КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

Лабораторна робота №2

Виконав:
студент 4-го курсу
спеціальність Математика
Шатохін Михайло

2016

1 Постановка задачі

Необхідно розв'язати нелінійну систему рівнянь

$$\begin{cases} \phi(y_1) = \gamma_1, \\ \frac{y_{i-1}-2y_i+y_{i+1}}{h^2} - p(x_i)y_i = -f(x_i, y_i), \quad i = \overline{2, n-1}, \\ \psi(y_n) = \gamma_2, \end{cases}$$

використовуючи ітераційний метод Ньютона та метод немонотонної прогонки. Введено позначення:

$a = 0, b = 1;$

$x_i = a + (i-1)h, i = \overline{1,n}; h = \frac{b-a}{n-1}, n = 101;$

y-вектор;

$\phi(t) = t^g + cos(t+k), \gamma_1 = 1, \psi(t) = t^3 + \frac{cos(t-k)^2}{k}, \gamma_2 = d;$

$p(t) = (t-a)^g(b-t), f(t,v) = exp(-t+v) - tv;$

$k = 21$ — номер за списком студента в группі;

$g = 1$ — номер группи;

$d = 7$ — день народження студента;

2 Опис алгоритму

Враховуючи методичні вказівки до розв'язання, спочатку знаходимо розв'язок першого та останнього рівнянь, далі лінійно апроксимуємо значення в проміжних точках, що використовуємо як початкове наближення для багатовимірного методу Ньютона, який відомий також як метод лінеаризації.

Метод Ньютона в даному випадку застосовується таким чином: нехай необхідно розв'язати рівняння $F(\overline{y}) = 0$. Тоді його можна по аналогії з одновимірним випадком лінеарізувати та отримати $F(\overline{\xi}) = grad(F)(\overline{y} - \overline{\xi})$, де $F(\overline{\xi}) = 0$, що дає відповідний ітераційний метод: $F(x_n) = grad(F)(x_{n+1} - x_n)$. В даному випадку $F(y) = \frac{y_{i-1}-2y_i+y_{i+1}}{h^2} - p(x_i)y_i + f(x_i, y_i)$. Враховуючи, що розбиття $\{x_i\}$ не залежить від номеру кроку, можна виділити лінійну частину F.

3 Практична реалізація

У функції main відбувається завантаження параметрів, виклик основних функції алгоритму та демонстрація отриманого результату.

main.m

```
function [] = main()
% main - main function.
%
% Solves a system of non-linear equations

```

```
    % Multiple experiments have shown that 1e-5 is a very good offset for this operation
     differentiate = @(func, point)(Diff(func, point, 1e-5));

    % First and last equations in the systems are different , so they have their own functions
    % There's a linear (as a function of y) part that has coefficient that depends on point
    [ firstEquationFunction , lastEquationFunction , func, coefficient , pointsCount, interval , precision , maxIterations ]...
     = Init ('params.mat');
    subintervalLength = ( interval (2) - interval (1)) / (pointsCount - 1);
    points = interval (1) : subintervalLength : interval (2);
    solution = zeros(pointsCount, 1);

    % Solving first and last equation separately, as they are different
    solution (1) = Newton( firstEquationFunction , interval );
    solution (pointsCount) = Newton(lastEquationFunction, interval );

    % Making starting approximation for other points
    solution (2 : end - 1) = solution (1) + ( solution (pointsCount) - solution (1)) .* ( subintervalLength : subintervalLength :1 - subintervalLength ) ';
    linearPart = -diag( coefficient ( points ')(2 : end - 1) + 2 / subintervalLength ^ 2) +...
     (diag(ones(1, pointsCount - 3), 1) + diag(ones(1, pointsCount - 3), -1)) / subintervalLength ^ 2;
    constants = zeros(pointsCount - 2, 1);
    constants (1) = solution (1) / subintervalLength ^ 2;
    constants (pointsCount - 2) = solution (pointsCount) / subintervalLength ^ 2;
    mainFunc = @(point)(func( points (2 : end - 1)', point) + constants );
    [ solution (2 : end - 1), iterationsElapsed ] = Newton(mainFunc, zeros(pointsCount - 2, 2), 1e-5, solution (2 : end - 1), linearPart , maxIterations );
    clf ;
    disp( sprintf ( ' Solution found in %d steps', iterationsElapsed ))
    showMe(solution, 1/pointsCount);
end;

function [ firstEquationFunction , lastEquationFunction , func, coefficient , pointNumber, interval , precision , maxIterations ] = Init (fileName)
% Initialises parameters

    load(fileName);
    disp( sprintf (' Initiating with parameters: group number = %d, student number = %d, birth date = %d', groupNumber, studentNumber, gamma2))
     firstEquationFunction = @(t)(phi(t, groupNumber, studentNumber) - gamma1);
     lastEquationFunction = @(t)(xi(t, studentNumber) - gamma2);
    func = f;
     coefficient = @(t)(p(t, groupNumber, intervalStart , groupNumber));
     interval = [ intervalStart , groupNumber];
    pointNumber = n;
end;

function [] = showMe(func, scale = 1)
```

```
    if ~strcmp(class(func), 'function_handle')
        y = func;
        x = ((1 : length(func)) - 1) * scale;
    else
        x = 0:scale:1;
        y = arrayfun(func, x);
    end;
    plot(x, y);
    grid on;
    axis square;
    print -deps result.eps;
end;
```

Функція Diff — допоміжна, створена для пошуку градієнту векторної функції.

Diff.m

```
function res = Diff(func, point, sideOffset)
% Differentiate vector function func

    if isrow(point)
        point = point';
    end;
    rows = length(point);
    res = zeros(length(func(point)), rows);
    for i = 1 : rows
        up = down = point;
        up(i) += sideOffset;
        down(i) -= sideOffset;
        res(:, i) = (func(up) - func(down)) / (2 * sideOffset);
    end;
end;
```

У функції Solve розв'язується тридіагональна система лінійних рівнянь методом немонотонної прогонки. Для забезпечення універсальності вона також може розв'язувати інші системи, проте це викликає відповідне попередження.

Solve.m

```
function solution = Solve(matrix)
% Solves linear system

    [rows, cols] = size(matrix);
    if (isTridiagonal(matrix))
        matrix = [[0; diag(matrix, -1)], diag(matrix), [diag(matrix, 1)], matrix(:, rows
            + 1 : end)];
        coefficients = zeros(rows - 1, 2);
        columns = zeros(1, rows - 1);
        currentColumn = 1;
        dependencies = zeros(1, rows - 1);
        for i = 1 : rows - 1
            if abs(matrix(i, 2)) >= abs(matrix(i, 3))
                coefficients(i, :) = [-matrix(i, 3), matrix(i, 4)] / matrix(i, 2);
                matrix(i + 1, [2, 4]) += [coefficients(i, 1), -coefficients(i, 2)] *
                    matrix(i + 1, 1);
```

```
                    dependencies(currentColumn) = i + 1;
                    columns(i) = currentColumn;
                    currentColumn = i + 1;
                else
                    coefficients (i, :) = [−matrix(i, 2), matrix(i, 4)] / matrix(i, 3);
                    matrix(i + 1, [2, 4]) = matrix(i + 1, [1, 4]) + [ coefficients (i, 1),
                        − coefficients (i, 2)] * matrix(i + 1, 2);
                    if i < rows − 1
                        matrix(i + 2, [1, 4]) = [0, matrix(i + 2, 4)] + [ coefficients (i, 1),
                            − coefficients (i, 2)] * matrix(i + 2, 1);
                    end;
                    dependencies(i + 1) = currentColumn;
                    columns(i) = i + 1;
                end;
            end;
            solution = zeros(rows, 1);
            solution (currentColumn) = matrix(rows, 4) / matrix(rows, 2);
            for i = rows − 1 : −1 : 1
                solution (columns(i)) = coefficients (i, 1) *
                    solution (dependencies(columns(i))) + coefficients (i, 2);
            end;
        else
            warning('Input␣matrix␣was␣not␣tridiagonal');
            rightSide = matrix (:, rows + 1 : end);
            matrix = matrix (:, 1:rows);
            solution = matrix \ rightSide;
        end;
end;

function result = isTridiagonal (matrix)
    [rows, cols] = size(matrix);
    core = matrix (:, 1 : rows);
    mask = 1 − (diag(ones(1, rows)) + diag(ones(1, rows − 1), 1) + diag(ones(1, rows −
        1), −1));
    if core .* mask == 0
        result = true;
    else
        result = false;
    end;
end;
```

У функції Newton реалізовані одновимірний метод Ньютона та метод лінеаризації. У випадку, якщо з’являється підозра на існування локального мінімуму (тобто, якщо похідна рівна нулю або градієнт має вироджений), то функція повертає попередження.

Newton.m

```
function [ solution, iterationsElapsed ] = Newton(func, interval, precision = 1e−5,
    approximation = 'default', linearPart = [], maxIterationSteps = 1000)
% Newton − solves non−linear equation F(x) = 0 using Newton's method
%
% Function may have explicit linear part
```

```
[funcInputLength, cols] = size(interval);
if cols ~= 2
    error('invalid interval format. expected nx2 matrix, got %dx%d',
        funcInputLength, cols);
end;
if strcmp(approximation, 'default')
    approximation = interval(:, 1) + rand(funcInputLength, 1) .* (interval(:, 2) -
        interval(:, 1));
end;
if length(linearPart) == 0
    differentiate = @(point)(Diff(func, point, 1e-5));
    fullValue = func;
else
    differentiate = @(point)(Diff(func, point, 1e-5) + linearPart);
    fullValue = @(point)(func(point) + linearPart * point);
end;
if funcInputLength == 1
    for iterationsElapsed = 1 : maxIterationSteps
        if (differentiate(approximation) == 0)
            warning('Newton method found local minimum.');
            return
        end;
        change = fullValue(approximation) / differentiate(approximation);
        solution = approximation - change;
        if abs(change) < precision
            return
        end;
        approximation = solution;
    end;
else
    for iterationsElapsed = 1 : maxIterationSteps
        change = Solve([differentiate(approximation), fullValue(approximation)]);
        solution = approximation - change;
        if (norm(fullValue(solution)) > norm(fullValue(approximation)))
            [eigenVectors, eigenValues] = eig(differentiate(approximation));
            [minimal, minimalIndex] = min(abs(diag(eigenValues)));
            minimalEigenVector = eigenVectors(:, minimalIndex);
            solution += sum(change .* minimalEigenVector) * minimalEigenVector;
            change -= sum(change .* minimalEigenVector) * minimalEigenVector;
            solution += Newton(@(t)(Diff(@(u)(norm(fullValue(solution + u *
                minimalEigenVector))), t, 1e-5)), [-1, 1]) * minimalEigenVector;
            if max(abs(change)) < precision
                warning('Newton method may have found local minimum.');
                return
            end;
        elseif max(abs(change)) < precision
            return
        end;

        approximation = solution;
    end;
```

```
    end;
end;
```

Параметри задачі задані у файлі params.mat. Тут представлені у вигляді, що зрозумілий людині.

params.mat

```
# name: precision
# type: scalar
 1e-5
# name: maxIterations
# type: scalar
 100
# name: n
# type: scalar
 101
# name: intervalStart
# type: scalar
 0
# name: groupNumber
# type: scalar
 1
# name: studentNumber
# type: scalar
 21
# name: gamma1
# type: scalar
 1
# name: gamma2
# type: scalar
 7
# name: phi
# type: function handle
 @<anonymous>
 @(t, g, k)(t ^ g + cos(t + k));
# name: xi
# type: function handle
 @<anonymous>
 @(t, k)(t ^ 3 + cos((t - k) ^ 2) / k);
# name: p
# type: function handle
 @<anonymous>
 @(t, g, a, b)((t - a) .^ g .* (b - t));
# name: f
# type: function handle
 @<anonymous>
 @(t, v)(exp(-t + v) - t .* v);
```

В результаті виконання програми отримується такий результат:

output.txt

```
>>main
 Initiating  with parameters: group number = 1, student number = 21, birth date = 7
Solution found in 5 steps
>>
```

Та графік розв’язку: