Programmiermethoden des Wissenschaftlichen Rechnens

Winter semester 2018/2019
Prof. Dr. Marc Alexander Schweitzer
Clelia Albrecht und Albert Ziegenhagel

UNIVERSITÄT BONN

Exercise sheet 2.

SymPy

Exercise 13. (System of linear equations)

Consider the matrix and the vector

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & -1 \\ \alpha & -1 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 3 \\ \beta \\ -3 \end{pmatrix}. \tag{1}$$

a) Determine the values $\alpha \in \mathbb{R}$, for which the homogeneous equation system $Ax = 0$ only has the trivial solution $x = 0$.

b) Are the column vectors of A linearly dependent for $\alpha = 0$?

c) Find the values for $\alpha, \beta \in \mathbb{R}$, for which the equation system $Ax = b$ has no solution.

d) Compute the solution set of $Ax = b$ for $\alpha = -3,\ \beta = 0$.

Exercise 14. (Eigenvalues)

Compute the eigenvalues $\lambda_i(\epsilon)$ and the eigenvectors $\varphi_i(\epsilon)$ of the matrix

$$A(\epsilon) := \begin{pmatrix} 1 + \epsilon\cos(2/\epsilon) & -\epsilon\sin(2/\epsilon) \\ -\epsilon\sin(2/\epsilon) & 1 + \epsilon\cos(2/\epsilon) \end{pmatrix}. \tag{2}$$

How do $A(\epsilon)$, $\lambda_i(\epsilon)$, and $\varphi_i(\epsilon)$ behave for $\epsilon \to 0$?

Exercise 15. (Nonlinear equations)

Find all solutions for the equation $x^3 + 3x - a = 0$ for the real parameter $a \in \mathbb{R}$. Show that for each choice of a, there is only *one* real-valued solution x. Sketch that solution x as a function of a in the interval $[-500, 500]$.

Exercise 16. (Newton III)

a) Modify the Newton solver from Exercise 6, such that it now computes the derivative function with SymPy. Test it for the equations $e^x + 2x = 0$ and $\cosh(x) = 2x$.

b) Modify the solution program in Exercise 7 to let SymPy compute the Jacobian.

Exercise 17. (ODEs II)

a) Solve the IVP

$$u'(x) + u(x) = x, \quad u(0) = 1,$$

from Exercise 10, this time using Sympy.

b) Solve the BVP

$$u''(x) = u(x), \quad u(0) = 0,\ u'(1) = -1.$$

Exercise 18. (Orthonormal polynomials)

We consider the *scalar product*

$$\langle f, g\rangle := \int_a^b f(x)g(x)\,dx \tag{3}$$

in the space $C[a, b]$ of continuous functions over an interval $[a, b]$. In $C[-1, 1]$ we want to construct an orthogonal basis $L_0, \ldots L_n$ of all polynomials of degree n. We start with a basis consisting of all *monomials*

$$P_i(x) := x^i, \quad i = 0, \ldots n,$$

and, applying the *Gram-Schmidt process*, we use them to define the so-called *Legendre polynomials*

$$L_0 := P_0, \quad L_m := P_m - \sum_{i=1}^{m-1} \frac{\langle P_m, L_i\rangle}{\langle L_i, L_i\rangle} L_i, \ \ m = 1, \ldots, n.$$

Write a SymPy procedure to generate the first n Legendre polynomials, and test, for instance, for L_3 and L_6, whether they are in fact orthogonal.

Simulations with Python

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.

Exercise 19. (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 exactly 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 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. To represent the grid, you could e.g. use a matrix whose entries correspond to the state of a cell. Choose a random initial condition and try to avoid unnecessary case distinctions for boundary cells.

Exercise 20. (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 value n". 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 5 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. Choose a random initial configuration.

Linear Algebra in C

Exercise 21. (A simple vector type)

We start by creating a simple type for numerical vectors.

a) Create a C type `Vector` that represents a contiguous dense vector of (double precision) floating point values. An object of the vector class should be created by

```
int vec_create(int size, Vector* vector);
```

where `size` the length of the resulting vector and `vector` is the input parameter of the vector to be created. The function should return 0 if the creation has been successful and a non-zero value (e.g. -1) on failure. The function

```
void vec_free(Vector* vector);
```

is then used to free all storage that has been allocated during the call of `vec_create`.

b) Create a function `vec_assemble` to assign values to an already created vector. The signature of that function should look as follows:

```
void vec_assemble(Vector vector, double* values, int size);
```

where `vector` is the vector where to assign the values to, `values` is the array of numbers to assign and `size` is the size of the `values` array (which needs to be the same size as `vector`).

c) Create a function `vec_dot` that computes the dot-product of two vectors. The signature of that function should look as follows:

```
double vec_dot(Vector v1, Vector v2);
```

where `v1` and `v2` are the arguments to the dot product and the result of the computation is returned by the function.

Exercise 22. (A sparse matrix type)

Consider a sparse matrix like the following

$$A = \begin{pmatrix} -7 & 4 & 0 & 2 & 0 & 0 \\ 1 & -5 & 0 & 4 & 0 & 0 \\ 0 & 2 & -11 & 3 & 6 & 0 \\ 0 & 0 & 0 & -2 & 0 & 2 \\ 0 & 1 & 2 & 3 & -9 & 3 \\ 0 & 0 & 7 & 0 & 0 & -9 \end{pmatrix} \tag{4}$$

A naive approach to store this matrix in a computer would be as a *dense* matrix where all entries are stored. A smarter way would be to store the matrix in a *sparse* fashion. Usually this means we store non-zero entries only. One of the most popular formats to store sparse matrices in numerical codes is the *compressed sparse row (CSR)* format. For a sparse $n \times m$ matrix with k non-zero entries, the CSR format stores three arrays `A`, `JA` and `IA`. The array `A` of length k contains all non-zero entries in row-wise order. The array `JA` of length k stores the column indices for each of the non-zero entries. The array `IA` of length $n + 1$ stores the indices of the elements of `A` of `JA` where a new row starts. For the matrix given above the three arrays would look like this:

A	-7	4	2	1	-5	4	2	-11	3	6	-2	2	1	2	3	-9	3	7	-9
JA	0	1	3	0	1	3	1	2	3	4	3	5	1	2	3	4	5	2	5

IA	0	3	6	10	12	17	19

a) Create a C type `CsrMatrix` that represents a sparse matrix in the format that has been described above. An object of the matrix should be created by the function

```
int csr_create(int rows, int columns, int nnz, CsrMatrix* matrix);
```

where `rows` corresponds to n, `columns` to m, `nnz` is the number of non-zeros k of the matrix and `matrix` is the input parameter of the matrix to be created. The function should return 0 if the construction has been successful or a non-zero value (e.g. -1) if there have been any problems during the construction. The function

```
void csr_free(CsrMatrix* matrix);
```

is then used to free all storage that has been allocated during the call of `csr_create`.

b) Create a function `csr_assemble` to initialize a matrix with values. The initialization is done by passing *triplets* to the assemble function, where a triplet is a 3-tuple of a value and its row and column index (a_{ij}, i, j). The signature of the function should be the following:

```
void csr_assemble(CsrMatrix matrix, double* values,
                  int* row_indices, int* column_indices, int nnz);
```

where `matrix` is the matrix object to fill with values, `values` are the values a_{ij} to store into the matrix, `row_indices` and `column_indices` are the corresponding indices i and j and `nnz` is the total number of values to be set.

c) Create a function `mat_vec_multiply` to compute a matrix vector product $r = Av$. The signature of the function should be the following:

```
void mat_vec_multiply(Vector r, CsrMatrix A, Vector v);
```

where `A` is left-hand-side matrix, `v` is right-hand-side vector and `r` is the vector where to save the result into.

Exercise 23. (Iterative solvers)

a) Create a function `gs_solve` that solves a linear equation system $Au = b$ via the iterative Gauss-Seidel method till the norm of the residual $r = Au - b$ is less than 10^{-10}. The signature of the function should be the following:

```
int gs_solve(CsrMatrix A, Vector u, Vector b);
```

where `A` and `b` are the input parameters and `u` is the resulting vector. The function should return 0 on success and a non-zero value if the equation system could not be solved. Test the function with the matrix given above. How many iterations are necessary to reach the given residual tolerance?

b) Download the file `read.c` from the course website. Within that file there is a function `csr_read` that reads data to create a CSR matrix from a file, but the function contains at least 3 errors. Find and correct all errors.

c) Download the files `test_matrix.txt` and `test_rhs.txt` from the course website. Use the functions `vec_read` and the corrected version of `csr_read` from `read.c` to load the data from the files, create a `CsrMatrix` and `Vector` from that data. Solve the equation system with the Gauss-Seidel algorithm from the previous exercise.