Valentin Helml
Dirk Praetorius
Julian Streitberger

Summer term 2022
March 18 2022

Exercise
Scientific programming in mathematics

Series 3

Exercise 3.1. Modify the Newton method from Exercise 2.6, the bisection method from slide 82, and the secant method from slide 119 such that all use the termination criterion from Exercise 2.6 together with an optional tolerance (which is 10^{-12} by default). Besides the approximate zero, the function should also return the finite sequences of iterates (x_k) and the corresponding function iterates $(f(x_k))$.

Exercise 3.2. Consider the function $f(x) := \cos^2(2x) - x^2$ on the interval $[a, b] = [0, 3/2]$. Use the Newton method, the bisection method, and the secant method from Exercise 3.1 to calculate the zero $x^\star$. Additionally, use Aitken's Δ^2-method from Exercise 2.7 to accelerate the sequences.. Plot the experimental convergence rates and visualize the error (and the convergence rate) in a log-log plot.
Hint: For a numerical scheme which approximates $y^\star$ with $(y_k)_{k\in\mathbb{N}_0}$, define the error $\widetilde{e}_k := |y_{k+1} - y_k|$, then one can compute the experimental convergence rate $\widetilde{p}_k$ via

$$\widetilde{p}_k = \frac{\log(\widetilde{e}_{k+2}) - \log(\widetilde{e}_{k+1})}{\log(\widetilde{e}_{k+1}) - \log(\widetilde{e}_k)} \quad \text{for } k \in \mathbb{N}_0.$$

Exercise 3.3. Let $m, n, N \in \mathbb{N}$. Let $I, J, a \in \mathbb{R}^N$ represent the coordinate format of a sparse matrix $A \in \mathbb{R}^{m\times n}$, i.e., for all $k = 1, \dots, N$, it holds that $A_{ij} = a_k$ with $i = I_k$, $j = J_k$. Write a MATLAB function

```
[II,JJ,AA] = naive2ccs(I,J,a,m,n)
```

which returns the corresponding vectors of the CCS format. If an entry A_{ij} occurs multiple times, the function should add the contributions.

Exercise 3.4. Let $A \in \mathbb{R}^{m\times n}$ be a sparse matrix, represented in CCS-format by the vectors $I, a \in \mathbb{R}^N$ and $J \in \mathbb{R}^{n+1}$. Write a MATLAB function `y = mvmsparse(I,J,a,m,x)` that realizes the matrix-vector-product $y = Ax$ for vectors $x \in \mathbb{R}^n$, using suitable (as few as possible) loops. Determine the computation complexity of your code.

Exercise 3.5. The following code computes a sparse matrix $A \in \mathbb{R}^{N\times N}$, which actually stems from the lowest-order FEM discretization of the Poisson problem:

```
function A = matrix(N)

    x = rand(1,N);
    y = rand(1,N);
```

```
    triangles = delaunay(x,y);
    n = size(triangles,1);

    A = sparse(N,N);
    for i = 1:n
        nodes = triangles(i,:);
        B = [1 1 1 ; x(nodes) ; y(nodes)];
        grad = B \ [0 0 ; 1 0 ; 0 1];
        A(nodes,nodes) = A(nodes,nodes) + det(B)*grad*grad'/2;
    end
end
```

Plot the computational time $t(N)$ over N and visualize the growth $t(N) = \mathcal{O}(N^\alpha)$ for $N = 100 \cdot 2^k$ and $k = 0, 1, 2, \ldots$. Which growth do you see? What is the reason for it? What is the bottleneck of this implementation? What can be done to improve the runtime behavior? Write an improved code `matrix2` which leads to a better computational time. Visualize its runtime in the same plot to show that the improved code is really superior. Which growth do you expect and see for your improved code?
Hint: You might want to use `sparse` in an appropriate way.

Exercise 3.6. 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(f,n,a,b)` 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 $I_N \approx \int_a^b f\ dx$, where $N := 2^n$. Take care that $b < a$ leads to $\int_a^b f\ dx = -\int_b^a f\ dx$. In this case, your code should give an additional warning that $b < a$.
- The call `midpointrule(f,n,a,b,'nodes')` shall return, additionally to I_N, the vector `nodes` of the points x_j with $j = 0, \ldots N$.

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

Exercise 3.7. 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. Use the function `norm`, as well as MATLAB arithmetics. Then, compare the runtime of `poweriteration` with the built-in MATLAB function `eig` and plot the runtimes for matrices of different sizes.

Exercise 3.8. Write a function `plotPotential`, which takes a function $f : \mathbb{R}^2 \to \mathbb{R}$, a domain $[a, b]^2$ and a step size $\tau > 0$, and plots the projection of $f(x, y)$ onto the 2D plane (i.e., `view(2)`). Add a `colorbar` to the plot. For the visualization, use a tensor grid with step size τ. You may assume, that the actual implementation of f takes matrices $x, y \in \mathbb{R}^{M\times N}$ and returns a matrix $z \in \mathbb{R}^{M\times N}$ of the corresponding function values, i.e., $z_{jk} = f(x_{jk}, y_{jk})$. Optionally, the function `plotPotential` takes a parameter $n \in \mathbb{N}$. For given n, add n (black or white) contour lines to the figure. To verify your code, write a MATLAB script which visualizes the potential $f(x, y) = x \exp(-x^2 - y^2)$.