Programming Methods in Scientific Computing

Winter semester 2018/2019
Prof. Dr. Marc Alexander Schweitzer
Clelia Albrecht, Albert Ziegenhagel
and Lukas Troska

UNIVERSITÄT BONN

Exercise sheet 6.

The assignment corresponding to this exercise sheet can be found at `https://classroom.github.com/g/J46qGHUT`. Accept the assignment and download the starter code repository. Copy your code from exercise sheet 5 to the corresponding files in this repository.

Parallelization of the Poisson solver

In this exercise we want to complete the parallelization of our finite differences solver. The main step that is missing is to parallelize the grid structure. We will do this by applying a *domain decomposition* approach. This means each node of the grid will be assigned to a single process that becomes the *owner* of that node. After that, each process performs work only on the nodes that it owns and additionally might need to do some extra work for nodes that have neighbors which are owned by a different process. There are two implications of this approach:

- Each process should get a similar amount of nodes, so that the work (aka. *load*) is distributed evenly.
- The number of nodes that have neighbors that belong to another process should be minimized.

In the last exercise each node could be identified by a multi-index α and a flat-index i. The flat-index i implied a global co-lexicographic ordering of the nodes (compare figure 1).
For the parallelization of the grid onto P processes we will now split the grid into P as close to equally sized parts as possible. Instead of just splicing the domain into P parts, we will try to choose parts that are as close to squares as possible. This will try to minimize the number of nodes that are located at process borders. To perform this splitting we need find the number of processes per space dimension P_d so that $P = \prod_{i=0}^{d-1} P_d$ (compare figure 2).
After that splitting each process can introduce local indices i_l and α_l that take into account only the nodes that are local to the process. The global multi-index α_g is assigned to each node the same way as in the non-parallel case. The global index i_g of a node that is owned by the process p is defined by $i_g = i_l + \sum_{q=0}^{p-1} N_q$ where N_q is the number of nodes owned by the process with the rank q (compare figure 3 and 4).

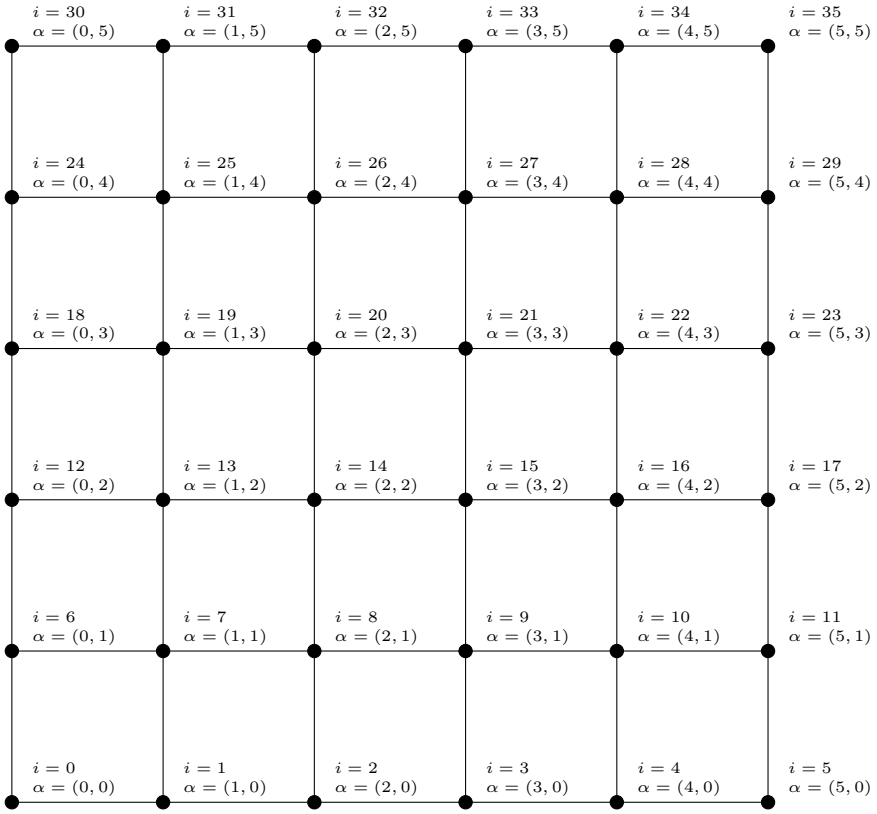

Figure 1: Non-distributed grid. Each node is identified by a co-lexicographically ordered flat-index i and a multi-index α.

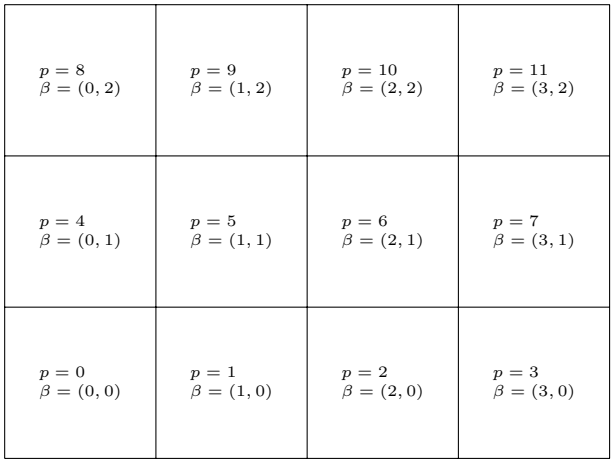

Figure 2: Domain decomposition onto $P = 12$ processes with $P_0 = 4$ and $P_1 = 3$. Each process is identified by its rank p as well as its coordinates β.

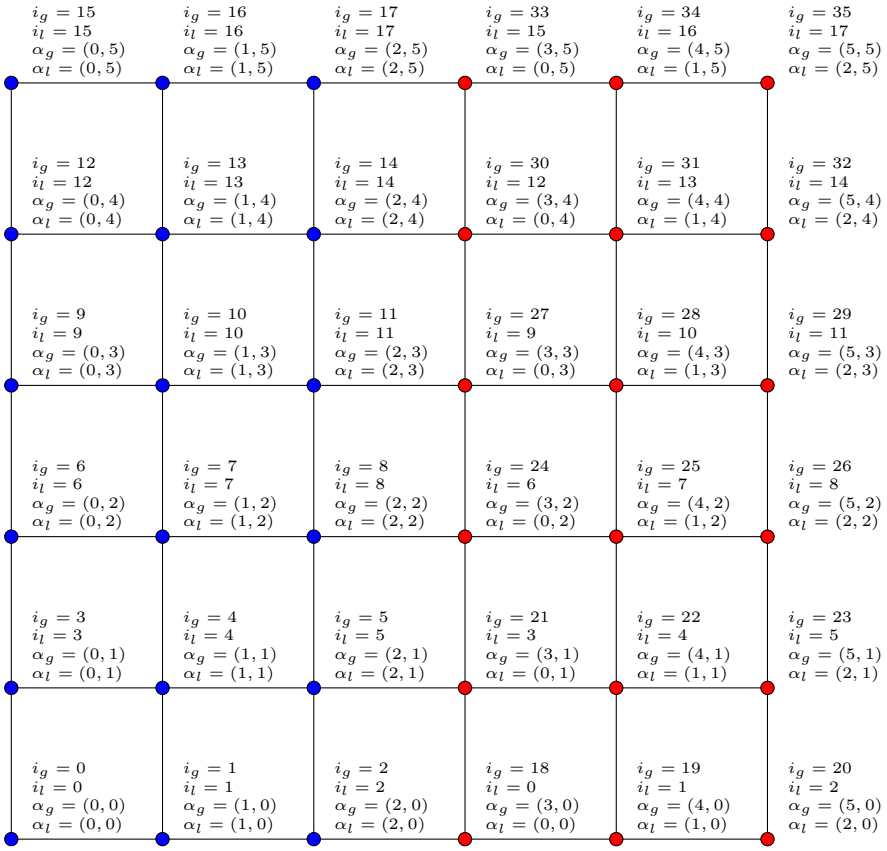

Figure 3: Grid distributed onto two processes (blue: $p = 0$, $\beta = (0,0)$, red: $p = 1$, $\beta = (1,0)$). Each node is identified by the local indices i_l and α_l as well as the global indices i_g and α_g.

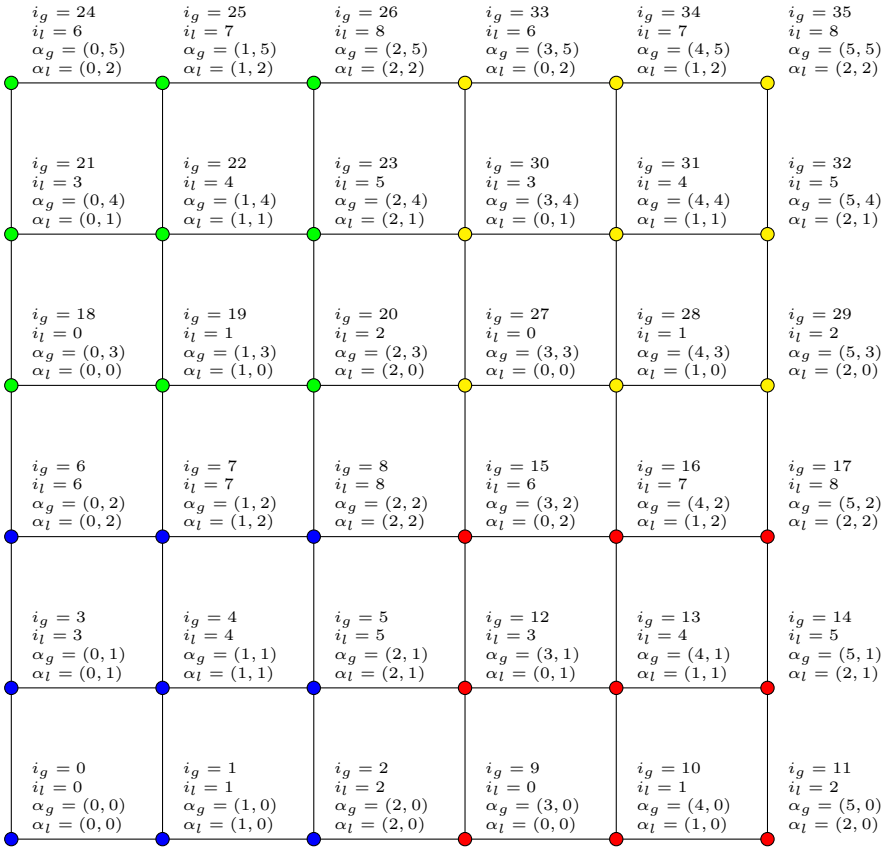

Figure 4: Grid distributed onto four processes (blue: $p = 0$, $\beta = (0,0)$, red: $p = 1$, $\beta = (1,0)$, green: $p = 2$, $\beta = (0,1)$, yellow: $p = 3$, $\beta = (1,1)$). Each node is identified by the local indices i_l and α_l as well as the global indices i_g and α_g.

Exercise 1. (Parallelization of the grid class)

a) First we will introduce an additional function to `ContiguousParallelPartition` that might come handy in the following implementations. Add the function

```
int to_global_index(int local_index, int owner_process) const;
```

That allows us to convert the `local_index` i_l of the `owner_process` p to its global index i_g.

b) Add a constructor to the the `RegularGrid` class with the following signature:

```
RegularGrid(MPI_Comm communicator,
            Point min_corner,
            Point max_corner,
            MultiIndex global_node_count_per_dimension);
```

This constructor should distribute the grid onto the P processes as described above. Use the functions `MPI_Dims_create` to compute the values for P_d. After that you can use `MPI_Cart_create` to create a new MPI communicator that has topological information and the coordinates β attached to it. Afterwards you can use `MPI_Cart_coords` to get the coordinates β of a process with rank q from that communicator and `MPI_Cart_rank` to the rank q of the process at the coordinates β. Create and store an object of `ContiguousParallelPartition` that uses the number of local nodes N_p as the local size.

c) Add a getter for the parallel partition of the grid:

```
const ContiguousParallelPartition& partition() const;
```

d) Add getters to retrieve the processes per dimension P_d and the coordinates β of the local process p from the grid.

```
MultiIndex processes_per_dimension() const;
MultiIndex local_process_coordinates() const;
```

e) Add getters for the node counts per dimension to the grid class:

```
MultiIndex global_node_count_per_dimension() const;

MultiIndex node_count_per_dimension() const;
MultiIndex node_count_per_dimension(int process_rank) const;
```

The first functions returns the total number of nodes per dimension of the whole grid over all processes. The second function returns the number of nodes in each direction that are owned by the current process p. The last function returns the number of nodes in each direction for any other process q.

f) Modify the meaning of the functions from the previous exercise in the grid:

```
int number_of_nodes() const;
int number_of_inner_nodes() const;
int number_of_boundary_nodes() const;
```

All this functions should return the number of all/inner/boundary nodes of the complete grid, and **not** only the part local to the current process.

g) Modify the meaning of the functions from the previous exercise in the grid:

```
int number_of_neighbors(int local_node_index) const;
int neighbors_of(int local_node_index,
                 std::array<std::pair<int, int>, space_dimension>& neighbors) const;

bool is_boundary_node(int global_node_index) const;

Point node_coordinates(int global_node_index) const;
scalar_t node_neighbor_distance(int local_node_index,
                                int neighbor_direction,
                                NeighborSuccession neighbor_succession) const;
```

All function arguments that are called `local_node_index` accept the **local** index i_l of a node. This means they can only be called for nodes that are owned by the current process p. Function arguments that are called `global_node_index` accept the **global** index i_g of a node. This means they accept any node and need to return correct results for those nodes even if the owner q of that specific node is not the local process p.

The function `neighbors_of` should put the **global** indices i_g of the neighbors into its output parameter `neighbors`.

Exercise 2. (Parallelization of the solvers)

In the last exercise we parallelized our linear algebra types (matrix and vector) and all operations already. Now we want to make sure that we can use the distributed matrices and vectors in our linear solvers. Since the solvers basically just use the already parallel matrix-vector product and vector dot-product, this is actually a pretty straight forward task to be done:

a) Check your code of `Solver`, `IterativeSolver`, `RichardsonSolver`, `CgSolver` and `JacobiIteration` (**not** the `GaussSeidelPreconditioner`) for all places where a temporary vector is created. Replace the call to the non-parallel vector constructor that takes the size of the vector with a call to the parallel vector constructor that takes the `ContiguousParallelPartition` as first argument.

b) Usually no other changes should be required to make the solvers/preconditioners above work with distributed matrix and vectors types. If your specific implementation requires additional changes, identify the code locations and perform the changes.

c) To parallelize the `GaussSeidelPreconditioner` we would need to apply some additional changes. Why is this the case and what changes would need to be done? Create a `*.txt` file with the answers to the questions and upload it into your repository. **optional:** Apply the proposed changes to your implementation of the `GaussSeidelPreconditioner` and make sure the preconditioner works with distributed matrices/vectors.

Exercise 3. (Export to Python)

Let us now make sure we can solve a Laplace problem via a Python script in parallel. Within the repository there is a file `python/module.cpp` that contains the export of some MPI related functionality. It exports a class `MpiCommWrapper` to python that allows to pass around MPI communicators in python. Objects of that class are *implicitly* convertable to and from an `MPI_Comm` object. Additionally there are exports of two functions `mpi_comm_world` and `mpi_comm_self` that allow you to access `MPI_COMM_WORLD` and `MPI_COMM_SELF` from within a python script.

a) Add all exports from the last exercise to the `python/module.cpp` file.

b) Export the MPI related constructors of your `SparseMatrix`, `Vector` and `RegularGrid` class to python. Use `MpiCommWrapper` instead of `MPI_Comm` when specifying the argument types of the constructors in the `py::init<>` calls.

c) Modify your python script that solves the Poisson equation from the last exercise to create a distributed grid over `mpi_comm_world`. You should be able to run the Python script in parallel via

```
mpiexec -n <NUMBER_OF_PROCESSES> python name_of_my_script.py
```

Note: on Linux and macOS you might need to use `python3` instead of `python`.

Time Dependent Problems

Now we want to solve a time dependent problem, namely the heat equation, given by

$$\frac{\partial \Phi}{\partial t} = \alpha \Delta \Phi + f, \quad \alpha \in \mathbb{R}, \tag{1}$$

where Φ is the temperature given as a function of space and time. Here $\Delta\Phi$ denotes, as before, the Laplacian of Φ with respect to space. The parameter α is related to the thermal diffusivity of the material which affects the speed and scale of the problem. A common simplification, which we will adopt, is to set α to 1.

One immediate observation is that the heat equation in this form looks very similar to the Poisson equation that you have encountered in exercise sheet 4 (with u replaced by the temperature Φ). The only difference is an additional term given by the time derivative of the temperature, $\frac{\partial \Phi}{\partial t}$. An intuitive way to understand this equation is given by the interpretation of the Laplace operator $\Delta\Phi(x)$ at some point x as the relation of the average of the "surrounding" values to the value at the point x: If $\Delta\Phi(x) < 0$, then the "surrounding" values are on average smaller than $\Phi(x)$ and if $\Delta\Phi(x) > 0$ they are larger. Therefore equation (1) can be interpreted as saying that the temperature at a point x rises (or falls) if the surrounding temperature is larger (or smaller). The term f simply represents a heat source or sink inside of the domain.

To be able to solve the heat equation (1) it needs to be accompanied by appropriate boundary conditions for the temperature in both space and time (which themselves can depend on time). Therefore, in our case, the continuous problem is given by

$$\begin{aligned} \frac{\partial \Phi}{\partial t}(x,t) &= \Delta\Phi(x,t) + f(x,t) && \text{for } x \in \Omega, && t \in (0,T], \\ \Phi(x,t) &= g(x,t) && \text{for } x \in \partial\Omega, && t \in (0,T], \\ \Phi(x,0) &= \hat{\Phi}_0(x) && \text{for } x \in \Omega, \end{aligned} \tag{2}$$

where $f(x,t)$ is a prescribed heat-source, $g(x,t)$ is a prescribed boundary condition in space and $\hat{\Phi}_0(x)$ is the prescribed initial temperature.

To be able to solve problem (2) the differential operators need to be discretized. From exercise sheet 4 you already know how to do this for the Laplacian $\Delta\Phi$ using finite differences. Here, we will also use finite differences to discretize the time derivative $\frac{\partial\Phi}{\partial t}$. For this we divide the time interval $[0, T]$ into $M + 1$ uniform time points, such that the k-th time step is given by $t_k := k\Delta t$, where $\Delta t := \frac{T}{M}$. We then define for each time t_k a space discretization of the temperature as in exercise sheet 4. Let Φ_i^k represent the temperature at time t_k at the i-th node. We approximate the time derivative by the first order differences:

- Backward difference: $(D^-\Phi)_i^k = \frac{\Phi_i^k - \Phi_i^{k-1}}{\Delta t}$
- Forward difference: $(D^+\Phi)_i^k = \frac{\Phi_i^{k+1} - \Phi_i^k}{\Delta t}$
- Central difference: $(D^\pm\Phi)_i^k = \frac{\Phi_i^{k+1} - \Phi_i^{k-1}}{2\Delta t}$

We will only consider the case of a backward difference in time, which is also known as a backwards or implicit Euler scheme. The reason for this is that the implicit Euler scheme, compared to the explicit Euler scheme given by forward differences in time, is unconditionally stable, which places no restrictions on the choice of the space discretization parameter h or the time discretization size Δt. On the other hand it is computationally more expensive since we need to solve a linear system for each time step.
Inserting these difference operators into the heat equation and rearranging yields the discretized form of (2) at time t_k

$$(D^-\Phi)^k - \Delta_h\Phi^k = f^k. \tag{3}$$

Similarly as in exercise sheet 4 for the Poisson problem, we can bring all spatial boundary conditions to the right hand side. This yields the system

$$(D^-\Phi)^k + \tilde{A}^k\Phi^k = b^k, \tag{4}$$

where $\tilde{A}^k$ and b^k are the usual system matrix and right hand side vector of the Poisson problem with incorporated boundary conditions.
All that is now left is to handle the finite difference quotient $(D^-\Phi)^k$. The idea is to split $(D^-\Phi)^k$ into unknown and known parts, namely

$$(D^-\Phi)_i^k = \frac{\Phi_i^k - \Phi_i^{k-1}}{\Delta t} = \frac{\Phi_i^k}{\Delta t} - \frac{\Phi_i^{k-1}}{\Delta t}. \tag{5}$$

We can then bring the known part $\frac{\Phi_i^{k-1}}{\Delta t}$ to the right hand side and define a new right hand side vector d^k such that

$$d_i^k := \begin{cases} b_i^k + \frac{\Phi_i^{k-1}}{\Delta t} & \text{node } i \text{ is an internal node} \\ b_i^k & \text{otherwise.} \end{cases} \tag{6}$$

Similarly, we modify the system matrix to deal with the unknown part, giving a new system matrix C with

$$C_{i,j}^k := \begin{cases} \tilde{A}_{i,j} + \frac{1}{\Delta t} & i = j \text{ and node } i \text{ is an internal node} \\ \tilde{A}_{i,j} & \text{otherwise.} \end{cases} \tag{7}$$

The final discretized linear system for time step t_k is then given by

$$C^k\Phi^k = d^k. \tag{8}$$

This method is also called the BTCS (Backwards Time, Centered Space) scheme. We can immediately see that we only need to store the temperature for the last time step to be able to compute the new solution. Therefore, putting it all together, the pseudocode of the algorithm looks as follows:

Algorithm 1 BTCS scheme

1: $\Phi_{\text{previous}} \leftarrow \hat{\Phi}_0$
2: **for** $1 \leq k \leq M$ **do**
3: Solve the discrete heat problem (8) for Φ^k (with $\Phi^{k-1} = \Phi_{\text{previous}}$)
4: $\Phi_{\text{previous}} \leftarrow \Phi^k$

Exercise 4. (Assembly of the heat equation matrix)

In this exercise, we will assemble the system matrix and right hand side of the discretized heat equation (8) for a given time step t.
Write a template function `assemble_heat_matrix` which should assemble the heat equation system matrix C and a right hand side d for a given grid, temperature of the previous time step, time, time step size, right hand side function and boundary value function. Thus, the function should have the following function header:

```
template<typename T>
std::pair<SparseMatrix<T>, Vector<T>> assemble_heat_matrix
      (const RegularGrid& grid,
          const GridFunction<T>& previous_temperature,
          const scalar_t t,
          const scalar_t delta_t,
          const std::function<T(const Point&, const scalar_t)>& rhs_function,
          const std::function<T(const Point&, const scalar_t)>& boundary_function)
```

Make sure this function also works correctly in parallel and export it to python using `pybind11` as on exercise sheet 5.

Exercise 5. (Solving the Heat equation with FDM)

Consider the heat equation (2) in two dimensions. Let $\Omega = [0,1]^2$, the right hand side function $f(\boldsymbol{x}, t) = -4$, boundary function $g(\boldsymbol{x}, t) = 1 + x_1^2 + 2x_2^2 + 2t$, initial temperature $\hat{\Phi}_0(\boldsymbol{x}) = g(\boldsymbol{x}, 0)$ and final time $T = 2$.

a) Write a python script implementing the BTCS algorithm 1 for this problem using your python module.

b) Use 11 time steps (i.e. $M = 10$) and your favorite number of nodes to solve the problem and write the solution for each time step to a VTK file. If you name the files in ascending order, e.g. `heat_timestep_#.vts`, ParaView can load them all at once and play them in an animation.

c) The analytical solution to this problem is given by the boundary value function g (please verify that for yourself). Visualize the discrete representation of this function in ParaView, too, and compare it to your solution. Use the same grids for both and try out different grid spacings as well as different numbers of time steps. What is the dependence of the accuracy of the solution on these parameters?

d) Create a png picture for the last time step and upload it to your repository.

Exercise 6. (Solving the Heat equation with FDM, again)

Consider again the heat equation (2) in two dimensions. We will now consider the famous diffusion problem of a Gaussian hill. Let $\Omega = [-2, 2]^2$, the right hand side function $f(\boldsymbol{x}, t) = 0$, boundary function $g(\boldsymbol{x}, t) = 0$, initial temperature $\hat{\Phi}_0(\boldsymbol{x}) = \gamma \exp(-\alpha x_1^2 - \beta x_2^2)$ and final time $T = 1$, where $\alpha = 1$ and $\beta = \gamma = 2$.

a) Write a python script implementing the BTCS algorithm 1 for this problem using your python module.

b) Try out different grid and time step sizes. Do you notice any impact on the solver or the computed solution?

c) Write the solution for each time step to a VTK file.

Measuring Errors

A common way to assess the quality of the computed solution is to measure errors of the form

$$e_h := u^* - u_h, \tag{9}$$

where u^* is the analytical solution to the problem (if known) or some approximation to it, usually computed on a very fine grid. One is then interested in the convergence behavior of the computed solution, namely the behavior of the error in a suitably chosen norm $\|e_h\|$ as the discretization becomes finer, i.e. $h \to 0$. In the finite difference setting, the most natural choice is the l_∞-norm

$$\|u_h\|_\infty := \max_{x \in \Omega} |u_h(x)| = \max_{x_i \in \Omega_h} |u_h(x_i)| = \max_i |u_{h,i}|, \tag{10}$$

where we used the fact that piecewise linear functions (like our grid functions) are maximal at the nodes.
From the theoretical standpoint one is interested in two key concepts: consistency and stability. Consistency measures how well the finite difference equations approximate the partial differential equation, whereas stability deals with the influence of errors from any source (e.g. truncation errors, errors in the data, ...) on the computed solution. From the theory it is also known that consistency of order k together with stability implies convergence of order k, i.e.

$$\|e_h\|_\infty \leq C h^k, \tag{11}$$

with $C > 0$ a constant independent of h.
For ease of implementation we now come back to the Poisson problem with Dirichlet boundary conditions. Translating the above considerations to this setting, one can show consistency of order 2 using a Taylor expansion. Stability can also be shown, however the proof is slightly more involved. In total we get quadratic convergence for our problem, where we set $h := \max_{i=1,\dots,d}\{h_{x_i}\}$, i.e. the maximum of all grid spacings.

Exercise 7. (Computing the l_∞-error)

In this exercise, we will compute the l_∞-error for a given analytical solution.
Write a template function `compute_l_infinity_error` which should compute the l_∞-error $\|e_h\|_\infty$ given a grid, a computed solution and an analytical solution. Thus, the function should have the following function header:

```
template<typename T>
T compute_l_infinity_error(const RegularGrid& grid,
                           const GridFunction<T>& computed_solution,
                           const std::function<T(const Point&)>& analytical_solution)
```

Implement it in such a fashion that it also works in parallel and export this function to python using `pybind11`.

Exercise 8. (Error measurement)

In this exercise, we will look at the behavior of the error $\|e_h\|_\infty$ for the Poisson problem as we decrease the grid size (or equivalently, increase the number of nodes). For this, consider the usual Poisson problem on $\Omega = [0,1]^d$ for dimension $d \in \{2,3\}$ with right hand side $f(\boldsymbol{x}) = d\pi^2 \Pi_{i=1}^d \sin(\pi x_i)$ and boundary data $g(\boldsymbol{x}) = \Pi_{i=1}^d \sin(\pi x_i)$.

a) Write a python script that solves the problem for a specified dimension d. Verify that the analytical solution is given by g and compute the error for a sequence of uniformly refined grids, i.e.

$$N = N_i = 2^m \quad \text{for all } i = 1, \dots, d$$

for increasing $m > 1$, where N_i is the number of nodes in spatial direction i. Use the parallel version of your code to be able to compute errors on very fine grids.

b) Plot the computed errors against N using `matplotlib`. What is the appropriate axis scaling to use?

c) Can you observe the theoretical convergence order? Does it depend on the dimension d? Upload one convergence plot for each of $d = 2$ and $d = 3$ to your repository.

d) What happens if you increase the number of nodes only in some spatial directions, i.e. keep at least one of the N_{x_i} fixed?