Dirk Praetorius
Winfried Auzinger

Sommersemester 2017
22.03.2017

**Übungen zur Vorlesung**
**Computermathematik**

**Serie 3**

**Aufgabe 3.1.** Aitken's $\Delta^2$-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 3.2.** 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 3.1.
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 3.3.** 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

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

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 3.4.** One can implement the *Neville-algorithm* from exercise 3.3 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 3.5.** One efficient way to compute the foward difference quotient $\Phi(h)$ from exercise 3.2 is the *Richardson-extrapolation of the foward difference quotient.* The (theoretical!) idea is the following: Use the values $\Phi(h_0), \dots, \Phi(h_n)$ to compute an interpolation polynimomial of degree $n-1$ with $(h_j, \Phi(h_j))$ für $j = 1, \dots, n$. 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 3.3.

**Aufgabe 3.6.** 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., $x^{(0)} = (1, \dots, 1) \in \mathbb{R}^n$, 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, $(\lambda_k)$ converges towards $\lambda$, and $(x^{(k)})$ converges towards an eigenvector associated to $\lambda$ (in an appropriate sense). Write a MATLAB function `poweriteration`, which, given a matrix $A$, a tolerance $\tau$ 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 $(\lambda_k)$ 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 $\lambda_k$ 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.

**Aufgabe 3.7.** Let $f : [a,b] \to \mathbb{R}$ be a continuous function. For $N \in \mathbb{N}$ and $x_j := a + j\,(b-a)/N$ with $j = 0, \ldots, N$, we define the *composite midpoint rule*

$$I_N := \frac{b-a}{N} \sum_{j=1}^{N} f\big((x_{j-1} + x_j)/2\big).$$

Since $I_N$ is a Riemann sum, we know that

$$\lim_{N\to\infty} I_N = \int_a^b f\,dx.$$

For $f \in C^2[a,b]$, one can even show that

$$\Big|\int_a^b f\,dx - I_N\Big| = \mathcal{O}(N^{-2}). \tag{3}$$

Write a MATLAB function

```
int = midpointrule(a,b,f,n)
```

which, for the sequence $N = 2^k$ and $k = 0, \ldots, n$ , computes and returns the vector `int` of the corresponding values $I_N$. Think about how you can test your code! What are suitable test examples? Experimentally verify the order of convergence order given in (3.7). **Hint:** Test your quadrature with polynomials of different degree. Calculate the result analytically. What do you notice?

**Aufgabe 3.8.** Modify the function `midpointrule` from exercise 3.7 in the following way.

- If `midpointrule(f,n)` is called without the interval boundaries $a, b$, then $\int_{-1}^{1} f\ dx$ is calculated.
- The call `midpointrule(f,n,a,b)` shall return as in Aufgabe 3.4 the vector $I_N \approx \int_a^b f\ dx$. Take care that in the case $b < a$ it holds $\int_a^b f\ dx = -\int_b^a f\ dx$. In this case additionally give a warning.
- In the case `midpointrule(f,n,a,b,'nodes')`, additionally to the vector `int`, the vector `nodes` of the points $x_j$ with $j = 0, \ldots 2^n$ shall be returned.