Constructing Generators of Cohomology Classes on Surfaces

Josua Rieder

October 2022

Contents

Chapter One

Introduction

1.1 Motivation

Given a bounded Lipschitz-polyhedron $\Omega_C \subseteq \mathbb{R}^3$ whose surface $\partial\Omega_C$ is endowed with a polygonal discretization Γ_h, we seek to compute a minimal generating set of surface edge cycles that are independent in $H_1(\Gamma_h, \mathbb{Z})$. For problems with trivial boundary conditions, an algorithm that terminates in $\mathcal{O}(|E|)$ steps is known where $|E|$ is the number of edges of Γ_h [1]. If, additionally, electric and magnetic constraints are imposed on some surface elements as in [2], new algorithms are necessary to, one, ensure that the boundary conditions don't interfere with the topological cycles and, two, to find the remaining generators corresponding to the electric and magnetic ports, which is what we aim to offer in this work.

1.2 Terminology and Notation

As our work is largely a discretized, computational continuation of the theoretical groundwork laid out in [2], we use the terms *field domain* Ω, *circuit domain* Ω_C, *ports* Γ_E^i, Γ_M^i, *cycles* γ as well as the notion of σ*-relativity* exactly as defined therein.

A *walk* on a graph $G = (V, E)$ is a sequence of vertices $w = (v_1, \ldots, v_n)$ where $v_i \in V$ and $\{v_i, v_{i+1}\} \in E$. We interpret all walks to be open per default, i.e. we don't treat $\{v_n, v_1\}$ to be among the edges belonging to the walk (and as such it may not even necessarily be an element of E). Whenever we want to construct a *loop*, we always include the first vertex as the last: $v_1 = v_n$.

A *path* is a walk where the vertices (bar the potential wraparound $v_1 = v_n$) are unique. The distinction is not reflected in notation nor code.

An *edge function* is any function defined on the edge set E of a graph. In the context of this thesis, we only work with edge functions mapping into the integers: $f : E \to \mathbb{Z}$. From a programming perspective, such functions are implemented as contiguous, dynamic array of integers and are thus called

edge vectors. The algorithm for how to convert walks into edge vectors is not presented as it is fairly trivial; the only point of caution is that the internal orientation of the edges has to be respected in this conversion.

For tuples, we let the absolute value notation $|\cdot|$ refer to their length and the subscript notation ${}_k$ to their k-th element.

For graphs G, we always allow denoting their vertex and edge set as V_G and E_G respectively, even if G was not previously defined as $G := (V_G, E_G)$.

We employ the subroutine BFS (initialism of breadth-first search) in numerous listings. It accepts as arguments the graph G, the set of starting points $\mathcal{S}$ and the set of end points $\mathcal{E}$ and returns a tuple containing the vertices of a shortest path found from $\mathcal{S}$ to $\mathcal{E}$: $\mathrm{BFS}(G, \mathcal{S} \subseteq V_G, \mathcal{E} \subseteq V_G) \to \bigcup_{i=0}^{\infty} V_G^i$. In case no path from $\mathcal{S}$ to $\mathcal{E}$ was found, it returns the empty tuple ().

We also employ the subroutine MST (initialism of minimum spanning tree). It accepts as arguments the connected graph G and optionally a weight function w and returns a subgraph of G: $\mathrm{MST}(G, w : E_G \to \mathbb{R}) \to (V_G, E_G^* \subseteq E_G)$.

Chapter Two

Problem Formulation

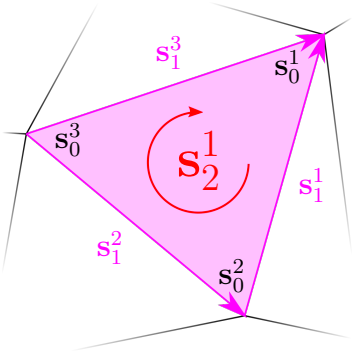
2.1 Setting

Let $\mathcal{S}_l$ denote the set of l-primitives for $l = 0, 1, 2$. Note that we don't restrict ourselves to l-simplices so as to not exclude quadrilateral discretizations which are commonly used in the context of the finite element method. The relationship between primitives of adjacent dimensions is captured by the oriented incidence relations $\iota^l_{l-1} : \mathcal{S}_{l-1} \times \mathcal{S}_l \to \{-1, 0, 1\}$. Every positive-dimensional primitive is thought of as possessing an *interior orientation* that manifests through its incidence relation with its substrate: $\iota(\mathbf{x}, \mathbf{y}) = \pm 1$, if the induced orientation of $\mathbf{x}$ with respect to $\mathbf{y}$ concurs/disagrees with the interior orientation of $\mathbf{x}$; $\iota(\mathbf{x}, \mathbf{y}) = 0$, if $\mathbf{x}$ is not contained in $\mathbf{y}$.

Analogously to [2], we denote the circuit domain *surface elements* as:

$$G := \left\{ \overline{\Gamma^1_E}, \ldots, \overline{\Gamma^{N_E}_E}, \overline{\Gamma^1_M}, \ldots, \overline{\Gamma^{N_M}_M}, \overline{\Gamma_I} \right\} \tag{2.1}$$

The *port function* $p : \mathcal{S}_2 \to G$ is the discrete counterpart of the continuous boundary conditions as described in [2] and serves as the third and final input.



$$\begin{aligned}
\iota^2_1(\mathbf{s}^1_1, \mathbf{s}^1_2) &= -1 & \iota^1_0(\mathbf{s}^1_0, \mathbf{s}^2_1) &= 0 \\
\iota^2_1(\mathbf{s}^2_1, \mathbf{s}^1_2) &= -1 & \iota^1_0(\mathbf{s}^2_0, \mathbf{s}^2_1) &= 1 \\
\iota^2_1(\mathbf{s}^3_1, \mathbf{s}^1_2) &= 1 & \iota^1_0(\mathbf{s}^3_0, \mathbf{s}^2_1) &= -1 \\
\\
\iota^1_0(\mathbf{s}^1_0, \mathbf{s}^1_1) &= 1 & \iota^1_0(\mathbf{s}^1_0, \mathbf{s}^3_1) &= 1 \\
\iota^1_0(\mathbf{s}^2_0, \mathbf{s}^1_1) &= -1 & \iota^1_0(\mathbf{s}^2_0, \mathbf{s}^3_1) &= 0 \\
\iota^1_0(\mathbf{s}^3_0, \mathbf{s}^1_1) &= 0 & \iota^1_0(\mathbf{s}^3_0, \mathbf{s}^3_1) &= -1
\end{aligned}$$

Figure 2.1: Ex. of an $\mathcal{S}_l$ and ι

2.2 Input Constraints

A number of constraints apply to the input:

1. The mesh defined by $\mathcal{S}_l$ is the surface of a Lipschitz-polyhedron. In particular, this implies that the mesh is connected and orientable.
2. Every port is *topologically trivial*, i.e. simply connected.
3. Every vertex belongs to at most one port.

Compare also assumption 1 in [2].

2.3 Objectives

Using the port function p defined above, we introduce the notion of *excluded* mesh elements:

- A face f is an *excluded face* iff it is affiliated with a port: $p(f) \neq \overline{\Gamma_I}$
- An edge e is an *excluded edge* iff at least one of its bordering faces belongs to an electric port or both bordering faces belong to a magnetic port.
- A vertex v is an *excluded vertex* iff it belongs to an electric port or is in the interior of a magnetic port.[1]

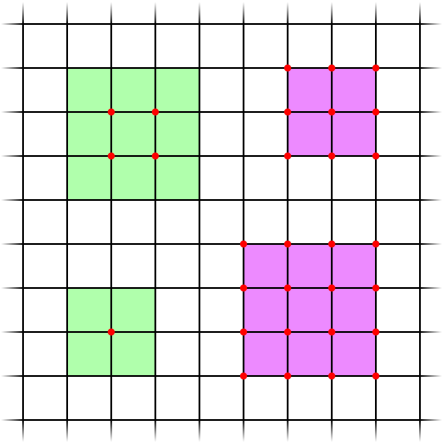


Figure 2.2: Excluded vertices in red

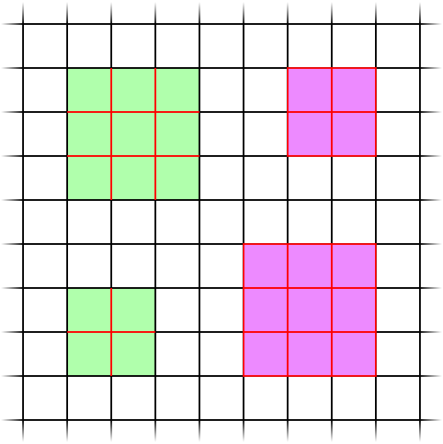


Figure 2.3: Excluded edges in red

Our objective is to find a minimal set of fundamental cycles $\gamma_1, \ldots, \gamma_N$ non-bounding relative to $\partial\Gamma_E$ which is thus restricted to the non-excluded edges. As discussed in section 3.2 of [2], such cycles fall into one of three classes:

[1] In the graph-theoretic setting, interiorness is verified by checking whether any neighboring vertex is unaffiliated with any port.

1. Topological cycles, of which there are $N_T = 2\beta_1(\Omega_C)$ where β_1 is the first Betti number and Ω_C is the circuit domain. As such, each one-dimensional hole in the circuit domain corresponds to a pair of cycles.
2. Electric connector cycles, of which there are $\max(N_E - 1, 0)$ in the case of a single connected circuit domain.
3. Magnetic port cycles, of which there are $\max(N_M - 1, 0)$ in the case of a single connected circuit domain.

These N cycles may then be used to obtain the discrete counterpart of the corresponding tangential cohomology vector fields $\mathbf{c}_1, \ldots, \mathbf{c}_N$ satisfying

$$\int_{\gamma_j} \mathbf{c}_m \cdot \mathrm{d}s = \begin{cases} 1 & m = j, \\ 0 & \text{else}, \end{cases} \quad j, m \in \{1, \ldots, N\} \tag{2.2}$$

2.4 Secondary Objectives

As a secondary objective, we desire cycles that are as short as possible. Shortness here refers to the number of contained edges, not distance in a metric space. The benefit of short cycles lies within superior numerical stability during the subsequent FEM step of the electromagnetic simulation.

2.5 Complexity Analysis

To facilitate the inquiry into our algorithms' complexity, it is beneficial to provide some (in-)equalities that we can draw on later.

First, we assume the maximum polygonal *arity*, the maximum number of edges belonging to a single polygon, to be constant:

$$a_{\max} = \max_{f \in \mathcal{S}_2} \sum_{e \in \mathcal{S}_1} \left| \iota_1^2 (l, f) \right| \tag{2.3}$$

This assumption is justified because, in practice, the maximum arity is a design choice not contingent on the fineness of the mesh.

Secondly, it can be observed that $\mathcal{O}(|E_{\mathcal{V}}|) = \mathcal{O}(|E_{\mathcal{F}}|)$ because the mesh is the surface of a bounded domain.

Lastly, for polyhedra with constant genus g and thus constant Euler characteristic $\chi = 2 - 2g$, we can derive the following with the help of Euler's polyhedron formula $\chi = V - E + F$:

$$\mathcal{O}(|V_{\mathcal{V}}| + |V_{\mathcal{F}}|) = \mathcal{O}(|E_{\mathcal{V}}| + \chi) = \mathcal{O}(|E_{\mathcal{V}}|) \tag{2.4}$$

$$\mathcal{O}(|V_{\mathcal{V}}|) \subseteq \mathcal{O}(|E_{\mathcal{V}}|) \tag{2.5}$$

$$\mathcal{O}(|V_{\mathcal{F}}|) \subseteq \mathcal{O}(|E_{\mathcal{V}}|) \tag{2.6}$$

Chapter Three

Proposed Algorithms

3.1 Precomputations

For given $\mathcal{S}_l$, ι, p, collectively referred to as the *mesh*, we populate a small number of data structures. All further computations are performed on these three data structures; they fully capture the problem input.

1. $\mathcal{V} = (V_\mathcal{V}, E_\mathcal{V})$: an undirected graph whose vertices are the mesh's **vertices** ($\mathcal{S}_0$) and whose edges are the mesh's **edges** ($\mathcal{S}_1$), called the *vertex graph*.
2. $\mathcal{F} = (V_\mathcal{F}, E_\mathcal{F})$: an undirected graph whose vertices are the mesh's **faces** ($\mathcal{S}_2$) and whose edges represent **face adjacency**, called the *face graph*.
3. C: an array of size $|E_\mathcal{V}|$ that stores the two adjacent faces ($\operatorname{argmax}_{f \in \mathcal{S}_2} \left|\iota_1^2(e, f)\right|$) and the two contained vertices ($\operatorname{argmax}_{v \in \mathcal{S}_0} \left|\iota_0^1(v, e)\right|$) for every edge e.

In listings, we will denote access to C as $\{f_1, f_2\} \leftarrow f_C(e)$ and $\{v_1, v_2\} \leftarrow v_C(e)$ respectively.

Side note: The graphs are implemented as adjacency lists following the array-of-arrays paradigm (the graph is represented as an array of the size of its vertex set whose every entry contains an array of outbound edges). Undirectedness is achieved by doubly inserting, i.e. for every edge $\{v_i, v_j\}$, not only is an entry in the outbound edge array of v_i maintained but also v_j.

These two graphs have a number of annotations (accompanying data) associated with their vertices and edges:

- $V_\mathcal{V}$ is annotated with the vertices' port information, denoted $p_V : \mathcal{S}_0 \to G$. In the code, the port information is a pair of enumerator (for the port type) and index to indicate which subset of 2.1 the element belongs to.
- $E_\mathcal{V}$ is annotated with the edges' indices and directions.
  The latter is denoted $\overline{v_C} : \mathcal{S}_1 \to \mathcal{S}_0^2$, a mapping from edges to ordered pairs of vertices.

In the code, the edge direction is an enumerator to indicate whether the direction of the edge dictated by ι is from the vertex with the lower index to the one with the higher index or the reverse.

- $V_{\mathcal{F}}$ is annotated with the faces' port information and vertex list.
  The latter is denoted $\overline{v_F} : \mathcal{S}_2 \to \bigcup_{i=0}^{\infty} \mathcal{S}_0^i$.
  In the code, the port information is identical to that of $V_{\mathcal{V}}$ and the vertex list is an array of vertex indices obeying the inherent order dictated by ι.

- $E_{\mathcal{F}}$ is annotated with the vertex graph edge index that the face graph edge crosses, denoted $e_{\mathcal{V}} : E_{\mathcal{F}} \to E_{\mathcal{V}}$.[1]
  The edges of the face graph signify face adjacency and therefore don't *directly* correspond to mesh edges. However, for meshes that discretize a bounded domain (which is one of our prerequisites), face graph edges always correspond to vertex graph edges because there are no borders.

While populating the annotation of the vertices' ($V_{\mathcal{V}}$) port information, we also make sure that every vertex belongs to at most one port.

Within computer memory, these annotations manifest as additional data stored per vertex (in the case of vertex annotations) and per outbound edge (in the case of edge annotations). This is taken care of by the graph theory library. See chapter 5 for more.

It may come as a surprise that our vertex graph is undirected despite the input mesh being endowed with an orientation. The directionality information is actually still preserved (as part the annotations of $E_{\mathcal{V}}$ and $V_{\mathcal{F}}$); defining the graph as undirected, however, makes implementing some of the algorithms - in particular the topological cycle algorithm - more comfortable. Asymptotically, there is no difference.

[1] This is achieved by constructing $E_{\mathcal{F}}$ and its annotation only after C has been populated. At that point, one can iterate over C and insert an element into $E_{\mathcal{F}}$ for every encountered face pair.

The following is a small collage illustrating the different structures associated with an exemplary torus mesh. The left and right, as well as the top and bottom edges of the diagrams are identified, i.e. they refer to the same edges which is the reason why it looks like there is a shortage of vertices in 3.2.

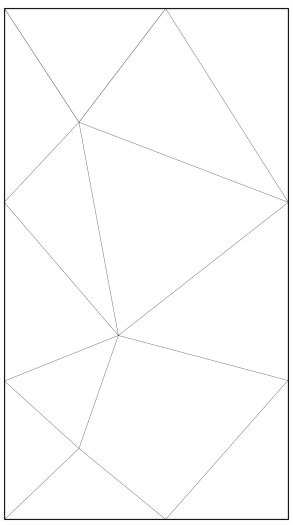


Figure 3.1: Mesh

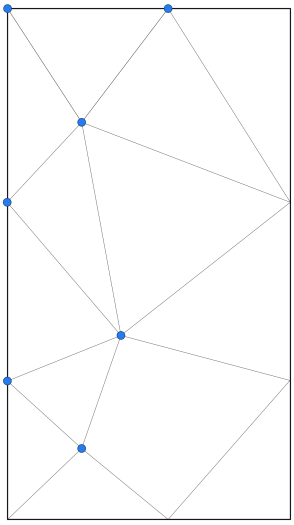


Figure 3.2: Mesh + $V_{\mathcal{V}}$

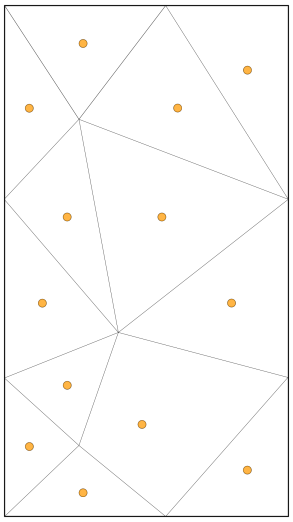


Figure 3.3: Mesh + $V_{\mathcal{F}}$[2]

All the computations presented in this section up until this point are trivial to perform, so we won't describe their derivation. A reference implementation is also available. The last step we will describe in a little more detail:

Once all structures have been populated, we will find an orientation of the mesh. The orientation of a face is stored in our data structures by the order in which the vertices appear in a face's vertex list ($\overline{v_F}$). Inverting the orientation of a face is equivalent to reversing this list. To compute an orientation, we perform the following steps for each connected component[3] of the face graph:

1. Pick a random face from the respective connected component.
2. Initiate a depth-first search starting at that face, storing a boolean array for which faces have been visited and storing the predecessor alongside on the DFS-stack that indicates which face has led to the discovery of that face.

   2.1. For every visited face, orient it in accordance with its predecessor:

[2]Note that the elements of $V_{\mathcal{F}}$ are not actually nodes with a physical position as this illustration might suggest. The orange dots are merely an abstract representation of the entire face they are illustrated to be inside of.

[3]Although there is only one connected component as per the first input constraint, the orientation algorithm trivially works for multiple disjoint meshes as well and thus we present it in that form.

2.1.1. Find the edge shared by the current and the predecessor face.

2.1.2. If the edge's endpoint vertices occur in the same direction[4], reverse this face's vertex array.

2.2. While enqueueing the adjacent unvisited faces, check whether the now oriented current face is oriented in agreement with the the adjacent visited faces, i.e. whether subroutine 2.1. executed on the current face and that adjacent visited face would be a no-op. If it is not, abort the algorithm because the mesh is not orientable.

3.2 Topological Cycles

For computing topological cycles, we largely rely on previous work [1]. The only novel contribution here consists of the intersection resolution code.

Algorithm 1 Topological Cycles

1: **procedure** TopologicalCycles($\mathcal{V}, \mathcal{F}$)
2: $\mathcal{F}^* \leftarrow \mathrm{MST}(\mathcal{F})$
3: $\mathcal{V}' \leftarrow (V_\mathcal{V}, E_\mathcal{V} \setminus \{e_\mathcal{V}(e) \mid e \in E_{\mathcal{F}^*}\})$
4: $\mathcal{V}^* \leftarrow \mathrm{MST}(\mathcal{V}')$
5: $B \leftarrow E_{\mathcal{V}'} \setminus E_{\mathcal{V}^*}$ ▷ B is the set of buckles
6: $C \leftarrow \{\mathrm{BFS}((V_{\mathcal{V}'}, E_{\mathcal{V}'} \setminus \{\{a, b\}\}), \{a\}, \{b\}) \mid \{a, b\} \in B\}$
7: $C \leftarrow \{\text{PushLoopsOutOfPorts}(\gamma, \mathcal{V}) \mid \gamma \in C\}$
8: **return** C
9: **end procedure**

[4] As an example, $(2, 3)$ occurs in the same direction in both $(1, 2, 3, 4)$ as well as $(5, 2, 3, 6)$, but not in $(4, 3, 2, 1)$. The wraparound also has to be considered: $(2, 3)$ occurs in the same direction in both $(1, 2, 3, 4)$ as well as $(3, 4, 1, 2)$.

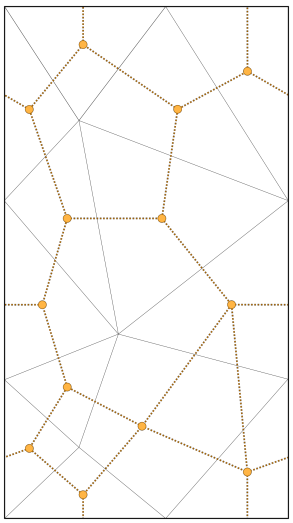


Figure 3.4: $\mathcal{F}$ in orange

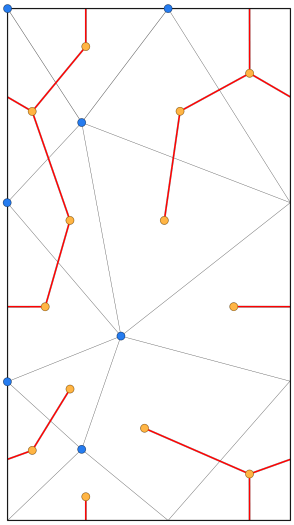


Figure 3.5: $E_{\mathcal{F}^*}$ in red

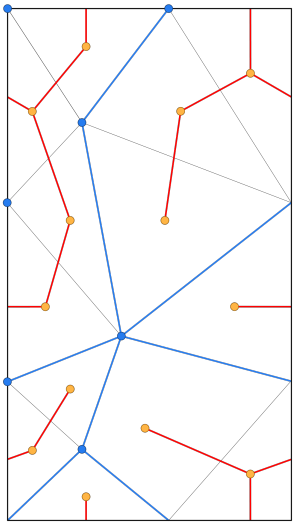


Figure 3.6: $\mathcal{V}'$ in blue

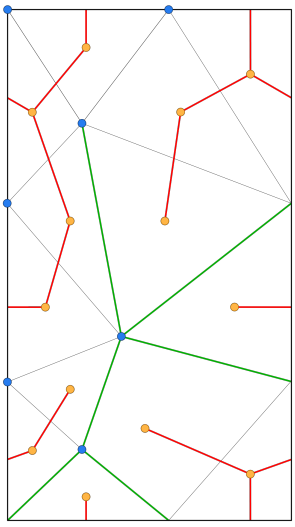


Figure 3.7: $E_{\mathcal{V}^*}$ in green

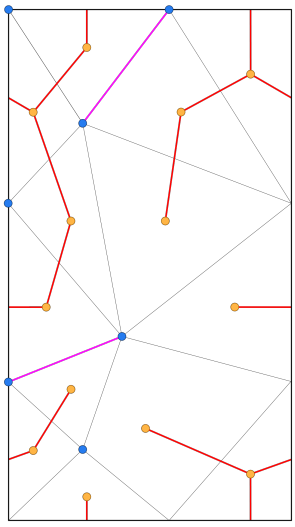


Figure 3.8: B in fuchsia

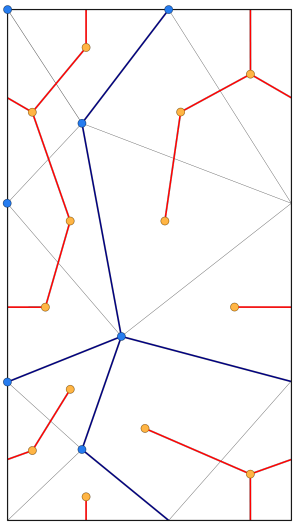


Figure 3.9: C in navy

3.2.1 Intersection Resolution

In the above algorithm, we computed the topological cycles on the mesh. Because we eventually want to convert these cycles into edge functions (cf. 2.2) and because not all edges are admissible in the domain of these functions, we need to reroute the cycles that happen to include excluded edges.

We now describe the PushLoopsOutOfPorts function used above in the topological cycle function. As the name suggests, this algorithm only works for loops. This is not an inherent limitation of the algorithm; an adaptation that works for open walks is not difficult to arrive at but we would have no use for it within this work because all topological cycles are loops.

1. If the given loop γ doesn't contain an excluded edge, terminate.[5]
2. If the last edge of the loop is an excluded edge, rotate[6] the loop until that is no longer the case.[7]
3. Iterate through the edges of the loop and for each consecutive run of excluded edges, do the following:
   - 3.1. Construct a *outline graph* for the smallest non-excluded outline around the corresponding port.[8]
   - 3.2. Find the shortest path on this outline graph from the start to the end of the run of excluded edges, the *detour*.
   - 3.3. Replace the run with the shortest path on the port's outline.

When dealing with electric ports, a lot of extra care has to be taken in step 3.1. when the outline graph is constructed. For PushLoopsOutOfPorts to not alter the homological properties of the topological cycles, the runs of excluded edges and their detours have to be topologically equivalent. As ports are constrained to be topologically trivial, this means that the outline graphs constructed in 3.1. also have to be topologically trivial. A naïve implementation,

[5] Even though walks (here: loops) are implemented as arrays of vertices as outlined in the terminology section, we freely switch back and forth between a vertex-centric and an edge-centric view. In fact, in the reference implementation, walks come with iterators that make them behave as though they were arrays of edges instead.

[6] Special care has to be taken for the last vertex: the correct algorithm is to remove the last vertex (which is equal to the first vertex), rotate as usual and to then reinsert the new first vertex at the back.

[7] This step is not strictly necessary. An equivalent algorithm that omits this step and instead works with cyclical indices may be devised but we opted for this variant because it makes the rest of the algorithm a bit easier and because it has no impact on the asymptotic performance of the algorithm.

[8] The same graph can later be reused, should another intersection with the same port occur. This is the reason why these graphs are tabulated in the reference implementation.

[9] As the outline graphs do not reside in the same space as the mesh and the vertex graph (cf. figure 3.19), what is illustrated here is merely a projected version "$\phi(G)$". Contrary to what the illustration suggests, the top and bottom "fangs" are not connected.

[10] When converting vertex walks to edge vectors, redundant back-and-forths are automatically removed because they cancel out (i.e. sum up to 0) when constructing the edge vector.

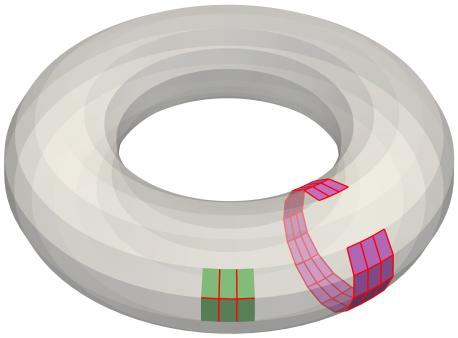


Figure 3.10: Excluded edges

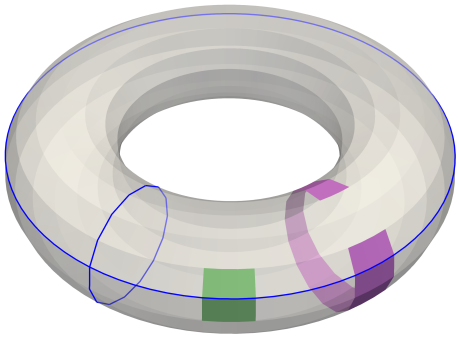


Figure 3.11: C before resolution

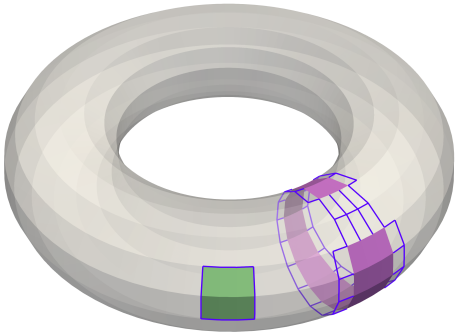


Figure 3.12: Outline graphs[9]

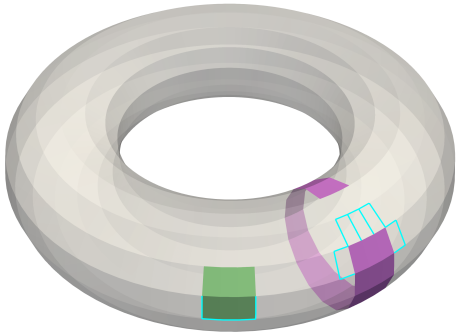


Figure 3.13: Detours

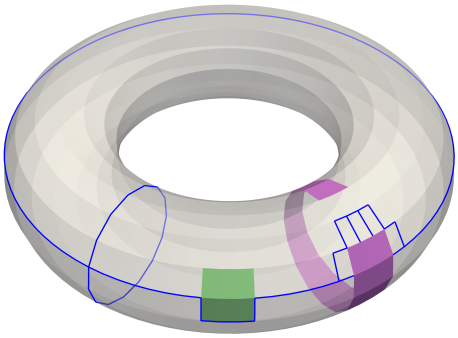


Figure 3.14: C after resolution

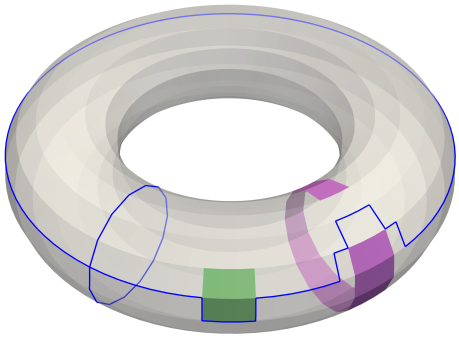


Figure 3.15: C as an edge vector[10]

for instance one that uses subgraphs of $\mathcal{V}$ as outline graphs, may fail to deliver on this in situations where electric ports are "close" to being topologically non-trivial but it may still at least be able to accurately tell when it fails, e.g. by computing the cyclomatic number r of the generated outline graph.

The following situation (Figure 3.16) is a trap for naïve implementations of

PushLoopsOutOfPorts: if the outline graph is constructed as a subgraph of the vertex graph, the detour chosen for the illustrated intersection "short-circuits" the topological cycle. The shortest path on the naïve outline graph between the upper intersection vertex and the lower intersection vertex comprises exactly the two edges of the topological cycle that aren't excluded. The topological cycle collapses to a loop of length 4 on the two edges between the the port. Upon constructing the edge vector, the cycle vanishes completely. The below collage demonstrates correct handling of this situation as performed by the reference implementation.

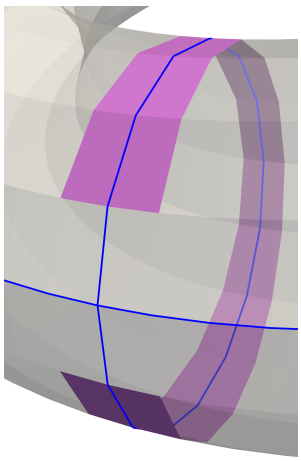


Figure 3.16:
C before resolution

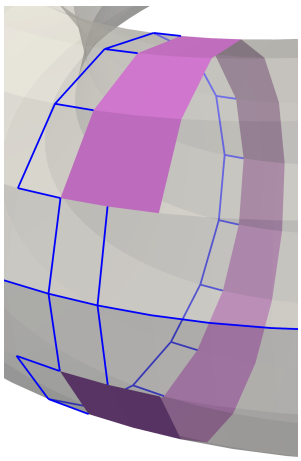


Figure 3.17:
C after resolution

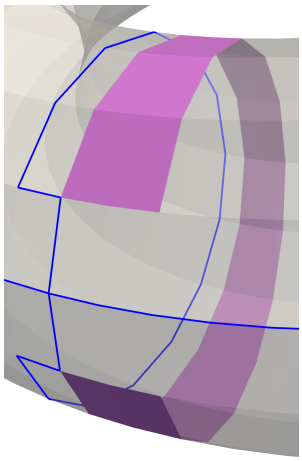


Figure 3.18:
C as an edge vector

A more sophisticated approach and also the one pursued in the reference implementation is to not use subgraphs of $\mathcal{V}$ as outline graphs for electric ports but to potentially insert multiple vertices into the outline graphs (the *local theater*) for one and the same vertex in the vertex graph (the *global theater*). The idea is to trace along the boundary of the ports and, in the case of electric ports, to draw *arcs* (α) of non-excluded edges from and into the boundary for every boundary edge. These arcs are to be inserted into the outline graph with no regard for previous arcs: if a global vertex that is part of the current arc was already part of a previous arc, it is nonetheless inserted again locally. The vertex in the global theater then corresponds to multiple vertices in the local theater while they among themselves are not in any way connected to each other. For magnetic ports, we simply take the tight boundary as our outline graph because magnetic port boundary edges are not excluded.

We now present the above ideas in a rigorous format. Note that v is any vertex on the boundary of the port whose outline graph we want to construct. We naturally have this v at hand when we search for runs of excluded edges in our intersection resolution code.

Algorithm 2 Construct Outline Graph

procedure ConstructOutlineGraph($\mathcal{V}, \mathcal{F}, v \in V_{\mathcal{V}}$)
  $\partial\overline{\Gamma_I} := \{e \in E_{\mathcal{V}} \mid (\exists f \in f_C(e) : p(f) = \overline{\Gamma_I}) \wedge (\exists f \in f_C(e) : p(f) \neq \overline{\Gamma_I})\}$
  $p \leftarrow (v, 1)$ ▷ previous vertex
  $c \leftarrow p$ ▷ current vertex
  $l \leftarrow c$ ▷ last (final) vertex
  $G \leftarrow (\{c\}, \{\})$
  while true **do**
    $e \in \{e \in \partial\overline{\Gamma_I} \setminus \{\{p_1, c_1\}\} \mid c \in e\}$ ▷ pick arbitrarily[11]
    $n_1 \in v_C(e) \setminus \{c\}$ ▷ pick unique
    $x \leftarrow [n_1 = l_1]$ ▷ abortion condition
    if x **then**
      $n \leftarrow l$
    else
      $n \leftarrow (n_1, |V_G| + 1)$
      $V_G \leftarrow V_G \cup \{n\}$
    end if
    if $\exists i : p_V(v) = \overline{\Gamma^i_M}$ **then**
      $E_G \leftarrow E_G \cup \{\{c, n\}\}$
    else if $\exists i : p_V(v) = \overline{\Gamma^i_E}$ **then** ▷ purely illustrative check
      $\alpha \leftarrow \mathrm{BFS}((V_{\mathcal{V}}, \mathring{E_{\mathcal{V}}}), \{c_1\}, \{n_1\})$ ▷ $\mathring{E_{\mathcal{V}}}$ is $E_{\mathcal{V}}$ w/o excluded edges
      $\alpha' \leftarrow ((\alpha_2, |V_G| + 1), (\alpha_3, |V_G| + 2), \ldots, (\alpha_{|\alpha|-1}, |V_G| + |\alpha| - 2))$
      $V_G \leftarrow V_G \cup \{\alpha'_1, \ldots, \alpha'_{|\alpha'|}\}$
      $E_G \leftarrow E_G \cup \{\{c, \alpha'_1\}, \{\alpha'_1, \alpha'_2\}, \ldots, \{\alpha'_{|\alpha'|}, n\}\}$
    end if
    if x **then**
      break
    end if
    $p, c \leftarrow c, n$
  end while
  return G
end procedure

To translate back and forth between the global and the local theater, there exists a left-unique, right-total relation between $V_{\mathcal{V}}$ and V_G. However, to be more in line with the implementation, we opt to rather denote this as $\phi : V_G \to V_{\mathcal{V}}, v \mapsto v_1$. The inverse is set-valued: $\phi^{-1} : \operatorname{ran}\phi \to \mathcal{P}(V_G) \setminus \{\{\}\}$. In the reference implementation, ϕ is realized by annotating the vertices V_G while ϕ^{-1} is realized by means of a non-unique hash table (from $V_{\mathcal{V}}$ to (multiple) V_G) that is updated with every insertion into V_G.

Detours between the port boundary vertices $a, b \in V_{\mathcal{V}}$ are then found by computing $\beta' \leftarrow \mathrm{BFS}(G, \phi^{-1}(a), \phi^{-1}(b))$ and by then pulling back the re-

[11] For the very first iteration, there are two choices. For all subsequent iterations, there is only one.

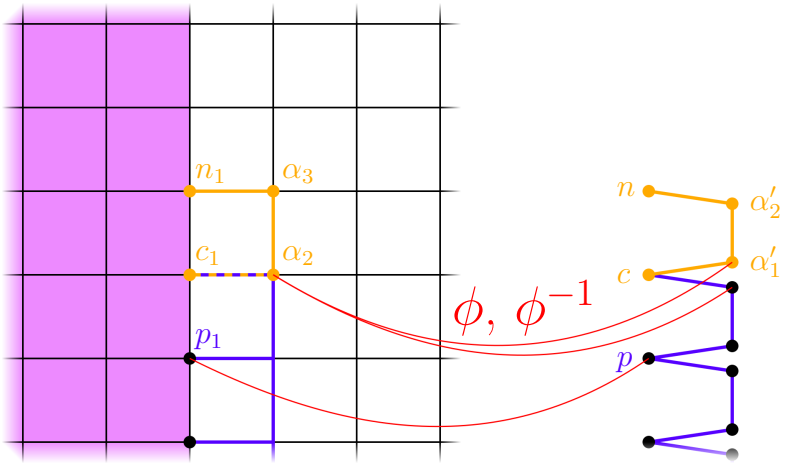


Figure 3.19: Snapshot of a ConstructOutlineGraph iteration:
global theater (left), local theater (right), previous arcs E_G, current arc α / α'

sulting path β' into the global theater by mapping its vertices with ϕ: $\beta \leftarrow (\phi(\beta'_1), \ldots, \phi(\beta'_{|\beta'|}))$. The resulting β is then woven into the topological cycle, replacing $(a, \ldots, b)$.

3.2.2 Time Complexity

An essential component of TopologicalCycles is the algorithm used to solve the MST problem. While, as of today, the time complexity of the optimal algorithm solving the general MST problem is not known [3], our task is massively simplified by virtue of our graphs being unweighted. For unweighted graphs, any spanning tree is a minimum spanning tree. Spanning trees can be found using a simple breadth-first search which terminates in $\mathcal{O}(|V| + |E|)$ steps, which is equivalent to $\mathcal{O}(|E|)$ for our types of graphs.

The time complexity of TopologicalCycles is obtained by summing up the time complexities of its steps:

2: $\mathcal{F}^* \leftarrow \mathrm{MST}(\mathcal{F})$
$\mathcal{O}(|E_\mathcal{F}|) = \mathcal{O}(|E_\mathcal{V}|)$

3: $\mathcal{V}' \leftarrow (V_\mathcal{V}, E_\mathcal{V} \setminus \{e_\mathcal{V}(e) \mid e \in E_{\mathcal{F}^*}\})$
$\mathcal{O}(|V_\mathcal{V}| + |E_\mathcal{V}| + |E_\mathcal{F}|) = \mathcal{O}(|E_\mathcal{V}|)$

4: $\mathcal{V}^* \leftarrow \mathrm{MST}(\mathcal{V}')$
$\mathcal{O}(|E_{\mathcal{V}'}|) \subseteq \mathcal{O}(|E_\mathcal{V}|)$

5: $B \leftarrow E_{\mathcal{V}'} \setminus E_{\mathcal{V}^*}$
$\mathcal{O}(|E_{\mathcal{V}'}| + |E_{\mathcal{V}^*}|) \subseteq \mathcal{O}(|E_\mathcal{V}|)$

6: $C \leftarrow \{\text{BFS}\left((V_{\mathcal{V}'}, E_{\mathcal{V}'} \setminus \{\{a,b\}\}), a, b\right) \mid \{a,b\} \in B\}$
$\mathcal{O}\left(|B| \cdot |E_{\mathcal{V}'}|\right) = \mathcal{O}\left(\beta_1 |E_{\mathcal{V}'}|\right) \subseteq \mathcal{O}\left(\beta_1 |E_{\mathcal{V}}|\right)$; this somewhat output-sensitive time complexity comes from the fact that every buckle corresponds to one topological cycle.

7: $C \leftarrow \{\text{PushLoopsOutOfPorts}\left(\gamma, \mathcal{V}\right) \mid \gamma \in C\}$
$\mathcal{O}\left(\beta_1 |E_{\mathcal{V}}|^2\right)$ or $\mathcal{O}\left(\beta_1 |E_{\mathcal{V}}|\right)$; see below.

The total time complexity of TopologicalCycles is thus concluded to be either $\mathcal{O}\left(\beta_1 |E_{\mathcal{V}}|^2\right)$ or $\mathcal{O}\left(\beta_1 |E_{\mathcal{V}}|\right)$ depending on the additional assumptions made. See below for more details.

We now examine the time complexity of PushLoopsOutOfPorts. It is obvious that steps 1 and 2 terminate after $\mathcal{O}\left(|E_{\mathcal{V}}|\right)$ operations.

Even though step 3.1 is within a loop, the loop does not really pertain to it in terms of complexity because the created outline graphs are tabulated, and neither does the number of topological cycles. We view step 3.1 as being executed once per port ($\mathcal{O}\left(N_M + N_E\right)$ times). For magnetic ports, ConstructOutlineGraph simply traces along the boundary and thus takes $\mathcal{O}\left(|E_{\mathcal{V}}|\right)$ steps to complete for all magnetic ports cumulatively. For electric ports, the time complexity is dominated by the BFS searches we perform for every arc. We compute $\mathcal{O}(|\partial\overline{\Gamma_E}|)$ arcs (the number of electrical boundary edges) and every BFS search takes $\mathcal{O}\left(|E_{\mathcal{V}}|\right)$ steps to complete. We conclude that step 3.1 takes $\mathcal{O}(|\partial\overline{\Gamma_E}||E_{\mathcal{V}}|)$ steps to complete. In many situations, e.g. if $|E_{\mathcal{V}}|$ increases through uniform refinement of the mesh, the number of electric port boundary edges is asymptotically equivalent to the number of vertex graph edges $\mathcal{O}(|\partial\overline{\Gamma_E}|) = \mathcal{O}\left(|E_{\mathcal{V}}|\right)$, leaving us with the complexity of $\mathcal{O}(|E_{\mathcal{V}}|^2)$.

Although the complexities presented for this step are relatively large, we conjecture that they are not tight upper bounds and that the true upper bound is closer to $\Theta\left(N_E |E_{\mathcal{V}}|\right)$. In fact, if we again consider the case where $|E_{\mathcal{V}}|$ increases as a consequence of uniform refinement of the mesh, we can expect that after a constant number of refinements, the BFS step becomes unable to discover other ports because it reaches n_1 too quickly. However, if other ports are out of the equation, the generated path α cannot be longer than the arity of the polygon that the edge $\{c_1, n_1\}$ is part of which is bounded by the constant $a_{\max}$ per 2.3. In that case, the electric port outlines are just a constant factor times longer than tight boundaries and computing them takes only $\mathcal{O}\left(|E_{\mathcal{V}}|\right)$ steps for all electric ports cumulatively.

Each execution of steps 3.2 and 3.3 takes $\mathcal{O}\left(|E_{\mathcal{V}}|\right)$ operations to terminate because the size of the number of edges in such an outline graph is only bounded from above by $\mathcal{O}\left(|E_{\mathcal{V}}|\right)$. Finding and inserting $\mathcal{O}\left(|E_{\mathcal{V}}|\right)$ shortest paths (thanks to loop 3), each of length $\mathcal{O}\left(|E_{\mathcal{V}}|\right)$, takes $\mathcal{O}\left(|E_{\mathcal{V}}|^2\right)$ steps to complete.

For an example where PushLoopsOutOfPorts really does take $\Theta\left(|E_{\mathcal{V}}|^2\right)$ steps to complete, consider figure 3.20: "the comb". The illustrated strip is the surface of a torus, $\Theta\left(|E_{\mathcal{V}}|\right)$ edges long and a constant number of edges wide. The given topological cycle intersects the single magnetic port $\Theta\left(|E_{\mathcal{V}}|\right)$ times while the average shortest path around the port is also $\Theta\left(|E_{\mathcal{V}}|\right)$ edges long.

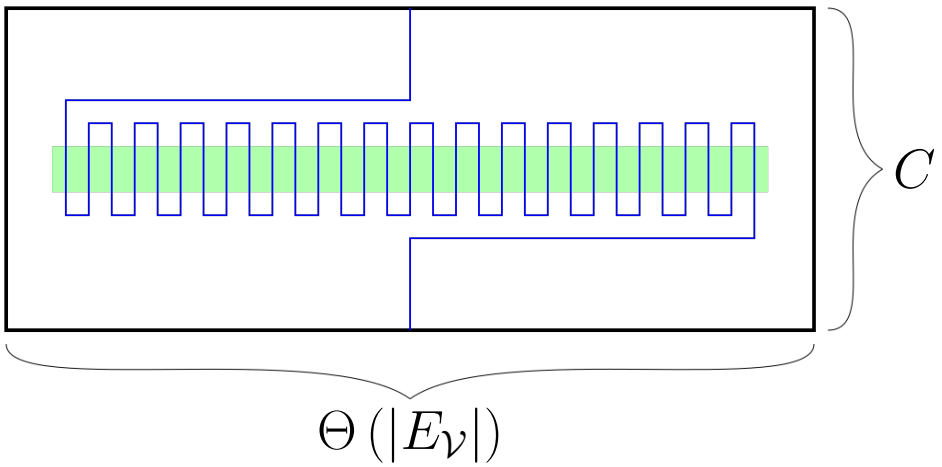


Figure 3.20: "The comb" (torus surface)

Note that it is entirely possible for the first 6 steps of TopologicalCycles to yield the illustrated topological cycle, provided that a sufficiently adversarial MST algorithm is used.

If we venture the assumption that the total number of intersections between each topological cycle and all ports (or equivalently, the number of consecutive runs of excluded edges in each topological cycle) is bounded by a constant, the time complexity shrinks because loop 3 is only performed a constant number of times. The new time complexity of steps 3.2 and 3.3 would become $\mathcal{O}(|E_\mathcal{V}|)$ including the loop.

Summing up the above complexities gives us a total complexity of either $\mathcal{O}\left(|E_\mathcal{V}|^2\right)$ or $\mathcal{O}(|E_\mathcal{V}|)$ for PushLoopsOutOfPorts, depending on the additional assumptions made. Combined with the fact that the subroutine is invoked $\Theta(|C|) = \Theta(|B|) = \beta_1$ times, this yields a total time complexity of $\mathcal{O}\left(\beta_1|E_\mathcal{V}|^2\right)$ or $\mathcal{O}(\beta_1|E_\mathcal{V}|)$ respectively for step 7.

3.3 Electric Connector Cycles

As described in more detail in [2], the sought-after electric connector cycles γ_i^E, $i = 1, \ldots, N_E$ are exactly those walks that start and end on the boundary of electric ports and that establish connections between all electric ports. We will take extra care to ensure that the generated electric connector cycles are optimal in the sense as defined in 2.4, i.e. contain the lowest number of edges possible.

Algorithm 3 Electric Connector Cycles

procedure ElectricConnectorCycles($\mathcal{V}, \mathcal{F}$)
  $\mathring{\mathcal{V}} \leftarrow (V_{\mathcal{V}}, \mathring{E_{\mathcal{V}}})$ ▷ $\mathring{E_{\mathcal{V}}}$ is $E_{\mathcal{V}}$ without the excluded edges
  for $i = 1, \ldots, N_E$ **do**
    $V_B^i \leftarrow \emptyset$ ▷ sets of boundary vertices
  end for
  for all $e \in E_{\mathcal{V}}$ **do**
    if $(\exists i \exists f \in f_C(e) : p(f) = \overline{\Gamma_E^i}) \wedge (\exists f \in f_C(e) : p(f) = \overline{\Gamma_I})$ **then**
      $V_B^i \leftarrow V_B^i \cup v_C(e)$
    end if
  end for
  $G = (V_G, E_G) \leftarrow (\{\overline{\Gamma_E^1}, \ldots, \overline{\Gamma_E^{N_E}}\}, \emptyset)$ ▷ port reachability graph
  $\gamma^E : E_G \to \bigcup_{i=0}^{\infty} V_{\mathcal{V}}^i$ ▷ codomain is the set of walks on $V_{\mathcal{V}}$
  for $i = 2, \ldots, N_E$ **do**
    for $j = 1, \ldots, i-1$ **do**
      $\gamma_{j,i}^E \leftarrow \mathrm{BFS}(\mathring{\mathcal{V}}, V_B^j, V_B^i)$
      if $|\gamma_{j,i}^E| \neq 0$ **then** ▷ if a path was found
        $e = \{\overline{\Gamma_E^j}, \overline{\Gamma_E^i}\}$
        $E_G \leftarrow E_G \cup \{e\}$
        $\gamma^E(e) \leftarrow \gamma_{j,i}^E$
      end if
    end for
  end for
  $w : E_G \to \mathbb{N}, e \mapsto |\gamma^E(e)|$ ▷ edge weight function
  $G^* \leftarrow \mathrm{MST}(G, w)$
  return $\{\gamma^E(e) \mid e \in E_{G^*}\}$
end procedure

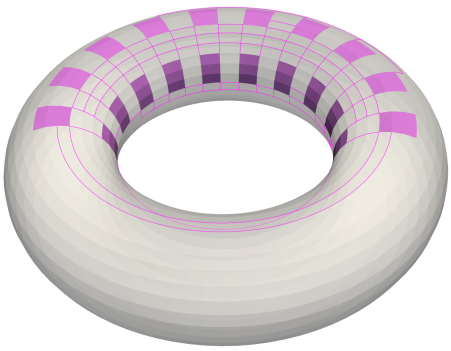


Figure 3.21: γ^E with many overlaps

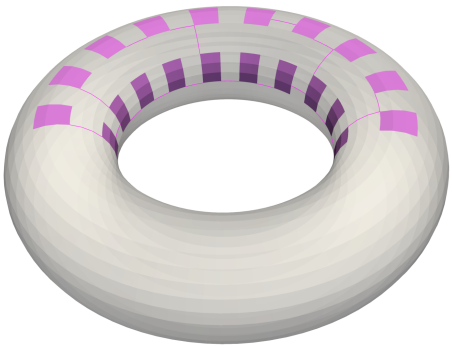


Figure 3.22: $\gamma^E|_{E_{G^*}}$

3.3.1 Time Complexity

It can easily be seen that the lines 2 to 12 terminate in $\mathcal{O}(|E_{\mathcal{V}}|)$ steps.

As for the nested loop from line 13 to 22, a naïve implementation may have a complexity of $\mathcal{O}\left(N_E^2|E_{\mathcal{V}}|\right)$ but this is not strictly necessary. It suffices to only perform $N_E - 1$ breadth first searches if, in every search, the shortest path from the starting port to every other port is recorded. This is the approach taken by the reference implementation and brings the complexity down to $\mathcal{O}(N_E|E_{\mathcal{V}}|)$.

As for line 23, it is clear that it inherits the above complexity: the paths γ^E have been constructed in $\mathcal{O}(N_E|E_{\mathcal{V}}|)$ steps so they cannot be longer in total than $\mathcal{O}(N_E|E_{\mathcal{V}}|)$ vertices and measuring them also cannot take longer than $\mathcal{O}(N_E|E_{\mathcal{V}}|)$ steps.

For a second time, we depend on the complexity of an MST subroutine for line 24 and for a second time, just as in 3.2.2, we find ourselves in a simplified case for which the complexity is a known result. As the edge weights for our port reachability graph stem from the number of vertices between the ports, we are restricted to integer weights for which a linear-time MST algorithm is known [4]. Thanks to this result, line 24 can be finished in $\mathcal{O}(N_E + |E_G|)$ steps. As for $|E_G|$, we can use the same argument as above and conclude that $|E_G|$ cannot grow faster than $N_E|E_{\mathcal{V}}|$ so we can bound the time complexity of this line with $\mathcal{O}(N_E|E_{\mathcal{V}}|)$ too.

Line 25, dealing with less than N_E paths, each of which is no longer than $|E_{\mathcal{V}}|$, trivially terminates in $\mathcal{O}(N_E|E_{\mathcal{V}}|)$ steps too.

We thus conclude that the time complexity of the entire ElectricConnectorCycles routine is $\mathcal{O}(N_E|E_{\mathcal{V}}|)$.

3.4 Magnetic Port Cycles

As discussed in [2], for our connected surface, the desired magnetic connector cycles each comprise of two loops: one loop around the cycle's respective magnetic port and one *disagreeing* loop around a singled-out magnetic port. Pursuant to our secondary objective, we choose the magnetic port with the shortest boundary for the role of the singled-out port. "Disagreeing" here refers to the constraint that, among the two loops of every cycle, one loop should agree with the orientation of the faces of the port it bounds whereas the other one should disagree with it. The function BoundaryOrientation serves to determine the agreement between a boundary loop and the faces of the port it bounds.

For the implementation of MagneticPortCycles, we first construct the edge vector ($\mathbb{Z}^{|E_{\mathcal{V}}|}$) for the boundary of every magnetic port. This is achieved by first finding any edge on the boundary of the magnetic port and then tracing along the boundary until the initial edge is rediscovered. Some care has to be taken to properly respect the edges' interior orientation ($\overline{v_C}$). The shortest among all boundaries is then removed from this set and added disagreeingly onto all other edge vectors.

For the implementation of BoundaryOrientation, we consider an edge of the boundary loop as well as the port face it touches and then compare the direction prescribed onto the edge by the boundary loop and by the face ($\overline{v_F}$).

Algorithm 4 Magnetic Port Cycles

procedure MagneticPortCycles($\mathcal{V}, \mathcal{F}$)$\to \mathcal{P}(\mathbb{Z}^{|E_\mathcal{V}|})$
  $\partial\overline{\Gamma^i_M} := \{e \in E_\mathcal{V} \mid \{p(f) \mid f \in f_C(e)\} = \{\overline{\Gamma^i_M}, \overline{\Gamma_I}\}\}$
  $B \leftarrow \{\}$ ▷ magnetic port boundary edge vectors
  for $i = 1, \ldots, N_M$ **do**
    $\boldsymbol{\gamma} \leftarrow \mathbf{0}_{|E_\mathcal{V}|}$ ▷ edge vector
    $e \in \partial\overline{\Gamma^i_M}$ ▷ pick arbitrarily
    $p \leftarrow \overline{v_C}(e)_1$ ▷ previous vertex
    $c \leftarrow \overline{v_C}(e)_2$ ▷ current vertex
    $l \leftarrow c$ ▷ last (final) vertex
    do
      if $p = \overline{v_C}(e)_1$ **then** ▷ check edge's interior orientation
        $\delta \leftarrow \ \ 1$
      else
        $\delta \leftarrow -1$
      end if
      $\boldsymbol{\gamma}_e \leftarrow \boldsymbol{\gamma}_e + \delta$
      $e \in \{e' \in \partial\overline{\Gamma^i_M} \setminus \{e\} \mid c \in e'\}$ ▷ pick unique
      $n \in v_C(e) \setminus \{c\}$ ▷ pick unique
      $p, c \leftarrow c, n$
    while $c \neq l$
    $B \leftarrow B \cup \{\boldsymbol{\gamma}\}$
  end for
  if $|B| < 2$ **then**
    return $\{\}$
  end if
  $\boldsymbol{\gamma}^* \in \operatorname{argmin}_{\boldsymbol{\gamma} \in B} |\{e \in E_\mathcal{V} \mid \boldsymbol{\gamma}_e \neq 0\}|$ ▷ pick arbitrarily
  $\varphi^* \leftarrow$ BoundaryOrientation$(\mathcal{V}, \mathcal{F}, \boldsymbol{\gamma}^*)$
  $C \leftarrow \{\}$
  for $\boldsymbol{\gamma} \in B \setminus \{\boldsymbol{\gamma}^*\}$ **do**
    $\varphi \leftarrow$ BoundaryOrientation$(\mathcal{V}, \mathcal{F}, \boldsymbol{\gamma})$
    $C \leftarrow C \cup \{\boldsymbol{\gamma} - \varphi\varphi^*\boldsymbol{\gamma}^*\}$
  end for
  return C
end procedure

Algorithm 5 Boundary Orientation

procedure BoundaryOrientation($\mathcal{V}, \mathcal{F}, \boldsymbol{\gamma} \in \mathbb{Z}^{|E_\mathcal{V}|}$)$\to \{-1, 1\}$
  $e \in \{e \in E_\mathcal{V} \mid \boldsymbol{\gamma}_e \neq 0\}$ ▷ pick arbitrarily
  $f \in \{f \in f_C(e) \mid \exists i : p(f) = \overline{\Gamma^i_M}\}$ ▷ pick unique
  $\overline{v} \leftarrow \overline{v_F}(f)$
  $i \in \{i \in \{1, \ldots, |\overline{v}|\} \mid \overline{v}_i = \overline{v_C}(e)_1\}$ ▷ pick unique
  if $i = |\overline{v}|$ **then**
    $j \leftarrow 1$
  else
    $j \leftarrow i + 1$
  end if
  if $\overline{v}_j = \overline{v_C}(e)_2$ **then**
    $\varphi \leftarrow \quad 1$
  else
    $\varphi \leftarrow -1$
  end if
  return $\varphi \operatorname{sgn}(\boldsymbol{\gamma}_e)$
end procedure

As an example, consider the following situation with two magnetic port cycles $\boldsymbol{\gamma}^1, \boldsymbol{\gamma}^2$. The round circles illustrate the faces' interior orientation. It can be seen how in the two cycles the loop $\boldsymbol{\gamma}^*$ occurs in different winding directions.

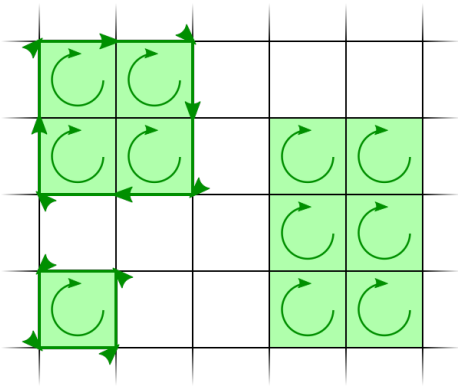


Figure 3.23: $\boldsymbol{\gamma}^1 \in C$

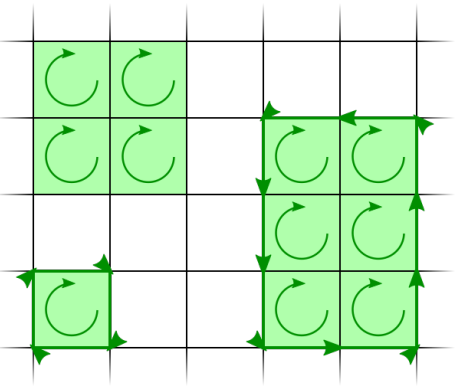


Figure 3.24: $\boldsymbol{\gamma}^2 \in C$

3.4.1 Time Complexity

The first part of BoundaryOrientation is to find a vertex graph edge that occurs within the provided boundary loop. As we cannot make any further assumption as to where within $\boldsymbol{\gamma}$ the first such edge occurs, this takes us $\Theta(|E_\mathcal{V}|)$ steps to complete.

The second and final non-constant-time part of BoundaryOrientation is the search for $\overline{v_C}(e)_1$ in $\overline{v}$ which takes $\Theta(|\overline{v}|)$ steps to complete. However, this can be

bounded by $\mathcal{O}(a_{\max})$ which we assumed to be constant in 2.3.

The total time complexity of BoundaryOrientation thus comes out as $\Theta(|E_{\mathcal{V}}|)$.

The first part of MagneticPortCycles, constructing all boundaries B, takes $\mathcal{O}(|E_{\mathcal{V}}|)$ steps to complete per magnetic port, hence $\mathcal{O}(N_M|E_{\mathcal{V}}|)$ in total (obviously $\Theta(|B|) = \Theta(|C|) = \Theta(N_M)$). As we assume that we are working with dense data structures[12], it can also be observed that this complexity cannot be undercut because the vectors in B contain $\Theta(N_M|E_{\mathcal{V}}|)$ scalars and thus necessitate at least $\Theta(N_M|E_{\mathcal{V}}|)$ steps to be constructed.

Finding the shortest boundary loop $\boldsymbol{\gamma}^*$ requires inspecting each of the scalars initialized in the previous part a constant number of times and, thus, doesn't change the complexity.

In the last relevant part, we invoke BoundaryOrientation ($\Theta(|E_{\mathcal{V}}|)$) and perform a linear algebra operation ($\Theta(|E_{\mathcal{V}}|)$). Both these steps are executed $\Theta(|B|) = \Theta(N_M)$ times, giving us a complexity of $\Theta(N_M|E_{\mathcal{V}}|)$ for this part, too.

The total time complexity of MagneticPortCycles is thus shown to be $\Theta(N_M|E_{\mathcal{V}}|)$.

[12] "Dense" here refers to an implementation paradigm of linear algebra (here, vector) data structures. Unlike sparse data structures, dense data structures incur complexity even if filled with zeros. Sparse data structures which would change the complexity of this step were, however, not investigated within the scope of this work.

Chapter Four

Verification

4.1 Derivation

We define the *boundary operator* for l-chains:

$$B_l : \mathbb{R}^{|\mathcal{S}_l|} \to \mathbb{R}^{|\mathcal{S}_{l-1}|},\ l = 1, 2 \tag{4.1}$$

Using the incidence function ι as defined in 2.1, this operator can be represented as a matrix:

$$\mathbf{B}_l \in \{-1, 0, 1\}^{|\mathcal{S}_{l-1}| \times |\mathcal{S}_l|} \tag{4.2}$$

$$(\mathbf{B}_l)_{j,k} = \iota_{l-1}^{l}\left(\mathbf{s}_{l-1}^{j}, \mathbf{s}_{l}^{k}\right) \tag{4.3}$$

where $j = 1, \ldots, |\mathcal{S}_{l-1}|$ and $k = 1, \ldots, |\mathcal{S}_l|$. The matrix $\mathbf{B}_l$ is the signed incidence matrix of l- and $l-1$-primitives.

We use the symbol $\mathring{}$ to denote removal of excluded primitives:

$$\mathring{\mathcal{S}}_l := \{\mathbf{s}_l \in \mathcal{S}_l \mid \mathbf{s}_l \text{ is not excluded}\} \tag{4.4}$$

This notion naturally extends to B_l (and, thus, $\mathbf{B}_l$) by restricting the domain and codomain:

$$\mathring{B}_l : \mathbb{R}^{|\mathring{\mathcal{S}}_l|} \to \mathbb{R}^{|\mathring{\mathcal{S}}_{l-1}|} \tag{4.5}$$

As derived in [2], our task consists of finding all 1-chains on the edge set $\mathcal{S}_1$ whose boundary is either empty or contained in $\partial\Gamma_E$. Finding these 1-chains is equivalent to finding the null space $\ker \mathring{\mathbf{B}}_1$.

It is known that the null space of $\mathring{\mathbf{B}}_1$ is contained in the range of $\mathring{\mathbf{B}}_2$, formally $\ker \mathring{\mathbf{B}}_1 \subseteq \operatorname{ran} \mathring{\mathbf{B}}_2$. The generating 1-chains $\{\mathring{\mathbf{c}}_1, \ldots, \mathring{\mathbf{c}}_M\} \subseteq \mathbb{Z}^{|\mathcal{S}_1|}$ are supposed to close this gap, hence the following equation must hold:

$$\ker \mathring{\mathbf{B}}_1 = \operatorname{ran} \mathring{\mathbf{B}}_2 + \operatorname{span}\{\mathring{\mathbf{c}}_1, \ldots, \mathring{\mathbf{c}}_M\} \tag{4.6}$$

This viewpoint places the verification problem wholly into the domain of linear algebra, allowing us to tackle it using linear algebra methods. We start off by defining $\mathring{\mathbf{C}} := \begin{bmatrix}\mathring{\mathbf{c}}_1 & \cdots & \mathring{\mathbf{c}}_M\end{bmatrix} \in \{-1, 0, 1\}^{|\mathring{\mathcal{S}}_1| \times M}$ which allows us to make the following simplifications:

$$\ker \mathring{\mathbf{B}}_1 = \operatorname{ran} \mathring{\mathbf{B}}_2 + \operatorname{span}\{\mathbf{c}_1, \ldots, \mathbf{c}_M\}$$

$$\ker \mathring{\mathbf{B}}_1 = \operatorname{ran} \mathring{\mathbf{B}}_2 + \operatorname{ran} \mathbf{C} \tag{4.7}$$

$$\ker \mathring{\mathbf{B}}_1 = \operatorname{ran} \begin{bmatrix} \mathring{\mathbf{B}}_2 & \mathbf{C} \end{bmatrix} \tag{4.8}$$

We give this composed matrix a name for convenience's sake:

$$\mathbf{X} := \begin{bmatrix} \mathring{\mathbf{B}}_2 & \mathring{\mathbf{C}} \end{bmatrix} \in \{-1, 0, 1\}^{|\mathring{\mathcal{S}}_1| \times |\mathring{\mathcal{S}}_2| + M} \tag{4.9}$$

Verifying 4.8 is equivalent to verifying the following pair of equations:

$$\mathring{\mathbf{B}}_1 \mathbf{X} = 0 \tag{4.10}$$

$$\ker \begin{bmatrix} \mathring{\mathbf{B}}_1 \\ \mathbf{X}^T \end{bmatrix} = \{0\} \tag{4.11}$$

the former of which proves $\operatorname{ran} \mathbf{X} \subseteq \ker \mathring{\mathbf{B}}_1$ while the latter proves that there are no linearly dependent and thus redundant 1-cycles among $\{\mathring{\mathbf{c}}_1, \ldots, \mathring{\mathbf{c}}_M\}$.

Equation 4.11 can be reformulated to a definiteness problem:

$$\ker \begin{bmatrix} \mathring{\mathbf{B}}_1 \\ \mathbf{X}^T \end{bmatrix} = \{0\}$$

$$\ker \left(\mathring{\mathbf{B}}_1^T \mathring{\mathbf{B}}_1 + \mathbf{X}\mathbf{X}^T \right) = \{0\} \tag{4.12}$$

$$\mathring{\mathbf{B}}_1^T \mathring{\mathbf{B}}_1 + \mathbf{X}\mathbf{X}^T \text{ is s.p.d.} \tag{4.13}$$

The last step follows from the observation that for any real-valued matrix $\mathbf{A}$, the matrix $\mathbf{A}\mathbf{A}^T$ is symmetric positive semi-definite which is preserved under matrix addition. Lastly, the fact that the null space is trivial (4.12) rids us of the "semi-".

4.2 Implementation Considerations

There are numerous ways to test whether a given symmetric matrix is positive definite, the most well-known of which is probably the Cholesky decomposition. A point worth making is, however, that despite us venturing into the domain of computational linear algebra in our verification routine, we did not necessarily lose determinacy. All matrices we've constructed are integer-valued and checking whether an integer-valued matrix is positive definite can be done deterministically by using rational numbers instead of floating point numbers.

Chapter Five

Implementation

A reference implementation written in C++20 is provided as a module of LehrFEM++ [5] and builds upon LehrFEM++'s API for handling meshes. However, the only part of the cohomological cycle computation pipeline that interfaces with LehrFEM++ is the constructor of the internal mesh data structure. Additionally, some of the auxiliary visualization utilities also operate on some of the data structures of LehrFEM++ but they are not relevant to the computation of cycles. By rewriting the aforementioned constructor to use a different input data structure, the cohomological cycle computation pipeline can be used independently of LehrFEM++.

The reference implementation including the verification and the auxiliary visualization modules is fully documented using Doxygen [6].

For the basic graph theoretical building blocks, the reference implementation draws on the C++ graph theory library Quiver [7], which provides a class for adjacency lists and routines for spanning tree computations and breadth-first search.

5.1 Testing

In the unit test file, we first test the correctness of some more isolated pieces of code including the correct rejection of input that fails to meet one of the preconditions.

The main part, however, consists of running the entire cohomology pipeline on a number of preconstructed test cases and invoking the verification routine on the generated cycles as well as verifying known relationships such as $N_T = 2\beta_1(\Omega_C)$. The set of test cases comprises two cubes, five tori and a sequence of handlebodies with genera ranging from 0 to 20. Some of these scenarios test intricate constellations such as the one presented in Figure 3.16. The meshes used in the different test cases range from anywhere between 8 and 3000 vertices in size while the number of ports ranges from 0 to 42. All tests complete successfully.

5.2 Illustrations

The 2D images were handcrafted using the program Inkscape [8].

The 3D images were stylized and rendered using the program ParaView [9]. The 3D models have been generated using the auxiliary visualization utilities included in

the reference implementation. The visualization utilities are able to export various of the internally used data structures to OBJ files.

Chapter Six

Conclusion and Further Work

We have presented algorithms to compute and verify cycles for electromagnetic problems with nontrivial boundary conditions and we have provided complexity analyses for these algorithms. We have also provided a reference implementation of the algorithms and of the verification routine with which we have been able to confirm the correctness of the generated cycles in all tested scenarios.

Further research question in this direction may include:

- Investigate the use and complexity benefit of sparse data structures as briefly alluded to in the analysis of the magnetic port cycle algorithm.
- Investigate alternative ways to generate the topological cycles in a setting with impermissible (excluded) edges, in particular whether it is possible to "guide" the MST algorithm so that the resulting topological cycles don't contain excluded edges, saving us the need to move them away from the excluded edges in a post-processing step.
- Investigate ways to shorten cycles without altering their homological properties.

Bibliography

[1] R Hiptmair and J Ostrowski. Generators of $H_1(\Gamma_h, \mathbb{Z})$ for triangulated surfaces: Construction and classification. Report 160, SFB 382, Universität Tübingen, Tübingen, Germany, 2001. *SIAM J. Computing*.

[2] Ralf Hiptmair and Jörg Ostrowski. Electromagnetic port boundary conditions: Topological and variational perspective. *International Journal of Numerical Modelling: Electronic Networks, Devices and Fields*, 34(3):e2839, 2021.

[3] Seth Pettie and Vijaya Ramachandran. An optimal minimum spanning tree algorithm. *J. ACM*, 49(1):16–34, jan 2002.

[4] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, 1994.

[5] Raffael Casagrande and Ralf Hiptmair. LehrFEM++. `https://github.com/craffael/lehrfempp`.

[6] Dimitri van Heesch. Doxygen. https://www.doxygen.nl/.

[7] Josua Rieder and Pascal Sommer. Quiver. `https://github.com/JosuaRieder/Quiver`.

[8] Inkscape Project. Inkscape. https://inkscape.org.

[9] Los Alamos National Laboratory Sandia National Laboratories, Kitware Inc. ParaView. https://www.paraview.org/.