Dirk Praetorius
Winfried Auzinger

Sommersemester 2018
11.04.2018

Übungen zur Vorlesung
Computermathematik

Serie 3

Aufgabe 3.1. A matrix $A \in \mathbb{R}^{n\times n}$ is strictly diagonally dominant if

$$\sum_{\substack{k=1\\k\neq j}}^{n} |A_{jk}| < |A_{jj}| \quad \text{für alle } j \in \{1,\dots,n\}.$$

Further, if a matrix is symmetric, strictly diagonally dominant and there holds $A_{jj} > 0$ for all $j \in \{1,\dots,n\}$, then A is positive definite. Write a function `constructSPDmatrix` which returns for a give dimension n a random symmetric and positive definite (SPD) matrix $A \in \mathbb{R}^{n\times n}$. **Hint:** Use `rand()`. Note that $A * A^T$ is symmetric for all $A \in \mathbb{R}^{n\times n}$. Use this to construct SPD-matrices.

Aufgabe 3.2. Let $b \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n\times n}$ be a symmetric and positive definite (SPD) matrix. In order to solve the linear system $Ax = b$, one can use the CG-method. Write a MATLAB function `cgsolve`, which gets A, b as well as a tolerance $\tau > 0$ and returns the solution x of the linear system. The algorithm is obtained as follows: For an arbitrary initial guess $x^{(0)} \in \mathbb{R}^n$ compute $r^{(0)} := b - Ax^{(0)}$ and $d^{(0)} := r^{(0)}$. For all $k \in \mathbb{N}$, we define the sequences

$$\alpha^{(k)} := \frac{r^{(k)} \cdot r^{(k)}}{d^{(k)} \cdot Ad^{(k)}}, \quad x^{(k+1)} := x^{(k)} + \alpha^{(k)} d^{(k)}, \quad r^{(k+1)} := r^{(k)} - \alpha^{(k)} A d^{(k)}$$

as well as

$$d^{(k+1)} := r^{(k+1)} + \frac{r^{(k+1)} \cdot r^{(k+1)}}{r^{(k)} \cdot r^{(k)}} d^{(k)}.$$

If the accurancy $|r^{(k+1)}| \le \tau$ is achieved, the CG-iteration should stop and return the current approximation $x^{(k+1)} \approx x$. Test your code with suitable examples. How does the number of CG-steps behaive for $\tau \to 0$ and big dimensions $n \in \mathbb{N}$. **Hinweis:** Use exercise 3.1.

Aufgabe 3.3. Aitken's Δ^2-method is a method for convergence acceleration of sequences. For an injective sequence (x_n) with $x = \lim_{n\to\infty} x_n$ one defines

$$y_n := x_n - \frac{(x_{n+1} - x_n)^2}{x_{n+2} - 2x_{n+1} + x_n} \tag{1}$$

Under certain assumptions for the sequence (x_n) it then holds

$$\lim_{n\to\infty} \frac{x - y_n}{x - x_n} = 0,$$

i.e., the sequence (y_n) converges faster to x than (x_n). Write a MATLAB function `aitken` which takes a vector $x \in \mathbb{R}^N$ and returns a vector $y \in \mathbb{R}^{N-2}$. Use suitable loops. Further, write an alternative MATLAB function `aitken_vec` which calculates the vector $y \in \mathbb{R}^{N-2}$ with suitable vector arithmetic instead of loops. Think about how you can test your code! What happens for a geometric sequence $x_n := q^n$ with $0 < q < 1$?

Aufgabe 3.4. Write a MATLAB function `diffaitken`, which computes the approximation of the derivative of a function f in a point x through the foward and central difference quotient

$$\Phi(h) = \frac{f(x+h) - f(x)}{h} \quad \text{resp.} \quad \Phi(h) = \frac{f(x+h) - f(x-h)}{2h}.$$

Given the function f, the point x and an initial parameter $h_0 > 0$, the function returns an approximation of the derivative obtained as follows: For $n \geq 1$, compute $h_n := 2^{-(n-1)}h_0$, $x_n := \Phi(h_n)$. Further, compute the sequence of the Aitken-extrapolation which is given by $\phi_n := x_n$ für $n = 1, 2$, and $\phi_n := y_{n-2}$ for $n \geq 3$. In this case y_n denote the sequence from exercise 3.3.
Additionally, compute the experimental rate of convergence for the foward and central difference quotient with and withoud Aitken-extrapolation. Visualize your results. What rates do you get?

Aufgabe 3.5. Consider the real nodes $x_1 < \cdots < x_n$ and function values $y_j \in \mathbb{R}$. Then, linear algebra provides a unique polynomial $p(t) = \sum_{j=1}^n a_j t^{j-1}$ of degree $n-1$, such that $p(x_j) = y_j$ for all $j = 1, \ldots, n$. Suppose a fixed evaluation point $t \in \mathbb{R}$. The *Neville-algorithm* is able to compute the point evaluation $p(t)$ without computing the vector of coefficients $a \in \mathbb{R}^n$. It consists of the following steps: First, define for $j, m \in \mathbb{N}$ with $m \geq 2$ and $j + m \leq n + 1$ the values

$$\begin{aligned} p_{j,1} &:= y_j, \\ p_{j,m} &:= \frac{(t - x_j)p_{j+1,m-1} - (t - x_{j+m-1})p_{j,m-1}}{x_{j+m-1} - x_j}. \end{aligned}$$

This implies $p(t) = p_{1,n}$. Write a MATLAB-function `neville` which computes $p(t)$ for a given evaluation point $t \in \mathbb{R}$ and vectors $x, y \in \mathbb{R}^n$. To do that, you can use the following scheme

$$\begin{array}{ccccccccccccc}
y_1 & = & p_{1,1} & \longrightarrow & p_{1,2} & \longrightarrow & p_{1,3} & \longrightarrow & \cdots & \longrightarrow & p_{1,n} & = & p(t) \\
 & & & \nearrow & & \nearrow & & & & \nearrow & & & \\
y_2 & = & p_{2,1} & \longrightarrow & p_{2,2} & & & & & & & & \\
 & & & \nearrow & & & & \nearrow & & & & & \\
y_3 & = & p_{3,1} & \longrightarrow & \vdots & & & & & & & & \\
\vdots & & \vdots & & \vdots & \nearrow & & & & & & & \\
y_{n-1} & = & p_{n-1,1} & \longrightarrow & p_{n-1,2} & & & & & & & & \\
 & & & \nearrow & & & & & & & & & \\
y_n & = & p_{n,1} & & & & & & & & & &
\end{array} \tag{2}$$

One easy way to implement this scheme is by building a matrix with entries $(p_{j,m})_{j,m=1}^n$. For testing, take an arbitrary polynomial resp. nodes, and compute $y_j = p(x_j)$.

Aufgabe 3.6. One can implement the *Neville-algorithm* from exercise 3.5 wihout using additional memory. Therfore, instead of storing the values $(p_{j,m})_{j,m=1}^n$ in a matrix, you can overwrite suitable entries in the given vector y. Write a MATLAB-function `neville2` which realizes the *Neville-algorithm* wihtout using additional memory.

Aufgabe 3.7. One efficient way to compute the foward difference quotient $\Phi(h)$ from exercise 3.4 is the *Richardson-extrapolation of the foward difference quotient.* The (theoretical!) idea is the following: Use the values $\Phi(h_0), \ldots, \Phi(h_n)$ to compute an interpolation polynimomial of degree $n-1$ with $(h_j, \Phi(h_j))$ für $j = 1, \ldots, n$. Then, there holds $p_n(h) \approx \Phi(h)$ and one can use the *Neville-algorithm* to compute the point evaluation at $h = 0$. (A proof of convergence for this scheme is given in the lecture Numerischen Mathematik.) Write a function `richardson` which computes an approximation of $f'(x)$ for a given function-handle f, evaluation point $x \in \mathbb{R}$, step-size h_0 and tolerance $\tau > 0$. First, define $h_n := 2^{-n}h_0$ and $y_n := p_n(0)$. Then, the function should return the first $y_{n+1} \approx f'(x)$ which satisfies

$$|y_n - y_{n+1}| \leq \begin{cases} \tau, & \text{falls } |y_{n+1}| \leq \tau, \\ \tau\,|y_{n+1}| & \text{else.} \end{cases}$$

Use the function `neville` from exercise 3.5.

Aufgabe 3.8. One possible algorithm for eigenvalue computations is the *Power Iteration.* It approximates (under certain assumptions) the eigenvalue $\lambda \in \mathbb{R}$ with the greatest absolute value of a symmetric matrix $A \in \mathbb{R}^{n \times n}$ as well as the corresponding eigenvector $x \in \mathbb{R}^n$. The algorithm is obtained as follows: Given a vector $x^{(0)} \in \mathbb{R}^n \backslash \{0\}$, e.g., $x^{(0)} = (1, \ldots, 1) \in \mathbb{R}^n$, define the sequences

$$x^{(k)} := \frac{Ax^{(k-1)}}{\|Ax^{(k-1)}\|_2} \quad \text{and} \quad \lambda_k := x^{(k)} \cdot Ax^{(k)} := \sum_{j=1}^n x_j^{(k)} (Ax^{(k)})_j \quad \text{for } k \in \mathbb{N},$$

where $\|y\|_2 := \left(\sum_{j=1}^n y_j^2\right)^{1/2}$ denotes the Euclidean norm. Then, under certain assumptions, (λ_k) converges towards λ, and $(x^{(k)})$ converges towards an eigenvector associated to λ (in an appropriate sense). Write a MATLAB function `poweriteration`, which, given a matrix A, a tolerance τ and an initial vector $x^{(0)}$, verifies whether the matrix A is symmetric. If this is not the case, then the function displays an error message and terminates (use `error`). Otherwise, it computes (λ_k) and $(x^{(k)})$ until

$$\|Ax^{(k)} - \lambda_k x^{(k)}\|_2 \leq \tau \quad \text{and} \quad |\lambda_{k-1} - \lambda_k| \leq \begin{cases} \tau & \text{if } |\lambda_k| \leq \tau, \\ \tau|\lambda_k| & \text{else,} \end{cases}$$

and returns λ_k and $x^{(k)}$. Realize the function in an efficient way, i.e., avoid unnecessary computations (especially of matrix-vector products) and storage of data. Then, compare `poweriteration` with the built-in MATLAB function `eig`. Use the function `norm`, as well as MATLAB arithmetic.