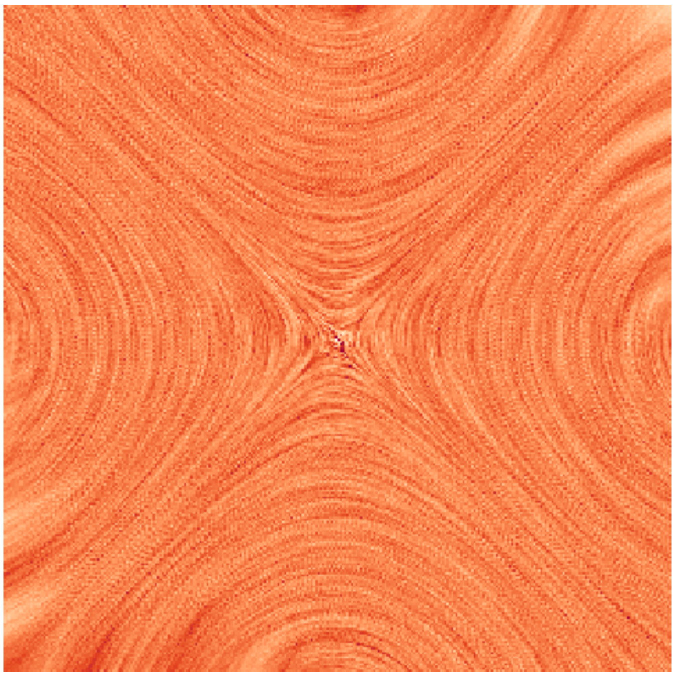
Nédélec Space in IPPL



Semester Project

Department of Mathematics

ETH Zürich

written by

Alexander Pietak

supervised by

Dr. A. Adelmann (ETH, PSI)

advised by

S. Mayani (ETH, PSI)

July 11, 2025

Abstract

Currently the Finite Element implementation in `IPPL` only supports the first order Lagrangian space, which limits its usability to scalar valued functions. In the future we would like to be able to solve problems from electromagnetism, which requires different finite element spaces. To this extent we are introducing the Nédélec space into `IPPL`. While the Lagrangian degrees of freedom are defined on the vertices, the Nédélec ones are defined on the edges, making a one to one translation of the current implementation, especially with regards to storing of degrees of freedom, not possible. To solve this problem we introduce a new `FEMVector` class used to store the values at degrees of freedom located at arbitrary positions, with which we then implement the Nédélec space.

In this report we will talk about why we need the Nédélec space, how we implemented it, and then finish of by showing the correctness and performance of our implementation.

Contents

List of Figures

Chapter 1

Introduction

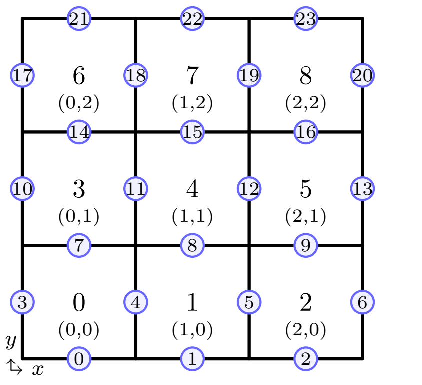
`IPPL` [1] stands for Independent Parallel Particle layer and is a performance portable C++ library designed for Particle-Mesh methods. One primary part of `IPPL` is a Particle In Cell (PIC) loop, in which particles are simulated in a Lagrangian frame, while simultaneously certain properties are calculated in an Eulerian frame (on the mesh), and then these properties are used to update the particles. So we have two independent building blocks, one is the simulation of particles, while the other one is the solving of PDEs in the Eulerian frame. One method that can be used to solve the PDEs and which already is part of `IPPL` is the Finite Element Method (FEM). For a general introduction into FEM please refer to [2] and for an explanation on how FEM is implemented in `IPPL` please refer to [3].

Currently the FEM framework in `IPPL` only supports first order Lagrange basis functions, which limits the range of solvable PDEs to scalar functions. In the future we would like to be able to solve problems from electromagnetism, therefore we need to define a new finite element space which supports such vector fields. A natural choice for the electric field in the case of electromagnetism are the Nédélec basis functions [4], in order to also have the magnetic one a different finite element space would have to be used, namely Raviart-Thomas. The Nédélec basis functions create a $H(\mathrm{curl}, \Omega)$-conforming finite element space, where the basis functions are vector fields defined on the edges of the mesh and they exist for 2D and 3D. In the rest of this report we will look into the implementation of the Nédélec space in `IPPL` and some numerical results in order to test the correctness of the implementation and to provide performance characteristics.

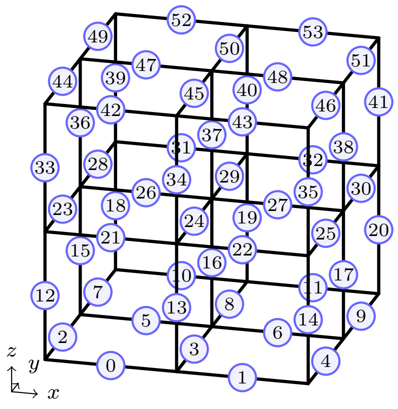
Chapter 2

Method

2.1 Theory



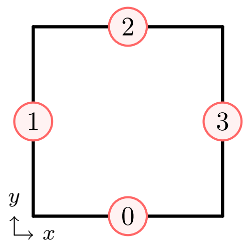
(a) Global DOF numbering scheme, element numbering, and multidimensional index in 2D, for a 4 × 4 mesh.



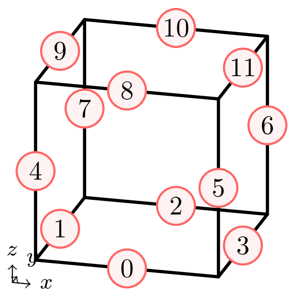
(b) Global DOF numbering scheme for 3D, given for a 3 × 3 × 3 mesh.

Figure 2.1: The global DOF numbering scheme (blue circles) for the Nédélec space in both 2D and 3D. For 2D also the element numbering (black number), and its multidimensional index (black pair) is given, for 3D this was omitted for visual clarity.

Currently `IPPL` only supports structured, rectilinear grid meshes. Therefore our FEM reference elements are the unit square in 2D and the unit cube in 3D, for which our Nédélec Degrees of Freedom (DOFs) reside on the edges. In order to be able to interact with them we assign each DOF a number, this numbering happens by sorting, low to high, the DOF positions according to first their z-coordinate (only in 3D), because there are multiple DOFs with same z-coordinate we have not created a unique ordering, but a block ordering, where we have multiple blocks of DOFs with same z-coordinate and these blocks are sorted correctly. Then in order to further sort we now take each of these blocks and sort them according to their y-coordinate (in 2D we would simply start here and order all DOFs according to their y-coordinate), but this again will create a block sorting for which we then sort each block according to their x-coordinate. After we have sorted the final blocks according to x we now should have created an unique ordering



(a) The reference element for the 2D case, with the local DOF numbering.



(b) The reference element for the 3D case, with the local DOF numbering.

Figure 2.2: The reference elements for the Nédélec space in both 2D and 3D with the DOF numbering scheme.

of the DOFs. An example can be seen in Figure 2.1. The elements are numbered in the same manner, but additionally also have a multidimensional index, which is a coordinate vector $\vec{p}$, that tells the position of an element relative to the mesh itself, so it gives you the offset in elements of an element relative to the element with smallest position, this can also be seen in Figure 2.1. During the FEM calculations we are going to work with a reference element, which in our case is either the unit square or the unit cube. On it we are also going to have to number the local DOFs, this is given in Figure 2.2. This notion of local numbering does not only apply to the reference element, but also for the elements inside of the mesh, where the local DOFs are simply the ones which lie on this element.

During the FEM calculations we are often required to retrieve the global shape functions associated with an element given by its multidimensional index, in other words: figuring out what the global indices of the local shape functions of an element are. The global DOF indices for an element with multidimensional index $\vec{p}$ living in a mesh with number of vertices per axis given by n_x, n_y, and n_z are given in Equation (2.1) for the 2D case and in Equation (2.2) for 3D, where they are ordered according to the local shape functions.

$$\text{DOFs} = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} \vec{v} \cdot \vec{p} \\ a + n_x - 1 \\ b + n_x \\ b + 1 \end{bmatrix}, \text{ with } \vec{v} = \begin{bmatrix} 1 \\ 2n_x - 1 \end{bmatrix} \tag{2.1}$$

$$\text{DOFs} = \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ i \\ j \\ k \\ l \end{bmatrix} = \begin{bmatrix} \vec{v} \cdot \vec{p} \\ a + n_x - 1 \\ b + n_x \\ b + 1 \\ \vec{v}_z \vec{p}_z + 2n_x n_y - n_x - n_y + \vec{p}_y n_x + \vec{p}_x \\ e + 1 \\ e + n_x \\ e + n_x + 1 \\ a + 3n_x n_y - n_x - n_y \\ i + n_x - 1 \\ j + n_x \\ j + 1 \end{bmatrix}, \text{ with } \vec{v} = \begin{bmatrix} 1 \\ 2n_x - 1 \\ 3n_x n_y - n_x - n_y \end{bmatrix} \tag{2.2}$$

The last thing left to do is to the state the formulae for the basis functions. Here we follow standard FEM methodology and do not evaluate the basis functions on the mesh, but define a reference element on which we evaluate them and then use a transformation in order to transform them to the actual element in the mesh. We define with $\vec{\beta}_i$ the i-th local shape function, $i \in \{0,1,2,3\}$ for 2D and $i \in \{0,1,2,...,11\}$ for 3D, of an element in the mesh, and with $\hat{\vec{\beta}}_i$ the corresponding shape function in the reference element. The formulae for the $\hat{\vec{\beta}}_i$ are readily available, such as in [5], and they are given in Collection 2.1. So we want to evaluate the function $f(\cdot)$, which depends on the shape function $\vec{\beta}_i$ transformed by an arbitrary operator $\mathcal{L}$, i.e., $f(\mathcal{L} \circ \vec{\beta}_i)$, using the reference element shape functions, such that we get:

$$f(\mathcal{L} \circ \vec{\beta}_i) = f(\mathcal{G} \circ (\mathcal{L} \circ \hat{\vec{\beta}}_i)), \tag{2.3}$$

where $\mathcal{G}$ is the operator that we are looking for. In the case of curl-conforming Nédélec space $\mathcal{L}$ will in most cases either be identity or the curl operator, for them we have that the $\mathcal{G}$ will look as follows:

$$f(\vec{\beta}_i) = f(\hat{\vec{\beta}}_i) \text{ and} \tag{2.4}$$

$$f(\nabla \times \vec{\beta}_i) = f(J^{-T}(\nabla \times \hat{\vec{\beta}}_i)), \tag{2.5}$$

so in the case of identity $\mathcal{G}$ is identity and for the curl it is J^{-T}, which is the inverse transpose Jacobian of the affine transformation between the reference element and the actual element.

2.2 Pipeline

The general pipeline we are using is shown in Figure 2.3. We define a solver, to solve a problem for $\vec{u}(\vec{x})$ of the form:

$$F(\vec{u}(\vec{x})) = \vec{g}(\vec{x}) \text{ in } \Omega, \tag{2.6}$$

$$\vec{u}(\vec{x}) \times \vec{n} = 0 \text{ on } \partial\Omega, \tag{2.7}$$

where $F(\cdot)$ defines some functional which determines the type of problem we are solving, $\vec{g}(\cdot)$ is the right hand side (RHS) function, and $\vec{n}$ is the normal vector of the domain Ω.

The first step of the pipeline regards the right hand side function $\vec{g}$, which will be provided to us, in some form, by the PIC code; We will later go into more detail on the exact way in which we represent this RHS. The next step in the pipeline consists of assembling the Galerkin system $A\vec{\mu} = \vec{b}$, where A is defined by $F(\cdot)$, $\vec{b}$ by $\vec{g}$, and $\vec{\mu}$ is the solution vector. During the assembly is where the finite element space comes into play, in our case this is the `NedelecSpace` class, as it knows about how the Galerkin discretization looks like and therefore how the matrix A and the vector $\vec{b}$ need to be constructed. Note that in our case we are doing this matrix-free, so instead of fully expressing the entries of the matrix A we construct a functional which returns the result of the multiplication $A\vec{v}$ for some arbitrary vector $\vec{v}$, as such a multiplication is the only information needed to solve such a system using the Conjugate Gradient (CG) method. When the information about the linear system is created we pass it to the CG solver, which will then solve it for us and returns the solution vector $\vec{\mu}$. Note that the solution vector $\vec{\mu}$ only stores the basis function coefficients of the Nédélec space and in order to retrieve the solution $\vec{u}(\vec{x})$ we

2D

$$\hat{\vec{\beta}}_0 = \begin{bmatrix}1-y\\0\end{bmatrix}, \qquad \hat{\vec{\beta}}_1 = \begin{bmatrix}0\\1-x\end{bmatrix}, \qquad \hat{\vec{\beta}}_2 = \begin{bmatrix}y\\0\end{bmatrix}, \qquad \hat{\vec{\beta}}_3 = \begin{bmatrix}0\\x\end{bmatrix}.$$

3D

$$\hat{\vec{\beta}}_0 = \begin{bmatrix}yz-y-z+1\\0\\0\end{bmatrix}, \qquad \hat{\vec{\beta}}_4 = \begin{bmatrix}0\\0\\xy-x-y+1\end{bmatrix}, \qquad \hat{\vec{\beta}}_8 = \begin{bmatrix}z(1-y)\\0\\0\end{bmatrix},$$

$$\hat{\vec{\beta}}_1 = \begin{bmatrix}0\\xz-x-z+1\\0\end{bmatrix}, \qquad \hat{\vec{\beta}}_5 = \begin{bmatrix}0\\0\\x(1-y)\end{bmatrix}, \qquad \hat{\vec{\beta}}_9 = \begin{bmatrix}0\\z(1-x)\\0\end{bmatrix},$$

$$\hat{\vec{\beta}}_2 = \begin{bmatrix}y(1-z)\\0\\0\end{bmatrix}, \qquad \hat{\vec{\beta}}_6 = \begin{bmatrix}0\\0\\xy\end{bmatrix}, \qquad \hat{\vec{\beta}}_{10} = \begin{bmatrix}yz\\0\\0\end{bmatrix},$$

$$\hat{\vec{\beta}}_3 = \begin{bmatrix}0\\x(1-z)\\0\end{bmatrix}, \qquad \hat{\vec{\beta}}_7 = \begin{bmatrix}0\\0\\y(1-x)\end{bmatrix}, \qquad \hat{\vec{\beta}}_{11} = \begin{bmatrix}0\\xz\\0\end{bmatrix}.$$

Collection 2.1: Formulae for the local shape functions on the reference element of the Nédélec space in both 2D and 3D.

have to reconstruct it, something which is done by the Nédélec space. After all this is done we can pass the solution back to PIC and we are done with the FEM-based solve.

One question we have to answer is how we store the vectors $\vec{b}$ and $\vec{\mu}$. Both of them are 1D vector which store values at the DOF positions and which need to provide functionality for halo exchanges and domain boundaries. Until now `IPPL` only supported the first order Lagrangian space, where the DOF positions coincide with the vertex positions and therefore an `ippl::Field` could be used, which is a data structure containing values at the mesh grid-points, i.e., vertex positions. However, for the Nédélec space the DOFs are edge centered and therefore cannot be stored inside of an `ippl::Field`. To this end we implemented a new class called `FEMVector`, which represents a 1D vector and provides functionality for halo exchanges (currently no boundary operations are implemented).

With this general overview provided, we will now discuss in more detail the different parts of the pipeline.

2.3 FEMVector

In its very core the `FEMVector`, Figure 2.4, is a wrapper around a 1D `Kokkos::View` which stores data of a templated type. So the `FEMVector` has a member called `data` which is a 1D `Kokkos::View` and then some methods to interact with it, like retrieving its size, an entry, or even itself. While this is neither revolutionary, nor particularly useful, its actual usefulness comes from

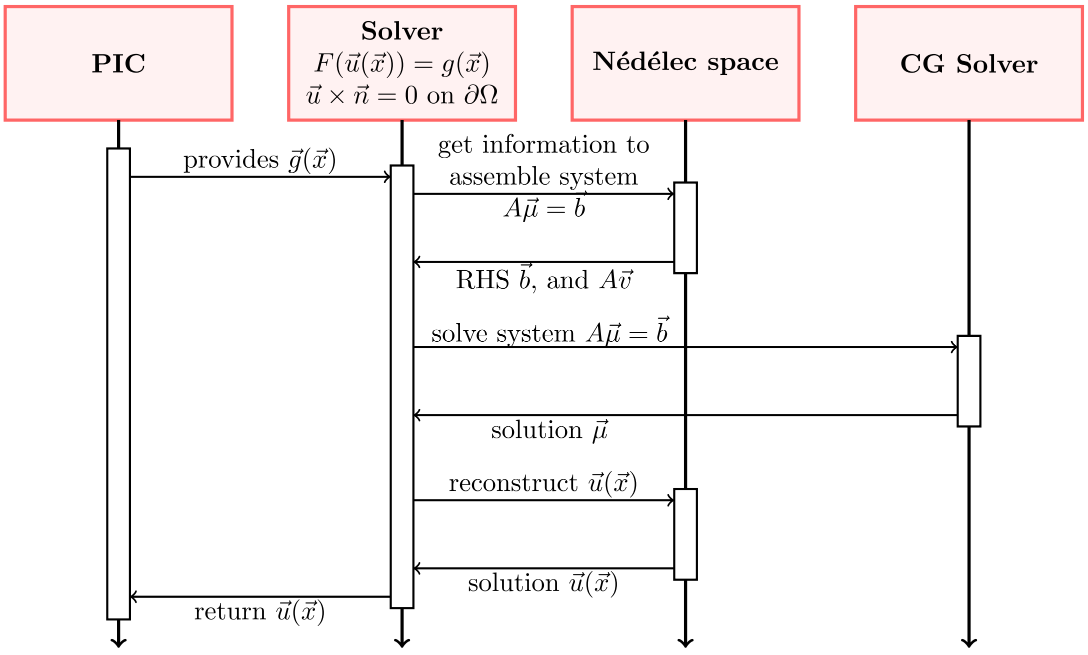

PIC
Solver
$F(\vec{u}(\vec{x})) = g(\vec{x})$
$\vec{u} \times \vec{n} = 0$ on $\partial\Omega$
Nédélec space
CG Solver
provides $\vec{g}(\vec{x})$
get information to assemble system $A\vec{\mu} = \vec{b}$
RHS $\vec{b}$, and $A\vec{v}$
solve system $A\vec{\mu} = \vec{b}$
solution $\vec{\mu}$
reconstruct $\vec{u}(\vec{x})$
solution $\vec{u}(\vec{x})$
return $\vec{u}(\vec{x})$

Figure 2.3: General pipeline diagram, for the solving of a problem using the Nédélec space.

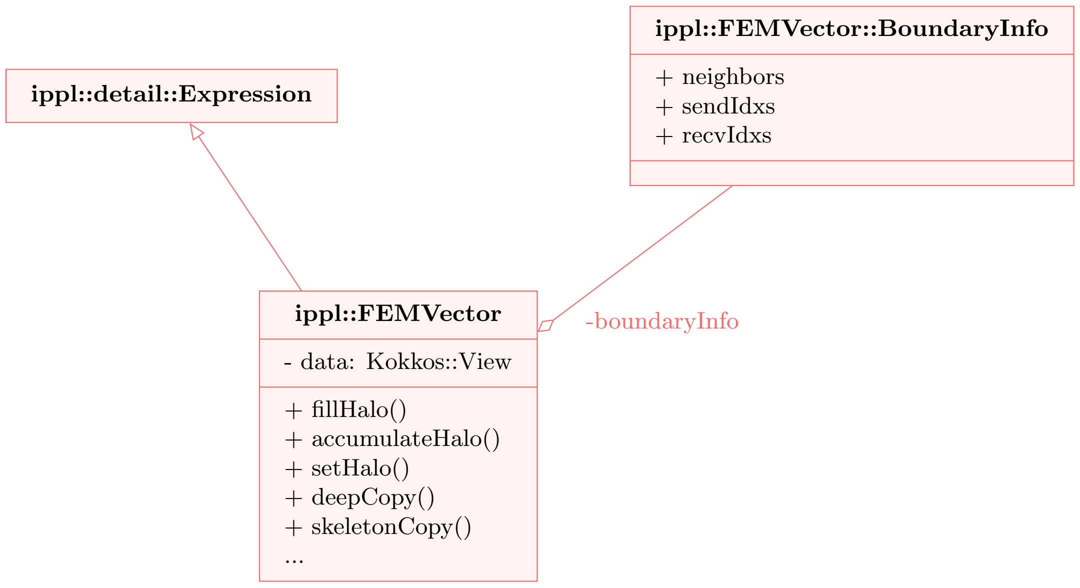


Figure 2.4: UML Class diagram for the `FEMVector`.

its ability to perform halo operations (used during parallelization with MPI). Additionally it also allows for the future implementation of new features, like the handling of domain boundaries.

In order to allow for the halo exchange the `FEMVector` exposes a set of functions to manipulate the halo, namely `fillHalo()`, `accumulateHalo()`, and `setHalo()`, where the first two are equivalent to their `ippl::Field` counter parts and the third one is used to set the value of the halo to what is passed as an argument. The way in which the halo is internally handled is by having three lists, the first one, called `neighbors`, simply stores the MPI ranks of all the other ranks from which information needs to be send or received. The other two lists, called `sendIdxs` and `recvIdxs`, are lists of lists which store for each rank in `neighbors` a list of indices of the underlying `Kokkos::View`, which are part of their halo and we therefore have to send (`sendIdxs`), or which are part of our halo and we therefore have to receive values into (`recvIdxs`). All these lists are passed to the `FEMVector` at construction time. As we are simply working with indices we are able to create an exchange mechanism that is completely independent of the underlying geometry of the problem and by already passing the lists in the constructor we can also have that the `FEMVector` is completely unaware of the geometry it is used upon, which adds flexibility.

These three lists are implemented using a `std::vector<size_t>` for `neighbors` and a `std::vector<Kokkos::View<size_t*>>` for both `sendIdxs` and `recvIdxs`. They are not directly stored inside of the `FEMVector` class, but inside of a helper struct, called `BoundaryInfo`, to which the `FEMVector` owns a pointer. The reason for having this separation between the `FEMVector` with its boundary exchange logic and the `BoundaryInfo` with the data needed to perform the exchange is to have a smaller footprint of the `FEMVector`, allowing for cheaper copying to device, and for the easy creation of `FEMVector`s which do not have any halo information; of course this means that when the entire `FEMVector` is copied to device any information about the boundary is lost. On the other hand we still have that the exchange of data needs to happen via the device, as the entries of the `FEMVector` are stored there, this is the reason why the `sendIdxs` and `recvIdxs` store `Kokkos::View`s, which then are passed (in the form of `Kokkos::View`s) by the CPU to the GPU during the exchange. The actual exchange logic then becomes rather trivial, and consists of looping over the entries in `neighbors` and for each of them copy data, on the device, according to either `sendIdxs` or `recvIdxs` to a buffer which then is sent over the network. For receiving we again loop over the `neighbors`, receive into the buffer and then copy into the `FEMVector` data according to either `sendIdxs` or `recvIdxs`. The logic used to handle the buffer is equivalent to the one utilized by the `ippl::HaloCells` class, which can be found in [6], and will therefore not be discussed in detail.

The last important thing that the `FEMVector` provides are methods for copying, namely a function called `deepCopy()` and one called `skeletonCopy()`. The function `deepCopy()` creates a new `FEMVector` which is identical to the current one and stores the same values. The function `skeletonCopy()` on the other hand will not copy over the values, it creates a new `FEMVector` of same size and boundary info, but it takes a template parameter which sets the value-type of the entries of the new object, allowing for the creation of a `FEMVector` with same structure, but which stores different types of values. This becomes especially useful when dealing with both `FEMVectors` that need to store vectors and `FEMVectors` that need to store scalars.

From all this we can see that the `FEMVector` is a general object, which is independent from any sort of mesh, DOFs, MPI layout, or even FEM itself. Meaning, that it can be utilized in a lot of different situations, from higher order Lagrange, to new finite element spaces, or something completely independent of FEM like general mesh data that needs to support communication, but this flexibility also means that when we use a `FEMVector` we need to create a supporting structure around it, that relates the `FEMVector` to its use case. This structure has to mainly provide two things, the first one is functionality to map between the entries of the `FEMVector` and the DOFs (or arbitrary points) inside of the mesh, and the second one is functionality to create `FEMVector`s, this includes figuring out the correct size and setting-up all the halo exchange information, given some representation of the MPI domain decomposition. So while the `FEMVector` is a general object we still need to custom tailor it to the use-case that we have, which in our case is all part of the Nédélec space.

2.4 Nédélec Space

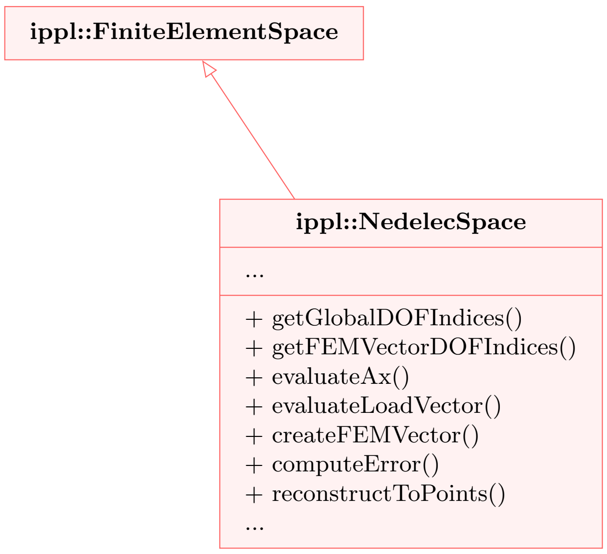


Figure 2.5: UML Class diagram for the Nédélec Space.

The `NedelecSpace` class (UML diagram in Figure 2.5) stores all the information regarding the Nédélec space. This means all the information about the basis functions, from the mapping between global and local index to their formula and their curl. It provides methods for the evaluation of the RHS given a representation of $g(\vec{x})$ and the evaluation of the multiplication $A\vec{v}$, between the Galerkin matrix and an arbitrary vector. It also creates the supporting structure around the `FEMVector`, namely the mapping between the DOF indices and the entries of the `FEMVector`, as well as routines for the creation of `FEMVectors`. Lastly it provides functionality for the calculation of the L2 error, given an analytical form, and the reconstruction of the solution given the Nédélec basis coefficients. All these points are discussed in more detail in the next few sections.

2.4.1 Basis Functions

As already mentioned we store information regarding the mapping from local to global DOFs, this equates to implementing the formulae defined in Equations (2.1) and (2.2). Similarly we also have methods to retrieve the shape function values at arbitrary points inside of the reference element, which is laid out by Collection (2.1). Next to simply retrieving the shape function values we also need the curl of it, defined by $\nabla \times \vec{\beta}_i$ with $\vec{\beta}_i$ the i-th basis function, as this is one of the most common operators needed in curl-conforming Nédélec FEM. In the future different operators can be implemented depending on what the problem to solve requires, as different PDEs require different operators.

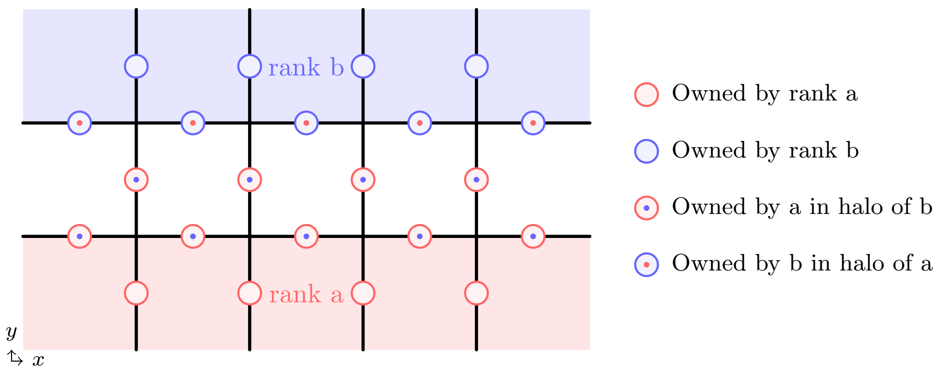


Figure 2.6: Example boundary setup in 2D for the exchange between two ranks. The image displays a north-south exchange, for a west-east exchange the image can simply be rotated clockwise by 90 degree. The colored areas indicate that these elements are completely owned by that rank, the uncolored elements are shared between the two ranks, i.e., both ranks have DOFs on them.

2.4.2 FEMVector

One of the main responsibilities of the `NedelecSpace` class with regards to the `FEMVector` is its creation. The `NedelecSpace` class provides a function, called `createFEMVector()`, which takes as input an MPI rank layout (calculated by `IPPL`) and returns a `FEMVector` for the MPI subdomain of the current rank with correct size and halo communication setup. We therefore need to discuss how the different boundaries with their halo cells are laid out. The layout provided by `IPPL` follows a vertex based approach, so it will provide a range of vertices which dictate the subdomains of each rank, this leads to some complication when having edge based DOFs, but a simple rule can be followed: a rank owns all the DOFs which are part of the elements that are engulfed by the vertex ranges of the layout. For example if the start of subdomain is marked by $(0, 1)$ and the end by $(1, 2)$, it would mean that this rank owns the elements $(0, 1)$, $(1, 1)$, $(0, 2)$, and $(1, 2)$. For a boundary where two MPI ranks meet, we have a hyperplane of elements which are owned by neither of the ranks, but both ranks have a DOF on opposing edges of the elements. There the rank for which this boundary is at the end of its domain will additionally own the DOFs on the edges connecting the two domains with the DOFs owned by the other rank in its halo. The rank for which the boundary is at the beginning of its domain will not get any additional DOFs, but will simply put all the ones the other rank owns into its halo. Naturally if the DOF is part of the halo of one rank (so part of the `recvIdxs`) it needs to be in the `sendIdxs` of the other rank. A visual representation of this for the 2D case can be seen in Figure 2.6, where a north south boundary is displayed, a west east would look identical just rotated 90 degrees clockwise. For 3D the same scheme is used, one simply adds a new dimension with everything else staying the same.

From these boundaries we can see that it appears as if we are growing one hyperplane of elements outwards, so if the layout dictates that the rank has a domain of size $\hat{m}_x \times \hat{m}_y \times \hat{m}_z$ elements it appears like we are actually interacting with a domain of size $\hat{m}_x+2 \times \hat{m}_y+2 \times \hat{m}_z+2 = m_x \times m_y \times m_z$. Note that we are also adding one additional hyperplane of elements on the mesh domain boundary, to stay in line with this definition and to also allow for more complicated boundary conditions (like periodic) in the future. From this we then have that the `FEMvector`

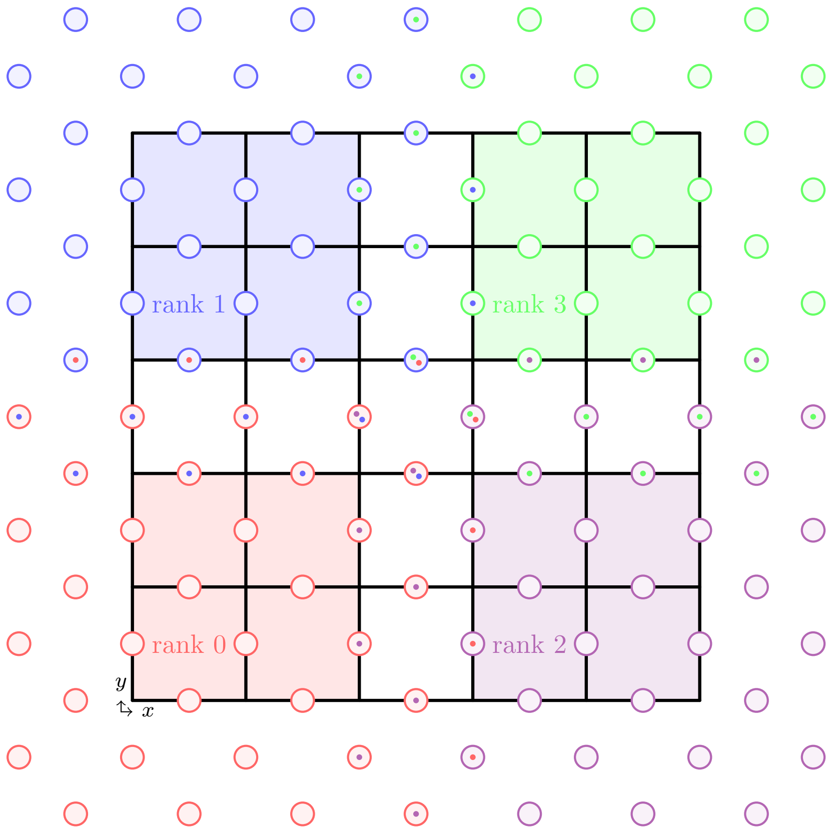


Figure 2.7: Example 2D 6×6 domain decomposition into 4 ranks. The colored areas indicate that these elements are completely owned by that rank, the uncolored elements are shared between different ranks, i.e., multiple ranks have DOFs on them. The small colored dots inside of the degrees of freedom indicate that that rank has this DOF as part of their halo. Note that the `FEMVectors` expand one hyperplane of elements further out than the mesh (black grid), this also means that the element numbering for the global domain and the MPI subdomains have different origins. Also note that we do not make a halo exchange between rank 1 and rank 2.

will store

$$\begin{cases} m_x(m_y-1)+m_y(m_x-1), & \text{for 2D} \\ (m_z-1)(m_x(m_y-1)+m_y(m_x-1)+m_xm_y)+m_x(m_y-1)+m_y(m_x-1), & \text{for 3D} \end{cases} \tag{2.8}$$

elements. An example 2D decomposition for 4 ranks is given in Figure 2.7. Note that we have to pay special attention at points where multiple ranks meet, as it can get somewhat complicated trying to figure out which rank owns what. Another interesting fact is that for this case we would in theory have that rank 1 and rank 2 would exchange values over the diagonal, but in our implementation we are not doing such an exchange. Our test showed that this does not have an impact on the final result or the convergence of the algorithm. A similar thing is observed in 3D, where we do handle the diagonal exchange over edges, but not the diagonal exchange over corners, but again no impact on the error can be observed. We are not entirely sure why this happens and further research is required.

Now that we have established the DOFs which are stored by each `FEMVector` and how the boundary is handled the last thing we need to look at is the order in which these DOFs are stored inside of the `FEMVector` and related to that how we map between the global DOF numbering scheme (as displayed in Figure 2.1) and the entries in the `FEMVector`, i.e., given the global number of a DOF, what is its index in the `FEMVector`. For the ordering we follow the idea laid out in Section 2.1, where we treat each MPI rank independently and do the numbering for its $m_x \times m_y \times m_z$ subdomain, note that when we are on the domain boundary the `FEMVector` is storing DOFs which are not part of the mesh, as the `FEMVector` is unaware of the mesh it will handle them like every other DOF. Then in order to translate between the global number of a DOF and its entry inside of the `FEMVector` we have to go over the element it belongs to, and there its local number (as displayed in Figure 2.2). Suppose we have an element with a global position $\vec{p}$ in the mesh. This element will belong to a subdomain of a rank, where it will have a subdomain relative position of $\bar{\vec{p}}$. Then we can use Equations (2.1) and (2.2), replace $\vec{p}$ by $\bar{\vec{p}}$ and n_x, n_y, n_z by m_x, m_y, m_z. These equations will then return the global DOF indices of all the DOFs related to this element, from them we then want to pick out the one which corresponds to our original DOF, which we do with the local shape function index, as we are working on the same element the local index of the original DOF will be the same as the one of DOF in the `FEMVector`. For example in Figure 2.7 we want to get the `FEMVector` index of the DOF on the south edge (so local shape function index 0) of the element with global position $\vec{p} = (3,1)$. This element lies in subdomain of rank 2 where it has a subdomain specific position of $\bar{\vec{p}} = (1,2)$, note that here we are considering elements for which the `FEMVector` stores DOFs and because the `FEMVector` has DOFs outside of the mesh we have to include those elements (which are not part of the mesh) in our calculations, furthermore rank 2 has dimensions $m_x \times m_y = 5 \times 5$ and from this using Equation (2.1) we figure out that the index of the DOF inside of the `FEMVector` is 19.

This might seem somewhat tedious to have to go over the element in order to translate indices, but in practice this does not lead to any problems, as for the assemble of the RHS, the evaluation of $A\vec{v}$, and the error metric calculations we are following an element based approach, where we loop over the elements and then for each of them do local calculations which we then translate to the global domain, therefore we always will be working with the shape functions of a given element, and it then is quite natural to both retrieve their global indices and their positions inside of the `FEMVector`.

2.4.3 RHS

The implementation of this is given by the function `evaluateLoadVector()`. In Galerkin FEM one generally transforms the problem in a linear system of equations $A\vec{\mu} = \vec{b}$, here we will now discuss how we obtain the RHS $\vec{b}$. The PDEs we are currently solving have a RHS which corresponds to a load term, which means that the I-th entry of the vector $\vec{b}$, of the linear system, is given by:

$$\vec{b}_I = \int_\Omega \vec{\beta}_I \cdot \vec{g}(\vec{x}) d\vec{x}, \tag{2.9}$$

where $\vec{\beta}_I$ is the basis function with global index I and $\vec{g}(\cdot)$ is the RHS function of the PDE we are solving. We implement this in standard FEM fashion using an element oriented assembling scheme, where we loop through all the elements, then for each of them calculate the local contribution of the local shape functions to the global basis indices, where we evaluate the integral using numerical quadrature. For a detailed description of the numerical quadrature used in `IPPL` refer to [3]. As we are currently only supporting zero Dirichlet boundaries in Nédélec FEM we skip all DOFs which are located on the boundary.

In theory the function $\vec{g}(\vec{x})$ will be provided by the PIC loop and how exactly this interface looks like is at this stage of the `IPPL` development not clear yet. Therefore we currently implemented it by passing a `FEMVector` which stores the values of $\vec{g}(\vec{x})$ at the Nédélec DOF positions (center of edges) and we then do a very crude interpolation to the quadrature points using the distance between the edge centers and the quadrature nodes. One caveat that needs to be remembered here is that a `FEMVector` can never exist by itself and that it always needs a supporting structure. In this case of passing the RHS as a `FEMVector` it means we need to define what DOFs the `FEMVector` is storing, especially with regards to MPI ranks, and how they are ordered inside of it (here we are ignoring the halo, as we do not need to make any exchanges). In theory this is defined by the `NedelecSpace` class, but this leads to a problem, because the PIC loop would have to provide this vector and it does not know (and should not know) anything about the Nédélec space. This means that in the future some other way needs to be defined on how the RHS should be provided, one that is independent of the underlying FEM space being used. This is currently ongoing research in `IPPL`, also with regards to how interpolation between particles and mesh should be handled. In our implementation we are circumventing this problem by defining that the `FEMVector` that gets passed to the `evaluateLoadVector()` function needs to have a structure identical ont the one described is Section 2.4.2, except for any of the halo information (so simply the DOFs it stores and the order of this). We are able to do this as we are currently only solving standalone test cases without any PIC for which we manually defined the RHS. Therefore we can use our “human” knowledge to construct `FEMVector`s for these specific cases.

2.4.4 Evaluate $A\vec{v}$

In the previous section we saw how we obtain the RHS $\vec{b}$ of the system $A\vec{\mu} = \vec{b}$, now we discuss how the left hand side (LHS) is calculated. In theory one can explicitly construct the matrix A and then solve the system directly, the problem with this approach is that for larger domain sizes the matrix A can become very large making it unfeasible to store it inside of memory, especially on GPUs. For this reason we are taking a matrix-free approach in which we provide a functional that calculates the matrix vector product $A\vec{v}$, for some arbitrary $\vec{v}$, and combine this with an iterative solver, currently we use CG. Like this a much smaller memory footprint is achieved making the problem solvable even for large domains and higher orders. One part of this functional is the member function `evaluateAx()`[1] that calculates the result of a multiplication between A and a vector $\vec{v}$.

The entry (I, J) for the global indices I and J of the matrix A are given by:

$$A_{I,J} = \int_\Omega \mathcal{F}(\vec{\beta}_I, \vec{\beta}_J) d\vec{x}, \tag{2.10}$$

where $\mathcal{F}(\vec{\beta}_I, \vec{\beta}_J)$ is related to the $F(\vec{u}(\vec{x}))$ of the `Solver` class through its variational formulation and it will be different for each type of problem that we want to solve. In order to obtain the product $A\vec{v} = \vec{s}$ we can say:

$$\vec{s}_I = \sum_J A_{I,J} \vec{v}_J \tag{2.11}$$

In order to implement this in code we take a very similar approach to the calculation of the RHS. We loop through all the elements and then for each element K calculate the local element Matrix $\bar{A}$ which stores the contributions of the local shape functions of the element to the global matrix A:

$$\bar{A}_{i,j} = \int_K \mathcal{F}(\vec{\beta}_i, \vec{\beta}_j) d\vec{x}. \tag{2.12}$$

Here the integral is solved using numerical quadrature and the $\mathcal{F}$ is provided to the `NedelecSpace` class by the `Solver` class through a functor that takes as input the basis function values and their curl at the quadrature points and returns the corresponding value of $\mathcal{F}(\cdot, \cdot)$. This approach allows for the solving of problems with arbitrary $F(\cdot)$, and in the future the parameters which are passed to the functional representation can be expanded to allow for the solving of problems which do not only rely on the basis function value and the curl. After the local element matrix $\bar{A}$ is constructed we use it to update $\vec{s}$ according to Equation (2.11) where we need to map the local DOF indices (i, j) to their global ones (I, J) using the local to global mapping discussed previously. Like with the RHS the DOFs on the boundaries are skipped, due to their zero Dirichlet nature. After all this is done we should have an $\vec{s}$ which corresponds to the product $A\vec{v}$ and we can then take this entire function and pass it to the CG solver as an operator.

2.4.5 Error metric

In order to gauge the accuracy and correctness of our implementation we added an error metric namely the continuous L2 error, implemented by the function `computeError()`. The error that we are solving for is:

$$\epsilon = \int_\Omega |\vec{u}(\vec{x}) - \vec{u}_{sol}(\vec{x})|^2 d\vec{x}, \tag{2.13}$$

where $\vec{u}(\cdot)$ is our numerical solution and $\vec{u}_{sol}(\cdot)$ in an analytical solution. In true FEM fashion we do not compute this directly on a global level, but do it element wise, such that we have:

$$\epsilon = \sum_{K \in \text{elements}} \int_K |\vec{u}(\vec{x}) - \vec{u}_{sol}(\vec{x})|^2 d\vec{x}. \tag{2.14}$$

[1]Note the somewhat different naming convention

As we can see this would require us to have a functional representation of our solution $\vec{u}(\cdot)$, but in practice we have a vector $\vec{\mu}$ which corresponds to the basis function coefficients. In order to overcome this we utilize interpolation, specifically interpolation based on the basis functions. In order to get the value $\vec{u}(\vec{x})$ for some $\vec{x}$ inside of an element K we use:

$$\vec{u}(\vec{x}) = \sum_{i \in \text{local DOFs of } K} \mu_i \vec{\beta}_i(\vec{x}), \tag{2.15}$$

where μ_i is the basis function coefficient of the local shape function i and $\vec{\beta}_i$ is the corresponding shape function. We then use numerical quadrature to evaluate the integral in Equation (2.14) and retrieve the final error. In standard FEM fashion we expect the error to decrease with second order when increasing the number of elements per dimension linearly.

2.4.6 Solution reconstruction

At the end of the day, after the problem has been solved, we need to pass the solution back to PIC, so we need a method for reconstructing the solution $\vec{u}(\vec{x})$ from the basis function coefficients $\vec{\mu}$. Like with the RHS we also have here that the exact interface between PIC and FEM has not been designed yet. What we currently provide is a function called `reconstructToPoints()`, which takes as input a list of points inside of the mesh, stored in a `Kokkos::View`, and then reconstructs the function value of $\vec{u}(\cdot)$ at these points and returns this as also as a `Kokkos::View`. In the future this might be used to directly retrieve the function value at the particle positions.

The implementation of this is rather straight forward and shares similarities to the error metric calculations. The general idea is that we loop through all the points, then for each point $\vec{w}$ figure out to which element K it belongs too and then reconstruct the function value $\vec{u}(\vec{w})$ using the formula of Equation (2.15), with $\vec{x}$ replaced by $\vec{w}$. Note that all this requires that the points passed to the function lie inside of the subdomain of the current MPI rank, which in turn means that every rank gets its unique set of positions. We think that this implementation makes more sense compared to one where a singular global set of positions is passed as each rank represents an individual unit that to a degree should be unaware of the global scheme and should generally only work on its subdomain.

2.5 Solver

Finally we have the `Solver` class, displayed in Figure 2.8, which contains the information for solving a problem of a specific type. This mainly boils down to providing a representation of the variational formulation of the LHS, corresponding to $\mathcal{F}$, and storing the `FEMVector` representations of the RHS $\vec{b}$ and the final solution vector $\vec{\mu}$. Because the $\mathcal{F}$ is unique for each problem that we are solving we have to create an individual class for each of these problem types. It also interacts with the Nédélec space to retrieve the RHS and the functor for the $A\vec{v}$ multiplication, which it then passes to the CG solver. For example we implemented a class `FEMMaxwellDiffusionSolver` which solves a problem of the kind

$$\nabla \times \nabla \times \vec{u}(\vec{x}) + \vec{u}(\vec{x}) = \vec{g}(\vec{x}) \text{ in } \Omega \tag{2.16}$$

$$\vec{u}(\vec{x}) \times \vec{n} = \text{ on } \partial\Omega, \tag{2.17}$$

which we will discussed in more detail in Section 3 as part of our experiments.

Figure 2.3 already provided a simplified overview of how this works. More concrete we pass the RHS $\vec{g}(\cdot)$ evaluated at the Nédélec DOFs to the constructor of the `Solver` class, which will

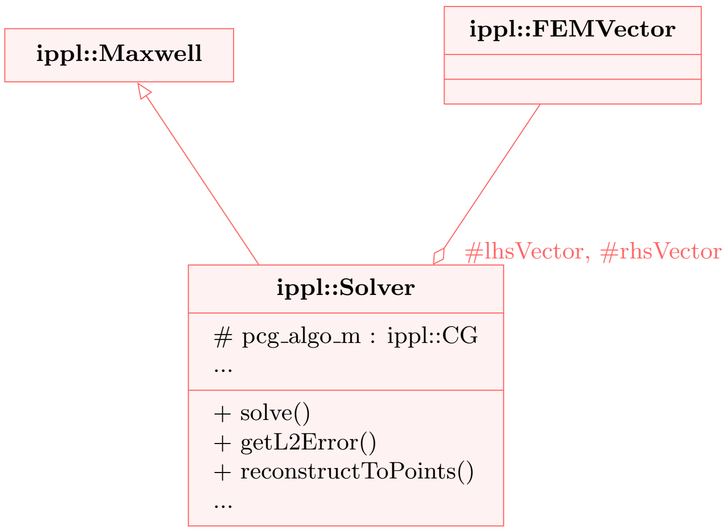


Figure 2.8: UML Class diagram for the `Solver` class.

assemble the RHS using the Nédélec space according to Section 2.4.3 and store this inside of its `rhsVector` member. Then in order to solve the actual problem the function `solve()` is called which will first retrieve the functor for the $A\vec{v}$ evaluation from the Nédélec space according to Section 2.4.4 with the help of its definition of $\mathcal{F}$. After this both the RHS and the functor for $A\vec{v}$ is passed to the CG solver which solves the entire problem and returns the basis function coefficients $\vec{\mu}$ which are then stored in `lhsVector`, completing the solve routine.

After the problem has been solved we normally would pass the solution back to the PIC loop, but as this interface is not yet fully designed we currently do not do anything with the finished result and we simply store the basis function coefficients. But we also expose two functions `getL2Error()` and `reconstructToPoints()` which are wrappers around the `NedelecSpace::computeError()` and `NedelecSpace::reconstructToPoints()` functions of the Nédélec space which can be used during testing in order to gauge the correctness of the implementation.

Chapter 3

Results

3.1 Setup

In this section we will test our implementation on one type of PDE, which is given by:

$$\nabla \times \nabla \times \vec{u}(\vec{x}) + \vec{u}(\vec{x}) = \vec{g}(\vec{x}) \text{ in } \Omega \tag{3.1}$$

$$\vec{u}(\vec{x}) \times \vec{n} = \text{ on } \partial\Omega, \tag{3.2}$$

where $\vec{g}(\vec{x})$ is the source function, $\vec{u}(\vec{x})$ the solution, $\vec{n}$ the domain normal (pointing outwards), and Ω the domain with $\partial\Omega$ being the boundary. For this PDE we define two different source functions $\vec{g}(\vec{x})$ and two different domains Ω. The first case, which we call "Trigonometric" has the following $\vec{g}(\vec{x})$ and Ω:

$$\vec{g}(\vec{x}) = \begin{cases} \begin{bmatrix} (1+k^2)\sin(ky) \\ (1+k^2)\sin(kx) \end{bmatrix} & \text{for 2D, with } \Omega = [1,3]^2 \\ \begin{bmatrix} (1+k^2)\sin(ky)\sin(kz) \\ (1+k^2)\sin(kx)\sin(kz) \\ (1+k^2)\sin(kx)\sin(ky) \end{bmatrix} & \text{for 3D, with } \Omega = [1,3]^3 \end{cases} . \tag{3.3}$$

The second case, which we call "Polynomial", has the following $\vec{g}(\vec{x})$ and Ω:

$$\vec{g}(\vec{x}) = \begin{cases} \begin{bmatrix} 2-(y^2-1) \\ 2-(x^2-1) \end{bmatrix} & \text{for 2D, with } \Omega = [-1,1]^2 \\ \begin{bmatrix} -2(z^2-1)-2(y^2-1)+(y^2-1)(z^2-1) \\ -2(x^2-1)-2(z^2-1)+(x^2-1)(z^2-1) \\ -2(y^2-1)-2(x^2-1)+(x^2-1)(y^2-1) \end{bmatrix} & \text{for 3D, with } \Omega = [-1,1]^3 \end{cases} . \tag{3.4}$$

We then have that the exact solution $\vec{u}(\vec{x})$ for the Trigonometric problem is:

$$\vec{u}(\vec{x}) = \begin{cases} \begin{bmatrix} \sin(ky) \\ \sin(kx) \end{bmatrix} & \text{for 2D} \\ \begin{bmatrix} \sin(ky)\sin(kz) \\ \sin(kx)\sin(kz) \\ \sin(kx)\sin(ky) \end{bmatrix} & \text{for 3D} \end{cases}, \tag{3.5}$$

and for the Polynomial one it is:

$$\vec{u}(\vec{x}) = \begin{cases} \begin{bmatrix} -(y^2-1) \\ -(x^2-1) \end{bmatrix} & \text{for 2D} \\ \begin{bmatrix} (y^2-1)(z^2-1) \\ (x^2-1)(z^2-1) \\ (x^2-1)(y^2-1) \end{bmatrix} & \text{for 3D} \end{cases}. \tag{3.6}$$

In order to solve the problem described by Equations (3.1) and (3.2) using FEM we need to find its variational formulation. Luckily this is a very common type of problem and therefore this has already be done for us in [5]. The variational formulation is given by:

$$\int_\Omega \nabla \times \vec{u}(\vec{x}) \cdot \nabla \times \vec{v}(\vec{x}) d\vec{x} + \int_\Omega \vec{u}(\vec{x}) \cdot \vec{v}(\vec{x}) d\vec{x} = \int_\Omega \vec{u}(\vec{x}) \cdot \vec{g}(\vec{x}) d\vec{x}, \tag{3.7}$$

where we enforce the zero dirichlet boundary by skipping DOFs located on the boundary. From this we now also can quite easily extract the $\mathcal{F}(\cdot,\cdot)$ of Equation (2.10):

$$\mathcal{F}\left(\vec{u}(\vec{x}), \vec{v}(\vec{x})\right) = \nabla \times \vec{u}(\vec{x}) \cdot \nabla \times \vec{v}(\vec{x}) + \vec{u}(\vec{x}) \cdot \vec{v}(\vec{x}). \tag{3.8}$$

During the element-based assembling scheme we are going to assemble the local matrices for the LHS and local parts of the vector for the RHS for every element in the mesh. In order to simplify calculations we are not going to work on the actual elements of the mesh, but rather on the reference element. In Section 2.1 we already introduced the transformations required, our problem transforms into:

$$A_{i,j} = \int_K \nabla \times \vec{\beta}_i \cdot \nabla \times \beta_j d\vec{x} + \int_K \vec{\beta}_i \cdot \vec{\beta}_j d\vec{x} = \int_{\hat{K}} J^{-T}(\nabla \times \hat{\vec{\beta}}_i) \cdot J^{-T}(\nabla \times \hat{\vec{\beta}}_j) d\vec{x} + \int_{\hat{K}} \hat{\vec{\beta}}_i \cdot \hat{\vec{\beta}}_j d\vec{x}, \tag{3.9}$$

$$s_i = \int_K \vec{\beta}_i \cdot \vec{g}(\vec{x}) d\vec{x} = \int_{\hat{K}} \hat{\vec{\beta}}_i \cdot \vec{g}(\vec{x}) d\vec{x}, \tag{3.10}$$

where J^{-T} is the inverse transpose Jacobian of the affine transformation between the reference and the actual element.

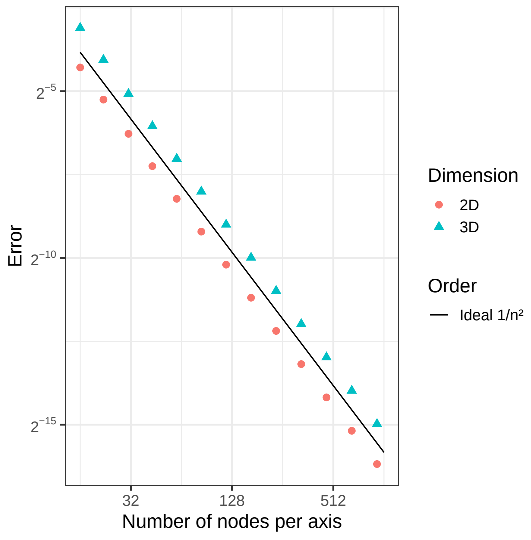
Trigonometric problem

	2D		3D	
Num nodes	L2 Error	Iterations	L2 Error	Iterations
16	$5.125 \cdot 10^{-2}$	13	$1.168 \cdot 10^{-1}$	5
22	$2.626 \cdot 10^{-2}$	19	$6.013 \cdot 10^{-2}$	8
31	$1.290 \cdot 10^{-2}$	29	$2.961 \cdot 10^{-2}$	15
43	$6.586 \cdot 10^{-3}$	42	$1.514 \cdot 10^{-2}$	22
60	$3.339 \cdot 10^{-3}$	61	$7.682 \cdot 10^{-3}$	35
84	$1.688 \cdot 10^{-3}$	89	$3.884 \cdot 10^{-3}$	54
118	$8.495 \cdot 10^{-4}$	165	$1.955 \cdot 10^{-3}$	93
166	$4.272 \cdot 10^{-4}$	248	$9.833 \cdot 10^{-4}$	233
234	$2.142 \cdot 10^{-4}$	335	$4.932 \cdot 10^{-4}$	343
330	$1.074 \cdot 10^{-4}$	1026	$2.474 \cdot 10^{-4}$	994
466	$5.379 \cdot 10^{-5}$	1609	$1.238 \cdot 10^{-4}$	2321
659	$2.686 \cdot 10^{-5}$	2963	$6.184 \cdot 10^{-5}$	3883
931	$1.345 \cdot 10^{-5}$	4261	$3.096 \cdot 10^{-5}$	6185

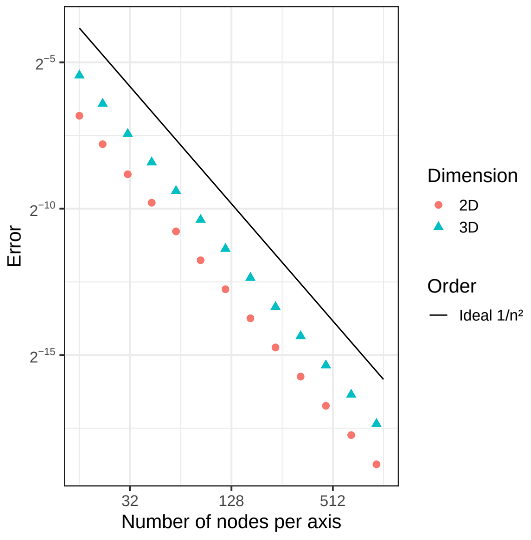
Polynomial problem

	2D		3D	
Num nodes	L2 Error	Iterations	L2 Error	Iterations
16	$8.824 \cdot 10^{-3}$	7	$2.293 \cdot 10^{-2}$	23
22	$4.502 \cdot 10^{-3}$	12	$1.171 \cdot 10^{-2}$	36
31	$2.206 \cdot 10^{-3}$	19	$5.741 \cdot 10^{-3}$	56
43	$1.125 \cdot 10^{-3}$	29	$2.930 \cdot 10^{-3}$	81
60	$5.703 \cdot 10^{-4}$	43	$1.485 \cdot 10^{-3}$	119
84	$2.882 \cdot 10^{-4}$	62	$7.503 \cdot 10^{-4}$	170
118	$1.450 \cdot 10^{-4}$	90	$3.776 \cdot 10^{-4}$	267
166	$7.291 \cdot 10^{-5}$	185	$1.899 \cdot 10^{-4}$	382
234	$3.657 \cdot 10^{-5}$	178	$9.522 \cdot 10^{-5}$	640
330	$1.834 \cdot 10^{-5}$	483	$4.776 \cdot 10^{-5}$	1039
466	$9.181 \cdot 10^{-6}$	1113	$2.391 \cdot 10^{-5}$	2086
659	$4.585 \cdot 10^{-6}$	2220	$1.194 \cdot 10^{-5}$	3592
931	$2.295 \cdot 10^{-6}$	3600	$5.977 \cdot 10^{-6}$	5804

Table 3.1: Tables with exact values of the convergence studies. Additionally the number of CG iterations is also given.



(a) Trigonometric case.



(b) Polynomial case.

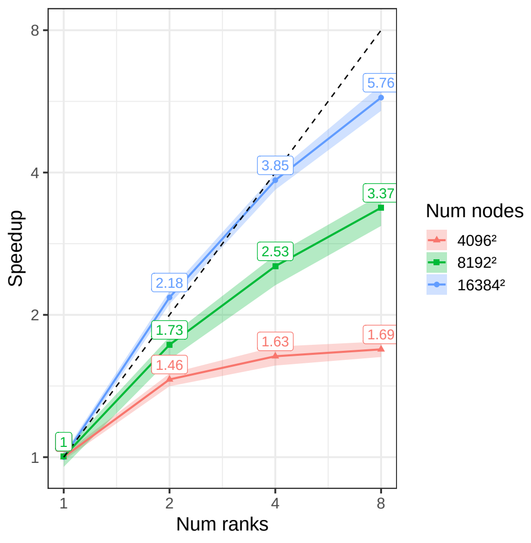
Figure 3.1: L2 error convergences for the different problems. As we can see, perfect order 2 convergence is achieved.

3.2 Correctness

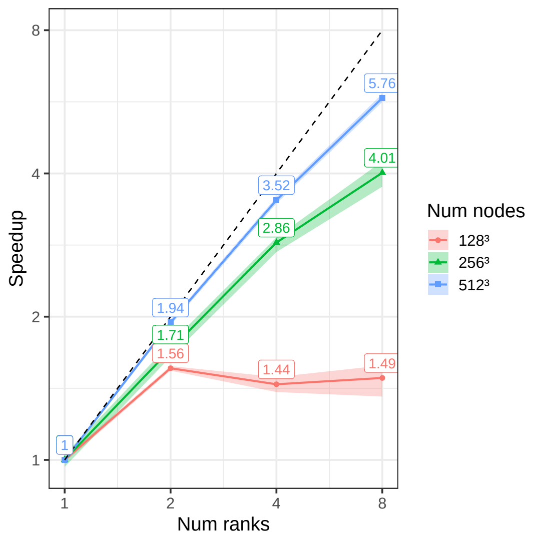
In order to gauge the correctness of the implementation a convergence study is done. To this extend the problems are solved multiple times while increasing the mesh resolution, for each case the L2 error as described in Section 2.4.5 is calculated and the result is plotted. For these experiments we went with number of nodes per dimension in the range of 16 to 931, a Gauss Jacobi quadrature using 5 nodes per dimension for the numerical quadrature is used, and the linear system is solved using a non-preconditioned conjugate gradient algorithm with tolerance set to 10^{-13} and maximum number of iterations set to 10000. The plots resulting from this can be seen in Figure 3.1 with exact values given in Table 3.1. From theory we would expect the error to converge with a rate of $\Theta(n^{-2})$ where n is the number of vertices per dimension. As we can see in the plots we have this convergence, as expected.

3.3 Scaling Analysis

We also perform a small scaling study of our implementation in the form of a strong and weak scaling analysis. The hardware setup we used for the runs consists of a two node setup, each equipped with a dual socket 128 core `AMD EPYC 7773X` and 1TB of memory, and four `NVIDIA A100`s with 80GB of memory each. For the network a 200Gb/s Infiniband connection is used, and the MPI is a cuda aware `OpenMPI 5.0.3` instance, the operating system is `Rocky Linux 9.4`. For the runs we only consider the Polynomial case, as the performance between the two cases should be the same. It was configured identically to the previous section, with the exception that for both strong and weak scaling the maximum number of CG iterations are set to 100 and the tolerance is set to a negative value, like this we have that the number of CG iterations stays constant over the different domain sizes and therefore the work required is directly related to

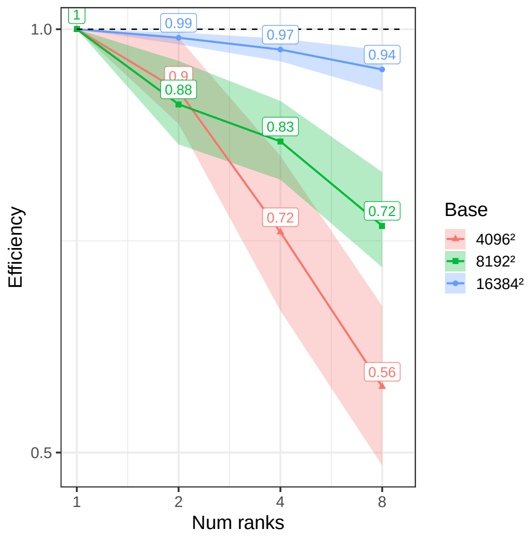


(a) 2D case

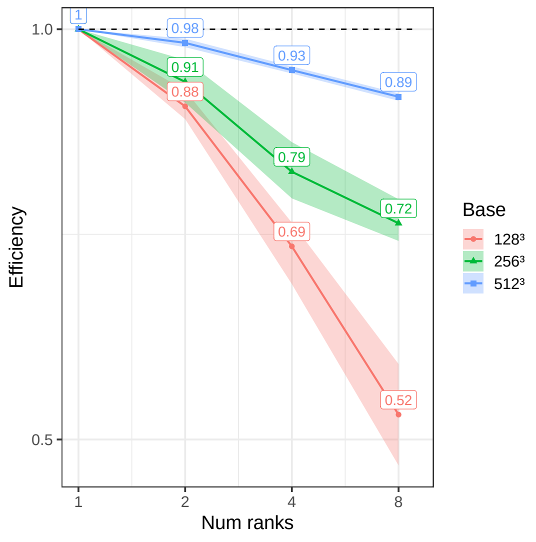


(b) 3D case

Figure 3.2: Strong scaling of the implementation. All runs where repeated 10 times with an additional warm-up run, mean and confidence intervals are plotted.



(a) 2D case



(b) 3D case

Figure 3.3: Weak scaling of the implementation. All runs where repeated 10 times with an additional warm-up run, mean and confidence intervals are plotted.

Num nodes	Time [s]
4096	0.6828832287
8192	2.4991507782
16384	11.3711869943

(a) 2D case

Num nodes	Time [s]
128	0.7920425589
256	5.7699309643
512	45.0464857090

(b) 3D case

Table 3.2: Runtimes for for the scaling experiments in the case of a singular rank. The mean over the runs is calculated. Note that here in the case of a singular rank it does not matter if we consider the strong or the weak scaling case.

the domain size (which is critical for the weak scaling). All runs where repeated 10 times with an additional warm-up run for which then mean and confidence intervals are calculated. For strong scaling in 2D we went with problem sizes of 4096^2, 8192^2, and 16384^2 which where run with 1, 2, 4, and 8 ranks (where only for the case of 8 ranks we move to two nodes), this can be seen in Figure 3.2(a), with base runtimes in Table 3.2a. For the strong scaling in 3D we went with problem sizes of 128^3, 256^3, and 512^3 with same range of ranks, this can be seen in Figure 3.2(b), with base runtimes in Table 3.2b. For the weak scaling in 2D the base sizes where 4096^2, 8192^2, and 16384^2, they where scaled with the number of ranks p with the formula $\sqrt{p}\, n_{base}$, where n_{base} is one of the base sizes. The number of processes was again in the range of 1 to 8, the results of this can be seen in Figure 3.3(a), with the base runtimes given in Table 3.2a. For the weak scaling in 3D the base sizes where 128^3, 256^3, and 512^3, the update formula for more processes is given by $\sqrt[3]{p}\, n_{base}$, with same number of processes as in all other experiments, this can be seen in Figure 3.3(b), with base runtimes given in Table 3.2b. Generally for all the experiments here we were not able to go to any larger problem sizes due to having not enough memory on the GPUs.

From the plots we get a somewhat mixed result, we can unsurprisingly see that for smaller problem sizes both the strong and weak scaling perform bad, for example in the strong scaling 3D case we have that for a problem size of 128 no more scaling can be seen for more than 2 ranks. If we move to larger problem sizes the scaling tends to get better, but they are still not satisfactory. For example for the largest problem size in 3D we already drop to around 90% parallel efficiency at 8 ranks, for 2D we are able to hold ourselves better and remain at 94%. For the speed-up in both 2D and 3D for the largest problem size and 8 ranks we already drop to 72% of the optimal speed-up. And all this is for the case of only 8 GPUs, in theory we would like to go to much more. So in theory the larger the problem size the better, but here we then start to run into a different issue: memory. We cannot go to larger problem sizes, because they simply stop fitting into GPU memory. For example in the 3D case with problem size of 1024^3 we have that already only the representation of the RHS $\vec{g}(\vec{x})$ at the DOFs using double precision requires around 77GB of memory, which barely fits into a GPU. So we have the problem of needing to parallelize because we do not have enough memory, but at the point at which we have to parallelize the program does not yet scale well and we would actually like to go larger. Trying to decrease the memory footprint would be rather difficult, as at one point we have to store the `FEMVectors` on device. One could maybe try to do something with regard to storing the entire data in CPU and then only moving it part-wise to GPU, this would help with the amount of memory that is needed on device, but would introduce more complexity and overhead every time data needs to be copied to device and back. Another approach would be to try to decrease the overhead generated by the parallelization, which in turn would lead to better scaling behaviour for all problem sizes and we therefore do not run into the issue of the scaling behaviour being bad for the sizes at which we parallelize. We therefore can see that still some work could be done in this area.

Chapter 4

Conclusion

The initial goal that we had was to implement the Nédélec space in `IPPL`, based on the method we laid out in Section 2 and the experimental results we presented in Section 3 we are rather confident that we succeeded in our initial goal, but there are still more things that could be done. For example right now we only support zero Dirichlet boundary conditions, here some more work can be done to expand to more types of boundaries such as non-zero Dirichlet or periodic, all this would entail expanding the functionalities of the `FEMVector`. Also the Nédélec space needs to be better integrated into the PIC loop, for both how information is passed from PIC to FEM and the other way around. Here some uniform interface could be designed, which is independent of the finite element space used. For the passing of information from PIC to FEM, the function values could be provided at the quadrature nodes such that no additional interpolation has to be done, this would lead to a dependence on the quadrature rule being used by the FEM space, but this seems like much less of a problem compared to a dependence on the finite element space and the DOFs within it. For the exchange of information from FEM to PIC we would say that our current setup of being able to evaluate the solution function at arbitrary points is already pretty good, as we could have that PIC provides the location of all the particles and FEM then simply returns the solution for each of those particles. One could also think about implementing higher order Nédélec spaces. One other interesting thing that should be looked into is that we are not exchanging values over the corners in the halo communication, as seen in Figure 2.7. In our test we did not see any difference in the resulting error between cases where the domain was decomposed in a way where subdomain corners exists where such corner values would have to be exchanged, and decompositions where we do not have such subdomain corners. It also is quite interesting that this phenomenon exists for both 2D and 3D, but here we have that the singular corner values which we do not exchange for the 2D case become a "line" for the 3D case where we have to exchange them, so the number of values go from $\Theta(1)$ to $\Theta(n)$. This leads us to believe that a reason behind why the values do not need to be exchange might be because there are only constant many of them and we are only missing a halo exchange therefore they are somewhat close to correct and then CG is able to take care of this, but this only makes limited sense, as FEM generally is sensitive to error in a singular DOF and we should loose the second order convergence. We are therefore not entirely sure what the reason behind this is and therefore more research in this area would be required. Based on all this we see that there is still more work that can be done, but we were able to lay the foundation of the Nédélec space.

While our implementation of the Nédélec space laid the foundation for more possibilities regarding it, we also have that the `FEMVector` laid the foundation for more development itself. Before we were limited to DOFs on the vertices of the mesh, which meant we only could use the first order Lagrange space, but with the development of the `FEMVector` we are now able to have DOFs at arbitrary points of the mesh, which makes it possible to implement higher order spaces and entirely new ones. For example the Lagrange space can now be expanded from only supporting first order to supporting second or third order. So while the `FEMVector` might not be the flashiest new thing we still think that it can play an important role in the future development of FEM inside of `IPPL`.

Bibliography

[1] S. Muralikrishnan, M. Frey, A. Vinciguerra, M. Ligotino, A. J. Cerfon, M. Stoyanov, R. Gayatri, and A. Adelmann, "Scaling and performance portability of the particle-in-cell scheme for plasma physics applications through mini-apps targeting exascale architectures," in *Proceedings of the 2024 SIAM Conference on Parallel Processing for Scientific Computing (PP)*, pp. 26–38, SIAM, 2024.

[2] J. N. Reddy, *Introduction to the Finite Element Method.* New York: McGraw-Hill Education, 4th edition ed., 2019.

[3] L. Bühler, "Building blocks for finite element computations in ippl." Available at `https://amas.web.psi.ch/people/aadelmann/ETH-Accel-Lecture-1/projectscompleted/phys/bachelor_thesis_buehlluk.pdf` (2025/06/03), 2023. Bachelor Thesis ETH Zürich.

[4] J.-C. Nédélec, "Mixed finite elements in $\mathbb{R}^3$," *Numerische Mathematik*, vol. 35, no. 3, pp. 315–341, 1980.

[5] P. Monk, *Finite Element Methods for Maxwell's Equations.* Oxford University Press, 04 2003.

[6] M. Frey, A. Vinciguerra, S. Muralikrishnan, Sonali, vmontanaro, Mohsen, A. Adelmann, manuel5975p, and F. Schurk, "Ippl-framework/ippl: Ippl-3.2.0," Mar. 2024.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. **In consultation with the supervisor**, one of the following two options must be selected:

[✓] I hereby declare that I authored the work in question independently, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

☐ I hereby declare that I authored the work in question independently. In doing so I only used the authorised aids, which included suggestions from the supervisor regarding language and content and generative artificial intelligence technologies. The use of the latter and the respective source declarations proceeded in consultation with the supervisor.

Title of paper or thesis:

Nédélec Space in IPPL

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Pietak	Alexander

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guidelines.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Zürich, 11.07.2025	[illegible]

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] For further information please consult the ETH Zurich websites, e.g. https://ethz.ch/en/the-eth-zurich/education/ai-in-education.html and https://library.ethz.ch/en/researching-and-publishing/scientific-writing-at-eth-zurich.html (subject to change).