Exercise 2023-24 Documentation

Table of Contents

[Exercise 2023-24 Documentation](#MW_T_7F0C718B)   
[Cholesky Factorization Execution Time Analysis](#MW_T_F29B8E81)   
 [Overview](#MW_H_C9793A41)   
 [Code Structure](#MW_H_E5612F63)   
 [Data :](#MW_H_C4CA4124)   
 [Execution Time :](#MW_H_D55F5A82)   
 [Cubic Polynomial :](#MW_H_8D136833)   
 [Prediction :](#MW_H_EC3E5C2A)   
 [Visualization :](#MW_H_F1D07CA9)   
 [Polynomial Degree Comparison :](#MW_H_56FE625F)   
 [Usage](#MW_H_1A183D85)   
 [Notes](#MW_H_FD5BF09E)   
[Special solvers and sparse matrix](#MW_T_DCCD7D7D)   
 [Overview](#MW_H_1F296975)   
 [Function :](#MW_H_7958A49C)   
 [Parameters :](#MW_H_CE7BA9DF)   
 [Output :](#MW_H_2FB57289)   
 [Function Overview :](#MW_H_6C4766F3)   
 [Initialization:](#MW_H_505CF564)   
 [Forward Elimination:](#MW_H_01A53C4C)   
 [Back-Substitution:](#MW_H_D47B9B37)   
 [Handling Sparse Matrices:](#MW_H_BE5FA7D9)   
 [Algorithm Modifications for Sparse Matrices :](#MW_H_31DEAA6E)   
 [Notes](#MW_H_21B39FAC)   
 [How to Use](#MW_H_F6C7D7B3)   
[Tensor Operations](#MW_T_402DB5F3)   
 [Overview](#MW_H_6E3B2CE7)   
 [ttt\_myid - Tensor Outer and Inner Products](#MW_H_594E6E1B)   
 [Function :](#MW_H_B17AC924)   
 [Description :](#MW_H_E361D2A3)   
 [Example](#MW_H_D957F218)   
 [ttm\_myid - Tensor Matrix Multiplication](#MW_H_93BB15E2)   
 [Function](#MW_H_0986D7CC)   
 [Description](#MW_H_C3DE1C79)   
 [Example](#MW_H_22525A5F)   
 [ttv\_myid - Tensor Vector Multiplication](#MW_H_D2862A0B)   
 [Function](#MW_H_01DB56D3)   
 [Description](#MW_H_27304BD6)   
 [Example](#MW_H_DA59084A)   
 [test\_tensor - Testing Function](#MW_H_DDA5208F)   
 [Funciton :](#MW_H_5BAA407F)   
 [Description](#MW_H_33F2E753)   
 [Example](#MW_H_E70B4A93)   
 [Recommendations](#MW_H_FEEB83E2)   
 [Conclusion](#MW_H_37EA4419)   
[Solving symmetric positive certain systems with PCG](#MW_T_A91FCB7A)   
 [Overview](#MW_H_0CD8EE2F)   
 [Function](#MW_H_A1D37C42)   
 [Relation to MATLAB's pcg Function](#MW_H_A3FAD802)   
 [Test Cases](#MW_H_CCC45AC0)   
 [Test Case Code](#MW_H_E05CD116)   
 [Performance Analysis](#MW_H_E9D307F1)   
 [Plot](#MW_H_AA1376C9)   
 [Results](#MW_H_E254DA39)   
 [Conclusion](#MW_H_ED29B131)   
 [Note](#MW_H_4345BAB1)

Cholesky Factorization Execution Time Analysis

# Overview

This is a documentation on analyzes the execution time of Cholesky decomposition for randomly generated positive definite matrices of varying sizes. The code measures execution times, fits a cubic polynomial, and predicts execution times for additional matrix sizes. Users can explore the effect of different polynomial degrees on the fit.

# Code Structure

## Data :

Generates positive definite matrices of sizes ranging from 100 to 2000.

A = randn(n\_values(i));

Unrecognized function or variable 'n\_values'.

A = A \* A';

A = A + n\_values(i) \* eye(n\_values(i));

## Execution Time :

Measures the execution time for Cholesky using the 'timeit' function.

timing = timeit(@() chol(A));

execution\_times(i) = timing;

## Cubic Polynomial :

Fits a cubic polynomial to the measured execution times.

[p, S, mu] = polyfit(n\_values, execution\_times, degree);

## Prediction :

Predicts execution times for all sizes and additional values.

predict\_time = polyval(p, n\_values\_all, S, mu);

predict\_times\_extra = polyval(p, n\_values\_extra, S, mu);

## Visualization :

Plots measured times, cubic fit, and predicted times for extra values.

figure;

plot(n\_values, execution\_times, 'o','LineWidth', 1, 'MarkerSize', 8, 'Color', 'red', 'DisplayName', 'Measured Times');

hold on;

plot(n\_values\_all, predict\_time, '-','LineWidth', 1, 'MarkerSize', 8, 'Color', 'green', 'DisplayName', 'Cubic Fit');

plot(n\_values\_extra, predict\_times\_extra, 'x','LineWidth', 1, 'MarkerSize', 8, 'Color', 'blue', 'DisplayName', 'Predicted Times (Extra)');

xlabel('Matrix Size ');

ylabel('Execution Time ');

title('Cholesky Decomposition Execution Time Analysis');

legend('Location', 'SouthEast');

## Polynomial Degree Comparison :

Compares the cubic fit with polynomial fits of degrees 2 and 4.

figure;

plot(n\_values, execution\_times, 'o', 'DisplayName', 'Measured Times');

grid on;

hold on;

plot(n\_values\_all, predict\_time, '-','LineWidth', 1, 'MarkerSize', 8, 'Color', 'red', 'DisplayName', 'Cubic Fit');

plot(n\_values\_all, predict\_times\_2, '--','LineWidth', 1, 'MarkerSize', 8, 'Color', 'green', 'DisplayName', 'Degree 2 Fit');

plot(n\_values\_all, predict\_times\_4, '-.','LineWidth', 1, 'MarkerSize', 8, 'Color','blue', 'DisplayName', 'Degree 4 Fit');

xlabel('Matrix Size');

ylabel('Execution Time');

title('Cholesky Decomposition Execution Time Analysis with polynomial 2 and 4');

legend('Location', 'SouthEast');

# Usage

1. Run the script to analyze Cholesky execution times.
2. Explore the impact of different polynomial degrees on the fit.

# Notes

* If an error occurs for a specific matrix size, it indicates that the matrix is not positive definite.
* Polynomial degree 3 (cubic) is the default, but users can experiment with degrees 2 and 4 for comparison.
* You can customize the size of the matrices and plot the figures as necessary

Special solvers and sparse matrix

# Overview

The provided MATLAB function solveTridiagonalSystem implements the Thomas algorithm for solving a tridiagonal system of linear equations with sparse matrices. This adaptation is particularly beneficial when dealing with large systems that exhibit sparsity. The documentation outlines the modifications made to the original algorithm to handle sparse matrices efficiently.

# Function :

function x = solveTridiagonalSystem(a, b, c, d)

### Parameters :

* a: Diagonal coefficients (column vector)
* b: Upper diagonal coefficients (column vector)
* c: Lower diagonal coefficients (column vector)
* d: Right-hand side coefficients (column vector)

### Output :

* x: Solution vector

## Function Overview :

### Initialization:

* Check input sizes for consistency.
* Initialize vectors alpha and s with values from b and d, respectively.

### Forward Elimination:

* Perform forward elimination to transform the system into an upper triangular form.
* Update vectors alpha and s accordingly.

### Back-Substitution:

* Use back-substitution to find the solution vector x.

### Handling Sparse Matrices:

* Utilize sparse matrix operations for efficient memory usage.
* Update the algorithm to work seamlessly with sparse matrices.

## Algorithm Modifications for Sparse Matrices :

1. Use Sparse Matrix Operations:

* Adjust the algorithm to use sparse matrix operations (e.g., spdiags) when dealing with sparse coefficients.

1. Optimized Storage:

* Leverage the sparsity of the matrix to optimize storage and computation.

# Notes

* The function assumes the input coefficients are column vectors of the same length.
* It is essential to provide valid input coefficients for a well-defined tridiagonal system.

# How to Use

1. Define the tridiagonal system coefficients (a, b, c, and d).
2. Call the solveTridiagonalSystemSparse function with the defined coefficients.
3. Retrieve the solution vector x.

Tensor Operations

# Overview

This documentation provides an overview of four MATLAB functions designed for tensor operations: ttt\_myid, ttm\_myid, ttv\_myid, and the testing function test\_tensor. These functions are implemented to perform outer and inner products of tensors, matrix-tensor multiplications, and tensor-vector products.

# ttt\_myid - Tensor Outer and Inner Products

### Function :

function C = ttt\_myid(A, B, varargin)

### Description :

* Computes the outer or inner product of two multidimensional arrays A and B.
* Supports both outer product (C = ttt\_myid(A, B)) and inner product (t = ttt\_myid(A, B, 'all')).
* Input tensors A and B must be non-empty numeric arrays.

### Example

A = randn(3, 3);

B = randn(3, 3);

ttt\_myid(A, B); % Outer product

ttt\_myid(A, B, 'all'); % Inner product

# ttm\_myid - Tensor Matrix Multiplication

### Function

function Y = ttm\_myid(X, V, N, tflag)

### Description

* Computes the n-mode product of tensor X with a matrix V.
* The integer N specifies the dimension in X along which V is multiplied.
* Supports an optional transpose flag (tflag) for matrix transposition.
* Input validation checks for appropriate types and dimensions.

## Example

X = randn(3, 3, 3);

V = randn(3, 2);

N = 2;

ttm\_myid(X, V, N); % No transposition

ttm\_myid(X, V, N, 't'); % With transposition

%Some time you might have to use the transportation depending on the random example

# ttv\_myid - Tensor Vector Multiplication

### Function

function Y = ttv\_myid(X, V, N)

### Description

* Computes the product of a multidimensional array X with a (column) vector V.
* The integer N specifies the dimension in X along which V is multiplied.
* Input validation checks for appropriate types and dimensions.

### Example

X = randn(3, 3, 3);

V = randn(3, 1);

N = 2;

ttv\_myid(X, V, N); % Tensor-vector multiplication

% You have to be very causious about types and dimensions of the inputs variables while using this ttv\_myid

All functions in a script must be closed with an 'end'.

### test\_tensor - Testing Function

### Funciton :

function test\_tensor

### Description

* Tests the correctness of the implemented tensor operations by comparing results with corresponding MATLAB Tensor Toolbox functions.
* Uses predefined test cases to check the validity of the functions.
* Provides detailed error messages in case of failures.

### Example

test\_tensor(); % Run the tensor operation tests

# Recommendations

* Ensure proper installation of the MATLAB Tensor Toolbox for result validation.
* Validate the functions using a diverse set of test cases to ensure correctness.
* Validate the funciton dimensions and types.
* Be careful with it

# Conclusion

These tensor operations functions provide a versatile set of tools for tensor manipulation and product computations. By adhering to MATLAB conventions and including thorough input validation, these functions offer reliability and accuracy in various tensor-related tasks.

Solving symmetric positive certain systems with PCG

# Overview

This documentation presents an in-depth analysis of the pcg\_myid function, an implementation of the Preconditioned Conjugate Gradient (PCG) method for solving linear systems. The primary objective is to provide users with insights into the function's behavior, performance under various conditions, and its applicability to different types of matrices.

The pcg\_myid function solves a linear system  busing the Preconditioned Conjugate Gradient (PCG) method. It takes several optional parameters such as tolerance (tol), maximum iterations (maxit), preconditioners (M1, M2), initial guess (x0), and the exact solution (xsol).

## Function

The pcg\_myid function serves as an implementation of the Preconditioned Conjugate Gradient (PCG) method for solving linear systems. It offers flexibility through various optional parameters, enabling users to customize the solver based on specific requirements. The function's primary objective is to efficiently find solutions to systems of linear equations in the form .

### Relation to MATLAB's pcg Function

It is important to note that the pcg\_myid function shares similarities with the built-in MATLAB function pcg. The pcg function, provided by MathWorks, also implements the Preconditioned Conjugate Gradient method with a similar set of parameters.

Users familiar with the pcg function may find the pcg\_myid function to be a customizable alternative, allowing for a more tailored approach to solving linear systems.

function [x, flag, relres, iter, resvec, errvec] = pcg\_myid(A, b, tol, maxit, M1, M2, x0, xsol, varargin)

% Call the original pcg function

[x, flag, relres, iter, resvec] = pcg(A, b, tol, maxit, M1, M2, x0, varargin{:});

% Check if xsol and x have the same dimensions

if numel(xsol) ~= numel(x)

error('The dimensions of xsol and x must agree.');

end

% Calculate the A-norm error2

errvec = norm(xsol - x);Test Cases

end

The function is tested on two matrices:  and  The right-hand side (b) and known exact solutions (xsol) are defined accordingly.

### Test Case Code

clc

clear all

% Generate matrices

n=30;

tol = 1e-6;

A\_poisson = generatePoissonMatrix(n);

A\_suitesparse = loadSuiteSparseMatrix();

x0\_poisson = (zeros(size(A\_poisson, 1), 1));

x0\_suiteparse = (zeros(size(A\_suitesparse,1),1));

% Define right sides

b\_poisson = (A\_poisson \* ones(size(A\_poisson, 1), 1));

b\_suitesparse =(( A\_suitesparse \* ones(size(A\_suitesparse, 1), 1)));

% Known exact solutions (for testing)

xsol\_poisson = exactSolutionPoisson(n); % exact solution

xsol\_suitesparse =ones(size(A\_suitesparse, 1), 1);% Define or calculate exact solution for SuiteSparse problem

% Call the modified pcg function for both matrices

maxit = 4 \* n;

%no presetting

[x1, flag1, relres1, iter1, resvec1, errvec1] = pcg\_myid(A\_poisson, b\_poisson, 1e-6, maxit, [], [], x0\_poisson, xsol\_poisson);

[x2, flag2, relres2, iter2, resvec2, errvec2] = pcg\_myid(A\_suitesparse, b\_suitesparse, 1e-6, maxit, [], [], x0\_suiteparse, xsol\_suitesparse);

% ichol presetting

[x5, flag5, relres5, iter5, resvec5, errvec5] = pcg\_myid(A\_suitesparse, b\_suitesparse, 1e-6, maxit, ichol(A\_suitesparse), [], x0\_suiteparse, xsol\_suitesparse);

[x3, flag3, relres3, iter3, resvec3, errvec3] = pcg\_myid(A\_poisson, b\_poisson, 1e-6, maxit, ichol(A\_poisson), [], x0\_poisson, xsol\_poisson);

%custom presetting

[x4, flag4, relres4, iter4, resvec4, errvec4] = pcg\_myid(A\_suitesparse, b\_suitesparse, 1e-6, maxit, custom\_preconditioner\_suite\_sparse(A\_suitesparse), [], x0\_suiteparse, xsol\_suitesparse);

[x6, flag6, relres6, iter6, resvec6, errvec6] = pcg\_myid(A\_poisson, b\_poisson, 1e-6, maxit, custom\_preconditioner(A\_poisson), [], x0\_poisson, xsol\_poisson);

## Performance Analysis

The performance of the pcg\_myid function is analyzed for different preconditioning strategies and compared visually.

## Plot

The performance analysis includes four distinct scenarios, each visualized through plots:

No Presetting:

* In this scenario, the pcg\_myid function is executed without any preconditioning. The plot illustrates the convergence behavior and the A-norm error over iterations.

figure;

sgtitle( 'No presetting' )

subplot(2, 2, 1);

semilogy(1:iter1, resvec1(1:iter1), '-o');

title('Matrix-poisson Residuals');

xlabel('Iteration');

ylabel('Relative Residual');

subplot(2, 2, 2);

semilogy(1:iter1, errvec1(1:min(iter1, length(errvec1))), '-o');

title('Matrix-poisson A-norm Error');

xlabel('Iteration');

ylabel('A-norm Error');

subplot(2, 2, 3);

semilogy(1:iter2, resvec2(1:iter2), '-o');

title('Matrix-suitesparse Residuals');

xlabel('Iteration');

ylabel('Relative Residual');

subplot(2, 2, 4);

semilogy(1:iter2, errvec2(1:min(iter2, length(errvec2))), '-o');

title('Matrix-suitesparse A-norm Error');

xlabel('Iteration');

ylabel('A-norm Error');

Presetting with Incomplete Cholesky (IC(0)):

* This scenario employs an incomplete Cholesky (IC(0)) preconditioner (L\_poisson) to enhance the convergence of the PCG method.

% ichol preconditioning

figure;

sgtitle( 'Precondtioned ichol' )

subplot(2, 2, 1);

semilogy(1:iter3, resvec3(1:iter3), '-o');

title('Matrix-poisson ichol Residuals');

xlabel('Iteration');

ylabel('Relative Residual');

subplot(2, 2, 2);

semilogy(1:iter3, errvec3(1:min(iter3, length(errvec3))), '-o');

title('Matrix-poisson ichol A-norm Error');

xlabel('Iteration');

ylabel('A-norm Error');

subplot(2, 2, 3);

semilogy(1:iter5, resvec5(1:iter5), '-o');

title('Matrix-suitesparse ichol Residuals');

xlabel('Iteration');

ylabel('Relative Residual');

subplot(2, 2, 4);

semilogy(1:iter5, errvec5(1:min(iter5, length(errvec5))), '-o');

title('Matrix-suitesparse ichol A-norm Error');

xlabel('Iteration');

ylabel('A-norm Error');

Custom Pre-Conditioner (ichol):

* In this scenario, a custom preconditioner (M\_custom) generated using the incomplete Cholesky factorization (ichol) is applied to the linear system.

% for custom preconditioning

figure;

sgtitle( 'custom Precondtioned' )

subplot(2, 2, 1);

semilogy(1:iter4, resvec4(1:iter4), '-o');

title('Matrix-suitesparse Custom Residuals');

xlabel('Iteration');

ylabel('Relative Residual');

subplot(2, 2, 2);

semilogy(1:iter4, errvec4(1:min(iter4, length(errvec4))), '-o');

title('Matrix-suitesparse Custom A-norm Error');

xlabel('Iteration');

ylabel('A-norm Error');

subplot(2, 2, 3);

semilogy(1:iter6, resvec6(1:iter6), '-o');

title('Matrix-poisson Custom Residuals');

xlabel('Iteration');

ylabel('Relative Residual');

subplot(2, 2, 4);

semilogy(1:iter6, errvec6(1:min(iter6, length(errvec6))), '-o');

title('Matrix-poisson Custom A-norm Error');

xlabel('Iteration');

ylabel('A-norm Error');

plot(1:length(errvec\_suitesparse), errvec\_suitesparse, '-\*');

title('No Presetting (SuiteSparse)');

## Results

The results are shown in the generated plots. Each subplot represents a different preconditioning strategy.

## Conclusion

In conclusion, the performance analysis of the pcg\_myid function provides insights into the efficiency of different preconditioning strategies when solving linear systems. The choice of preconditioner significantly impacts the convergence behavior of the PCG method.

# Note

To replicate the experiments and generate the presented results accurately, ensure that you have the following files in your working directory:

* 1138\_bus.mat for the SuiteSparse matrix (A\_suitesparse).
* The code for generating the Poisson matrix using the generatePoissonMatrix function.

These files are essential for running the test cases and plotting the results as demonstrated in this documentation. Make sure to have the required data and code available to reproduce the presented findings effectively.

## 