Numerical Mathematics I, 2016/2017, Lab session 5

Keywords: rootfinding, nonlinear equations, fixed point iteration, Newton method

Remarks

- Make a new folder called `NM1_LAB_5` for this lab session, save all your functions in this folder.
- Whenever a new `MATLAB` function is introduced, try figuring out yourself what this function does by typing `help <function>` in the command window.
- Make sure that you have done the preparation before starting the lab session. The answers should be worked out either by pen and paper (readable) or with any text processing software (LaTeX, Word, etc.).

1 Preparation

1.1 Fixed point iteration

1. Study (Textbook, Section 2.6 & 2.7).
2. For a twice continuously differentiable function $\phi : [a, b] \to [a, b]$ we approximate its fixed point α with the iteration process

$$x^{(k+1)} = \phi(x^{(k)}), \quad k = 0, 1, 2, ... \tag{1}$$

   Prove the following two recurrence relations (2) and (3) for the error $e^{(k)} = x^{(k)} - \alpha$:

$$e^{(k+1)} = \phi'(\xi^{(k)})e^{(k)}, \tag{2}$$

$$e^{(k+1)} = \phi'(\alpha)e^{(k)} + \frac{\phi''(\eta^{(k)})}{2}(e^{(k)})^2, \tag{3}$$

   where $\xi^{(k)}$ and $\eta^{(k)}$ lie in between α and $x^{(k)}$.
3. Consider solving the scalar root-finding problem $f(\alpha) = 0$ using the fixed point iteration

$$x^{(k+1)} = x^{(k)} + cf(x^{(k)}). \tag{4}$$

   If the root α is simple (i.e., $f'(\alpha) \neq 0$), show that the "best" convergence is obtained if the constant c is chosen equal to $-1/f'(\alpha)$.

   Since $f'(\alpha)$ is generally unknown in advance, we make the alternative choice $c = -1/f'(x^{(0)})$. Determine the corresponding order of convergence and asymptotic convergence factor.
4. Consider the Aitken extrapolation formula (Textbook, Equation 2.31). Suppose we are given some fixed point iteration function ϕ, we define $\phi_\Delta(x)$ as

$$\phi_\Delta(x) = \frac{x\phi(\phi(x)) - [\phi(x)]^2}{\phi(\phi(x)) - 2\phi(x) + x}.$$

This is the first extrapolation of ϕ. The convergence order of $\phi_\Delta(x)$ is improved as compared to that of ϕ, as follows from (Textbook, Theorem 2.2). However, $\phi_\Delta(x)$ by itself is again an iteration function. Hence we can recursively define

$$\phi_\Delta^{(i+1)}(x) = \frac{x\phi_\Delta^{(i)}(\phi_\Delta^{(i)}(x)) - [\phi_\Delta^{(i)}(x)]^2}{\phi_\Delta^{(i)}(\phi_\Delta^{(i)}(x)) - 2\phi_\Delta^{(i)}(x) + x}, \tag{5}$$

with $\phi_\Delta^{(0)}(x) := \phi(x)$, and hence $\phi_\Delta^{(1)}(x) = \phi_\Delta(x)$. What are, according to the textbook, the orders of convergence of $\phi_\Delta^{(1)}$, $\phi_\Delta^{(2)}$ and $\phi_\Delta^{(3)}$ if ϕ converges linearly to a simple root of f? What if ϕ converges quadratically to a simple root of f?

1.2 The Newton method for systems

1. Study (Textbook, Section 2.3 & 2.5).

2. The scalar equation $x^2 = b$ has two solutions $x \in \mathbb{C}$ if $b \in \mathbb{R}\backslash\{0\}$. A similar result holds for matrices. A square root of a matrix $\mathbf{B} \in \mathbb{R}^{n\times n}$ is defined as a matrix $\mathbf{X} \in \mathbb{C}^{n\times n}$ such that

$$\mathbf{X}^2 = \mathbf{B}.$$

   Show that if $\mathbf{B}$ has n distinct nonzero eigenvalues, then $\mathbf{B}$ has at least 2^n square roots. Hint: If a matrix has distinct eigenvalues, it is always diagonalisable.

3. *Optional.* Consider applying Newton's method to a scalar rootfinding problem $f(\alpha) = 0$, where the root α is simple (i.e., $f'(\alpha) \neq 0$). Suppose we replace $f'(x)$ in Newton's iteration function ϕ by a finite difference approximation $\delta f(x)$ of a given order p,

$$\delta f(x) = f'(x) + C(x)h^p. \tag{6}$$

   Show that the resulting method has order of convergence $p = 1$ with asymptotic convergence factor given by

$$\phi'(\alpha) = \frac{C(\alpha)h^p}{f'(\alpha) + C(\alpha)h^p}.$$

2 Lab experiments

2.1 Fixed point iteration

Introduction

In this exercise you will investigate the convergence properties of several fixed point methods. As indicated in the preparation, the idea of Aitken extrapolation leads to a family of iterative methods defined by its "parent" iteration function $\phi = \phi_\Delta^{(0)}$.

Experiment

Write an iteration function `staticIteration` for the static iteration (1) with the iteration function (4). The header of your function should look like

```
% INPUT
% f          function of rootfinding problem
% c          static parameter: xnew = x + c*f(x)
% x0         initial guess
% tol        desired tolerance
% maxIt      maximum number of iterations
% OUTPUT
% root       root of f
% flag       if 0: attained desired tolerance
%            if 1: reached maxIt nr of iterations
% convHist   convergence history
function [root, flag, convHist] = staticIteration(f, c, x0, tol, maxIt)
```

Test the function `staticIteration` on the following rootfinding problem

$$\text{Find } \alpha \text{ such that } f(\alpha) = 0, \text{ where } f(x) = \log(xe^x).$$

Use as initial guess $x^{(0)} = 3$, and use `tol = 1E-12`, `maxIt = 25`. Let `c` be such that $\phi'(x^{(0)}) = 0$.

Experiment with Aitken extrapolation

Repeat the previous experiment, but now use the recursive Aitken extrapolation as described in (5). Write a function called `aitkenIteration`, which should look the same as the `staticIteration`, except that it takes an extra argument called `depth`, which specifies the value of i in (5). For $i = 0$, your function should do exactly the same as the function you wrote in the previous exercise. Note that recursive function calls should perform only a single iteration, and should therefore be of the form

```
aitkenIteration(f, c, ?, tol, 1, depth-1).
```

Compare the convergence histories for the first three extrapolations $\phi_\Delta^{(1)}$, $\phi_\Delta^{(2)}$ and $\phi_\Delta^{(3)}$ to the convergence history of the `staticIteration` of the previous exercise (hence in total you will get at least 4 convergence histories). Plot the logarithm of the error estimate against the iteration number. Whenever possible, calculate the convergence order.

2.2 The Newton method for systems

Introduction

In preparation question 1.2.2 we saw that any real $n \times n$ matrix $\mathbf{B}$ with distinct nonzero eigenvalues has 2^n square roots. In this lab experiment we will address the question how to compute the square root(s) of more general matrices $\mathbf{A} \in \mathbb{R}^{n\times n}$. We start with the remark that although not every matrix is diagonalisable, it can always be factorised in the following form*,

$$\mathbf{A} = \mathbf{PBP}^{-1},$$

where $\mathbf{B}$ is an upper triangular matrix with only two diagonals. The main diagonal contains the eigenvalues of $\mathbf{A}$, and the first upper diagonal only contains zeros and ones. More precisely, the

*The matrix $\mathbf{B}$ is called the Jordan normal form of $\mathbf{A}$.

matrix $\mathbf{B}$ is a block-diagonal matrix where each diagonal block $\mathbf{B}_i$ has the following form,

$$\mathbf{B}_i = \begin{pmatrix} \lambda_i & 1 & & \\ & \ddots & \ddots & \\ & & \ddots & 1 \\ & & & \lambda_i \end{pmatrix}.$$

It can be shown that each block $\mathbf{B}_i$ has an eigenspace of dimension one, so it cannot be diagonalised (unless it is a 1×1 matrix). Once we have computed $\mathbf{P}$ and $\mathbf{B}$, the computation of the square root(s) of $\mathbf{A}$ is reduced to the computation of the square root(s) of the diagonal blocks $\mathbf{B}_i$. One can show that if $\lambda_i \neq 0$ then the matrix $\mathbf{B}_i$ has exactly two square roots $\pm\mathbf{X}$. (If $\lambda_i = 0$ then the matrix $\mathbf{B}_i$ has no square root unless it is the 1×1 zero matrix.)

In the following experiment you will use Newton's method for computing the square root of such a matrix $\mathbf{B}_i$, which we will denote by $\mathbf{B}(\lambda_i)$ (for $\lambda_i \neq 0$). One way of approaching this problem is to compute the root of the function $\mathbf{F} : \mathbb{R}^{n\times n} \to \mathbb{R}^{n\times n}$ defined by

$$\mathbf{F}(\mathbf{X}) = \mathbf{X}^2 - \mathbf{B}(\lambda). \tag{7}$$

However, the way Newton's method is usually formulated is for functions of the form $\mathbf{F} : \mathbb{R}^m \to \mathbb{R}^m$. Hence we rewrite (7) by introducing the following notation (the vector representation of a matrix)

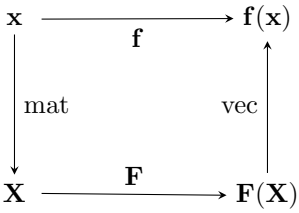
$$\mathrm{vec}(\mathbf{M}) := \begin{pmatrix} \mathbf{m}_1 \\ \vdots \\ \mathbf{m}_n \end{pmatrix} \in \mathbb{R}^{n^2},$$

where $\mathbf{m}_i$ are the columns of $\mathbf{M}$. The inverse operation (which maps the vector representation of $\mathbf{M}$ back to the original matrix form) is denoted by "mat". In `MATLAB` you can transform $\mathbf{X}$ to $\mathrm{vec}(\mathbf{X})$ by using `vecX = reshape(X, n^2, 1)`, and reverse this transformation by `X = reshape(vecX, n, n)`.

We also define the function $\mathbf{f} := \mathrm{vec}(\mathbf{F}) : \mathbb{R}^{n^2} \to \mathbb{R}^{n^2}$ as

$$\mathbf{f}(\mathbf{x}) := \mathrm{vec}[\mathbf{F}(\mathrm{mat}(\mathbf{x}))].$$

The relations described above are summarised in the following commutative diagram



The Jacobian matrix $\mathbf{J}$ of $\mathbf{f}$ is defined as

$$J_{ij}(\mathbf{x}) := \frac{\partial f_i}{\partial \mathbf{x}_j}(\mathbf{x}) \quad (i, j = 1, \ldots, n^2).$$

Experiment

Consider the matrix $\mathbf{B}(\lambda)$ given by

$$\mathbf{B}(\lambda) = \begin{pmatrix} \lambda & 1 & 0 \\ 0 & \lambda & 1 \\ 0 & 0 & \lambda \end{pmatrix}, \tag{8}$$

and let $\lambda = 2$.

Write a `MATLAB` function `matFunc` that evaluates $\mathbf{f} = \text{vec}(\mathbf{F})$ given the input $\mathbf{x} = \text{vec}(\mathbf{X})$. The header of your function should look like

```
% INPUT
% vecX      vector (n^2 x 1) representing a matrix X of size
%           n x n
% B         matrix B of size n x n
% OUTPUT
% vecF      vector (n^2 x 1) representing the matrix
%           F = X^2 - B
function vecF = matFunc(vecX, B)
```

You are given the function `matJac` that, depending on an input flag, either returns the exact Jacobian matrix $\mathbf{J}$ or its finite difference approximation $\mathbf{J}^h$.

Test the consistency of the finite difference approximation by computing

$$\max_{i,j} \left| \mathbf{J}^h_{ij}(\mathbf{x}) - \mathbf{J}_{ij}(\mathbf{x}) \right|$$

for $h = 10^{-l}$, $l = 0, \ldots, 14$. Note that this is very similar to the consistency test you performed in lab session 2. Find the optimal value h_{opt} for which the error is smallest. Use a random vector of length 3^2 for $\mathbf{x}$.

Now write a `MATLAB` implementation of the Newton method for systems (which should be very similar to the function you wrote in the previous exercise). The header of your function should look like

```
% INPUT
% f         function of rootfinding problem
% df        function returning the Jacobian matrix
% x0        initial guess
% tol       desired tolerance
% maxIt     maximum number of iterations
% OUTPUT
% root      if successful, root is an approximation to the
%           root of the nonlinear equation
% flag      flag = 0: tolerance attained, flag = 1: reached maxIt
% iter      the number of iterations
% convHist  convergence history
function [root, flag, iter, convHist] = newton(f, df, x0, tol, maxIt)
```

Compute a square root of the matrix defined in (8). The input for `newton` should be of the following form.

```
dfFunc = @(x) matJac(x, 'exact', [], B);
vecRoot = newton(@(x) matFunc(x, B), dfFunc, x0, tol, maxIt);
```

Use as initial guess the 3×3 identity matrix. Compare the convergence histories for using the exact Jacobian matrix and the approximation using $h = h_{\text{opt}}$. Also try $h = 10^{-1}$. Use a Newton tolerance of 10^{-10} and a maximum number of iterations of 25.

Optional

Computing (an approximation to) the Jacobian matrix is very expensive and hence we would like to avoid this. If we solve the linear system during the Newton iteration using an iterative

method (like the one considered in lab session 4) we only need the matrix-vector product $\mathbf{J}(\mathbf{x})\mathbf{v}$. Therefore, we can replace this matrix-vector product by a finite difference approximation to the directional derivative $\nabla_{\mathbf{v}}\mathbf{f}(\mathbf{x})$. If we do this, we obtain

$$\mathbf{J}(\mathbf{x})\mathbf{v} \approx \frac{\mathbf{f}(\mathbf{x}+h\mathbf{v}) - \mathbf{f}(\mathbf{x})}{h}.$$

Implement this approach in a function called `newtonIter`, this function should call `iterMethod` [†] at each Newton iteration. Think about what tolerance you set for the linear solve (perhaps it should decrease during Newton iteration), let the maximum number of iterations of the linear solve be given by the size of the of linear system that is being solved.

Consider the analogous problem for $\mathbf{B}(\lambda) \in \mathbb{R}^{n\times n}$, where $n = 5, 10, 50$. Compare the efficiency of this implementation to the standard implementation using `tic` and `toc`. Also try using a second-order central approximation.

3 Discussion

3.1 Fixed point iteration

1. Do the found convergence orders agree with the theory? Why is it difficult to check whether the theoretical orders of convergence actually are attained in your experiments for higher order methods?

2. Further to preparation question 1.1.3, suppose we take a different value for the constant c in every iteration step by setting $c = C(x^{(k)})$ for some function C. Show that we have (at least) quadratic convergence if and only if $C(\alpha) = -1/f'(\alpha)$. Can Newton's method be seen as a method of this type?

3. For the rootfinding problem $f(\alpha) = 0$, what are the pros and cons when comparing Newton's method to the (static) iteration (4) with the first Aitken extrapolation?

4. Suppose that two methods, say method A and method B, attain convergence within the same tolerance, but A does so in less iterations. Does this always mean that method A is more efficient? Why, or why not?

3.2 The Newton method for systems

1. Consider Newton's method with the exact derivative $f'(x)$ replaced by the finite difference approximation (6). Does this method maintain quadratic convergence if h is chosen too large? Hint: use the result of the optional question 1.2.3. Did your experiments confirm this?

2. If an iterative method (such as the one considered in lab session 4) is used for solving the linear system of equations, is it necessary to compute (an approximation of) the *full* Jacobian matrix $\mathbf{J}$ or is it sufficient to have a function that can evaluate (an approximation of) the matrix-vector product $\mathbf{Jv}$ for some vector $\mathbf{v}$? Explain.

3. *Optional.* How did you choose the tolerance for the iterative method at each step of Newton iteration? Explain.

[†] If you haven't done so already, read the optional exercise of lab session 4 to be able to use a function instead of a matrix as input for `iterMethod`.

4. *Optional.* Discuss the efficiency of `newtonIter` by comparing it to `newton`. Explain why the second-order central approximation works so well for this application.