Winfried Auzinger
Gregor Gantner
Alexander Haberl
Dirk Praetorius

Sommersemester 2016
16. März 2016

Übungen zur Vorlesung Computermathematik

Serie 3

Aufgabe 3.1. 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. 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 an alternative MATLAB function `aitken_vec` which calculates the vector $y \in \mathbb{R}^{N-2}$ from Aufgabe 3.1 with suitable vector arithmetic instead of loops.

Aufgabe 3.3. Write a MATLAB function `diffaitken`, which computes the approximation of the derivative of a function f in a point x through the central difference quotient

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

Given the function f, the point x, an initial parameter $h_0 > 0$ and a tolerance $\tau > 0$, the function returns an approximation of the derivative obtained as follows: For $n \geq 1$, compute $h_n := 2^{-(n-1)} h_0$, $x_n := \Phi(h_n)$, and ϕ_n defined by

$$\phi_n := \begin{cases} x_n & \text{if } n = 1, 2, \\ y_{n-2} & \text{if } n \geq 3, \end{cases}$$

where, for $n \geq 3$, we apply the Δ^2-method from Aufgabe 3.1–3.2 (define y_{n-2} through (1)). The iteration stops when $n \geq 2$ and

$$|\phi_n - \phi_{n-1}| \leq \begin{cases} \tau & \text{if } |\phi_n| \leq \tau, \\ \tau|\phi_n| & \text{else,} \end{cases}$$

and the function returns ϕ_n as approximation of the derivative. Think about how you can test your code!

Aufgabe 3.4. 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

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

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? **Hint:** Test your quadrature with polynomials of different degree. Calculate the result analytically. What do you notice?

Aufgabe 3.5. Modify the function `midpointrule` from Aufgabe 3.4 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.

Aufgabe 3.6. Alternatively to the bisection method from the lecture one can use the *Newton-method* for the calculation of a root of a function $f : [a, b] \to \mathbb{R}$. Given an initial value x_0 one inductively defines the sequence (x_n): For given x_k let x_{k+1} be the root of the tangent on the graph of f in the point $(x_k, f(x_k))$, i.e. $x = x_{k+1}$ satisfies $0 = f(x_k) + f'(x_k)(x - x_k)$. Solving for x shows

$$x_{k+1} = x_k - f(x_k)/f'(x_k).$$

Implement the Newton-method in a function `newton(f,fprime,x0,tau)` where the iteration is stopped if

$$|f'(x_n)| \le \tau$$

or

$$|f(x_n)| \le \tau \quad \text{and} \quad |x_n - x_{n-1}| \le \begin{cases} \tau & \text{for } |x_n| \le \tau, \\ \tau |x_n| & \text{else} \end{cases}$$

In each case, return x_n as approximation of the root, where in the first case, additionally give a warning. Beside x_n, return the sequence $(x_0, \dots, x_n)$ of the approximative roots and the corresponding function values. Test your implementation with the function $f(x) = x^2 + e^x - 2$.

Aufgabe 3.7. Think about at least three non-trivial examples to test your implementation of the *Newton-method*. Write a MATLAB-function `testnewton(f,fprime,x0,tau)` to visually verify your solution. Plot the test function $f(x)$ and the approximation of the root. Take care for suitable scaling in the plot in order to be able to check your solution as good as possible! **Hint:** You can use `scatter` to plot single points.

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., $x^{(0)} = (1, \dots, 1) \in \mathbb{R}^n$, 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, (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

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

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.