

CMDIE WS2002/03

8. Mai 2003

**Finding an Eigenvector and Eigenvalue, with  
Newtons Method for Solving Systems of Nonlinear  
Equations**

**Report**

**Harald Röck**

hroeck@cosy.sbg.ac.at

Academic Supervisor Prof. Roman Trobec

Department of Scientific Computing  
University of Salzburg

# 1 Introduction

Let  $A$  be an  $n \times n$  Matrix, we wish to find a nonzero vector  $x$  and a scalar  $\lambda$  such that

$$Ax = \lambda x$$

Such a scalar  $\lambda$  is called Eigenvalue, and  $x$  is a corresponding eigenvector.

Newton's method for solving systems of nonlinear equations is a fixed point iteration algorithm for finding roots of a function. A root of a function  $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$  is a vector  $x \in \mathbb{R}^m$  such that  $f(x) = O$ . Such a root is not unique, for Newton's method it depends on the initial seed which root will be found.

Newton's method is based on the truncated Taylor series of a differentiable function  $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$  :

$$f(x+h) \approx f(x) + J_f(x) \cdot h$$

where  $J_f(x)$  is the Jacobian matrix of  $f$ ,  $\{J_f(x)\}_{ij} = \frac{\partial f_i(x)}{\partial x_j}$ .

If  $J_f(x) \cdot h = -f(x)$ , then  $f(x+h)$  is taken as an approximate zero, and  $x+h$  is an approximation of a root of this function. That follows we have replaced the system of nonlinear equation with a system of linear equations which have to be solved at each iteration step.

## 1.1 Algorithm

**Algorithm [Newton's Method]:**

$x_0 =$  initial guess

**for**  $k = 0, 1, 2, \dots$

    calculate  $J_f(x_k)$  and  $f(x_k)$

    solve  $J_f(x_k)h_k = -f(x_k)$  for  $h_k$

$x_{k+1} = x_k + h_k$

**end**

## 2 Solution

How can we use this algorithm to calculate an eigenvalue  $\lambda$  and a corresponding eigenvector  $x$  of a  $n \times n$  Matrix  $A$ ?

$$A := \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} \quad x := \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

### 2.1 A New Function

We define the function  $f : \mathbb{R}^{n+1} \rightarrow \mathbb{R}^{n+1}$

$$f(x, \lambda) = \begin{pmatrix} Ax - \lambda x \\ x^T x - 1 \end{pmatrix}; \quad x \in \mathbb{R}^n \text{ and } \lambda \in \mathbb{R}$$

We see  $f(x, \lambda) = 0$  if and only if  $Ax = \lambda x$  and  $x^T x = 1$ . This is only satisfied if  $x$  is a normalized eigenvector and  $\lambda$  its corresponding eigenvalue.

To use Newton's Method we need the Jacobian Matrix  $J_f$  of  $f$ . We devise the Jacobian Matrix as follows: If we write  $f$  as an vector we get

$$f(x, \lambda) = \begin{pmatrix} a_{1,1}x_1 + a_{2,1}x_2 + \cdots + a_{1,n}x_n - \lambda x_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n - \lambda x_2 \\ \vdots \\ a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n - \lambda x_n \\ x_1^2 + x_2^2 + \cdots + x_n^2 - 1 \end{pmatrix} =: \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \\ f_{n+1} \end{pmatrix}$$

The Jacobian matrix is defined as

$$J_f(x, \lambda) := \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial \lambda} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\partial f_{n+1}}{\partial x_1} & \frac{\partial f_{n+1}}{\partial x_2} & \cdots & \frac{\partial f_{n+1}}{\partial \lambda} \end{pmatrix} = \begin{pmatrix} a_{1,1} - \lambda & a_{1,2} & \cdots & -x_1 \\ a_{2,1} & a_{2,2} - \lambda & \cdots & -x_2 \\ \vdots & \vdots & \ddots & \vdots \\ 2x_1 & 2x_2 & \cdots & 0 \end{pmatrix}$$

$$\implies J_f(x, \lambda) = \begin{pmatrix} A - I^n \lambda & -x \\ 2x^T & 0 \end{pmatrix}$$

## 2.2 Iteration

Now we should have all what we need for the Newton iteration. In the  $k$ 'th iteration step we have to solve the system of linear equations for  $[h_k \ \delta_k]^T$ :

$$\begin{pmatrix} A - I^n \lambda & -x_k \\ 2x_k^T & 0 \end{pmatrix} \begin{pmatrix} h_k \\ \delta_k \end{pmatrix} = - \begin{pmatrix} Ax_k - \lambda_k x_k \\ x_k^T x_k - 1 \end{pmatrix}$$

and update the current approximation with:

$$\begin{pmatrix} x_{k+1} \\ \lambda_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ \lambda_k \end{pmatrix} + \begin{pmatrix} h_k \\ \delta_k \end{pmatrix}$$

At this point we have to talk about a stopping criteria and the initial guess for the iteration.

For my implementation I used only a simple check of the size of the updating vector  $[h_k \ \delta_k]^T$  as stopping criteria. The algorithm will stop if  $\|[h_k \ \delta_k]^T\|_{\text{inf}} < \varepsilon$ , and  $\varepsilon$  can be defined by the user, if not it will be  $\varepsilon = 0.001$ . It will also stop if a maxima of iteration steps are reached. The standard value for this parameter is 100, but it can be changed by the user.

To start our iteration we need an initial guess for  $x_0$  and  $\lambda_0$ . For  $x_0$  we use a random vector and normalize it, and for  $\lambda$  we use  $\lambda = x_0^T A x_0$

## 3 Results

Followed you find some testing results

- $A = \begin{pmatrix} 2.9766 & 0.3945 & 0.4198 & 1.1159 \\ 0.3945 & 2.7328 & -0.3097 & 0.1129 \\ 0.4198 & -0.3097 & 2.5675 & 0.6079 \\ 1.1159 & 0.1129 & 0.6079 & 1.7231 \end{pmatrix} \longrightarrow \lambda = 4, x = \begin{pmatrix} 0.7606 \\ 0.1850 \\ 0.3890 \\ 0.4858 \end{pmatrix}$
- $A = \begin{pmatrix} 4 & -0.5 & 0 \\ 0.6 & 5 & -0.6 \\ 0 & 0.5 & 3 \end{pmatrix} \longrightarrow \lambda = 3.1357, x = \begin{pmatrix} 0.1498 \\ 0.2589 \\ 0.9542 \end{pmatrix}$

I also tested the program with some random matrices of size  $100 \times 100$  and  $1000 \times 1000$  and compared the results with the build in function of Matlab(`eig`). The result of the function `my_eig` is most times the biggest real eigenvalue.

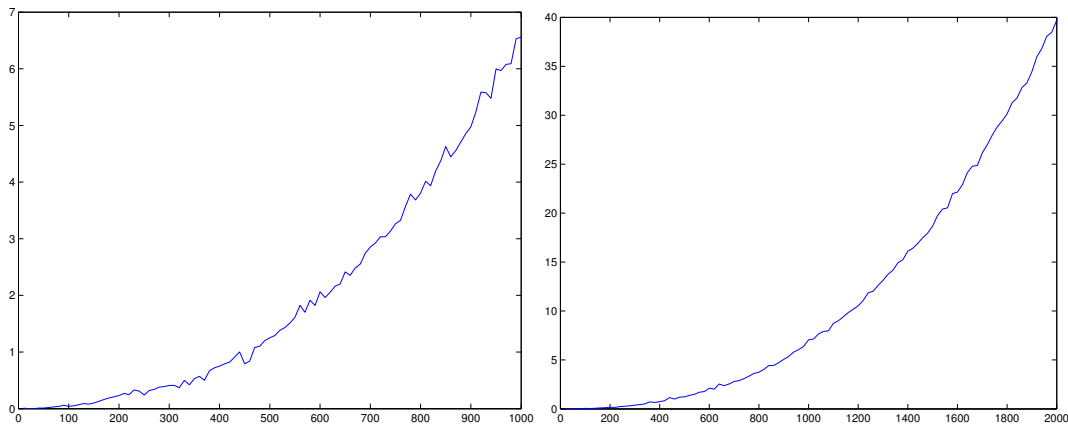


Abbildung 1: The graphs are showing the time used to calculate eigenvalues/eigenvectors for increasing matrix dimensions

## 4 Complexity

The complexity of this algorithm depends on the used algorithm for the solution of the linear equations. The Matlab "profiler" found the same part of the code. For small matrices the profiler detected the calculation of the different matrices as most expensive, but for larger systems its only the part where the linear equations are solved.

I did some timing measurements and compared the results with the build-in function of Matlab. The build-in function `eig` calculates all real and complex eigenvalues of a matrix. This function is surely implemented very efficient and uses an algorithm wich depends on the input matrix. That means it uses implicit a different algorithm for different matrices. Matlab internaly uses Lapack routines to compute eigenvalues and eigenvectors.

The method `my_eig` is only implemented in Matlab script language and not in binary form as the algorithm used by Matlab. It calculates only one eigenvector and eigenvalue, and this is always a real eigenvalue. If the matrix has no real eigenvalue the output isn't correct. i.e. the maximal count of iterations will be reached and it outputs a warning that the result may not be correct.

Therefore the function `eig` is faster although it calculates all eigenvectors and eigenvalues. But for big matrices, i.e. matrices with more elements then  $100 \times 100$ , my implementation will be faster most times.

At figure 1 you can find the results of my timing measurements. The y-axe is the time in seconds and on the x-axe you find the matrix size. The biggest matrix was of size  $2000 \times 2000$ . I tested some bigger matrices but matlab runs out of memory

after a few minutes.

## 5 Comments

The method for calculating one eigenvector/eigenvalue pair described above is just one way to find such a pair. This method finds also only *one* pair, and this only of the real space. But sometimes eigenvalues/eigenvectors are of the complex number space. It's not easy to find out whether a matrix has a real eigenvalue/eigenvector or not. If not this method is not useable. We know for special forms of matrices their eigenvalues, but not for general matrices. Note that a matrix with real numbers can have only complex eigenvalues/eigenvectors.

For some reasons it is possible that we have to know more or all eigenvalues of a matrix. For this problem we can use this method in combination with "deflation"(comp. SC M.Heath). But this approach is less than ideal. The better way would be a simultaneous iteration, as we know from our lecture.

The described method in this paper is also an example how numerical analysis works. The basic general strategy for finding a solution to a problem, is to replace the problem by an already known one with a similar solution. In this special case we used an algorithm for solving nonlinear equations to find an eigenvalue. But this algorithm known as Newton's Method is just an iteration of an algorithm to solving linear equations. To solve the linear equation we also use an approximation. We see that we are using two steps of approximation:

**Eigenvalue Problem**  $\longrightarrow$  **Nonlinear Equations**  $\longrightarrow$  **Linear Equations**

Although we are using more steps the solution can be very accurate.

This strategy is the usual process how to find a new algorithm.

## 6 Code

The implementation in Matlab can be found in the file `my_eig.m`

```
function [val, vec]= my_eig(A, eps, max, debug)
% MY_EIG eigenvalue/vector calculation
% my_eig calculates an eigenvector and the
% corresponding eigenvalue with Newton's Method
% for solving systems of nonlinear equations.
% Compare with 'Computer Problem 5.29' in
% 'Scientific Computing' of Michael T. Heath
%
```

```
% MY_EIG(A) provides the eigenvalue of the square Matrix A
%
% [val,vec] = MY_EIG(A) saves the eigenvalue in 'val' and
%           the eigenvector in 'vec'
%
% MY_EIG(A,eps,max) can be used to change the
%   accuracy of the solution.
%   eps: minimal correction of a Newton step
%   max: maximal step count
%   Standardvalues: eps=0.001 (i.e. 3 correct digits)
%                   max=100

%check how many parameters are used
if nargin < 4
    debug = 0;
end
if nargin < 3
    max = 100;
end
if nargin < 2
    eps = 0.0001;
end

% 'A' must be square
t=size(A);
if t(1) ~= t(2)
    if debug
        disp(['Matrix size: ']);
        disp(size(A));
    end
    error('input Matrix must be square')
end

n = length(A) + 1;

% x will be the eigenvector
x = rand(n-1,1); % start with a random vector
x = x/(norm(x)); % normalize

if debug
    disp(['Startvector: ']);
    disp(x);
end
```

```
% lambda will be the eigenvalue for x
lambda = x'*A*x;

% start the iteration
for j = 1:max
    if debug
        disp(['Step: ',int2str(j)])
    end

    % create the Jacobian Matrix M
    M = [ [A -x] ; 2*x' 0 ];
    for i = 1:n-1
        M(i,i) = M(i,i) - lambda;
    end

    % calculate the function vector
    B = [ A*x-lambda*x ; x'*x-1 ];

    % Newton step
    s = M \ -B;
    x = x+s(1:n-1);
    lambda = lambda+s(n);

    % stop if the maximal difference between two steps
    % is smaller then 'eps'
    if norm(s,inf) < eps
        break;
    end
end

if j == max
    warning(['maximal count was reached solution may' ...
        ' not be correct. ' , 'Please repeat with a higher value of "max"'])
end

% return values
val = lambda;
if nargin == 2 % check number of outputs
    vec = x;
end
return
```