Numerical Methods II

Project 3

Task 7: Calculating largest and smallest (in modulus) root of polynomial:

$$w(x) = x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0$$

by the means of the power method and inverse power method applied to the companion matrix.

January 25, 2019

Rafał Ziomek

Group: EA2

Contents

METHOD DESCRIPTION

Problem statement

For given polynomial:

$$w(x) = x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0$$

program calculates largest and smallest (in modulus) root. Largest root is calulated using power method, smalles using inverse power method. Both are applied to the companion matrix C_w, defined as:

$$\mathrm{C}_w = \begin{pmatrix} -a_{n-1} & -a_{n-2} & \dots & -a_1 & a_0 \\ 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 \end{pmatrix}$$

Gaussian elimination with complete pivoting is used to solve the system of linear equations in the inverse power method. The eigenvalues of companion matrix coincide with roots of polynomial w, hence above methods might be aplied.

Power method

In order to apply the power method on matrix M, following conditions must be met:

- Matrix M must be diagonalizable
- Matrix M must have dominant eigenvalue

Companion matrix C_w meets these requiremetns, when polynomial $w(x)$ has dominant root. Power method also may be used to approximate corresponding eigenvetor (this property is ignored in my program). Method starts with approximation x_0 of eigenvector and is described by recurrence relation:

$$x_{k+1} = \frac{C_w x_k}{||C_w x_K||}$$

Dominant eigenvalue is then caluclated from ratio:

$$r_k = \frac{\phi(x_{k+1})}{\phi(x_k)}$$

where $\phi(x)$ is is some linear function. Result of the power method is largest in modulus eigenvalue (from ratio r).

Inverse power method

Applying inverse power method requires that matrix C_w is nonsingular, since 0 is not an eigenvalue. It is possible to compute inverse of smallest eigenvalue (in modulus) of matrix C_w, by applying the power method to the inverse of C_w. However, it is not a good idea to compute it's inverse first and then use $x^{k+1} = C_w x^k$, rather we obtain x^{k+1} by solving the equation

$$C_w x^{k+1} = x^k$$

This can be done efficiently by using Gaussian elimination with complete pivoting (or any other method for solving system of equations).

Algorithm description

In order to apply mentioned methods for given polynomial, my program consists of a few modules.

- Main function for calculating smallest and largest root
- Function used to create companion matrix C_w for given polynomial $w(x)$
- Function implementing power method
- Function implementing inverse power method
- Function for Gaussian elimination with complete pivoting

Power method implementation

Algorithm takes created from given polynomial $w(x)$ companion matrix C_w and starts with random initial vector v_0. Then described power method is applied, where $\phi(x)$ is simply first value from the approximated eigenvector.

Inverse power method implementation

In order to optimize calculations, the algorithm performs Gaussian elimination with complete pivoting (GECP) on companion matrix C_w, then applies inverse power method and use GECP result to calculate smallest root.

MATLAB FUNCTIONS CODE

createCompanionMatrix

```
%Creating companion matrix out of polynomial coefficients
%Input: coefs – polynomial coefficients
%Output: compMatrix – generated companion matrix
function [compMatrix] = createCompanionMatrix(coefs)
    [~,n]=size(coefs);
    diagonal=ones(1,n-1);
    compMatrix = diag(diagonal,-1);
    compMatrix(1,:)=-coefs;
end
```

powerMethod

```
%Calculating max root using power method
%Input: companionMatrix – companion matrix of a polynomial, steps –
    number
%of steps to perform
%Output: max_root – largest in absolute value eigenvalue of a matrix
function [max_root] = powerMethod(companionMatrix, steps)
    [~,n] = size(companionMatrix);
    x = rand(n,1);
    for i=1:steps
        y = companionMatrix*x;
        max_root = y(1)/x(1);
        x = y/max(abs(y));
    end
end
```

inversePowerMethod

```
%Function calculates smallest (in modulus) eigenvalue (which in this
    problem is also smallest root of polynomial) of a matrix
%Input: companionMatrix - companion matrix of a polynomial, steps -
    number
%of steps to perform
%Output: min_root - smallest in absolute value eigenvalue of a matrix
function [min_root] = inversePowerMethod(companionMatrix, steps)
    [~,n] = size(companionMatrix);
    x = rand(n,1);
    [L,U,P,Q] = GECP(companionMatrix);
    PQ = P*Q;
    for i=1:steps
        y = solveSubstitute(L,U,PQ,x);
        min_root = y(1)/x(1);
        x=y/max(abs(y));
    end
    min_root = 1/min_root;
end
```

GECP

```
%Gaussian elimination with complete pivoting, function returns all
    matrices
%needed to perform substitution
%Input: A - matrix on which GECP is performed
%Output : L, U, P, Q - matrices, L and U are lower and upper triangular
%rexpectively, P and Q are transformation matrices
function [L, U, P, Q] = GECP(A)
[n, ~] = size(A);
p = 1:n;
q = 1:n;
for k = 1:n-1
    [max_val, rows] = max( abs(A(k:n, k:n)));
    [~, columns] = max(max_val);
```

```
13      row = rows(columns)+k-1;
14      column = columns+k-1;
15      A( [k, row], : ) = A( [row, k], : );
16      A( :, [k, column] ) = A( :, [column, k] );
17      p( [k, row] ) = p( [row, k] ); q( [k, column] ) = q( [column, k] );
18      if A(k,k) == 0
19        break
20      end
21      A(k+1:n,k) = A(k+1:n,k)/A(k,k);
22      i = k+1:n;
23      A(i,i) = A(i,i) - A(i,k) * A(k,i);
24  end
25  L = tril(A,-1) + eye(n);
26  U = triu(A);
27  P = eye(n);
28  P = P(p,:);
29  Q = eye(n);
30  Q = Q(:,q);
```

solveSubstitute

```
1  %Solving system of equation Ly = PQB -> Ux = y by substitution
2  %Input: matrices L, U, PQ, vector b; L, U and PQ are from GECP
3  %Output: x - calculated result vector
4  function [x] = solveSubstitute(L, U, PQ, b)
5      PQb = PQ*b;
6      [~,n] = size(L);
7      x = zeros(n,1);
8      for i=1:n
9          x(i) = PQb(i) / L(i,i);
10         for j=i+1:n
11             PQb(j) = PQb(j) - x(i) * L(j,i);
12         end
13     end
14     for i=n:-1:1
15         x(i) = x(i) / U(i,i);
```

```
        for j=(i-1):-1:1
            x(j) = x(j) - x(i)*U(j,i);
        end
    end

end
```

findMinMaxRoots

```
%Function takes coefficients matrix, without coef of x^n since it is 1 and
%returns min and max root in absolute value, number of steps is fixed
function [root_min,root_max] = findMinMaxRoots(coefVector, steps)
    compMatrix = createCompanionMatrix(coefVector);
    root_max=powerMethod(compMatrix, steps);
    root_min=inversePowerMethod(compMatrix,steps);
end
```

generatePolyWithGivenRoots

```
%Function used for testing, it generates polynomial coefficients for given
%roots
%Input: rootsVector - vector of roots
%Output: coefs - coefficients vector
function [coefs] = generatePolyWithGivenRoots(rootsVector)
x= sym('x');
poly = expand(prod(x-rootsVector));
coefs=coeffs(poly);
coefs=fliplr(coefs);
coefs=coefs(1,2:end);
end
```

stepsTest

```
%Function for testing. It runs an algorithm to the point where given
%tolerance is reached. However some tests gave results that were not
```

```
%converging, hence the limit 2000 for steps, so the program does not enter
%infinite (or very very long) loop
%Inputs: roots- vector of roots, tolerance - wanted tolerance
%Output: min_root - calculated smallest root, max_root - calculated largest
%root, steps - number of steps needed for given tolerance, if >2000, then
%it hold error
function [min_root, max_root, steps] = stepsTest(roots, tolerance)

coefs = generatePolyWithGivenRoots(roots);
err = 1;
steps = 1;
[~,idx_min] = min(abs(roots));
[~,idx_max] = max(abs(roots));
root_min = roots(idx_min);
root_max = roots(idx_max);
while (err>tolerance)
    [min_root, max_root] = findMinMaxRoots(coefs,steps);
    err = max(abs([root_min-min_root, root_max-max_root]));
    steps = steps + 1;
    if (steps>2000)
        disp("Number of steps more than 2000");
        steps = err;
        return
    end
end

end
```

test

```
%Test script, tests 1-5 are with roots that are significantly different
%from each other (in absolute value). In other tests, the difference is
%small.
tolerance = 1e-10;

%TEST 1
disp('Roots: -1,2,3');
roots = [-1,2,3];
[minRoot, maxRoot, steps] = stepsTest(roots, tolerance)

%TEST 2
disp('Roots: -7,9,3,5');
roots = [-7,9,3,5];
[minRoot, maxRoot, steps] = stepsTest(roots, tolerance)

%TEST 3
disp('Roots: 1,2,3,3,4,5,6,7,8,9,10');
roots = [1,2,3,3,4,5,6,7,8,9,10];
[minRoot, maxRoot, steps] = stepsTest(roots, tolerance)

%TEST 4
disp('Roots: -100, 10000, 0.00000000000001');
roots = [-100, 10000, 0.00000000000001];
[minRoot, maxRoot, steps] = stepsTest(roots, tolerance)

%TEST 5
disp('Roots: -0.001, 123, 0.000003');
roots = [-0.001, 123, 0.000003];
[minRoot, maxRoot, steps] = stepsTest(roots, tolerance)

%TEST 6
disp('Roots: 1, 3, 5, 7, 9, 11');
roots = [1, 3, 5, 7, 9, 11];
```

```
34 [minRoot, maxRoot, steps] = stepsTest(roots, tolerance)
35
36
37 %TEST 7
38 disp('Roots: 14, 14.01');
39 roots = [ 14, 14.01];
40 [minRoot, maxRoot, steps] = stepsTest(roots, tolerance)
41
42 %TEST 8
43 disp('Roots: -1.000123, 1.000132');
44 roots = [ -1.000123, 1.000132];
45 [minRoot, maxRoot, steps] = stepsTest(roots, tolerance)
46
47
48 %TEST 9
49 disp('Roots: 1.000123, 1.000132');
50 roots = [ 1.000123, 1.000132];
51 [minRoot, maxRoot, steps] = stepsTest(roots, tolerance)
52
53 %TEST 10
54 disp('Roots: 2, -2.05');
55 roots = [ 2, -2.05];
56 [minRoot, maxRoot, steps] = stepsTest(roots, tolerance)
```

NUMERICAL EXAMPLES AND ANALYSIS

Running test from previous section gives following results:

Test:

1. $roots = (-1, 2, 3)$
   $min_{calculated} = -1$ $max_{calculated} = 3$
   $steps = 56$

2. $roots = (-7, 3, 5, 9)$
   $min_{calculated} = 3$ $max_{calculated} = 9$
   $steps = 95$

3. $roots = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$
   $min_{calculated} = 1$ $max_{calculated} = 10$
   $steps = 234$

4. $roots = (-100, 10000, 0.00000000000001)$
   $min_{calculated} = 1e-14$ $max_{calculated} = 1e+04$
   $steps = 10$

5. $roots = (-0.001, 123, 0.000003)$
   $min_{calculated} = 3e-06$ $max_{calculated} = 123$
   $steps = 6$

6. $roots = (1, 3, 5, 7, 9, 11)$
   $min_{calculated} = 1$ $max_{calculated} = 11$
   $steps = 124$

7. $roots = (14, 14.01)$
   $min_{calculated} = 13.9968$ $max_{calculated} = 14.0132$
   $steps = 2000$

8. $roots = (-1.000123, 1.000132)$
   $min_{calculated} = 3.2388$ $max_{calculated} = 2.4520$
   $steps = 2000$

9. $roots = (1.000123, 1.000132)$
   $min_{calculated} = 0.9996$ $max_{calculated} = 1.0006$
   $steps = 2000$

10. $roots = (2, -2.05)$
    $min_{calculated} = 2$ $max_{calculated} = -2.05$
    $steps = 995$

Tests 1-6 analysis

In tests 1-6, the roots of polynomial were different from each other by large (at least 1) value (in absolute). In these cases, as expected, the power method and inverse power method produced accurate values. Tolerance for testing was of magnitude $1e-10$. Even with such small tolerance, number of steps needed to calculate the root was not exceeding 250. The test script was run more then 10 times. Number of steps needed for calculation varied, but that is because of random initial values used in algorithms. Difference in steps however, was always less then 5. In test cases 4 and 5, number of steps did not exceeded 10, I think it is because the difference in absolute values of a roots are very significant in these cases. Other property I have noticed is that when the number of roots of polynomial increases, so does the number of steps (hence steps needed in polynomial with 10 roots, around 230 steps were needed).

Tests 7-10 analysis

For tests 7-10 I have chosen polynomials with roots that are very close to each other in magnitude. In tests 7-9, even 2000 steps were insufficient to reach tolerance of $1e-10$. In tests 8 and 9, the roots are the same in absolute value, but one of the roots in test 8 is opposite to the one in test 9. Test 8 produced numbers not even close to the actual result, but roots calculated in test 9 were not that much off. The only difference was that root in test 8 was negative and in test 9 positive. I do not know the source of such divergence in solutions. In test 10, roots are different by 0.05 in absolute value, and such difference was enough to calculate the roots with given precision. Number of steps however was close to 1000.