Simulation and High-Performance Computing

University of Kiel, September 28th to Oktober 9th, 2020.

Exercises: Day Three

CSC Computational Science Center

Main Exercises: Applications of Lecture 05

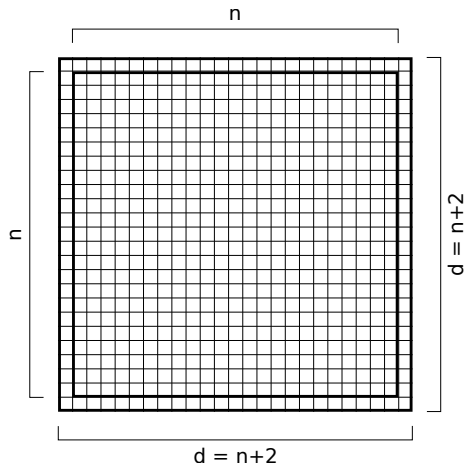
a) Grid functions and Poisson-Problem in 2D

When discretizing an equation like *Poisson's* equation in 2D with a finite difference scheme, one has to work with *grid functions* $u_h : \bar{\Omega}_h \to \mathbb{R}$.

We have to distinguish between *inner points*, which make up the degrees of freedom and *outer points*, which describe the boundary conditions and are therefore fixed.

For the sake of simplicity, our data structure for these grid functions consists of an array that is capable of storing $(n+2)^2 = d^2$ coefficients. Of these coefficients, n^2 elements are reserved for the inner points, the remaining are for the boundary values.

The storage pattern looks as follows:



When solving Poisson's equation, we have to write down the boundary values into the right-hand-side. Hence, we end up with a linear system of the form

$$A\,(u_h)_{|\Omega_h} = b,$$

where the matrix A alongside the boundary data stored in b correspond to the discrete Laplacian Δ_h. In addition to the boundary data, b can also contain the contribution of outside forces.

Since the matrix $A \in \mathbb{R}^{\Omega_h \times \bar{\Omega}_h}$ only operates on inner points and also just takes values from inner points, one has to take care of many special cases when implementing it.

For these exercises, we will only look at the simpler but frequently encountered case of zero boundary conditions, i.e. $(u_h)_{|\partial\Omega_h} = 0$. In that situation, we can 'cheat' and instead of A use the rectangular matrix $L \in \mathbb{R}^{\Omega_h \times \bar{\Omega}_h}$. We then do *not* need to include the boundary condition into the vector b.

Within the function `addeval_5point_stencil_gridfunc2d` **implement** a matrix-vector multiplication with the matrix L. Remember that values on the boundary are zero and, thus, do not contribute to the computations.

Test this operation by multiplying some grid functions with the matrix L and by then solving a linear system using the same matrix and the result of the previous multiplication as right-hand side. You should end up with the same grid function that you started with.

Usefull functions are: `clear_gridfunc2d`, `solve_gridfunc2d` and `init_sine_gridfunc2d` as a test grid function. Beware: the given solver computes the solution in-place, i.e. when b was the right-hand-side before the solver was called, then b will contain the computed solution x.

b) UL Decomposition Using BLAS

In the lectures, a given matrix $A \in \mathbb{R}^{n \times n}$ was written as the product of a lower triangular matrix $L \in \mathbb{R}^{n \times n}$ and an upper triangular matrix $U \in \mathbb{R}^{n \times n}$, i.e., $A = LU$. The lectures showed that this can be done by *only* using BLAS routines. We want you to redo and comprehend that procedure by implementing a *UL* decomposition instead.

We begin with an equation of the form

$$\begin{pmatrix} A_{**} & A_{*n} \\ A_{n*} & a_{nn} \end{pmatrix} = \begin{pmatrix} U_{**} & U_{*n} \\ & u_{nn} \end{pmatrix} \begin{pmatrix} L_{**} & \\ L_{n*} & l_{nn} \end{pmatrix}$$

Starting from this equation, work out the UL decomposition of A into the product of the upper triangular matrix U and the lower triangular matrix L, i.e., $A = UL$. `Implement` this operation - using *only* BLAS routines - in the function `decomp_ul` in the file `linalg.c`.

Now we look at the multiplication of an UL factorization of A with a given vector $x \in \mathbb{R}^n$, i.e., $Ax = (UL)x$. In the file `linalg.c`, `implement` that operation into the function `eval_ul` by first `implementing` functions `eval_l` and `eval_u` which describe the multiplication of a lower or an upper triangular matrix with a vector. Make sure that all operations are *in-place*, i.e., the result is written directly into the given vector x. And, once again, use only BLAS routines.

`Test` your UL decomposition and multiplication algorithms on various matrices.

Additional Exercises: Applications of Lecture 06

a) Method of lines for the Wave Equation in 2D With Euler and Leapfrog

So far, we learned how to deal with ordinary differential equations - i.e., differential equations that depend only on time - by using time-stepping methods. We also learned how to deal with a partial differential equation that depends only on spatial variables by using the finite difference method.

Now we want to look at equations that combine both difficiulties, i.e., they contain temporal *and* spatial derivatives. Unsurprisingly, our approach to those equations will be a combination of what we learned previously. First, we discretize in space and replace the spatial derivatives with discrete counterparts via the finite difference method. Since the time dependancy is still there, we, thus, obtain a system of ordinary differential equations which we can then solve with any time-stepping method. This ordering of the discretization - first in space, then in time - is called *the method of lines*.

The example we want to look at is the wave equation in 2D.

Your task is to **implement** both the explicit Euler method and the Leapfrog method for a finite difference discretization of the wave equation. Every 100 timesteps the current state of the position grid function will be written to disk and calling the gnuplot script will generate an animation of your results.

Please provide feedback about the format/difficulty/length of the exercises so that we can adjust accordingly. Contact us via Email or on the OpenOLAT forum.