Georg-August-Universität Göttingen
Institut für Theoretische Physik
Prof. Dr. F. Heidrich-Meisner
Dr. E. Bothmann
S. L. Villani
SoSe 2021

Übungen zur Vorlesung
Computergestütztes Wissenschaftliches Rechnen
Blatt 9

Lernziele dieses Übungsblattes

- Reading data from a file
- Using the LAPACK library
- Implementing the Jacobi method and the Gauss–Seidel method

Aufgabenmodus

Die Aufgabe 22 dieses Arbeitsblattes verfügt über verschiedene kurze Guides, die Sie am Ende des Dokumentes finden. Abhängig von Ihren Vorkenntnissen können Sie die Aufgabe entweder eigenständig bearbeiten, oder mit Hilfe dieser Guides. Beachten Sie, dass es sich bei den Guides nicht um ein Tutorial zur Lösung der Aufgabe handelt. Sie dienen lediglich zur Hilfestellung bei verschiedenen Aspekten der Aufgabe.

Auch diese Woche sind die Aufgaben auf Englisch.

Übungsaufgaben

Aufgabe 22 *Using the LAPACK library (Guides)*

Hint: This exercise has several guides and example code that can be downloaded from Stud.IP to familiarise oneself with the new concepts of this exercise. Neither is the solution itself.

In this exercise you are going to learn how to use the LAPACK library, a widely used numerical library to perform linear algebra operations such as linear system solving, QR decomposition, LU decomposition and so on. You will also study the CPU time needed to solve a linear system varying the system/matrix dimension n.

1. Install the LAPACK library (see the installation guide). LAPACK is natively written in Fortran but in recent versions it comes with an API for C (called LAPACKE) and other languages. To use LAPACKE functions in your code you have to add the line `#include <lapacke.h>`.
2. The file `matrices.dat`, which can be downloaded from Stud.IP (as a zipped tarball, `22_Matrices.tar.gz`), lists 30 million numbers randomly generated between 0 to 10. Use it to populate the matrices needed in this exercise. See the file-reading guide to learn how to read data from a file in C.

3. Since we are interested in studying the CPU time needed to solve a linear system versus the system's dimension, it's useful to implement the following sub-tasks within a function that has as an argument the dimension of the system, e.g.:

```
double solve_linear_system(lapack_int n)
{
   ...
}
```

   It should return the time needed for finding the solution.

4. The linear system to solve is of the kind

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{1}$$

   where $\mathbf{A}$ is an $n \times n$ symmetric matrix and the vectors $\mathbf{x}$ and $\mathbf{b}$ have dimension n. According to the conventions of LAPACK (and as we have done on the last sheet), initialise both $\mathbf{A}$ and $\mathbf{b}$ as one-dimensional arrays `double A[n*n]` and `double b[n]` (in our application, we do not need any $\mathbf{x}$ vector since the LAPACK function will overwrite $\mathbf{b}$ with the solution).

5. Fill $\mathbf{A}$ and $\mathbf{b}$ with random data read from the file.

6. Solve the linear system using the LAPACK function described in the LAPACK API guide.

7. Calculate the CPU time (see the corresponding guide) needed to run the LAPACK function for systems of dimension $n \in [500, 5000]$. It is sufficient to sample ten different n in that range.

8. Plot the CPU time (in seconds) versus the system dimension n. Find and discuss the power behaviour in n.

Aufgabe 23 *Solving linear systems with the Jacobi and Gauss–Seidel methods*

These methods are two very similar methods used to solve systems of linear equation $\mathbf{A}\mathbf{x} = \mathbf{b}$. They are based on DLU decomposition. The matrix $\mathbf{A}$ is written as a sum of three matrices: $\mathbf{D}$ containing the diagonal terms, while $\mathbf{L}$ and $\mathbf{U}$ contain the lower and upper off-diagonal terms, respectively.

The *Jacobi* method states that instead of solving the original system one can reorganise it based on the above decomposition and solve the following new system iteratively

$$\mathbf{D}\,\mathbf{x}^{(k+1)} = \mathbf{b} - (\mathbf{L} + \mathbf{U})\mathbf{x}^{(k)}, \tag{2}$$

where $\mathbf{x}^{(0)}$ is an initial guess (which might be random). There is a proof that a sufficient but not necessary condition for convergence to the correct solution is that $\mathbf{A}$ is diagonally dominant.

The *Gauss–Seidel* method follow a very similar idea but it keeps on the l.h.s. also the strictly lower triangular matrix $\mathbf{L}$:

$$(\mathbf{D} + \mathbf{L})\mathbf{x}^{(k+1)} = \mathbf{b} - \mathbf{U}\mathbf{x}^{(k)}. \tag{3}$$

This series has a radius of convergence that is bigger than for the Jacobi method: Convergence is guaranteed if the matrix is either diagonally dominant or positive-definite.

Computationally, the Gauss–Seidel method has the advantage of overwriting **x** during the calculation of the next iteration while for the Jacobi method one needs to use two arrays to store both $\mathbf{x}^i$ and $\mathbf{x}^{i+1}$. Hence Gauss–Seidel method is more suited for problems with large n. On the other hand, for the Jacobi method the calculation of the elements of $\mathbf{x}^{i+1}$ can be parallelised while for Gauss–Seidel it can not. It is therefore important to know both methods since they can perform best different situations.

In this exercise you are going to add these algorithms to your code library and compare their convergence.

1. Create a function for the Jacobi method, with the following signature:

```
void lgs_jacobi_solve(int n, double A[], double b[],
                      double x[], int k_max)
{
   ...
}
```

   where you have to implement the following computation:

$$x_i^{(k+1)} = \frac{1}{A_{ii}} \left(b_i - \sum_{j \neq i} A_{ij} x_j^{(k)} \right). \tag{4}$$

   `k_max` is the number of iterations to perform.

2. Now, implement Gauss-Seidel within

```
void lgs_gs_solve(int n, double A[], double b[],
                  double x[], int k_max)
{
   ...
}
```

   the computation to perform being

$$x_i^{(k+1)} = \frac{1}{A_{ii}} \left(b_i - \sum_{j=1}^{i-1} A_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} A_{ij} x_j^{(k)} \right). \tag{5}$$

   where again `k_max` is the number of iterations.

3. The files `23_matrix.dat` and `23_b.dat` within the `23_LGS_Files.zip` that can be downloaded from Stud.IP contain the data for the linear system you have to solve. The file `23_matrix.dat` is the matrix **A** and `23_b.dat` is the vector **b**. Read in the file according to what you have learnt in the previous exercise.

4. Solve the system using both methods with 40 iterations, and compare the two results.

5. For both methods, plot the modulus of the solution, $||x^{(k)}|| = \sqrt{\sum x_i^{(k)2}}$, for each iteration k. Also plot $||x^{(k)} - x^{(k+1)}||$ in dependence of k. Discuss the difference in the convergence behaviour between the two methods. Which one performs better?

Selbsttest

- Can you think of a reason why allocating memory for a one-dimensional array with n^2 elements might give better performance (both for the allocation itself and for the iteration over its elements) than allocating memory for each row of a two-dimensional $n \times n$ array of the same overall size?
- Which are the three main standard functions to read in data from a file? How much of the file do they read for each call?
- Describe the difference between the Jacobi and the Gauss–Seidel methods. What are their respective advantages?

Guides

Aufgabe 22 - Using the LAPACK library

1) **Installation**:

   This guide covers the installation of a recent version of LAPACK (v3.9.1 in our case) and the required compilation flags to use under Linux, macOS and Windows. The steps required to use LAPACK in your C program are similar to the ones we have used for the *GLS* library: We first make sure the library is installed, and then add compilation flags to ensure that i) the included headers are found and ii) the required libraries are found and linked.

   i) *Linux*: `sudo apt install liblapacke-dev libblas-dev`
   With this command the libraries will be installed in the default directory tree, so to use them it should be enough to add the compilation flags `-llapacke -lblas`, to link the required libraries.

   ii) *macOS*: `brew install lapack`.
   This command installs the library in a non-default location, because macOS ships with an older version of LAPACK, and the `brew` command does not want to override it. So it might be necessary to specify the path for the library, i.e. using
   `-I$(brew --prefix lapack)/include \`
   `-L$(brew --prefix lapack)/lib -llapacke`.

   iii) *Windows*: `pacman -S mingw-w64-x86_64-lapack`.
   With this command the libraries will be installed in the default directory tree, so to use them it should be enough to add the compilation flags `-llapacke -lblas`, to link the required libraries.

2) **Read from file**:

   To read from a file you need to first open it, specifying that we are going to read from it. The command is the one used to write to a file:

   ```
   FILE *file = fopen("filename", "r");
   ```

   where the second argument of `fopen()` is now `"r"`, which stands for *read*. Once the file is open in reading mode there are three ways to access its content. One is using the function

```
char fgetc(FILE *file);
```

which reads and returns a single char from the file. The second is

```
char *fgets(char *buf, int n, FILE *file);
```

this function reads up to `n-1` characters from the file and it copies the read string into the buffer `buf`, appending a `NULL` character to terminate the string. If this function encounters a newline character `\n` or the end of file character `EOF` before having read the maximum number of characters, `buf` only contains the characters read up to that point (including the new line character in the first case).

Finally there is the function

```
int fscanf(FILE *file, const char *format, our_variable);
```

which stops parsing whenever a whitespace character is encountered. For our case at hand (and the way the file `matrices.dat` is written), the best option is to use `fscanf` so that we don't have to bother about any data parsing as long as each value is separated by whitespace.
Be careful when specifying the format and passing the variable you want to write the parsed value to, they have to be consistent with each other. Using our use case as an example, to copy a value into **A** you have to write:

```
fscanf(file, "%lf", &A[i]);
```

where `lf` stays for *long float* (i.e. then double precision) while the ith entry of `A` is passed through via its address. You will find an example of this in a file you can download from Stud.IP, `22_LAPACK_Vorlage.zip`.

3) **LAPACK subroutines**

These have the form `LAPACK_pmmaaa`, where `p` says the type of the elements in the matrix, e.g. `s` and `d` stand for real floating point number at single and double precision, `mm` is the type of matrix, e.g. `ge` stands for general matrix, `sy` for symmetric, and finally `aaa` represents the operation we want to perform on the system, e.g. `ls` solves a least square problem or `bdr` stands for bidiagonal reduction. For our case, the subroutine we need is `LAPACK_dsysv`, a symmetric linear system with floating point entries at double precision. The arguments of this function can be easily found in the LAPACK manual, but for simplicity you'll find a description in the example files within `22_LAPACK_Vorlage.zip` on Stud.IP.

4) **CPU time**

To calculate the CPU time of an operation in a C program you need to include the header `time.h`. From this header we have access to the function `clock()` which returns the current number of clocks, the type of which is `clock_t`. Its precision depend on your processor so it might not be the same for everyone, but it should at least resolve milliseconds. This is not an issue since our calculation will take seconds.

To calculate the time of an operation, initialise two `clock_t` variables, `begin` and `end`. Write the returned value from `clock()` to `begin` before executing the LAPACK function, and the returned value from another call to `clock()` to `end` after

the execution. The operation's duration is then given by the subtraction of the two variables, i.e. `end-begin`.

The result is not in our time convention of *seconds*, but in *clocks*. To convert the result to seconds, divide the CPU time duration just found by the macro variable `CLOCKS_PER_SEC`. Before doing this, you have to cast the CPU time into `double`, otherwise integer division is used.

So to summarise, here is a snippet of the just described procedure:

```
#include <time.h>

clock_t begin,end;
begin = clock();
...
operation;
...
end = clock();
double cpu_time = (double) (end-begin)/CLOCKS_PER_SEC;
```