Numerical Methods: Assignment #2

Hong Chenhui
drredthered.github.io

Zhejiang University — March 31, 2024

Warning: The Chinese translation of this assignment has some severe problems on the physical essence for this model. One would be required to read the actual original content on this matter.

1 Problem

- **Question** A stage extraction process is depicted in Eq.(1). In such systems, a stream containing a weight fraction Y_{in} of a chemical enters from the left at a mass flow rate of F_1. Simultaneously, a solvent carrying a weight fraction X_{in} of the same chemical enters from the right at a flow rate of F_2. Thus, for stage i, a mass balance can be represented as

$$F_1 Y_{i-1} + F_2 X_{i+1} = F_1 Y_i + F_2 X_i \tag{1}$$

  At each stage, an equilibrium is assumed to be established between Y_i and X_i as in

$$X_i = K Y_i \tag{2}$$

  where K is called a distribution coefficient. Eq.(2) can be solved for X_i and substituted into Eq.(1) to yield

$$Y_{i-1} - (1 + \frac{F_2}{F_1} K) Y_i = (\frac{F_2}{F_1} K) Y_{i+1} \tag{3}$$

- If $F_1 = 500kg/h$, $Y_{in} = 0.5$, $F_2 = 300kg/h$, $X_{in} = 0$, and $K = 4$, determine the values of Y_{out} and X_{out} if a n-stage reactor is used.$(n = 3, 5, 10, 20, 25, 50, 100)$
- If $F_1 = 500kg/h$, $n = 20$, $F_2 = 300kg/h$, $X_{in} = 0$, and $K = 4$, determine the values of Y_{out} and X_{out} if the inflow $Y_{in} = 0.3, 0.5, 0.7, 0.9$. Note that Eq.(3) must be modified to account for the inflow weight fractions when applied to the first and last stages.

1.1 Theoretical viewpoint

Question

For any multistage counter-current extraction process, the linear equation would always be in the form of a tridiagonal matrix. Typical 3×3 matrix of this problem would be

$$\begin{bmatrix} -\frac{17}{5} & \frac{12}{5} & 0 \\ 1 & -\frac{17}{5} & \frac{12}{5} \\ 0 & 1 & -\frac{17}{5} \end{bmatrix} \begin{bmatrix} Y_1 = \frac{X_{out}}{4} \\ Y_2 \\ Y_3 = Y_{out} \end{bmatrix} = \begin{bmatrix} -Y_{in} \\ 0 \\ X_{in} \end{bmatrix}$$

$n \times n$ matrix follows the similar rules, hence the solution to this problem enumerating as follow.

Question

Algorithm 1: Naive Gauss Elimination

Data: A matrix A of size $n \times n$ and a vector b of size n
Result: The solution vector x of size n
for $k = 1$ **to** $n - 1$ **do**
  for $i = k + 1$ **to** n **do**
    $r = \frac{A(i,k)}{A(k,k)}$;
    $A(i, k : n) = A(i, k : n) - r \cdot A(k, k : n)$;
    $b(i) = b(i) - r \cdot b(k)$;
  end
end
for $k = n$ **to** 1 **do**
  $x(k) = \frac{b(k) - \sum_{j=k+1}^{n} A(k,j) \cdot x(j)}{A(k,k)}$;
end

Algorithm 2: Thomas Algorithm

Data: A tridiagonal matrix A of size $n \times n$ and a vector b of size n
Result: The solution vector x of size n
for $k = 1$ **to** $n - 1$ **do**
  $r = \frac{A(k+1,k)}{A(k,k)}$;
  $A(k + 1, k : n) = A(k + 1, k : n) - r \cdot A(k, k : n)$;
  $b(k + 1) = b(k + 1) - r \cdot b(k)$;
end
for $k = n$ **to** 1 **do**
  $x(k) = \frac{b(k) - A(k,k+1:n) \cdot x(k+1:n)}{A(k,k)}$;
end

Algorithm 3: Gaussian Elimination with Partial Pivoting (GEPP)

Input : A, b: coefficient matrix and right-hand side vector of $Ax = b$
Output: x: solution vector
for $k = 1$ **to** $n - 1$ **do**
  Find $p \in \{k, k + 1, \ldots, n\}$ such that $|a_{pk}| = \max_{i=k}^{n} |a_{ik}|$;
  Swap rows k and p of A and b;
  for $i = k + 1$ **to** n **do**
    $\ell_{ik} = \frac{a_{ik}}{a_{kk}}$;
    for $j = k + 1$ **to** n **do**
      $a_{ij} = a_{ij} - \ell_{ik} a_{kj}$;
    end
    $b_i = b_i - \ell_{ik} b_k$;
  end
end
for $i = n$ **to** 1 **do**
  $x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=i+1}^{n} a_{ij} x_j \right)$;
end

Question

Algorithm 4: Jacobi Method

Data: A matrix A of size $n \times n$ and a vector b of size n
Result: The solution vector x of size n
Input : A, b: coefficient matrix and right-hand side vector of $Ax = b$
Output: x: solution vector
Initialize x with zeros;
while *not converged* **do**
  for $i = 1$ **to** n **do**
    $sum = 0$;
    for $j = 1$ **to** n **do**
      if $j \neq i$ **then**
        $sum = sum + A(i, j) \cdot x(j)$;
      end
    end
    $x(i) = \frac{1}{A(i,i)} \cdot (b(i) - sum)$;
  end
end

Compare to see the difference.

2 Implementation

The result, for the sake of simplicity, obtains 14-digits precision.

1. when n varies:

   - Naive Gauss Elimination / Thomas Algorithm

n	NaiveGauss		Thomas	
	Y_{out}	X_{out}	Y_{out}	X_{out}
3	0.021754263835712	0.797076226940480	0.021754263835712	0.797076226940480
5	0.003682214843391	0.827196308594348	0.003682214843391	0.827196308594348
10	0.000046004776225	0.833256658706291	0.000046004776225	0.833256658706291
20	0.000000007255410	0.833333321240983	0.000000007255410	0.833333321240983
25	0.000000000091118	0.833333333181470	0.000000000091118	0.833333333181470
50	0	0.833333333333333	0	0.833333333333333
100	0	0.833333333333333	0	0.833333333333333

   - GEPP/Jacobi Method

n	GEPP		Jacobi		
	Y_{out}	X_{out}	Y_{out}	X_{out}	$iter$
3	0.021754263835712	0.797076226940480	0.021754263835712	0.797076226940480	81
5	0.003682214843391	0.827196308594348	0.003682214843391	0.827196308594348	146
10	0.000046004776225	0.833256658706291	0.000046004776225	0.833256658706291	241
20	0.000000007255410	0.833333321240983	0.000000007255410	0.833333321240982	293
25	0.000000000091118	0.833333333181470	0.000000000091118	0.833333333181469	298
50	0	0.833333333333333	0	0.833333333333333	300
100	0	0.833333333333333	0	0.833333333333333	300

2. when Y_{in} varies:

   - Naive Gauss Elimination / Thomas Algorithm

Y_{in}	NaiveGauss		Thomas	
	Y_{out}	X_{out}	Y_{out}	X_{out}
0.3	0.000000004353246	0.499999992744590	0.000000004353246	0.499999992744590
0.5	0.000000007255410	0.833333321240983	0.000000007255410	0.833333321240983
0.7	0.000000010157574	1.166666649737376	0.000000010157574	1.166666649737376
0.9	0.000000013059738	1.499999978233770	0.000000013059738	1.499999978233770

   - GEPP/Jacobi Method

Y_{in}	GEPP		Jacobi		
	Y_{out}	X_{out}	Y_{out}	X_{out}	$iter$
0.3	0.000000004353246	0.499999992744590	0.000000004353246	0.499999992744589	287
0.5	0.000000007255410	0.833333321240983	0.000000007255410	0.833333321240982	293
0.7	0.000000010157574	1.166666649737376	0.000000010157574	1.166666649737376	297
0.9	0.000000013059738	1.499999978233770	0.000000013059738	1.499999978233769	298

3 Analysis

3.1 Not much difference on the final result

This lack of discrepancy in the results indicates that all four methods are providing accurate and consistent solutions to the given problem. It suggests that the choice of method does not significantly affect the final results and any of the four methods can be used with confidence. Or last least under 14-digits precision. The only exception arises on the only iteration procedure, which is Jacobi Method, and it is with a cause. We deliberately set the discrepancy threshold at 14-digit precision, which would result in some variation on the last digit. With condition small enough, the method could technically receive any precision one craves for.

Yet under higer precision requirements, the situation differs. For instance, these are the precise Y_{out} values put forward by 3 of the methods when $n = 100$ and $Y_{in} = 0.3$:

Algorithm	Y_{out}
Naive Gauss	1.666916622996875e-39
Thomas Method	1.666916622996854e-39
GEPP	1.666916622996875e-39

Naive Gauss and GEPP are the same, while Thomas Method is slightly different. Applying the far-more accurate MATLAB matrix solver, one could see the Gaussian one as the true value, and the faster Thomas one not.

3.2 Why Thomas, and why GEPP?

Although Gauss elimination or conventional LU decomposition can be employed to solve banded equations, they are inefficient, because if pivoting is unnecessary none of the elements outside the band would change from their original values of zero. Thus, unnecessary space and time would be expended on the storage and manipulation of these useless zeros. If it is known beforehand that pivoting is unnecessary, very effi cient algorithms can be developed that do not involve the zero elements outside the band. Because many problems involving banded systems do not require pivoting, these alternative algorithms, as described next, are the methods of choice.

For such systems, the solution can be obtained in $O(n)$ operations instead of $O(n^3)$ required by Gaussian elimination. A first sweep eliminates the a_i's, and then an (abbreviated) backward substitution produces the solution.

Thomas' algorithm is not stable in general, but is so in several special cases, such as when the matrix is diagonally dominant (either by rows or columns) or symmetric positive definite; for a more precise characterization of stability of Thomas' algorithm, see Higham Theorem. If stability is required in the general case, Gaussian elimination with partial pivoting (GEPP) is recommended instead.

4 Codes

//functions

NaiveGauss.m

```
function x = naiveGauss(A, b)
% Check if the matrix is square
[m, n] = size(A);
if m ~= n
        error('Matrix A must be square');
end

if n ~= length(b)
        error('Dimensions of A and b are inconsistent');
end

Ab = [A, b];

% Forward elimination
for k = 1:n-1
        for i = k+1:n
                factor = Ab(i,k) / Ab(k,k);
                Ab(i,k:n+1) = Ab(i,k:n+1) - factor * Ab(k,k:n+1);
        end
end

% Back substitution
x = zeros(n, 1);
x(n) = Ab(n,n+1) / Ab(n,n);
for i = n-1:-1:1
        x(i) = (Ab(i,n+1) - Ab(i,i+1:n)*x(i+1:n)) / Ab(i,i);
end
end
```

thomasAlgorithm.m

```
function x = thomasAlgorithm(a, b, c, d)

n = length(d);
c_prime = zeros(n, 1);
d_prime = zeros(n, 1);

% Forward elimination
c_prime(1) = c(1) / b(1);
d_prime(1) = d(1) / b(1);
for i = 2:n-1
        c_prime(i) = c(i) / (b(i) - a(i) * c_prime(i - 1));
        d_prime(i) = (d(i) - a(i) * d_prime(i - 1)) /
        (b(i) - a(i) * c_prime(i - 1));
end
```

```
d_prime(n) = (d(n) - a(n) * d_prime(n - 1)) /
(b(n) - a(n) * c_prime(n - 1));

% Backward substitution
x = zeros(n, 1);
x(n) = d_prime(n);
for i = n-1:-1:1
        x(i) = d_prime(i) - c_prime(i) * x(i + 1);
end

end
```

gaussianEliminationWithPartialPivoting.m

```
function x = gaussianEliminationWithPartialPivoting(A, b)

augmentedMatrix = [A, b];

n = size(augmentedMatrix, 1);

% Forward elimination
for k = 1:n-1
        % Partial pivoting
        [~, maxIndex] = max(abs(augmentedMatrix(k:n, k)));
        maxIndex = maxIndex + k - 1;
        if maxIndex ~= k
                % Swap rows k and maxIndex
                augmentedMatrix([k maxIndex], :) =
                augmentedMatrix([maxIndex k], :);
        end

        % Perform elimination
        for i = k+1:n
                factor = augmentedMatrix(i, k)
                / augmentedMatrix(k, k);
                augmentedMatrix(i, k+1:end) = augmentedMatrix(i, k+1:end) -
                factor * augmentedMatrix(k, k+1:end);
        end
end

% Back substitution
x = zeros(n, 1);
x(n) = augmentedMatrix(n, n+1) / augmentedMatrix(n, n);
for i = n-1:-1:1
        x(i) = (augmentedMatrix(i, n+1) -
        augmentedMatrix(i, i+1:n) * x(i+1:n)) / augmentedMatrix(i, i);
end

end
```

jacobi.m

```
function [x, iter] = jacobi(A, b, x0, tol, max_iter)

n = size(A, 1);

% Initialize iteration counter
iter = 0;
```

```
% Initialize solution vector
x = x0;

% Main loop for Jacobi iteration
while iter < max_iter
        % Increment iteration counter
        iter = iter + 1;

        x_old = x;

        for i = 1:n
                sigma = 0;
                for j = 1:n
                        if j ~= i
                                sigma = sigma + A(i, j) * x_old(j);
                        end
                end
                x(i) = (b(i) - sigma) / A(i, i);
        end

        % Check for convergence
        if norm(x - x_old, inf) < tol
                break;
        end
end

end
```

//script
Extraction.m

```
clear;
clc;

%initialization

n = 100;
Yin = 0.3;

%for normal ones

for i=1:n
        for j=1:n
                if (j == i-1)
                        A(i,j) = 1;
                elseif(j == i)
                        A(i,j) = -17/5;
                elseif(j == i+1 )
                        A(i,j) = 12/5;
                else A(i,j) = 0;
                end
        end
        if(i == 1)
                b(i) = -Yin;
        else
                b(i) = 0;
        end
end
```

```
b = transpose(b);

x1 = naiveGauss(A,b);

%for thomas

for i = 1:n
        if(i == 1)
                a2(i) = 0;
                d2(i) = -Yin;
        else
                a2(i) = 1;
                d2(i) = 0;
        end
        % Subdiagonal
        b2(i) = -17/5;
        % Main diagonal
        if(i == n)
                c2(i) = 0;
        else
                c2(i) = 12/5;
        end
        % Superdiagonal
end


a2 = transpose(a2);
b2 = transpose(b2);
c2 = transpose(c2);
d2 = transpose(d2);

%a2 = [0; 1; 1; 1]; % Subdiagonal
%b2 = [-23/3; -23/3; -23/3; -23/3];   % Main diagonal
%c2 = [20/3; 20/3; 20/3; 0]; % Superdiagonal
%d2 = [-1/2; 0; 0; 0];   % Right-hand side vector

x2 = thomasAlgorithm(a2, b2, c2, d2);

%for GEPP

x3 = gaussianEliminationWithPartialPivoting(A, b);

 %for Jacobi%

% Initial guess for the solution vector
x0 = zeros(size(b));

% Set tolerance and maximum number of iterations
tol = 0.5e-16;
max_iter = 1000;

% Call Jacobi method function
[x4, iter] = jacobi(A, b, x0, tol, max_iter);

 %Jacobi Ends%
```

```
%displays%
disp('Solution vector:');
disp(x1);

disp('Y-out');
disp(x1(n));

disp('X-out');
disp(4*x1(1))

disp('Solution vector:');
disp(x2);

disp('Y-out');
disp(x2(n));

disp('X-out');
disp(4*x2(1))

disp('Solution vector:');
disp(x3);

disp('Y-out');
disp(x3(n));

disp('X-out');
disp(4*x3(1))

disp('Solution vector:');
disp(x4);

disp('Y-out');
disp(x4(n));

disp('X-out');
disp(4*x4(1))

disp(['Number of iterations: ', num2str(iter)]);

x5 = A\b;
disp(x5(n));
```