Valentin Helml
Dirk Praetorius
Julian Streitberger

Summer term 2022
8 April 2022

# Exercise
# Scientific programming in mathematics

## Series 6

**Exercise 6.1.** Explain the differences between variables and pointers. What are advantages and disadvantages of each of them? Write a function `swap` that swaps the contents of two variables $x$, $y$. What is the problem with the following code?

```
void swap(double x, double y)
{
        double tmp;
        tmp = x;
        x = y;
        y = tmp;
}
```

Save your source code as `swap.c`.

**Exercise 6.2.** Write a function `void cut(double* x, int* n, double cmin, double cmax)`, which, given a vector $x \in \mathbb{R}^n$ and thresholds $c_{\min}, c_{\max} \in \mathbb{R}$, removes from the vector all entries $x_j$ such that $x_j < c_{\min}$ or $x_j > c_{\max}$. For instance, for $c_{\min} = 0$ and $c_{\max} = 10$, the vector $x = (-4, 3, -5, 1, 7, 3, 11, -1) \in \mathbb{R}^8$ should be replaced by the vector $x = (3, 1, 7, 3) \in \mathbb{R}^4$. Work with dynamically allocated memory (the input vector must be overwritten with the shortened one, in particular the length must be adjusted accordingly). Write a main program, which provides the length $n$, the vector $x \in \mathbb{R}^n$, and the thresholds $c_{\min}, c_{\max}$, calls the function, and prints both vectors (the original one and the shortened one) to the screen. Test your implementation with suitable examples. Save your source code as `cut.c`. Determine the computational cost of your code! How did you test your code?

**Exercise 6.3.** Write a *recursive* function `void quickSort(double* x, int n)`, which sorts a given vector $x \in \mathbb{R}^n$ in ascending order using the *quicksort* algorithm. Pick an arbitrary entry from the vector $x$, called the pivot. Reorder the vector so that all elements with values less than the pivot come before the pivot, while all the values greater than the pivot come after it (equal values can go either way). After this procedure, the pivot is in its final position. Recursively apply the above steps to the subvector of elements with smaller values and separately to the subvector of elements with greater values. Work with dynamically allocated memory. Moreover, write a main program that provides the input vector $x \in \mathbb{R}^n$, calls the function, and prints both the input and the sorted vector to the screen. Save your source code as `quickSort.c`.

**Hint:** Choose $x_1$ as the pivot. Starting with $j = 2$, search for an element $x_j$ with $x_j \geq x_1$, i.e., $x_j$ belongs to the subvector $x^{(\geq)}$ of all elements greater than or equal to the pivot. Then, starting with $k = n$, search for an element $x_k$ with $x_k < x_1$, i.e., belongs to the subvector $x^{(<)}$ of all elements less than the pivot. In that case, swap $x_j$ and $x_k$. If $j$ and $k$ coincide, then $x$ has the form $x = (x_1, x^{(<)}, x^{(\geq)})$. Then the form $(x^{(<)}, x_1, x^{(\geq)})$ can be obtained immediately and the pivot is in its final position. It remains to sort $x^{(<)}$ and $x^{(\geq)})$ recursively.

**Exercise 6.4.** Write a function `void unique(double x*, int* n)`, which sorts a given vector $x \in \mathbb{R}^n$ in ascending order and eliminates all entries that appear more than once. For instance, the vector $x = (4, 3, 5, 1, 4, 3, 4) \in \mathbb{R}^7$ should be replaced by the vector $x = (1, 3, 4, 5) \in \mathbb{R}^4$. Work with dynamically allocated memory (the input vector must be overwritten with the shortened one, in particular, the length must be adjusted accordingly). To sort the vector, use a sorting algorithm of your choice, e.g., `quickSort` or `bubbleSort` from one of the previous exercises. Write a main program `unique.c` that provides the vector $x \in \mathbb{R}^n$ and its length $n$, calls the function, and prints both vectors (the input vector and the shortened one) to the screen. Provide suitable examples. Determine the computational cost of your code! How did you test your code?

**Exercise 6.5.** Write a function `int checkOccurrence(char* string, char character)`, which, given a string $s$ and a character $c$, returns how many times $c$ occurs in $s$. Both the lowercase and uppercase versions of $c$ contribute to the number of occurrences. Then, write a main program `checkOccurrence.c`, which reads $s$ and $c$ from the keyboard, calls the function, and prints its result to the screen. Test the program appropriately.

**Exercise 6.6.** Write a structure `Date` for the storage of all dates since January 1, 1900 (01.01.1900). The structure consists of three member (day, month and year) of type `int`. Write the functions

- `Date* newDate(int d, int m, int y)`,
- `Date* delDate(Date* date)`,

as well as the access functions

- `void setDateDay(Date* date, int d)`,
- `void setDateMonth(Date* date, int m)`,
- `void setDateYear(Date* date, int y)`,
- `int getDateDay(Date* date)`,
- `int getDateMonth(Date* date)`,
- `int getDateYear(Date* date)`.

Moreover, implement the function `int isMeaningful(Date* date)`, which determines whether a given date is admissible. The function returns the values `1` if the date is admissible, and `0` otherwise. For instance, the date 31.02.2022 is not admissible. Do not forget to consider leap years! Finally write a main program to test your implementation in an appropriate way. Save the source code by splitting it into a header file `date.h` and `date.c`.

**Exercise 6.7.** From the MATLAB exercises you already know that the *secant method* can be used to approximate the root $x^\star$ of a function $f : [a, b] \to \mathbb{R}$. Given two initial guesses $x_0$ and $x_1$, the algorithm defines the sequence of approximations $(x_n)_{n \in \mathbb{N}_0}$ for $n \geq 2$ as

$$x_n := x_{n-1} - f(x_{n-1}) \frac{x_{n-2} - x_{n-1}}{f(x_{n-2}) - f(x_{n-1})},$$

i.e., the approximation $x_n$ is the root of the line that connects the points $(x_{n-2}, f(x_{n-2}))$ and $(x_{n-1}, f(x_{n-1}))$. Write a function `double secant(double (*f)(double), double x0, double x1, double tau)`, which performs the above iteration until either

$$|f(x_n) - f(x_{n-1})| \le \tau$$

or

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

In the first case, print a warning to inform the user that the result is presumably wrong. The function returns $x_n$ as an approximation of a zero $x^\star$ of $f$. Use `assert` to check $\tau > 0$. The function requires as input a suitable implementation `double f(double x)` of the object function $f$. Moreover, write a main program `secant.c`, which reads $x_0$, $x_1$, and $\tau$ from the keyboard, calls the function, and prints the approximate zero $x_n$ and the function value $f(x_n)$ to the screen. How can you test your code? What are good examples?

**Exercise 6.8.** In this exercise, we reconsider the *Sieve of Eratosthenes* computing all prime numbers smaller than or equal to $n_{\max}$. Recall that the algorithm consists of the following steps:

- Initialize a list of natural numbers $(2, \dots, n_{\max}) \in \mathbb{R}^{n_{\max}-1}$.
- Sweep the list of all multiples of lowest entry (except the lowest entry itself).
- Consider the subsequent entry (if any) and sweep all its multiples from the list. Repeat this procedure until you reach the final element of the list.

Write a structure `Eratosthenes` for the storage of the result of the algorithm. The structure consists of the upper bound $n_{\max}$ (`int`), the number of prime numbers $n \le n_{\max} - 1$ (`int`), and the vector in $N^n$ of the prime numbers (`int*`). Implement also necessary access functions to work with this structure. Moreover, write a function `Eratosthenes* = doEratosthenesSieve(int nmax)`, which realizes the Sieve of Eratosthenes and uses the structure `Eratosthenes` to return the result. Realize the sweeping in a suitable and efficient way. Note that the vector containing the prime numbers must be of minimal length. Test your implementation accurately! Save the source code by splitting it into `Eratosthenes.h` and `Eratosthenes.c`.