Programming Methods in Scientific Computing

Winter semester 2019/2020

Prof. Dr. Marc Alexander Schweitzer
Clelia Albrecht and Albert Ziegenhagel

UNIVERSITÄT BONN

Exercise sheet 1.

The assignment corresponding to this exercise sheet can be found at `https://classroom.github.com/a/fvUmEpUl`. Accept the assignment and download the starter code repository.

Exercise 1. (Taylor series)

The most common way to define the exponential function $\exp : \mathbb{R} \to \mathbb{R}$ is via the power series:

$$\exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$

Similarly, we have the following identity for the sine function:

$$\sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}.$$

Please implement the following functions in the file `taylor.py`, where you can find the corresponding function heads.

a) Use the first identity to write a `Python` function `exp_approx(epsilon, x)` for the approximative computation of $\exp(x)$ to an error epsilon. Formulate the loop condition without referencing the true value. Find a way to avoid division by very large integers (and still be accurate enough).

b) Plot your function on the interval $[-1, 4]$ using matplotlib. Furthermore consider the `exp()` function in the math package (`import math`) and plot the error between your function and the one from the math package for different values of epsilon.

c) Do the same for $\sin(x)$ on suitable intervals using the second identity above.

Exercise 2. (A rudimentary matrix class)

In the file `matrix.py` you can find a class for the representation of matrix computations. The member functions of that class are not yet implemented. Only comments are present that describe what each of the functions is supposed to do. Implement all missing functionality of the matrix class. This includes:

a) The initialization function that creates a new matrix from a given set of values via `A = Matrix([[1, 2], [3, 4]])`.

b) Functions to get the row and column dimensions of the matrix via `A.rows()` and `A.cols()`.

c) Functions to access and modify specific values of the matrix via `A[i,j]`.

d) Generating a string representation of a given matrix via `str(A)` or to print it directly via `print(A)`. The representation should be in the usual form as a vertical sequence of rows, e.g.

```
1. 2.
3. 4.
```

e) A test for equality using the infix-operator via `A == B`.

f) A matrix-matrix product via `A * B`.

g) Implement the function `lu_decomposition(A)` that computes the LU-decomposition with partial pivoting $PA = LU$ of suitable regular matrices. Here L and U are lower and upper triangular matrices and P is a permutation matrix to reorder A. Use the matrix class from the first exercises to represent the input A and the resulting matrices L, U and P. Test the program for the inputs

$$A = \begin{pmatrix} 3 & 2 & 1 \\ 6 & 6 & 3 \\ 9 & 10 & 6 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

h) Implement the function `inverse(A)` to compute the inverse A^{-1} of an invertable square matrix A. You can use the LU-decomposition from the last exercise within your implementation. Test the implementation with at least the following matrix

$$A = \begin{pmatrix} 3 & -1 & 2 \\ -3 & 4 & -1 \\ -6 & 5 & -2 \end{pmatrix}.$$

i) Write a short text to answer the following questions:

- Why did you implement your matrix class the way you did?
- What makes your implementation efficent or maybe inefficient? What could you do to change that?
- Why is overloading operators a good idea for such a matrix class? What are the benefits?

Exercise 3. (Gauß-Seidel)

The Gauß-Seidel method is an iterative method to solve a linear system of equations $Ax = b$. The matrix A is decomposed into a diagonal matrix D, a strictly lower triangular component L and a strictly upper triangular component U, such that $A = L + D + U$. Starting with an initial value $x^{(0)}$, the following sequence is then computed iteratively:

$$x^{(k+1)} = (D + L)^{-1}(b - Ux^{(k)}).$$

We stop iterating when the norm of the update $|x^{(k+1)} - x^k|_\infty$ is less than some tolerance ϵ or we reach an iteration limit $k_{\max}$.

a) Implement the Gauss-Seidel iteration in a function `gauss_seidel(A,b,x_0)`. Use your own matrix class from the first exercise to represent A and a simple `Python` list to represent the vectors b and $x^{(k)}$. The return value of the function is supposed to be the final result vector after reaching one of the termination criteria. Use $\epsilon = 1\text{e}{-5}$ and $k_{\max} = 1000$ for your tests.

b) Consider at least the linear equation system for your tests:

$$\begin{pmatrix} 4 & 3 & 0 \\ 3 & 4 & -1 \\ 0 & -1 & 4 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 24 \\ 30 \\ -24 \end{pmatrix}$$

c) Implement the same Gauss-Seidel iteration using matrices and vectors from the `Python` package `NumPy`.

d) Compare the implementation of your own matrix class with the the one using `NumPy` and write a short text about this comparison. Within the file `data.py` in the repository you can find matrices and right-hand-side vectors for equation systems with 3, 5, 10 and 20 unknowns. Solve these systems of equations and try to get an impression of how the runtime of the two implementations compare to each other. What is the main takeaway?

Cellular automata

Finite cellular automata are a useful tool to model spatially discrete dynamic systems. They are characterized by the following properties:

1. A finite cellular automaton consists of a finite number of (commonly) one- or two-dimensional cells. All of these cells are of the same shape and size.
2. Every cell of an automaton is assigned just one of a finite number of states.
3. The state of a cell in time step t_{n+1} only depends on the state of itself and its neighbors in time step t_n. This dependence is determined by a local transition function.

Choose at least one of the following exercises. In these exercises, the choice of objects, functions and interfaces is up to you, therefore there are no tests your solution has to pass.

Exercise 4. (Conway's Game of Life)

The Game of Life takes place on a quadrilateral, two-dimensional grid. Every cell of this grid can be either dead or alive. The neighborhood of a cell is given by the eight adjacent cells. To determine the state of a cell in the next time step, the following rules apply:

1. A living cell with less than two and more than three living neighbors dies. A living cell with two or three living neighbors lives on.
2. A dead cell with exactly three living neighbors rises from the dead and is alive again.

a) Assure yourself (in writing) that the Game of Life does indeed fulfill all the requirements of a cellular automaton.

b) Implement a `Python` simulation of the Game of Life. One way to represent the grid could be to use a matrix whose entries correspond to the state of the cells. Choose a random initial condition and try to avoid unnecessary case distinctions for boundary cells.

c) To check that your implementation is correct you need some kind of animation. The easiest way to achieve that would be to print a matrix that represents the grid and then clear the terminal again for every time step.

Exercise 5. (A simple traffic simulation)

We want to use cellular automata for a simple traffic simulation. We consider a simple one-way, one-lane road with periodic boundary conditions, i.e. a circle. Each cell of our automaton represents a fixed length of the road, which is either occupied by a car driving with a certain speed or not occupied at all. The set of states is therefore "no car" or "car with speed x". To simplify things we only consider discrete speed values and measure

speed in cells per time step. Furthermore, our model should be collision-free and no cars should vanish or appear out of thin air. We set a speed limit at `speed_limit` cells per time step and assume that each car wants to drive this fast if it is able to, i.e. if it would not collide with another car.

a) Assure yourself again, that this simple traffic model does fulfill all requirements of a cellular automaton. What is a sensible definition of a neighborhood in this case?

b) Think of a local transition function that does meet all our requirements as stated above and write an algorithm for the traffic simulation in pseudo code.

c) Write a `Python` implementation of your traffic simulation algorithm, which should be able to handle variable road lengths and speed limits. Choose a random initial configuration.

d) Animate your traffic simulation. The easiest way might be to use an array to represent the street, print it out and then clear the terminal for every time step.