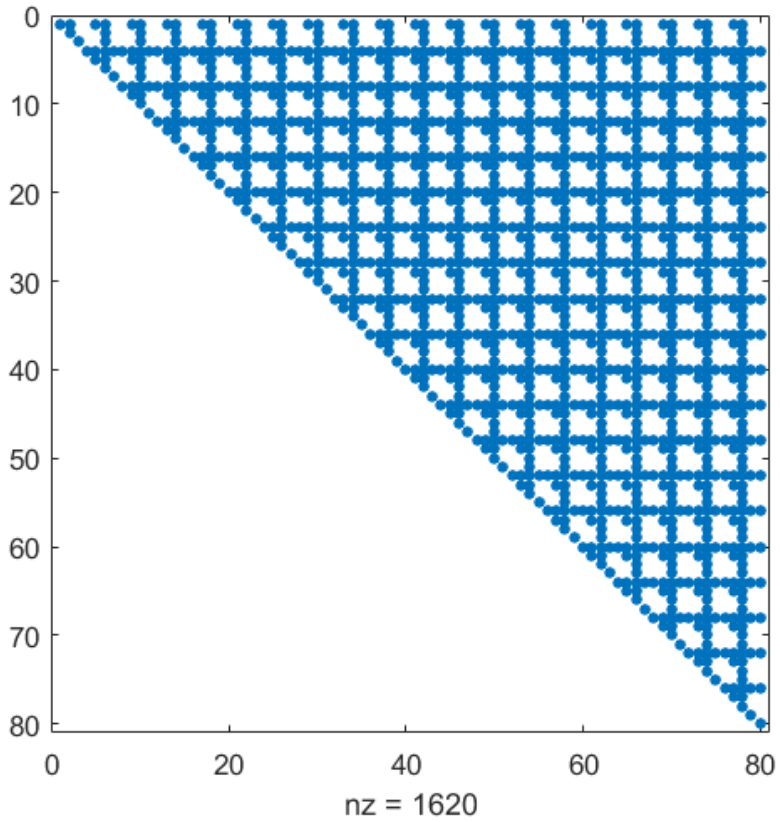

Illustration 3.3: spy(U)

Το γέμισμα, χρησιμοποιώντας τον τύπο που δίνεται, είναι 6.6529.

Για την μετατροπή του A από μορφή ΒΔ βέλους σε μορφή ΝΑ βέλους, χρησιμοποιείται το μητρώο μετάθεσης W που έχει 1 στην αντιδιαγώνιο και 0 οπουδήποτε αλλού. Αυτό δημιουργείται με τη χρήση της εντολής fliplr πάνω σε ένα ταυτοτικό μητρώο.

Ακολουθούν απεικονίσεις των νέων μητρώων B, LB, UB.

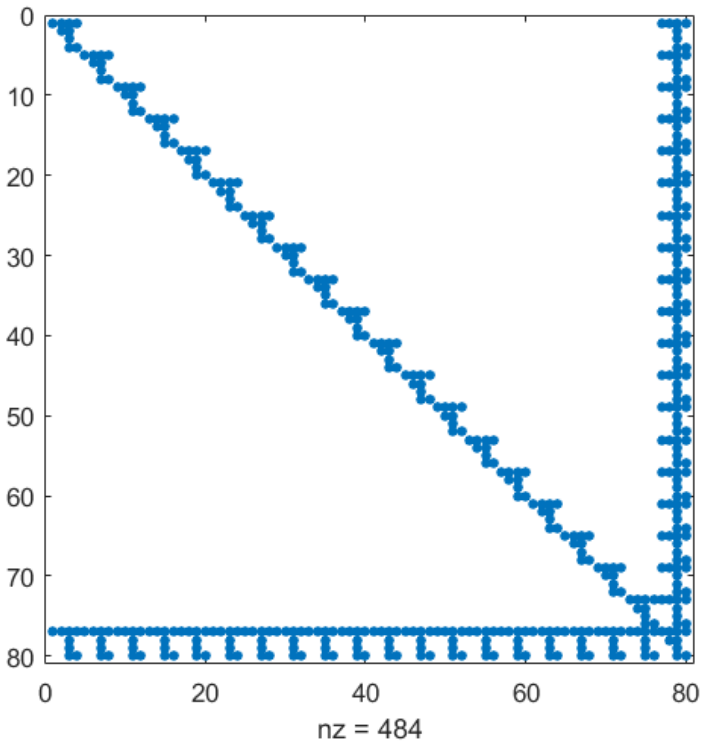

Illustration 3.4: spy(B)

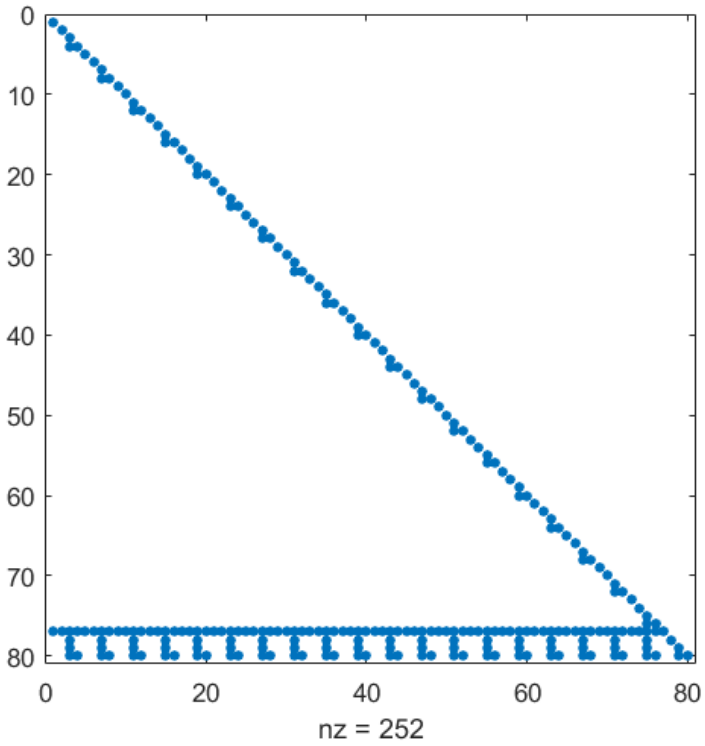

Illustration 3.5: spy(LB)

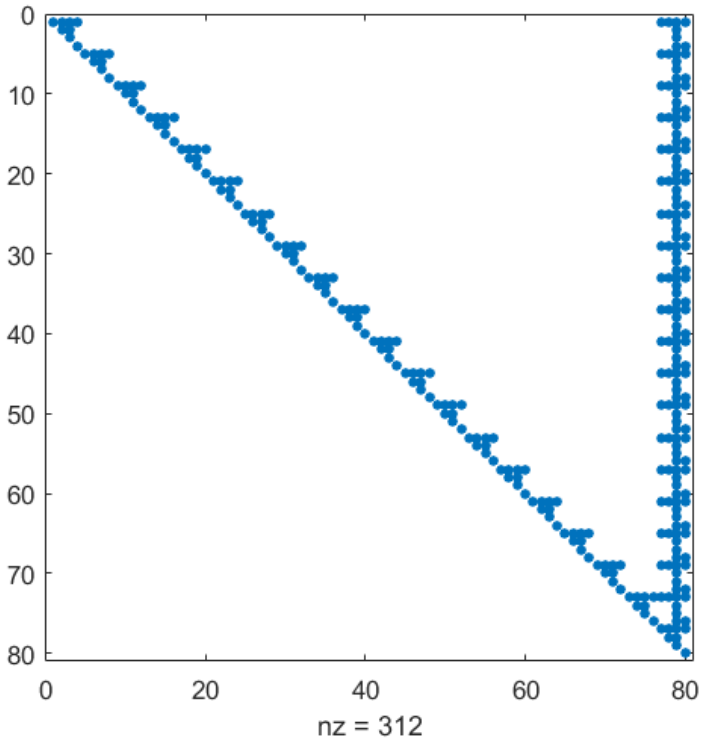

Illustration 3.6: spy(UB)

Το νέο γέμισμα που προκύπτει από τα νέα μητρώα είναι ακριβώς 1 (μονάδα). Προφανώς η διαφορά από το προηγούμενο γέμισμα (6.6529) είναι τεράστια.

Στη συνέχεια δημιουργήθηκε το διάνυσμα e = ones(m*n,1) και το b = A*e. Ακολουθούν τα ζητούμενα αποτελέσματα για τους διάφορους συνδιασμούς των m και n.

A, B - Not Sparse	m = 4; n = 250	m = 8; n = 125	m = 8; n = 250	m = 16; n = 125
Ax=b (Time)	0.0154	0.0169	0.0856	0.0912
BWx=Wb (Time)	0.0127	0.0132	0.0775	0.0795
Ax=b (Rel. Error)	1.6683e-11	5.7621e-12	2.2048e-10	1.0484e-11
BWx=Wb (Rel. Error)	3.3396e-13	3.7503e-13	1.0592e-10	7.4074e-13

Όπως είναι φανερό ο χρόνος επίλυσης μειώνεται αισθητά με την μέθοδο της μετάθεσης, όπως επίσης μειώνεται και το σχετικό σφάλμα σε όλες τις περιπτώσεις.

A, B - Sparse	m = 4; n = 250	m = 8; n = 125	m = 8; n = 250	m = 16; n = 125
Ax=b (Time)	0.0020	0.0019	0.0041	0.0064
BWx=Wb (Time)	0.0017	0.0020	0.0041	0.0064
Ax=b (Rel. Error)	3.6093e-13	4.6652e-13	1.1539e-11	1.2939e-12
BWx=Wb (Rel. Error)	6.3727e-14	3.4905e-13	1.7475e-12	4.9116e-13

Σε αντίθεση με πριν που τα A και B ήταν αποθηκευμένα σε πυκνή μορφή, μετά την μετατροπή τους σε αραιά μητρώα παρατηρείται πως ο χρόνος επίλυσης είναι ο ίδιος και στις δυο περιπτώσεις με μια μικρή διαφορά στα περιθώρια του λάθους. Ο πολύ μικρότερος χρόνος εκτέλεσης πιθανώς οφείλεται στο γεγονός οτί η MATLAB εκμεταλλεύεται τα sparse μητρώα χρησιμοποιώντας αλγορίθμους για πλοκάδες όπως επίσης πιθανώς βοηθάει το γεγονός ότι τα A και B αποτελούνται από αντίγραφα δυο μόνο πλοκάδων, των D και T. Όπως και πριν, το σχετικό σφάλμα με την μέθοδο της μετάθεσης είναι μειωμένο για κάθε συνδιασμό m και n.

meros3.m

```
myHash = 9683;

%m = 4; n = 20;
m = 4; n = 250;
%m = 8; n = 125;
%m = 8; n = 250;
%m = 16; n = 125;

rng(myHash);
T = full(sprand(m,m,0.6));
A = arrowNW(T,n);
[L,U] = lu(A);

%spy(A);
%spy(L);
%spy(U);

W = fliplr(eye(m*n));
B = W*A*transpose(W);
[LB,UB] = lu(B);

%spy(B);
%spy(LB);
%spy(UB);

fill = (nnz(L)+nnz(U)-m*n)/nnz(A);
fillB = (nnz(LB)+nnz(UB)-m*n)/nnz(B);

e = ones(m*n,1);
b = A*e;
Wb = W*b;

%A = sparse(A);
%B = sparse(B);

% A * x = b
f1 = @() A\b;
t1 = timeit(f1);
x1 = f1();
rerr1 = norm(e-x1,Inf)/norm(e,Inf);

% B * Wx = Wb
f2 = @() B\Wb;
t2 = timeit(f2);
Wx = f2();
x2 = transpose(W)*Wx;
rerr2 = norm(e-x2,Inf)/norm(e,Inf);
```

Μέρος Δ:

Για το μέρος αυτό δημιουργήθηκαν 3 αρχεία MATLAB. Ακολουθούν τα περιεχόμενα τους:

Tx.m

```
function [ c, t, mu ] = Tx( n, deg )
    t = zeros(1,length(n));

    for i = 1:length(n)
        A = randn(n(i));
        f = @() qr(A);
        t(i) = timeit(f);
    end

    [c,~,mu] = polyfit(n,t,deg);
end
```

Tqr.m

```
function [ c, t, mu ] = Tqr( n, deg )
    t = zeros(1,length(n));

    for i = 1:length(n)
        A = randn(n(i));
        f = @() qr(A);
        t(i) = timeit(f,2);
    end

    [c,~,mu] = polyfit(n,t,deg);
end
```

meros4.m

```
n1 = [200:200:1400];
n2 = [250:200:1750];
deg = 3;

% X = qr(A)

[c_x,t_x_real1,mu_x] = Tx(n1,deg);
t_x_polyval1 = zeros(1,length(n1));
for i = 1:length(n1)
    t_x_polyval1(i) = polyval(c_x,n1(i),[],mu_x);
end

t_x_real2 = zeros(1,length(n2));
t_x_polyval2 = zeros(1,length(n2));
for i = 1:length(n2)
    A = randn(n2(i));
    f = @() qr(A);
    t_x_real2(i) = timeit(f);
    t_x_polyval2(i) = polyval(c_x,n2(i),[],mu_x);
end

% [Q,R] = qr(A)

[c_qr,t_qr_real1,mu_qr] = Tqr(n1,deg);
t_qr_polyval1 = zeros(1,length(n1));
for i = 1:length(n1)
    t_qr_polyval1(i) = polyval(c_qr,n1(i),[],mu_qr);
```

```
end

t_qr_real2 = zeros(1,length(n2));
t_qr_polyval2 = zeros(1,length(n2));
for i = 1:length(n2)
    A = randn(n2(i));
    f = @() qr(A);
    t_qr_real2(i) = timeit(f,2);
    t_qr_polyval2(i) = polyval(c_qr,n2(i),[],mu_qr);
end

% PLOTS

figure('Name','X = qr(A) for n = [200:200:1400]');
plot(n1,t_x_real1,'+',n1,t_x_polyval1,'--');
figure('Name','X = qr(A) for n = [250:200:1750]');
plot(n2,t_x_real2,'+',n2,t_x_polyval2,'--');

figure('Name','[Q,R] = qr(A) for n = [200:200:1400]');
plot(n1,t_qr_real1,'+',n1,t_qr_polyval1,'--');
figure('Name','[Q,R] = qr(A) for n = [250:200:1750]');
plot(n2,t_qr_real2,'+',n2,t_qr_polyval2,'--');
```

Οι συναρτήσεις Tx και Tqr παίρνουν σαν ορίσματα το μέγεθος n του πίνακα A και τον βαθμό deg του πολυωνύμου που θα ταιριάξει η polyfit στα data sets των χρονομετρήσεων. Και οι δύο συναρτήσεις επιστρέφουν το διάνυσμα c με τα coefficients του πολυωνύμου, το διάνυσμα t με τους πραγματικούς χρόνους εκτέλεσης που μετρήθηκαν και το διάνυσμα mu με τον αριθμητικό μέσο και το standard deviation των n για χρήση στην polyval.

Στην αρχή του script ορίζονται τα δύο σετ των n (n1 και n2) καθώς και ο βαθμός deg του πολυωνύμου. Στη συνέχεια για κάθε μια από τις δύο μορφές της qr(A) εκτελείται η αντίστοιχη συνάρτηση για τα n1, γίνονται οι προβλέψεις των χρόνων για τα n1 και n2, χρονομετρούνται οι εκτελέσεις για τα n2 και στο τέλος δημιουργούνται τα ζητούμενα plots.

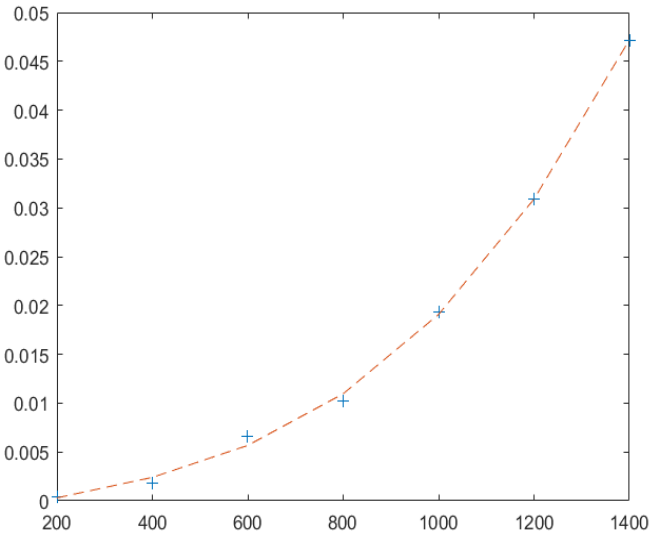

Illustration 4.1: X = qr(A) for n = [200:200:1400] (deg. 3)

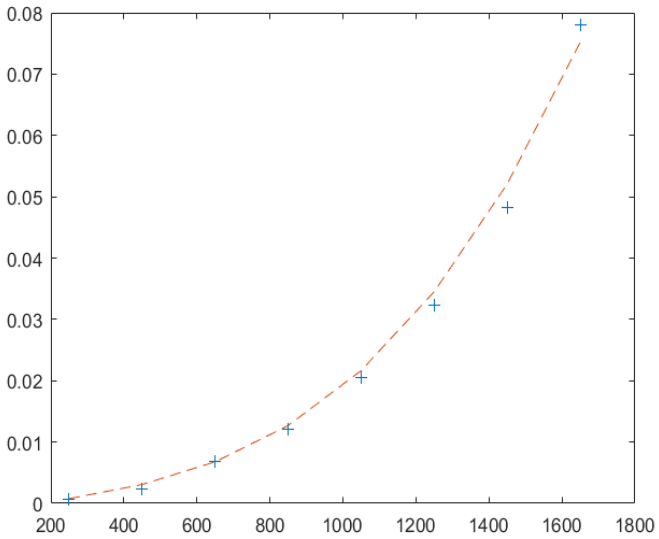

Illustration 4.2: X = qr(A) for n = [250:200:1750] (deg. 3)

Τα γραφήματα 4.1 και 4.2 αφορούν την X = qr(A). Αριστερά είναι το γράφημα με τις μετρήσεις και τις προβλέψεις για τα n1. Δεξιά είναι οι μετρήσεις και οι προβλέψεις για τα n2. Όπως είναι φανερό, οι προβλέψεις με το πολυώνυμο 3ου βαθμού είναι αρκετά ακριβείς.

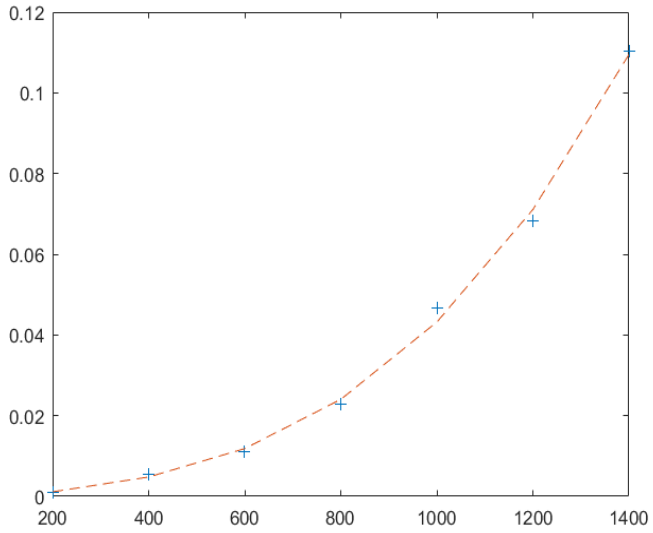

Illustration 4.3: [Q,R] = qr(A) for n = [200:200:1400] (deg. 3)

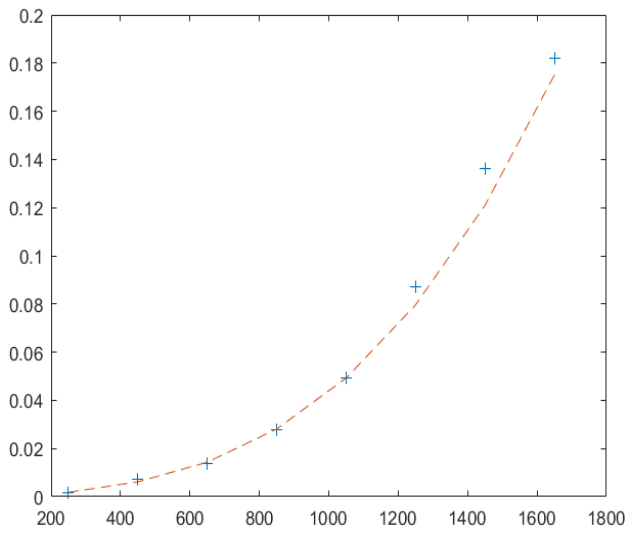

Illustration 4.4: [Q,R] = qr(A) for n = [250:200:1750] (deg. 3)

Τα γραφήματα 4.3 και 4.4 αφορούν την [Q,R] = qr(A). Όπως και πριν, αριστερά είναι οι μετρήσεις και οι προβλέψεις για τα n1 και δεξιά για τα n2. Ξανά είναι φανερό πως οι προβλέψεις είναι αρκετά ακριβείς.

Ακολουθούν τα ακριβή νούμερα των συντελεστών από την εκτέλεση του script:

Function	C3 (highest)	C2	C1	C0 (constant)
X = qr(A)	0.0014	0.0066	0.0142	0.0110
[Q,R] = qr(A)	0.0030	0.0162	0.0332	0.0240

Τα γραφήματα 4.5 έως 4.8 (παρακάτω) είναι τα ίδια με πριν, χρησιμοποιώντας πολυώνυμο 2ου βαθμού. Τα γραφήματα 4.9 έως 4.12 (παρακάτω) είναι επίσης τα ίδια, χρησιμοποιώντας πολυώνυμο 4ου βαθμού. Όπως φαίνεται και στις δυο περιπτώσεις οι προβλέψεις έχουν αρκετά χειρότερη ακρίβεια απ' ότι με το πολυώνυμο 3ου βαθμού.

Επομένως συμπεραίνουμε ότι η καλύτερη προσέγγιση γίνεται με πολυώνυμο 3ου βαθμού, πράγμα που θεωρητικά βασίζεται στο ότι η χρονική πολυπλοκότητα της QR παραγοντοποίησης είναι κυβική.

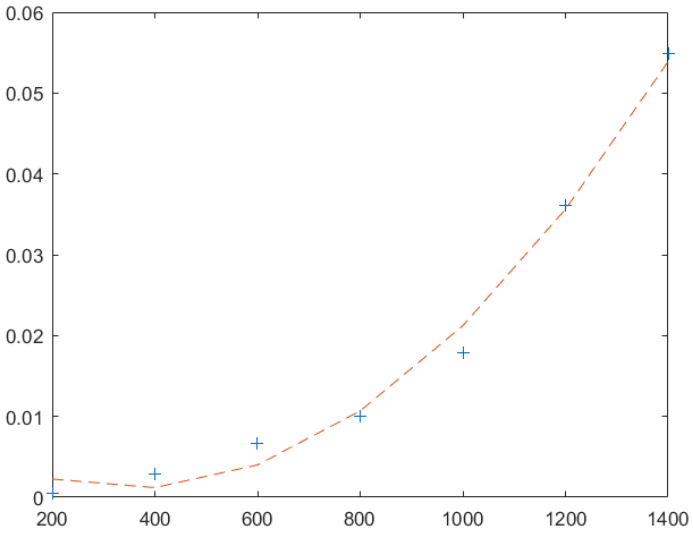

Illustration 4.5: X = qr(A) for n = [200:200:1400] (deg. 2)

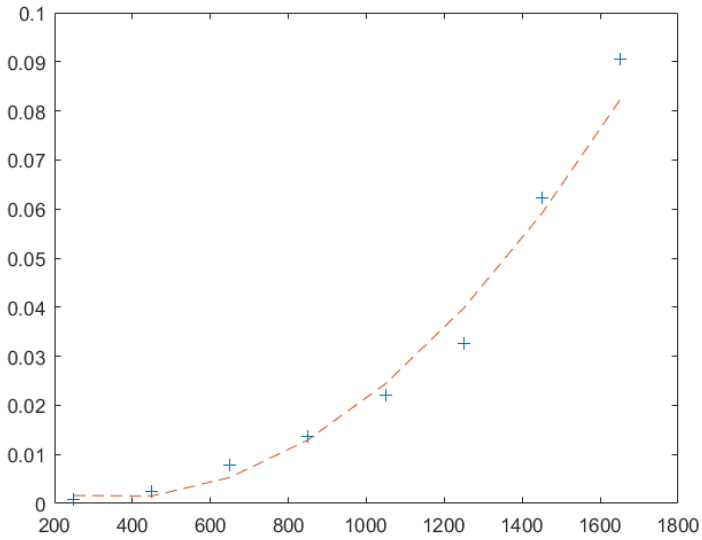

Illustration 4.6: X = qr(A) for n = [250:200:1750] (deg. 2)

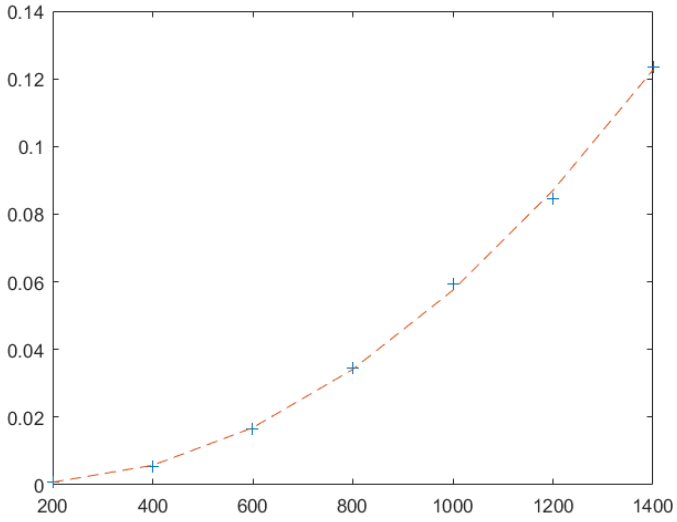

Illustration 4.7: [Q,R] = qr(A) for n = [200:200:1400] (deg. 2)

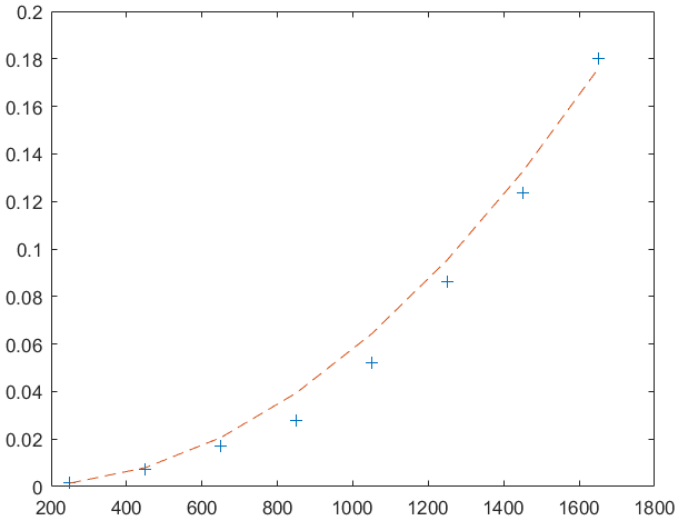

Illustration 4.8: [Q,R] = qr(A) for n = [250:200:1750] (deg. 2)

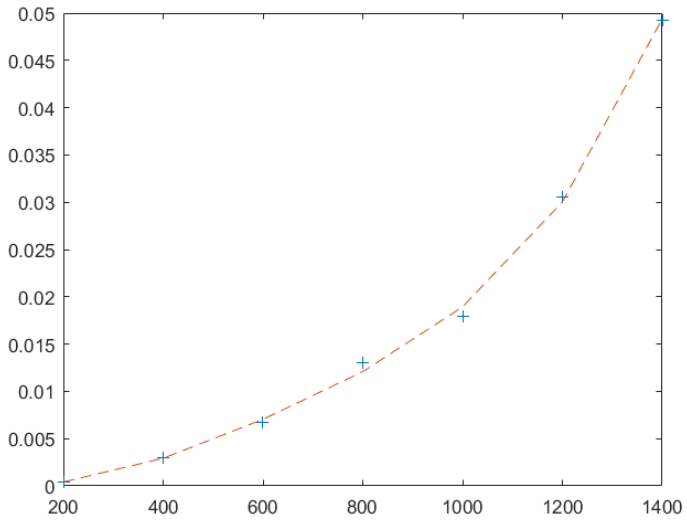

Illustration 4.9: X = qr(A) for n = [200:200:1400] (deg. 4)

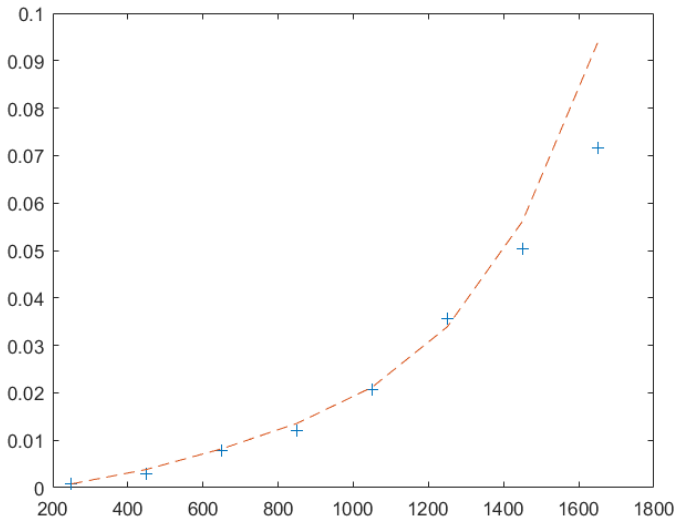

Illustration 4.10: X = qr(A) for n = [250:200:1750] (deg. 4)

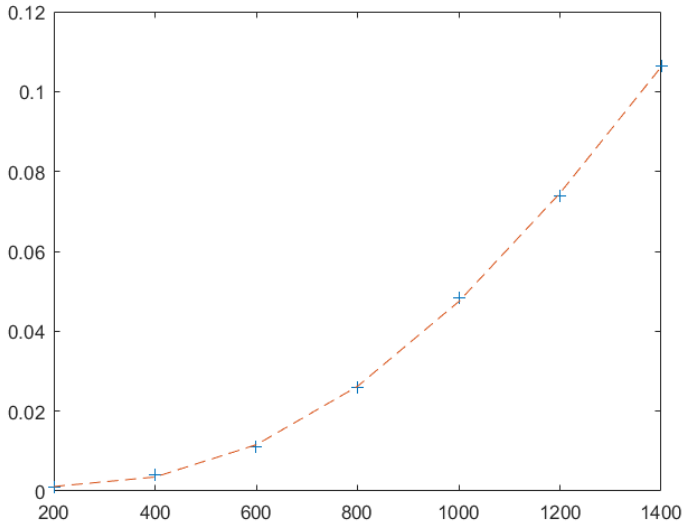

Illustration 4.11: [Q,R] = qr(A) for n = [200:200:1400] (deg. 4)

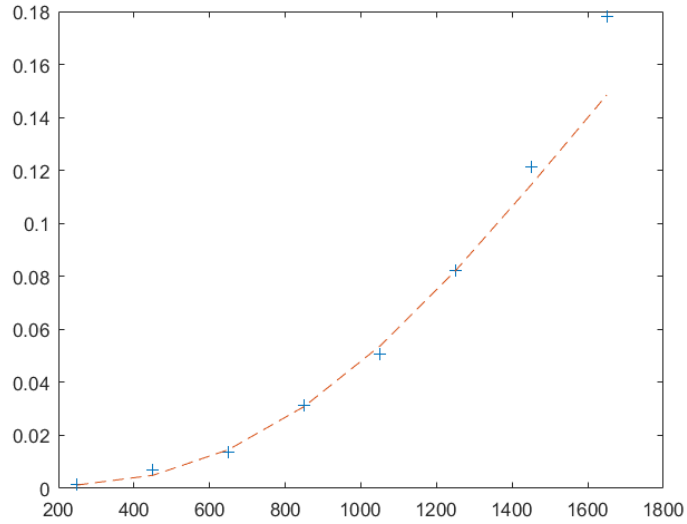

Illustration 4.12: [Q,R] = qr(A) for n = [250:200:1750] (deg. 4)

Αναφορές:

1. Intel 64 and IA-32 Architectures Optimization Reference Manual, Sept. 2014:

http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf