Επιστημονικός Υπολογισμός
Εργαστηριακή Άσκηση

Γεώργιος Μερμίγκης
ΑΜ:1084639

Ιανουάριος 2024

Περιεχόμενα

0 Στοιχεία υπολογιστικού συστήματος **3**

1 Χρονομετρήσεις **4**
1.1 Ερώτημα 1 4
1.2 Ερώτημα 2 6
1.3 Ερώτημα 3 7
1.4 Ερώτημα 4 8
1.4.1 Δεύτερης τάξης πολυώνυμο 8
1.4.2 Τέταρτης τάξης πολυώνυμο 9

2 Ειδικοί επιλυτές και αραιά μητρώα **10**

3 Τανυστές **12**
3.1 Ερώτημα 1 12
3.2 Ερώτημα 2 14

4 Επίλυση ΣΘΟ συστημάτων με PCG **17**
4.1 SuiteSparse: 138_bus HB 20
4.2 2D Poisson Matrix - Dirichlet 23

0 Στοιχεία υπολογιστικού συστήματος

Table 1: Απαιτούμενες Πληροφορίες

Χαρακτηριστικό	Απάντηση
Έναρξη/λήξη εργασίας	10/01/24 - 17/01/24
model	προσωπικό λάπτοπ Macbook Air [1]
O/S	macOS Sonoma 14.2.1
processor name	M1 [2]
processor speed	3.2 GHz
number of processors	1
total # cores	8
total # threads	8
FMA instruction	yes
L1 cache	2MB
L2 cache	16MB
L3 cache	8MB
Gflops/s	2.6 Tflops
Memory	8GB
Memory Bandwidth	66.67GB/s
MATLAB Version	23.2.0.2409890 (R2023b) Update 3 [3]
BLAS	native
LAPACK	native

Computer Type	LU	FFT	ODE	Sparse	2-D	3-D
Mac mini, Apple M2 @ 3.50 GHz	0.5551	0.1516	0.0767	0.2470	0.1682	0.1430
Windows 11, Intel Core i9-12900 @ 2.4 GHz	0.2516	0.1523	0.0881	0.4521	0.1890	0.2359
Windows 11, AMD Ryzen Threadripper(TM) 3970x @ 3.7 GHz	0.1945	0.1662	0.1723	1.2129	0.1981	0.1274
Windows 11, Intel Core i7-1185G7 @ 3.00 GHz	0.6690	0.3013	0.1433	0.3440	0.2774	0.1461
iMac, macOS 13.2.1, Intel Core i9 @ 3.6 GHz	0.3347	0.2679	0.1336	0.2840	0.6960	0.3816
Debian 11(R), AMD Ryzen Threadripper 2950x @ 3.50 GHz	0.3384	0.2465	0.1597	1.2545	0.2516	0.1971
Windows 11, AMD Ryzen(TM) 5 Pro 6650U @ 2.9 GHz	0.6419	0.3626	0.1273	0.5449	0.3753	0.2346
This machine	0.6724	0.4111	0.0972	0.3047	0.9162	0.6612
MacBook Pro, macOS 11.7.2, Intel(R) Core(TM) i7 @ 2.9 GHz	0.8710	0.5960	0.2006	0.5297	1.7433	0.7368

Figure 1: Αποτελέσματα της bench σε M1.

[1] `https://www.apple.com/gr/macbook-air-m1/`
[2] `https://www.apple.com/macbook-air-m1/specs/`
[3] Εντολή version στη MATLAB

1 Χρονομετρήσεις

1.1 Ερώτημα 1

Αρχικά ορίζω ένα array με διάφορα μεγέθη μητρώων που θα χρησιμοποιήσω και στη συνέχεια αρχικοποιώ το μητρώο **execution_times** που θα αποθηκεύσει τους χρόνους εκτέλεσης για κάθε μέγεθος μητρώου. Με ένα loop δημιουργώ το **A** μητρώο μου, το οποίο είναι συμμετρικά θετικά ορισμένο (ΣΘΟ), τρέχω τη cholesky για αυτό και αποθηκεύω τον χρόνο που χρειάστηκε στο execution_times. Επειδή θέλω να κάνω fit μια κυβική συνάρτηση στα data points μου, ορίζω βαθμό πολυωνύμου 3 και με την εντολή **polyfit** ψάχνω να βρω τους συντελεστές του κυβικού πολυωνύμου:

$$f(x) = ax^3 + bx^2 + cx + d$$

Τα αποτελέσματα διαφέρουν κάθε φορά λόγω της rand. Παραθέτω μερικά ενδεικτικά:

$$\begin{aligned} a &= 0.0000 \times 10^{-3} \\ b &= -0.0000 \times 10^{-3} \\ c &= 0.0053 \times 10^{-3} \\ d &= -0.6606 \times 10^{-3} \end{aligned}$$

Τέλος, κάνω plot το πολυώνυμο και παρατηρώ την προσέγγιση του στα measured_times.

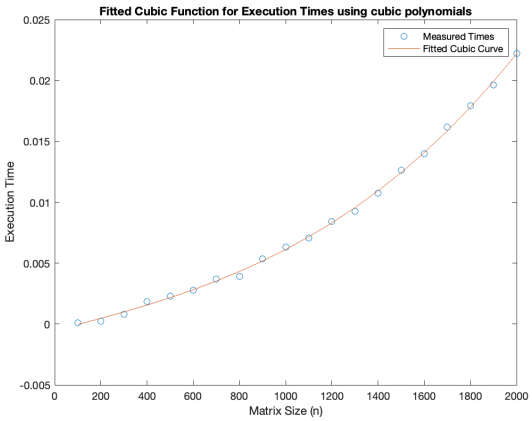

Figure 2: Fitted Cubic Function for Execution Times using cubic polynomials

Συμπερασματικά, βάση των συντελεστών του πολυωνύμου, ο κυβικός και ο τετραγωνικός χρόνος δεν επηρεάζουν έντονα τον χρόνο εκτέλεσης. Ο γραμμικός όρος φαίνεται πως είναι ο κυρίαρχος.

```
%Measure execution times
n_values = 100:100:2000;
execution_times = zeros(size(n_values));

for i = 1:length(n_values)
    n = n_values(i);
    A = randn(n, n);
    A = A * A';
    A = A + n * eye(n);
    timing = timeit(@() chol(A));
    execution_times(i) = timing;
end

%Fit a cubic function using polyfit
degree = 3;
coefficients = polyfit(n_values, execution_times, degree);

%coefficients
disp('Cubic function coefficients:');
disp(coefficients);

%plot data and fitted curve
fit_curve = polyval(coefficients, n_values);
figure;
plot(n_values, execution_times, 'o', n_values, fit_curve, '-
    ');
xlabel('Matrix Size (n)');
ylabel('Execution Time');
title('Fitted Cubic Function for Execution Times using cubic
    polynomials');
legend('Measured Times', 'Fitted Cubic Curve');
```

1.2 Ερώτημα 2

Για να υπολογίσω τις τιμές που προβλέπονται για τους χρόνος από την παραπάνω συνάρτηση, αξιοποιώ τους συντελεστές πολυωνύμου που βρήκα στο προηγούμενο υποερώτημα και τη συνάρτηση **polyval** με ορίσματα τους συντελεστές και τις n τιμές.

Για **n: 100:100:2000** οι τιμές που παίρνω είναι:

−0.0002	0.0005	0.0012	0.0020	0.0027	0.0034	0.0041
0.0048	0.0056	0.0063	0.0070	0.0077	0.0084	0.0092
0.0099	0.0106	0.0113	0.0120	0.0128	0.0135	

Για **n: 150:100:1550** οι τιμές που παίρνω είναι:

0.0002	0.0009	0.0016	0.0023	0.0030	0.0038	0.0045
0.0052	0.0059	0.0066	0.0074	0.0081	0.0088	0.0095
0.0102						

Και στις 2 περιπτώσεις παρατηρώ πως οι τιμές που προβλέπονται αυξάνονται όσο μεγαλώνει το n και η αύξηση είναι περίπου γραμμική, πράγμα που επιβεβαιώνει τον κυρίαρχο όρο να είναι ο γραμμικός.

```
%coefficients obtained from polyfit
coefficients = [0.0000; -0.0000; 0.0072; -0.9224] * 1e-03;

%values of n for prediction
n_values = 100:100:2000;
n_values1= 150:100:1550;

%calculate predicted execution times
predicted_times1 = polyval(coefficients, n_values);
predicted_times2 = polyval(coefficients, n_values1);

disp('Predicted Execution Times for 100:100:2000 (using
    cubic polynomials):');
disp(predicted_times1);
disp("");
disp('Predicted Execution Times for 150:100:1550 (using
    cubic polynomials):');
disp(predicted_times2);
```

1.3 Ερώτημα 3

Ξαναχρησιμοποιώ τη συνάρτηση **polyval** με ορίσματα τους συντελεστές του κυβικού πολυωνύμου και τα values από [100:100:2000, 150:100:1550]. Έτσι υπολογίζω τα predicted και κάνω plot μαζί με τους πραγματικούς χρόνους που υπολόγισα στο πρώτο υποερώτημα.

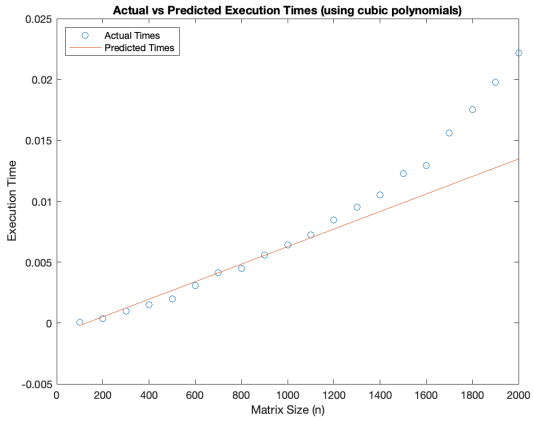

Figure 3: Actual vs Predicted Execution Times

Παρατηρώ πως μέχρι matrix size 1250 γίνεται αρκετά καλό prediction. Αυτό συμβαίνει διότι οι πραγματικές τιμές αρχίζουν να χάνουν τον γραμμικό τους "χαρακτήρα".

```
%predicted execution times for both sets of n values
predicted_n_values = [100:100:2000, 150:100:1550];
predicted_execution_times = polyval(coefficients,
    predicted_n_values);

%visualize the results
figure;
plot(n_values, execution_times, 'o', predicted_n_values,
    predicted_execution_times, '-');
xlabel('Matrix Size (n)');
ylabel('Execution Time');
title('Actual vs Predicted Execution Times (using cubic
    polynomials)');
legend('Actual Times', 'Predicted Times', 'Location', '
    NorthWest');
```

1.4 Ερώτημα 4

Τα παρακάτω αποτελέσματα αποδεικνύουν πως ο γραμμικός συντελεστής είναι και ο καθοριστικός. Παρόλο που αυξάνω την τάξη του πολυωνύμου, τα νέα πολυώνυμα δεν κάνουν καλύτερο fit.

1.4.1 Δεύτερης τάξης πολυώνυμο

Για αυτό το ερώτημα ακολούθησα ακριβώς την ίδια τακτική με το 1ο υποερώτημα της άσκησης αλλά για **degree=2**. Ψάχνω τους συντελεστές του πολυώνυμου 2ης τάξης:

$$f(x) = ax^2 + bx + c$$

Τα αποτελέσματα διαφέρουν κάθε φορά λόγω της rand. Παραθέτω μερικά ενδεικτικά:

$$a = 0.0000 \times 10^{-3}$$
$$b = -0.0002 \times 10^{-3}$$
$$c = 0.4070 \times 10^{-3}$$

Τέλος, κάνω plot το πολυώνυμο και παρατηρώ την προσέγγιση του στα measured_times.

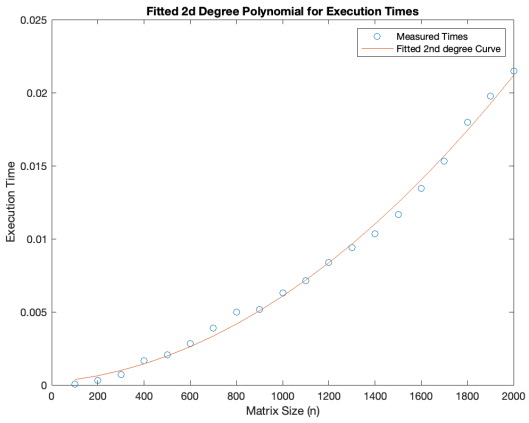

Figure 4: Fitted 2d Degree Polynomial for Execution Times

1.4.2 Τέταρτης τάξης πολυώνυμο

Για αυτό το ερώτημα ακολούθησα ακριβώς την ίδια τακτική με το 1ο υποερώτημα της άσκησης αλλά για **degree=4**. Ψάχνω τους συντελεστές του πολυώνυμου 2ης τάξης:

$$f(x) = ax^4 + bx^3 + cx^2 + dx + e$$

Τα αποτελέσματα διαφέρουν κάθε φορά λόγω της rand. Παραθέτω μερικά ενδεικτικά:

$$\begin{aligned} a &= -0.0000 \times 10^{-3} \\ b &= 0.0000 \times 10^{-3} \\ c &= -0.0000 \times 10^{-3} \\ d &= 0.0073 \times 10^{-3} \\ e &= -0.9268 \times 10^{-3} \end{aligned}$$

Τέλος, κάνω plot το πολυώνυμο και παρατηρώ την προσέγγιση του στα measured_times.

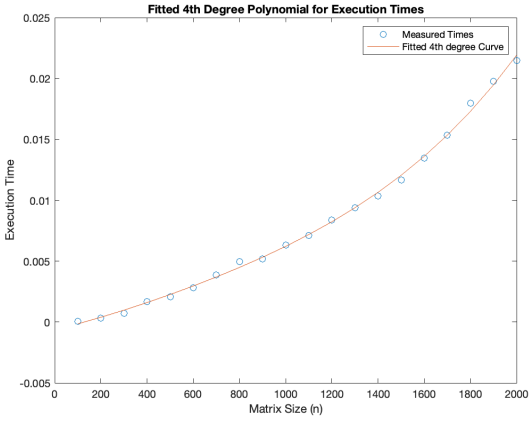

Figure 5: Fitted 4th Degree Polynomial for Execution Times

2 Ειδικοί επιλυτές και αραιά μητρώα

Η διαδικασία που θα ακολουθήσω για την επίλυση τριδιαγώνιων συστημάτων είναι:

$A = D - L - U$ όπου D διαγώνιο, L κάτω τριγωνικό και U άνω τριγωνικό. Για την επίλυση του $Ax = b$:

$$(D + L + U) \cdot D^{-1} \cdot (D - L - U) \cdot x = (D + L + U) \cdot D^{-1} \cdot b$$

Αυτό υλοποιώ στην **solveTriagonal** συνάρτηση που υλοποιήσα. Δέχεται ως ορίσματα ένα διαγώνιο μητρώο D, ένα κάτω τριγωνικό μητρώο L και ένα άνω τριγωνικό μητρώο U. Αρχικοποιώ το μητρώο της λύσης και προχωράω σε LU παραγοντοποιήση. Χρησιμοποιώ ένα loop για επανάληψη από τη δεύτερη έως την τελευταία γραμμή. Σε κάθε βήμα υπολογίζω έναν παράγοντα με βάση το τρέχον στοιχείο του L και το αντίστοιχο στοιχείο του D και αφού υπολογιστέι ενημερώνεται το D και το διάνυσμα b. Αφού ολοκληρωθεί η παραγοντοποίηση LU, η συνάρτηση προχωράει σε αντικατάσταση πρως τα πίσω για να βρει τα στοιχεία του διανύσματος λύσης. Ξεκινώντας από την τελευταία σειρά, υπολογίζει τη λύση για την τελευταία μεταβλητή (x(n, :)) με βάση το τροποποιημένο διάνυσμα της δεξιάς πλευράς b και το ενημερωμένο διαγώνιο μητρώο D. Στη συνέχεια επαναλαμβάνεται προς τα πίσω μέσω του συστήματος, υπολογίζοντας τις λύσεις για τις υπόλοιπες μεταβλητές.
Για μητρώο:

```
n = 5;
D = diag(4*ones(1, n));
L = diag(-ones(1, n-1), -1);
U = diag(-ones(1, n-1), 1);
b = randn(n, 1);
```

Τα ενδεικτικά αποτελέσματα που παίρνω είναι τα εξής:

$$-0.1052 \quad 0.2294 \quad -0.0412 \quad 0.0189 \quad 0.2375$$

Υλοποίησα και την **solveTridiagonalHad** όπου ακολουθεί την ίδια διαδικασία με την προηγούμενη υλοποιώντας Hadamard operations και έτρεξα διάφορα παραδείγματα. Η συγκεκριμένη σε μεγαλύτερου μεγέθους προβλήματα πετυχαίνει καλύτερους χρόνους από την απλή.
Για μητρώο:

```
n = 1000;
D_large = diag(4 * ones(1, n));
L_large = diag(-ones(1, n-1), -1);
U_large = diag(-ones(1, n-1), 1);
b_large = randn(n, 1);
```

Πετυχαίνει χρόνο 0.011269 seconds ενώ η πρώτη συνάρτηση πετυχαίνει χρόνο 0.017432 seconds.

Υπολογίζοντας τις νόρμες των υπερ/υπό-διαγωνίων, παρατηρώ πως σε κάθε μητρώο η νόρμα του άνω τριγωνικού μέρους είναι ίση με αυτή του κάτω τριγωνικού μέρους. Αυτό σημαίνει πως τα μητρώα δεν έχουν έντονες διακυμάνσεις στις τιμές τους, γι'αυτό και λαμβάνουμε καλούς χρόνους κατά την επίλυσή τους.
Οι συναρτήσεις που υλοποίησα είναι οι εξής:

```
function x = solveTridiagonal(D, L, U, b)
    %D, L, U are the tridiagonal matrices
    %b is the right-hand side vector(s)

    n = length(b);  %size of the system
    x = zeros(n, size(b, 2));  %initialize the solution
        matrix

    %LU decomposition
    for k = 2:n
        factor = L(k, k-1) / D(k-1, k-1);
        D(k, k) = D(k, k) - factor * U(k-1, k);
        b(k, :) = b(k, :) - factor * b(k-1, :);
    end

    %backward substitution
    x(n, :) = b(n, :) / D(n, n);
    for k = n-1:-1:1
        x(k, :) = (b(k, :) - U(k, k+1) * x(k+1, :)) / D(k, k
            );
    end
end
function x = solveTridiagonalHad(D, L, U, b)

    n = length(b);
    x = zeros(n, size(b, 2));  % Initialize the solution
        matrix

    %LU decomposition
    for k = 2:n
        factor = L(k-1) ./ D(k-1);
        D(k) = D(k) - factor .* U(k-1);
        b(k, :) = b(k, :) - factor .* b(k-1, :);
    end

    %Backward substitution
    x(n, :) = b(n, :) ./ D(n);
    for k = n-1:-1:1
        x(k, :) = (b(k, :) - U(k) .* x(k+1, :)) ./ D(k);
    end
end
```

3 Τανυστές

3.1 Ερώτημα 1

Επέλεξα να υλοποιήσω την **Tensor times Vector** function. Η συνάρτηση μου θα δέχεται ως ορίσματα ένα Multidimensional Array (MDA), ένα vector και τον τρόπο (mode) όπου θα γίνεται ο τροπικός πολλαπλασιασμός. Για αρχή κάνω τους σχετικούς ελέγχους:

1. Αν τα στοιχεία εισόδου είναι σωστού τύπου, δηλαδή μητρώο και διάνυσμα.
2. Αν το mode που έχει δοθεί είναι ένα από τα επιτρεπτά, δηλαδή 1, 2 ή 3.
3. Αν οι διαστάσεις είναι κατάλληλες για να πραγματοποιηθεί ο τροπικός πολλαπλασιασμός.

Αρχικοποιώ το μητρώο της λύσης και προχωράω στους πολλαπλασιασμούς. Στο mode = 2, ουσιαστικά από κάθε "φέτα" παίρνω τις γραμμές μία-μία και πολλαπλασιάζω κάθε στοιχείο με το εκάστοτε του διανύσματος. Στο mode = 1, από κάθε "φέτα" παίρνω τις στήλες μία-μία και πολλαπλασιάζω κάθε στοιχείο με το εκάστοτε του διανύσματος. Τέλος στο mode = 3, παίρνω 1 στοιχείο από κάθε "φέτα" (στοιχείο ίδιας θέσης) και πολλαπλασιάζω το καθένα με το εκάστοτε του διανύσματος. Επαναλαμβάνω μέχρι να τελειώσουν τα στοιχεία.
Για να ελέγξω αν τα αποτελέσματα μου είναι σωστά, συγκρίνω με αυτά που παράγονται από το **ttv operation** του Tensor Toolbox (TT). Για MDA και vector:

```
tensor_MDA = randi([1, 10], 3, 3, 3);
vector = [2; 3; 4];
```

Τα αποτελέσματα που παίρνω για mode 1, 2, 3 χρησιμοποιώντας τη δική μου συνάρτηση είναι τα εξής:

$$\text{Mode 1:} \quad A = \begin{bmatrix} 70 & 59 & 47 \\ 26 & 35 & 29 \\ 57 & 40 & 42 \end{bmatrix} \tag{1a}$$

$$\text{Mode 2:} \quad B = \begin{bmatrix} 30 & 51 & 36 \\ 62 & 27 & 47 \\ 50 & 50 & 34 \end{bmatrix} \tag{1b}$$

$$\text{Mode 3:} \quad C = \begin{bmatrix} 52 & 47 & 28 \\ 35 & 25 & 62 \\ 74 & 26 & 40 \end{bmatrix} \tag{1c}$$

Και είναι ακριβώς ίδια με αυτά που προκύπτουν από την **ttv operation** του TT. Ο κώδικας της συνάρτησης μου μπορεί να βρεθεί στην επόμενη σελίδα.

```
function result = ttv_1084639(tensor, vec, mode)
    disp(tensor);
    disp(vec);
    [m, n, p] = size(tensor);

    %check if matrix & vector are of the correct type
    if ~ismatrix(tensor) && ndims(tensor) > 2
        disp('Input tensor is a higher-dimensional array.');
    else
        error('Tensor must be a matrix or a higher-
            dimensional array.');
    end
    if ~isvector(vec)
        error('Vector must be a vector.');
    end

    %check if mode is 1,2 or 3
    if mode ~= 1 && mode ~= 2 && mode ~= 3
        error('Invalid mode. Mode must be 1, 2, or 3.');
    end

    %check if dims are compatible
    if numel(vec) ~= n
        error('Vector dimension does not match the size of
            the specified mode in the tensor.');
    end
    %initialize the result matrix
    result = zeros(m, p);

    if mode == 2
        %ttv operation for mode 2
        for i = 1:p
            result(:, i) = tensor(:,:,i) * vec;
        end
    elseif mode == 1
        %ttv operation for mode 1
        for i = 1:p
            result(:, i) = tensor(:,:,i).' * vec;
        end
    elseif mode == 3
        %ttv operation for mode 3
        for i = 1:m
            result(i, :) = tensor(i, :, 1) * vec(1) + tensor
                (i, :, 2) * vec(2) + tensor(i, :, 3) * vec(3)
                ;
        end
    end
end
```

3.2 Ερώτημα 2

Στο εξωτερικό γινόμενο περιμένουμε ως αποτέλεσμα ένα MDA, ενώ στο εσωτερικό γινόμενο περιμένουμε έναν αριθμό ως αποτέλεσμα. Η συνάρτηση που υλοποίησα δέχεται ως ορίσματα δύο MDAs και ένα operation argument που καθορίζει αν θα υπολογιστεί το εσωτερικό ή το εξωτερικό γινόμενο. Αν τα ορίσματα είναι λιγότερα από 3, δηλαδή 2, ορίζω η πράξη που θα συμβεί να είναι το εξωτερικό γινόμενο (σύμφωνα με την εκφώνηση). Αν τα ορίσματα είναι λιγότερα από 2, τυπώνεται σχετικό μήνυμα λάθους. Στη συνέχεια ελέγχω αν τα 2 MDAs έχουν το ίδιο πλήθος διαστάσεων γιατί αλλιώς δε θα μπορούν να υλοποιηθούν οι πράξεις εσωτερικού, εξωτερικού γινομένου. Επίσης έλεγχος γίνεται και για το operation argument. Αν είναι διαφορετικό του outer, all ή κενό, τυπώνεται μήνυμα λάθους (η συνάρτηση μου εκτελεί εξωτερικό γινόμενο για operation argument κενό ή outer).
Όσον αφορά το **εξωτερικό γινόμενο**, αρχικοποιείται με μηδενικά το C που θα "δεχτεί" τα αποτελέσματα και στη συνέχεια υπολογίζεται το γινόμενο. Σε κάθε επανάληψη το εκάστοτε στοιχείο του C υπολογίζεται πολλαπλασιάζοντας το (i, j) στοιχείο του A με το (k, l) στοιχείο του B. Το αποτέλεσμα αποθηκεύεται στο (i, j, k, l) στοιχείο του C. Το i "κινείται" στις γραμμές του A, το j στις στήλες του A, το k στις γραμμές του B και το l στις στήλες του B. Ουσιαστικά είναι σαν να υπολογίζεται το γινόμενο Kronecker μεταξύ των A, B.
Όσον αφορά το **εσωτερικό γινόμενο** κάνω element wise πολλαπλασιασμό των στοιχείων του A και του B και προσθέτω όλα τα αποτελέσματα πρώτα κατά στήλες. Το αποτέλεσμα είναι ένα vector και τώρα προσθέτω τα στοιχεία κατά γραμμές. Αυτό είναι και το τελικό αποτέλεσμα που επιστρέφω. Ουσιαστικά υπολογίζεται το Forbenius εσωτερικό γινόμενο των A, B.
Για να ελέγξω αν τα αποτελέσματα μου είναι σωστά, συγκρίνω με αυτά που παράγονται από το **ttt operation** του TT. Για MDAs:

```
A = [1, 2; 3, 4];
B = [5, 6; 7, 8];
```

Το αποτέλεσμα που παίρνω για εσωτερικό γινόμενο χρησιμοποιώντας τη δική μου συνάρτηση είναι:

$$\begin{bmatrix} \begin{bmatrix} 5 & 10 \\ 15 & 20 \end{bmatrix} & \begin{bmatrix} 7 & 14 \\ 21 & 28 \end{bmatrix} \\ \begin{bmatrix} 6 & 12 \\ 18 & 24 \end{bmatrix} & \begin{bmatrix} 8 & 16 \\ 24 & 32 \end{bmatrix} \end{bmatrix}$$

Για εξωτερικό γινόμενο: 70.
Είναι ακριβώς ίδια με αυτά που προκύπτουν από την **ttt operation** του TT. Ο κώδικας της συνάρτησης μου μπορεί να βρεθεί στην επόμενη σελίδα.

```
function C = ttt_1084639(A, B, operation)
    %check the number of input arguments
    if nargin < 3
        operation = 'outer';
    end
    if nargin < 2
        error('Arguments number can''t be < 2.');
    end

    %check if input MDA tensors have the same number of
        dimensions
    if ndims(A) ~= ndims(B)
        error('Input tensors must have the same number of
            dimensions.');
    end

    %outer product that will be called if operations="outer"
         or if the
    %nargin < 3
    if strcmp(operation, 'outer')

        C = zeros([size(A), size(B)]);

        %in each iteration, the corresponding element of C
            is computed by
        %multiplying the element of index (i,j) from A and
            index (k,l)
        %from B. The result is a tensor with dims [size(A),
            size(B)].
        % i: rows of A, j: cols for A, k: rows of B, l: cols
             for B
        for i = 1:size(A, 1)
            for j = 1:size(A, 2)
                for k = 1:size(B, 1)
                    for l = 1:size(B, 2)
                        C(i, j, k, l) = A(i, j) * B(k, l);
                    end
                end
            end
        end

    %inner product
    elseif strcmp(operation, 'all')
        C = sum(sum(A .* B));
    else
        error('Invalid operation. Use ''outer'' or ''all''.'
            );
    end
end
```

Ακόμη, δημιούργησα μια δική μου **test_tensor** συνάρτηση που δέχεται ως ορίσματα ένα MDA, ένα vector, ένα mode και δύο ακόμη MDAs A, B. Καλώ τις συναρτήσεις που έφτιαξα και συγκρίνω τα αποτελέσματα που παίρνω με αυτά που παράγουν οι συναρτήσεις του Tensor Toolbox. Τέλος, όρισα μια ακόμη συνάρτηση **test_tensor_given**, όπου ξανά ελέγχω την ttv function που έφτιαξα, με τον τρόπο που ορίζει η εκφώνηση.

4 Επίλυση ΣΘΟ συστημάτων με PCG

Για να βρω τον κώδικα της **Preconditioned Conjugate Gradient (PCG)** πήγα στο directory: /Applications/MATLAB_R2023b.app/toolbox/matlab/sparfun. Οι αλλαγές που έκανα είναι οι εξής:

1. Η PCG στην αρχή του κώδικας της ελέγχει αν το A είναι μητρώο ή function με την εντολή: iterchk. Επειδή δεν μπορώ να χρησιμοποιήσω την εντολή αυτή, άλλαξα τον έλεγχο αυτό σε:

```
if isnumeric(A)
    atype = 'matrix';
    afun = @(x) A * x;
else
    atype = 'function_handle';
    afun = A;
end
```

2. Η PCG με ξανά με την iterchk ελέγχει αν ο preconditioner M1 είναι μητρώο ή function. Άλλαξα την εντολή αυτή με:

```
if isnumeric(M1)
    m1type = 'matrix';
    m1fun = M1;
elseif isa(M1, 'function_handle')
    m1type = 'function_handle';
    m1fun = M1;
else
    error('Invalid␣preconditioner␣M1');
end
```

   Αντίστοιχη αλλαγή έκανα και για τον preconditioner M2.

3. Η PCG αν δεχτεί περισσότερα από 7 ορίσματα τυπώνει μήνυμα λάθους: too many inputs. Στη δική μου εκδοχή αυτό αλλάζει. Τυπώνεται μήνυμα λάθους: too many inputs, για περισσότερα από 9 ορίσματα.

4. Η PCG υπολογίζει το υπολειπόμενο r ως τη διαφορά μεταξύ του διανύσματος της δεξιάς πλευράς b και του αποτελέσματος της εφαρμογής ενός γραμμικού τελεστή (που καθορίζεται από το afun) στο τρέχον διάνυσμα λύσης x, χρησιμοποιώντας την εντολή: iterapp. Επειδή δεν μπορώ να χρησιμοποιήσω την εντολή αυτή, άλλαξα τον κώδικα αυτό σε:

```
    if strcmp(atype, 'matrix')

        r = b - A * x;
    elseif strcmp(atype, 'function_handle')
        r = b - afun(x);
    else
        error('Unsupported atype for matrix-vector
            multiplication.');
    end
```

5. Αρχικοποιώ το **errvec** διάνυσμα με μέγεθος ίσο με αυτό του resvec διανύσματος: errvec = zeros(maxit+1, 1);

6. Παιρνώντας στο loop των maxit iterations, αν υπάρχει preconditioner M1, η PCG υπολογίζει το διάνυσμα ενημέρωσης y εφαρμόζοντας τον γραμμικό τελεστή που αντιπροσωπεύεται από m1fun στο υπολειπόμενο διάνυσμα r χρησιμοποιώντας τον τελεστή ανάστροφης. Επειδή δεν μπορώ να χρησιμοποιήσω την εντολή αυτή, άλλαξα τον κώδικα αυτό σε:

$$y = m1fun\backslash r;$$

   Αντίστοιχη αλλαγή έκανα και για τον preconditioner M2.

7. Η PCG υπολογίζει το q να είναι το γινόμενο του afun (που καθορίζεται από τη συνάρτηση ή το μητρώο afun) και το διάνυσμα p. Επειδή δεν μπορώ να χρησιμοποιήσω την εντολή αυτή, άλλαξα τον κώδικα αυτό σε:

```
    if strcmp(atype, 'matrix')

        q = A * p;
    elseif strcmp(atype, 'function_handle')
        q = afun(p);
    else
        error('Unsupported atype for matrix-vector
            multiplication.');
    end
```

8. Για να υπολογίσω την A - norm του error:

```
    if ~isempty(xsol)
        %Calculate the norm of the difference
        %errvec(ii+1, 1) = norm(x - xsol);
        errvec(ii+1, 1) = sqrt((x - xsol)' * A * (x -
            xsol));

    else
        %If xsol is not provided, set error to NaN
        errvec(ii+1, 1) = NaN;
    end
```

9. Η PCG θέλοντας να υπολογίσει το r εκτελεί $A*x$ με την iterapp και αφαιρεί από το b. Επειδή δεν μπορώ να χρησιμοποιήσω την εντολή αυτή, άλλαξα τον κώδικα αυτό σε:

```
    if strcmp(atype, 'matrix')
        r = b - A * x;
    elseif strcmp(atype, 'function_handle')
        r = b - afun(x);
    else
        error('Unsupported atype for matrix-vector
            multiplication.');
    end
```

10. Αλλάζω την: r_comp = b - iterapp('mtimes',afun,atype,xmin,varargin{:}) με:

```
    if strcmp(atype, 'matrix')
        r_comp = b - A * xmin;
    elseif strcmp(atype, 'function_handle')
        r_comp = b - afun(xmin);
    else
        error('Unsupported atype for matrix-vector
            multiplication.');
    end
```

    Ξανά επειδή δεν μπορώ να χρησιμοποιήσω την iterapp.

11. Ανάλογα με το flag, στο τέλος του κώδικα της PCG, χειρίζομαι και το errvec (παρόμοια με το resvec).

4.1 SuiteSparse: 138_bus HB

Για να δοκιμάσω τη μέθοδο μου στο 138_bus, επισκέφτηκα τη σελίδα της SuiteSparse και "κατέβασα" το μητρώο. Για b διάνυσμα δημιουργώ ένα τυχαίο διάνυσμα με μήκος ίσο με τον αριθμό των γραμμών του A. Για ακριβή λύση ορίζω το:

$$A \setminus b$$

και καλώ τη **pcg_1084639**. Για προρύθμιση χρησιμοποιώ την επίλυση cholesky του A. Στη συνέχεια με τα επιστρεφόμενα αποτελέσματα, υπολογίζω το σχετικό υπόλοιπο και το σχετικό σφάλμα και τα κάνω plot.
Εκτέλεση **pcg_1084639** χωρίς προρύθμιση και max iterations 4:

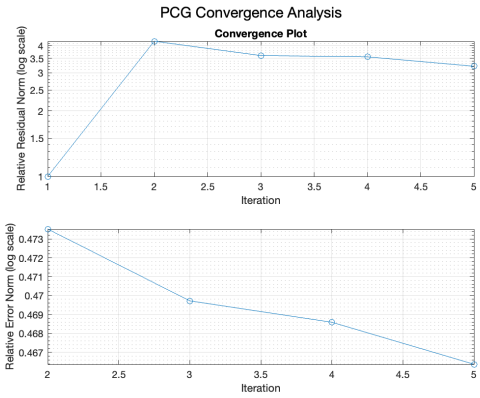

Figure 6: Without Preconditioners

Εκτέλεση **pcg_1084639** με προρύθμιση ichol(A) και max iterations 4:

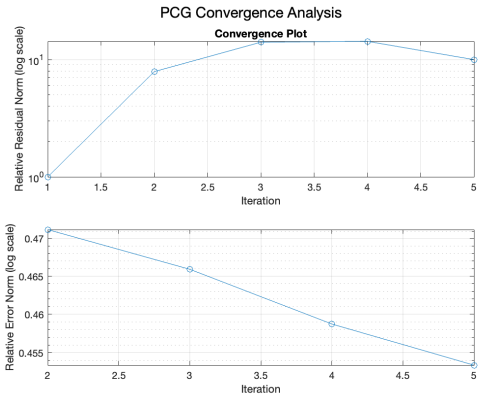

Figure 7: With Preconditioner

Εκτέλεση **pcg_1084639** χωρίς προρύθμιση και max iterations 20:

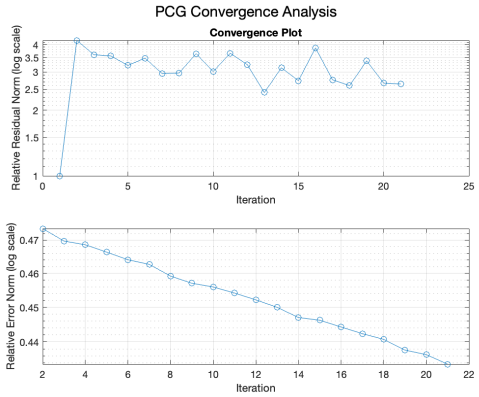

Figure 8: Without Preconditioners

Εκτέλεση **pcg_1084639** με προρύθμιση ichol(A) και max iterations 20:

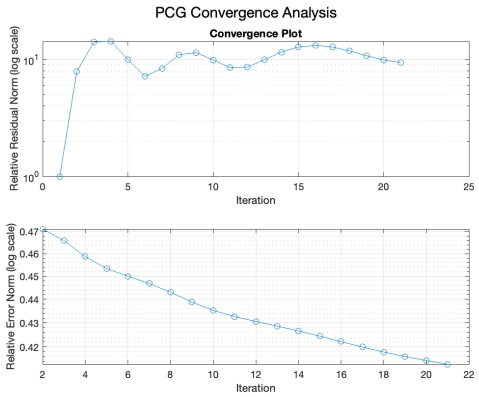

Figure 9: With Preconditioner

Παρατηρώ πως με την ichol(A) ως προρύθμιση δεν κερδίζω σε iterations, αλλά ούτε σε χρόνο. Με έκα καλύτερο preconditioner θα υπήρχε σίγουρα διαφορά σε αυτούς τους 2 παράγοντες.
Δοκιμάζω αντί για προρύθμιση με ichol(A) να χρησιμοποιήσω προρύθμιση ilu(A), με αποτέλεσμα η μέθοδος να συγκλίνει καλύτερα. Ουσιαστικά προσπαθώ να μειώσω το condition number του συστήματος.

Εκτέλεση **pcg_1084639** με προρύθμιση ilu(A) και max iterations 4:

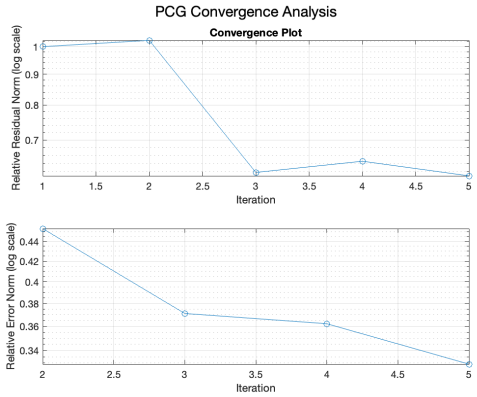

Figure 10: Iterations 4

Εκτέλεση **pcg_1084639** με προρύθμιση ilu(A) και max iterations 20:

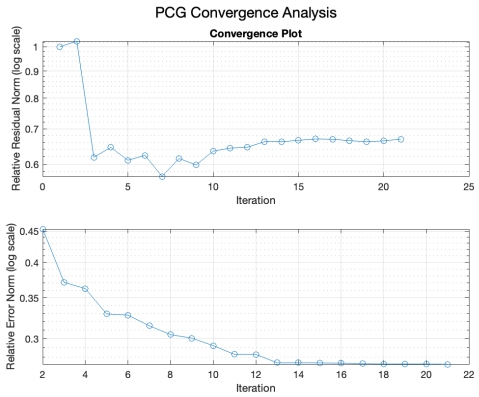

Figure 11: Iterations 20

4.2 2D Poisson Matrix - Dirichlet

Για να δοκιμάσω τη μέθοδο μου στο κλασικό μητρώο που προέρχεται από τη διακριτοποίηση με κεντρισμένες διαφορές της εξίσωσης Poisson με συνθήκες Dirichlet στο τετράγωνο, φτιάχνω το μητρώο με τον εξής τρόπο:

```
%Discretization
h = 1 / (n + 1);
x = linspace(0, 1, n + 2);
y = linspace(0, 0.5, m + 2);

%Create the matrix A
AA = sparse(n * m, n * m);

%loop over internal nodes (n)
for i = 2:n + 1
    %loop over points (m)
    for j = 2:m + 1
        %indices
        k = (j - 2) * n + i - 1;

        %diagonal entry
        AA(k, k) = -4 / h^2;

        %neighbors
        if i > 2
            AA(k, k - 1) = 1 / h^2;
        end
        if i < n + 1
            AA(k, k + 1) = 1 / h^2;
        end
        if j > 2
            AA(k, k - n) = 1 / h^2;
        end
        if j < m + 1
            AA(k, k + n) = 1 / h^2;
        end
    end
end
```

Για b διάνυσμα δημιουργώ ένα τυχαίο διάνυσμα με μήκος ίσο με τον αριθμό των γραμμών του A. Για ακριβή λύση ορίζω το:

$$AA \setminus bb$$

και καλώ τη **pcg_1084639**. Για προρύθμιση χρησιμοποιώ την επίλυση LU του A. Στη συνέχεια με τα επιστρεφόμενα αποτελέσματα, υπολογίζω το σχετικό υπόλοιπο και το σχετικό σφάλμα και τα κάνω plot.

Εκτέλεση **pcg_1084639** χωρίς προρύθμιση και max iterations 4:

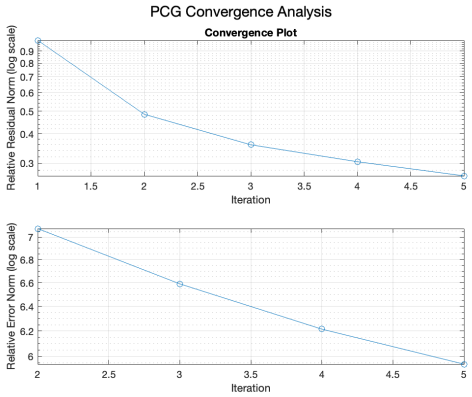

Figure 12: Without Preconditioners

Εκτέλεση **pcg_1084639** με προρύθμιση ilu(AA) και max iterations 4:

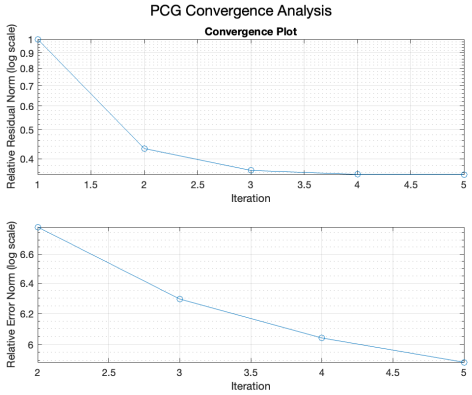

Figure 13: With Preconditioner

Εκτέλεση **pcg_1084639** χωρίς προρύθμιση και max iterations 20:

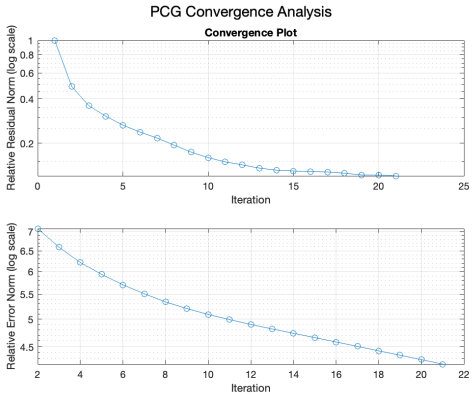

Figure 14: Without Preconditioners

Εκτέλεση **pcg_1084639** με προρύθμιση ilu(AA) και max iterations 20:

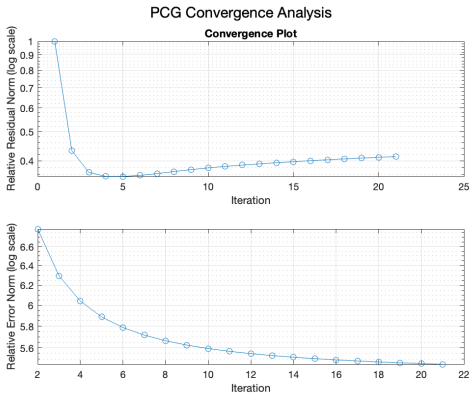

Figure 15: With Preconditioner

Παρατηρώ πως με την προρύθμιση η μέθοδος συγκλίνει καλύτερα και απαιτεί λιγότερα iterations.
Αναφορικά, για max iterations 4: χωρίς προρύθμιση η μέθοδος εκτελεί 4 βήματα, ενώ με προρύθμιση εκτελεί ξανά 4 βήματα, αλλά στη γραφική φαίνεται ξεκάθαρα η ταχύτερη σύγκλιση.
Για max iterations 20: χωρίς προρύθμιση η μέθοδος εκτελεί 20 βήματα, ενώ με προρύθμιση εκτελεί μόνο 4 βήματα.

Table 2: Relative Residual, Relative Error, Solve Time

Matrix	Preconditioner	Max Iterations	Iterations	Residual	Error	Time (s)
138_bus	none	4	0	1.0000	0	0.0015962
				4.1908	0.4735	
				3.5994	0.4697	
				3.5513	0.4686	
				3.2149	0.4664	
138_bus	ichol(A)	4	0	1.0000	0	0.0017692
				7.8969	0.4712	
				14.0738	0.4659	
				14.3186	0.4587	
				9.9963	0.4534	
138_bus	ilu(A)	4	4	1.0000	0	0.0072991
				1.0245	0.4536	
				0.6184	0.3711	
				0.6454	0.3622	
				0.6103	0.3292	
Poisson	none	4	4	1.0000	0	0.0015477
				0.4847	7.0780	
				0.3595	6.5895	
				0.3046	6.2145	
				0.2652	5.9393	
Poisson	ilu(A)	4	4	1.0000	0	0.071891
				0.4325	6.7911	
				0.3658	6.2942	
				0.3546	6.0411	
				0.3546	5.8884	