—End Term Assignment—

BP-22 G: Scientific Programming

Dr. Marcel Giar & Prof. Dr. Christian Heiliger

February 17, 2020 Winter Term 2019/2020

Sparse Matrices

The following lines are meant as some sort of guideline for your implementations. Some of them are mandatory, however (those printed in bold-face letters).

- **For all your implementations use the Python programming language or libaries that can be used with Python (e.g. Numpy or Scipy).**
- **For each function that you implement, add the name of the author.**
- **Include example calculations that exemplify how your code should be used and what it can do.**
- **Implement automatic tests for your functions in order to check the correctness of your implementation. You do not necessarily have to write low-level unit tests for all functions; however, consider automatically testing functions that solve certain subtasks. Writing your code in a way that enables testing with ease will make your code modular which makes it easier to understand and to maintain.**
- Try to keep your code "clean" by wrapping parts of code that are used more often inside functions. This will help you with structuring your code and makes it much easier to follow your implementations. It also avoids long lines of boilerplate code and oftentimes helps avoiding unnecessary errors.
- The main function of your program should only contain the calls to the functions doing the work for accomplishing the task. It should read like a rough outline of what the current program at hand will be doing. Do not clutter your main file with code lines that would be should better be put inside a function. This will enable any other person that did not co-author the code help understand the purpose of your coding much easier. Indeed, it will also help you understand your own lines of code better if you choose to revisit it at some point in the future.
- Put your source code in separate source files. For example, functions that in the whole serve a certain purpose could be gathered in the same source file. This also is a means to add structure to your software project. In Python you usually build modules and `import` from them.
- Consider using a version control system like Git. View these tasks as a chance to practise with version control.
- Add comments when needed but only as much as necessary. Indeed, while adding lots of comments is a good intention, they can easily become outdated. As people tend to believe what is written in a comment, this can lead to misunderstandings. You should rather try to make your code as self-explanatory as possible (meaningful names for variables and functions, reasonable structure, ...).
- Do not try to reinvent the wheel (although this can be tempting from time to time): If there is some external library that does some part of your task that you are not explicitly required to implement yourself, use it! You may always use external libraries for reference, of course.

The following lines contain some hints on the content of your presentations and how to present your work related to this project.

- Your presentation should not necessarily only show the results obtained from your calculations done with your codes. Technical details of your implementations (e.g., the kind of library you used) may also be of importance. Showing some lines of code can be helpful at some point if it serves a dedicated purpose. However, please refrain from just scrolling through your raw source code without any special intention. All in all, choose a way to present your projects that you consider best to demonstrate your efforts, *both* concerning the implementations and the results.
- Whenever possible, visualise your results in a suitable form. This can be flowcharts showing program design or graphs depicting results computed with your code.
- If possible, always compare your numerical results to analytical solutions or solutions from already-existent implementations (like Numpy or Scipy). Even more importantly, please include these kinds of comparison in your presentation.

The tasks listed in the exercises define a set of requirement that must be accomplished for your project being rated "good". There is plenty of room for creativity and extra effort (extra/alternative implementations, additional tests, etc.).

Generalities

Amongst the matrices discovered in scientific applications so-called "sparse matrices" form a very common case. For example, these matrices arise when solving partial differential equations (e.g., Poisson's equation) where the derivatives are approximated using finite differences schemes. These finite difference schemes result in large matrices that are very "sparse", meaning that the vast majority of their elements are zero.

In the context of modern scientific computing it is essential to exploit the sparsity of these matrices which results in special storing schemes. In order to have an example we shall consider the following matrix having $N_z = 12$ non-zero entries:[2]

$$A = \begin{pmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{pmatrix} \tag{1}$$

Essentially, an efficient storage scheme only will take care about the values $A_{ij} \neq 0$. In what follows, three one-dimensional arrays will be needed:

(i) An array `AVAL` to store the elements with $A_{ij} \neq 0$.
(ii) An array `JCOL` to store the *column* indices j for which $A_{ij} \neq 0$.
(iii) An array `IROW` to keep track of the *row* indices i for which $A_{ij} \neq 0$.

The most popular scheme is the so-called "Compressed Sparse Row" (CSR) format. Using this kind of layout, the three arrays mentioned above look like the following for our matrix form eq. (1):

index	1	2	3	4	5	6	7	8	9	10	11	12
AVAL	1	2	3	4	5	6	7	8	9	10	11	12
JCOL	1	4	1	2	4	1	3	4	5	3	4	5
IROW	1	3	6	10	12	13						

The array `IROW` needs a bit of explanation: It contains the index pointing to the *beginning* of a new row in arrays `AVAL`, `JCOL`. Hence, the content `IROW(i)` is the position in the latter two arrays where the ith row starts. The length of `IROW` is $n + 1$ (the size of A is $n \times n$), with the last element `IROW(n + 1)` marking the position where a fictitious row with index equal to $(n+1)$ starts: `IROW(n + 1)` = `IROW(1)` + N_z.

For other possible layouts that can be used to store sparse matrices and how to perform matrix vector operations see Ref. [2].

Outline of your tasks

Your task will be to solve sparse linear systems of equations. As a direct solution technique implement Gaussian elimination based on sparse matrices using a storage scheme for the

matrix $A \in \mathbb{R}^{n\times n}$ that takes into account the "sparsity level" of the matrix (for example you can use the CSR scheme outlined above). As a further method implement an iterative scheme like the Conjugate Gradient (CG) method. This method is explained in quite some detail in [2, 1].

Note that the CG method relies on the matrix A being symmetric and positive definite. For your analysis you will most likely be using randomly generated matrices. You may symmetrise these matrices and make them positive definite by letting $A \longrightarrow P = I_{n\times n} + (A + A^T)/2$, where $I_{n\times n}$ is the identity matrix of size $n \times n$ (cf. fig. 1). In order to generate a linear system of equations $P\mathbf{x} = \mathbf{b}$, generate a random vector $\mathbf{v}$ of length n and compute $\mathbf{b} = P\mathbf{v}$ which can then be used as target vector of the linear system.

The functions found in [4] can generate sparse matrices of many kinds. **For the present purposes it is sufficient to focus on "banded matrices" (cf. fig. 1), i.e., matrices that only have non-zero values on the diagonal and a few sub-/super-diagonals** (for example, see the command `scipy.sparse.diags` in [4]). Indeed, these are the kinds of matrices that regularly occur in the numerical solution of partial differential equations. You shall implement the

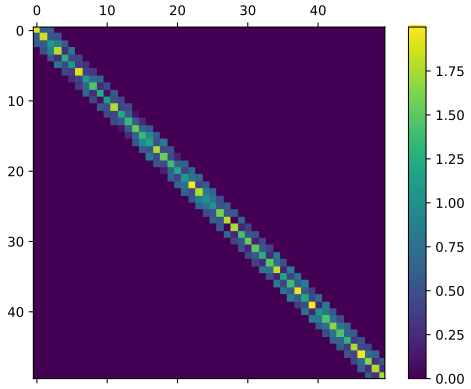

Figure 1: A matrix plots of matrix $P = I_{n\times n} + (A + A^T)/2$ determined from a randomly generated banded matrix A.

storage layout and necessary mathematical operations (matrix-vector products etc.) yourself. For purposes of cross-checking you might want to use the functions provided by `scipy` [4].

The following points shall serve as an overview what we expect you to work on. Feel free to add or replace certain tasks. In case you replace some of the tasks, please make sure that tasks you have chosen are comparable to the replaced ones in terms of complexity and work load.

- Based on a storage scheme applicable to sparse matrices implement the CG algorithm as an iterative method to solve a linear system of equations (implement it for dense matrices as well). As direct solution procedure for *dense* matrices implement LU-decomposition.[3]
- Particularly the CG algorithm heavily relies on matrix-vector operations. Exploit the chosen storage scheme for the matrix to implement functions computing these matrix-vector operations more efficiently than for the case when the full matrix (including all zeros) stored in a standard quadratic array is used.

- Ideally, your functions achieving the task of solving the linear system of equations should be able to deal with full as well as sparse matrices. This will require to analyse the matrix before starting to compute the solution and then to chose the appropriate implementation for the calculation. Particularly for the CG method, it is advisable to start with the implementation for the full matrices. You might, for instance, write a function where you make use of `numpy`/`scipy` for the matrix-vector/vector-vector operations in order to get a feeling for the algorithm.
- Being an iterative method, the CG method needs an initial guess in order to converge to the true solution. Experiment with the initial guess to get a feeling for how sensitive the algorithm is with respect to different starting values for the iteration. If you feel like it, try out preconditioning the CG algorithm [1, 2], although this is not a mandatory requirement.
- Analyse the runtime of your implementations (CG method: dense and sparse matrices; LU-decomposition: dense matrices) and compare them.
- Is there a certain level of sparsity (#non-zero elements$/n^2$, where the dimension of the matrix is $n \times n$) for matrices where using a sparse storage layout is less efficient (in terms of #bytes needed to store all information associated with the matrix) than using a dense storage layout?

References

[1] *Conjugate gradient method*. URL: `https://en.wikipedia.org/wiki/Conjugate_gradient_method`.

[2] *Iterative Methods for Sparse Linear Systems*. URL: `https://www-users.cs.umn.edu/~saad/IterMethBook_2ndEd.pdf`.

[3] *LU decomposition*. URL: `https://en.wikipedia.org/wiki/LU_decomposition`.

[4] *Sparse matrices (scipy.sparse)*. URL: `https://docs.scipy.org/doc/scipy/reference/sparse.html`.