Computing invariants of knotted manifolds

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Kelvin Killeen

Supervised by: Professor Graham Ellis

School of Mathematical & Statistical Sciences
University of Galway

A thesis submitted for the degree of
Doctor of Philosophy
September 2022

Abstract

In this thesis, we develop algorithms in computational topology for working with regular CW-complexes and calculating associated homological invariants. We adapt the classical notion of broken surface diagrams for describing embeddings of surfaces in 4-dimensional space, and describe algorithms for constructing regular CW-complex structures on the complements of these embedded surfaces. In particular, we develop an algorithm for inputting a virtual knot/link diagram and returning a regular CW-structure on the complement in S^4 of Satoh's Tube map of the virtual knot/link. We also develop algorithms that input classical knot/link diagrams and return regular CW-structures on the complements of the knot/link in S^3 as well as algorithms that perform Dehn surgery on these complements to produce any closed, orientable 3-manifold. We use homological algebra to design isotopy invariants of codimension two embeddings and describe algorithms for computing these invariants. Key illustrations of the algorithms include:

- A new method for distinguishing between the granny knot and the reef knot; these knot complements have isomorphic fundamental groups and homology groups.
- Recovering a calculation of Faria Martins and Kauffman that distinguishes between the spun Hopf link and the Tube of the welded Hopf link.

All algorithms in this thesis are implemented in the `GAP` computer algebra system and are publicly available as part of the `GAP` package `HAP`.

Acknowledgements

First and foremost, I would like to thank my supervisor Professor Graham Ellis for providing me with consistent invaluable guidance throughout these four years and without whom this thesis would surely not exist.

I am grateful to my examiners Dr. Mathieu Dutour-Sikirić and Dr. John Burns for their incredibly helpful feedback on my thesis work.

I wish to express immense gratitude to the Irish Research Society for funding my PhD and affording me this opportunity to begin with.

I would like to extend my thanks to my fellow postgraduate students at the School of Mathematical & Statistical Sciences and, of course, to the extremely patient and helpful administrative staff there too.

Finally, I thank my family and friends for their constant support and encouragement.

Declaration of Authorship

I, Kelvin Killeen, certify that this thesis is all my own work and that I have not obtained a degree in this University, or elsewhere, on the basis of any of the work described in this thesis.

Signed: Kelvin Killeen

Contents

List of Figures

List of GAP Sessions

The runtimes for each of the above `GAP` sessions are given in hours (h), minutes (min), seconds (s), and milliseconds (ms). These computations were all performed on the same laptop: a Dell Latitude 5590 with an eighth generation eight core Intel i7-8650U CPU clocked at 1.9GHz, 24GB of DDR4 RAM, and a 500GB NVMe SSD.

Symbols

S^n	the unit sphere in $\mathbb{R}^{n+1}$	$\{x \in \mathbb{R}^{n+1} : \|\|x\|\| = 1\}$
B^n	the closed unit ball in $\mathbb{R}^n$	$\{x \in \mathbb{R}^n : \|\|x\|\| \leq 1\}$
$\mathring{B}^n$	the open unit ball in $\mathbb{R}^n$	$\{x \in \mathbb{R}^n : \|\|x\|\| < 1\}$
I	the unit interval	$[0, 1]$
$\simeq$	homotopy equivalence	
$\cong$	homeomorphism equivalence	
$\oplus$	direct sum	
$\otimes$	tensor product	
$\#$	connected sum	
M_N	N^{th} prime knot on M crossings	

Chapter 1

Introduction

1.1 Thesis synopsis

In this thesis, we show how the classical notions of CW-complex, cohomology with local coefficients, covering space, homeomorphism equivalence, simple homotopy equivalence, tubular neighbourhood, and Artin spinning can be encoded on a computer and used to calculate ambient isotopy invariants of continuous embeddings $N \hookrightarrow M$ of one topological manifold into another. Of particular interest are continuous cellular embeddings $S^n \hookrightarrow S^{n+2}$ of the n-sphere into the $(n+2)$-sphere for $n \in \{1, 2\}$. We describe an algorithm for computing the homology $H_n(X, A)$ and cohomology $H^n(X, A)$ of a finite connected CW-complex X with local coefficients in a $\mathbb{Z}\pi_1 X$-module A when A is finitely generated over $\mathbb{Z}$. This algorithm can be used to compute the integral cohomology $H^n(\widetilde{X}_H, \mathbb{Z})$ and induced homomorphism $H^n(X, \mathbb{Z}) \to H^n(\widetilde{X}_H, \mathbb{Z})$ for the covering map $p : \widetilde{X}_H \to X$ associated to a finite index subgroup $H < \pi_1 X$, as well as the corresponding homology homomorphism. Our implementation of this algorithm is used to show that: (i) the degree 2 homology group $H_2(\widetilde{X}_H, \mathbb{Z})$ distinguishes between the homotopy types of the complements $X \subset \mathbb{R}^4$ of the spun Hopf link and Satoh's Tube map of the welded Hopf link (whose complements have isomorphic fundamental groups and integral homology); (ii) the degree 1 homology homomorphism $H_1(p^{-1}(B), \mathbb{Z}) \to H_1(\widetilde{X}_H, \mathbb{Z})$ distinguishes between the homeomorphism types of the complements $X \subset \mathbb{R}^3$ of the granny knot and the reef knot, where $B \subset X$ is the knot boundary (again, whose complements have isomorphic fundamental groups and integral homology). We conclude by showing how our algorithms for computing regular CW-decompositions of link complements

allow us to obtain regular CW-decompositions of all closed, orientable, connected 3-manifolds by way of Dehn surgery.

Our main goals with this thesis are as follows: (i) to develop algorithms which are capable of proving some classical knot theory results; (ii) to show how one can obtain efficient regular CW-decompositions of codimension two embeddings of spaces; (iii) to provide an open-source implementation of all of these associated algorithms so that the user can experiment further.

1.2 General motivation and goals

From the very beginnings of topology a central theme has been to find invariants and determine properties captured by them, often with the ultimate goal of completely classifying certain categories of topological spaces. For example, in 1921 Brahana [1] proved that the Euler characteristic and orientability completely classify closed surfaces (proofs with varying degrees of rigour had previously been given by Möbius in 1861, Jordan in 1866, von Dyck in 1888, Dehn and Heegaard in 1907). A second example is the ongoing classification of knots in $\mathbb{R}^3$ up to ambient isotopy. Inspired by the now defunct vortex theory of the atom, in which chemical elements were to be represented by knotted tubes in the aether, Scottish physicists W. Thomson and P. G. Tait began a classification of knots with low crossing number. They were encouraged in their endeavour by James Clerk Maxwell and enlisted the help of mathematicians T. P. Kirkman and C. N. Little. The development and study of knot invariants is an ongoing and active branch of topology and very much related to the considerable ongoing efforts of mathematicians and physicists to understand more about invariants and classification of low-dimensional manifolds. Furthermore, invariants of manifolds (such as Hecke Operators on their cohomology with local coefficients) are in turn intimately connected with current number-theoretic questions concerning automorphic forms [2]. This thesis aims to provide computational tools for calculating certain invariants of manifolds.

Two desirable properties of an invariant are that it should be powerful enough to distinguish between a good range of manifolds and also be simple enough to calculate. Despite the discovery by F. Waldhausen [3] of a *complete* knot invariant capable of distinguishing between any two non-isotopic knots, the search for more easily computable invariants still continues. On the other hand, the ease of computation

```
gap> trefoil:=PureCubicalKnot(3,1);;
gap> granny:=ArcPresentation(KnotSum(trefoil,trefoil));;
gap> granny:=KnotComplementWithBoundary(granny);;
gap> reef:=ArcPresentation(KnotSum(
>    trefoil,
>    ReflectedCubicalKnot(trefoil)
> ));;
gap> reef:=KnotComplementWithBoundary(reef);;
gap> inv_granny:=Set(FirstHomologyCoveringCokernels(granny,6));;
gap> inv_reef:=Set(FirstHomologyCoveringCokernels(reef,6));;
gap> inv_granny=inv_reef;
false
```

GAP Session 1.2.1: Showing that the granny and reef knots are not ambient isotopic.
Runtime: 1min 7s 760ms.

of the Euler characteristic has lead to its widespread use as a computational tool, even in some quite unexpected settings. For instance, G. Carlsson et al. [4] have used the Euler characteristic (in the form of persistent homology) to identify the Klein Bottle surface with a space generated by many samples of 3×3 pixel arrays sampled from many digital photographs of natural images collected in Holland. In subsequent work their Klein Bottle representation of data has been used to develop novel techniques for image compression. Knot invariants have also found uses outside of mathematics. For example, the Alexander polynomial seems to be the default tool used by biologists when attempting to identify a knot in a protein molecule [5]. A goal of this thesis is to implement tools for calculating invariants of topological spaces as software that lends itself both to the computations of interest in pure mathematics and also to computations arising in an applied setting. The topological spaces we consider are *cell spaces*—spaces obtained by *gluing* together *cells* of various dimensions.

We have opted to implement our algorithms in the GAP system for computational algebra [6]. An illustration of our software is given in GAP Session 1.2.1.

This illustration computes a certain invariant $\mathfrak{I}_c(K)$ of the knot complement $S^3 \setminus K$ for the granny and reef knots of Figure 1.2.1 and proves that these two knots aren't ambient isotopic. The two knots have complements $S^3 \setminus K$ with isomorphic fundamental group $\pi_1(S^3 \setminus K)$ known as the *knot group*. Thus, the knot group fails to distinguish between the two knots, as does the classical Alexander polynomial which is derived from the knot group. Though more sophisticated polynomial invariants

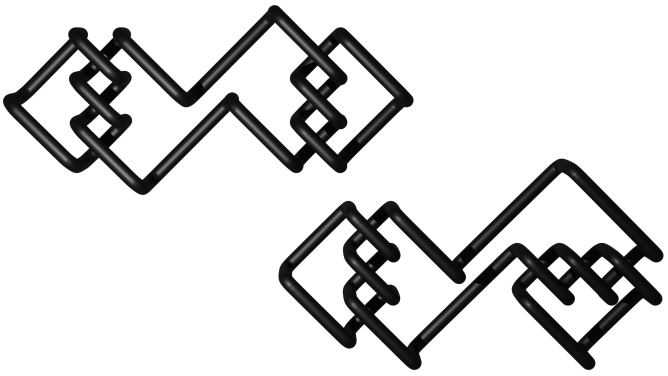


FIGURE 1.2.1: The granny and reef knots.

such as the HOMFLY polynomial or Kauffman polynomial are able to distinguish between the granny and reef knots. Restricting attention to prime knots, it is known (see for instance [7]) that neither the HOMFLY nor Kauffman polynomial distinguishes all 249 prime knots on ≤ 10 crossings. In contrast, it is shown in [8] that a weaker invariant than $\mathfrak{I}_c(K)$, depending only on the knot group, does distinguish between all prime knots on ≤ 12 crossings. Consequently, the invariant $\mathfrak{I}_c(K)$ is a useful complement to the HOMFLY and Kauffman polynomials. It is shown in [7] that the Links-Gould polynomial invariant [9] distinguishes between all 249 prime knots on ≤ 10 crossings. We do not have access to an implementation of the algorithm in [7] to evaluate the Links-Gould invariant on prime knots with 11 and 12 crossings.

Our implemented invariants are based on standard topological constructions such as fundamental groups, covering spaces, cohomology with local coefficients, boundaries of compact manifolds, tubular neighbourhoods of compact manifolds, complements of open tubular neighbourhoods, spun manifolds and so forth. These constructions apply to general cell complexes X and our implementation reflects this fact. For the most part we illustrate our invariants on complements of knots in S^3 and knotted surfaces in S^4 since codimension two embeddings are of particular interest to topologists (*cf.* Introduction to Chapter 2), but our implementation can be applied to other very different topological settings. To achieve the desired level of generality of input we intentionally avoid working directly with combinatorial notions such as knot diagrams and broken surface diagrams except to the extent that algorithms for converting such combinatorial descriptors into cell complexes are implemented. Readers who wish to use the implementation in other topological settings need only provide algorithms for converting descriptors from those settings into cell complexes. The topological/geometric interpretation of our invariants is immediate from their

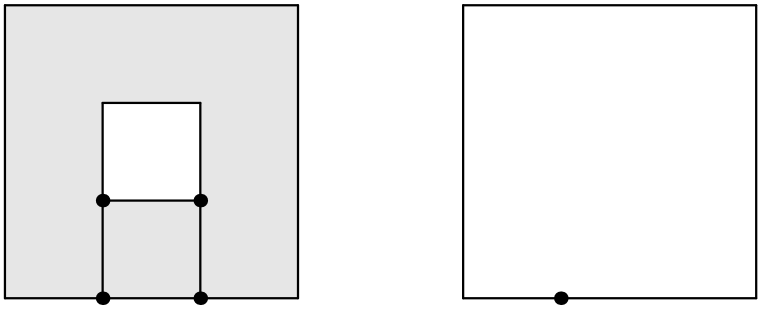


FIGURE 1.2.2: A regular CW-complex $\mathbb{S}$ and a non-regular CW-complex $\mathfrak{S}$.

definitions, in contrast to invariants defined using skein relations. In the case of polynomial invariants that have a topological interpretation it should be possible to implement them using the approach of cell complexes. Indeed, the Alexander Polynomial is implemented this way in `HAP` [10].

One example where our software could find use in the future is in relation to the Berge Conjecture and variants of this conjecture. Quoting from J. Greene [11] who made progress on this conjecture:

> What are all the ways to produce the simplest closed 3-manifolds by the simplest 3-dimensional topological operation? From the cut-and-paste point of view, the simplest 3-manifolds are the lens spaces $L(p, q)$, these being the spaces (besides S^3 and $S^1 \times S^2$) that result from identifying two solid tori along their boundaries, and the simplest operation is Dehn surgery along a knot $K \subset S^3$. With these meanings in place, the opening question goes back forty years to Moser, and its definitive answer remains unknown.

Algorithms implemented as part of this thesis form part of a pipeline that can be used to determine the topological type of a Lens space formed from Dehn surgery along a knot $K \subset S^3$. See the `HAP` manual [10] for details of the entire pipeline. The pipeline could potentially be used to generate examples related to the Berge conjecture, and to variants of the question concerning knotted surfaces $K \subset S^4$.

A first issue to address in any machine computation of topological invariants of a cell space X is the machine representation of X. Let us briefly consider four possible representations, and then opt for a fifth which seems better suited to our needs. Consider the torus $\mathbb{T} = S^1 \times S^1$ with CW-structure involving a single 0-cell, two

1-cells, and a single 2-cell obtained by taking the product of two copies of the CW-complex S'' shown in Figure 1.2.2 (right). One possibility is to represent $\mathbb{T}$ as a free presentation of its fundamental group, involving 2 generators and 1 relator. This representation is readily adapted to 2-dimensional CW-complexes with more than one 0-cell, and is used by [12] in their algorithm for computing fundamental groups and finite covers of 2-dimensional cell complexes. However, the notion of a group presentation does not readily adapt to higher dimensional CW-complexes such as the product $S^1 \times S^1 \times S^1 \times S^1$ of four copies of the CW-complex $\mathfrak{S}$. The difficulty with implementing higher dimensional CW-complexes is the question of representing how cells are attached. Consider for instance the complex projective plane $P^2(\mathbb{C})$ which admits a CW-structure $P^2(\mathbb{C}) = e^0 \cup e^2 \cup e^4$ with just three cells of dimensions 0, 2 and 4 respectively. How should we record the manner in which e^4 is attached to $e^0 \cup e^2$? A second possibility is to subdivide the CW-structure on $\mathbb{T} = S^1 \times S^1$ in a way that produces a simplicial complex. It is possible to triangulate $\mathbb{T}$ using 7 vertices, 21 edges and 14 triangular faces; the resulting simplicial complex can be stored as a collection of subsets of the vertex set. The approach readily generalises to CW-complexes of arbitrary dimension, but the number of simplices can become prohibitively large. For instance, the n-sphere S^n admits a CW-structure with just 2 cells, whereas any homotopy equivalent simplicial complex requires at least $2^{n+2} - 2$ simplices. A third possibility is to obtain smaller simplicial cell structures by relaxing the requirements of a simplicial complex to those of a simplicial set. The n-sphere can be represented as a simplicial set with just two non-degenerate cells. Simplicial sets, their fundamental groups, and their (co)homology with trivial coefficients have been implemented by John Palmieri in `SAGE` [13]. We opt against using simplicial sets as the setting for our algorithms because it seems to be a non-trivial task to find small simplicial set representations of some of the CW-complexes of interest to us, such as CW-complexes arising as the complements of knotted surfaces in S^4. An alternative simplicial approach, and the one used in the `SnapPy` software system [14] for studying 3-manifolds, is to represent a space X as a simplicial complex corresponding to a fundamental domain for the action of its fundamental group on its universal cover $\widetilde{X}$. This is a standard method for representing 3-manifolds, but it does not readily lend itself to more general spaces. In this thesis we opt to represent a CW-complex X as a *regular* CW-complex Y (*i.e.* one whose attaching maps restrict to homeomorphisms on cell boundaries) together with a homotopy equivalence $Y \simeq X$. More specifically, we work with a regular CW-complex Y, rather than a simplicial complex, and endow it with an *admissible discrete vector field*

whose *critical cells* are in one-one correspondence with the cells of X. Definitions of italicised terms are given in the thesis. The *arrows* in a discrete vector field represent elementary simple homotopies, and can be viewed as analogues of the degeneracy maps of a simplicial set. Regular CW-complexes have the advantage that they are determined up to homeomorphism by their face lattice and can thus be stored by listing the $(n-1)$-cells incident with each cell of dimension n. The homotopy equivalence $h\colon Y \stackrel{\simeq}{\to} X$ is represented as an *admissible discrete vector field* on Y. For instance, in `HAP` we can represent $X = P^2(\mathbb{C}) = e^0 \cup e^2 \cup e^4$ as a regular CW-complex Y with 111 cells together with the homotopy equivalence $h\colon Y \stackrel{\simeq}{\to} X$ to the non-regular space X with 3 cells.

This representation of spaces as a homotopy equivalence $h\colon Y \stackrel{\simeq}{\to} X$ with Y a finite regular CW-complex is used extensively in [8]. One contribution of this thesis is to adapt the representation to the case of the universal covering space $\widetilde{X}$ which will have infinitely many cells if the fundamental group $\pi_1 X$ is infinite. Using the group action of $\pi_1 X$ on the universal cover we implement a representation as a homotopy equivalence $h\colon \widetilde{Y} \stackrel{\simeq}{\to} \widetilde{X}$. The group $\pi_1 X$ is stored as a finitely presented group and the space $\widetilde{Y}$ is stored as an equivariant regular CW-complex with just finitely many orbits of cells.

The typically non-regular CW-complex X is used to make computations efficient. For instance, we might be faced with some topological construction that can be implemented as a cellular map $f\colon Y \to Y'$ of regular CW-complexes. In many instances the construction of f will not be functorial but will be 'functorial up to homotopy' and induce for example a functorial map $H^*(f)\colon H^*(Y', A) \to H^*(Y, A)$ in cohomology with local coefficients in a $\pi_1 Y'$-module A. It might be that the regular CW-complexes Y and Y' are extremely large and computing their cohomology is prohibitively expensive. In this scenario we would aim to use the homotopy equivalences $Y \stackrel{\simeq}{\to} X$ and $Y' \stackrel{\simeq}{\to} X'$ to provide a *lazy* implementation of the diagram of chain complexes

$$\begin{array}{ccc} C_*\widetilde{Y} & \xrightarrow{C_*(f)} & C_*\widetilde{Y'} \\ \uparrow \simeq & & \simeq \downarrow \\ C_*\widetilde{X} & & C_*\widetilde{X'} \end{array}$$

in which the images of elements in the chain groups are computed only when required. Here the chain groups are free abelian of possibly infinite rank, but they are finitely generated free π_1-modules assuming X, X' have only finitely many cells. On

passing to cohomology we would then get a lazy implementation of the diagram

$$\begin{array}{ccc} H^*(Y,A) & \xleftarrow{H^*(f)} & H^*(Y',A) \\ \downarrow \simeq & & \simeq \uparrow \\ H^*(X,A) & & H^*(X',A). \end{array}$$

Assuming that the non-regular CW-complexes X and X' are small enough for their cohomology to be computed using the Smith Normal Form algorithm then we can attempt to evaluate the homomorphism $H^*(f)\colon H^*(Y',A) = H^*(X',A) \to H^*(X,A) = H^*(X',A)$ on each of the finitely many generators of $H^*(X',A)$. Note that this strategy requires having explicit homotopy equivalences $Y \to X$, $Y' \to X'$ to hand. It does not suffice to have just an efficient (sparse) implementation of the Smith Normal Form to hand. See Section 2.3 for a specific illustration.

A second issue to address in machine computations of topological invariants is the credibility of the computed results. How can we be confident that they are correct? As with any mathematical work, our confidence is in part based on a careful mathematical explanation/proof of the mathematical method. But computer implementations of correct methods may have bugs that even a careful reading of the code will not discover. The situation is comparable to long and complicated mathematical proofs in highly cited mathematical publications that may contain some subtle error at some step which has remained undetected by the many citing mathematicians. Indeed, such a subtle error in one of his highly cited papers led Vladimir Voevodsky to work on Univalent Foundations[15] with the aim of getting computers to spot the subtle errors in mathematical proofs. It is easier to gain confidence in computer implementations of correct methods. There are two standard approaches, both of which have been used in this thesis work; in particular, they have been used to check our computer proofs of Theorems 3.3.1 and 4.2.1. Firstly, one can apply the implementation to substantially different input data that theoretically should yield the same output. Secondly, one can feed some input data to two substantially different implementations of methods for computing the task to hand.

Consider the above computation of the invariant $\mathfrak{I}_c(K)$ for the granny knot K. The input is any arc presentation of the granny knot. For instance, we could input both of the arc presentations of Figure 1.2.3 and observe that we get the same value for the invariant in each case.

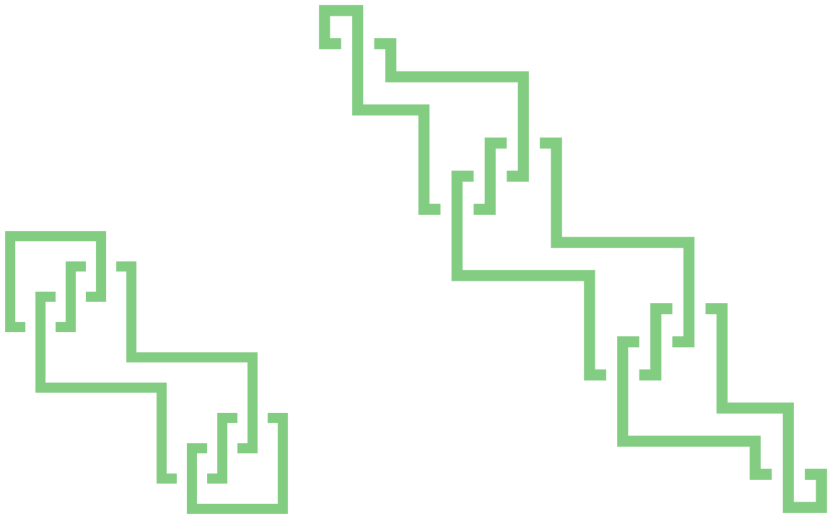


FIGURE 1.2.3: Two arc presentations for the granny knot.

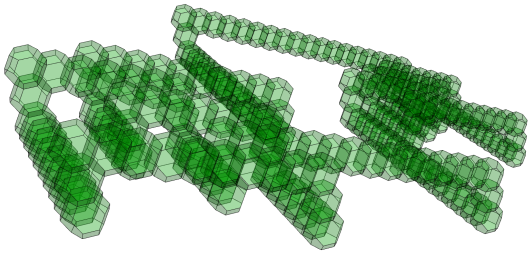


FIGURE 1.2.4: A polyhedral complex representation of the granny knot.

In each case the computer uses an algorithm described in the thesis to construct a regular CW-complex structure on the space $X = S^3 \setminus K$. The two inputs will yield substantially different (though homeomorphic) CW-complexes X. The computer then constructs many 6-fold covering spaces of X and outputs abelian groups derived from these covers. A bug in the implementation would most likely cause the computer to crash, and if not crash then cause the computer to return distinct values of the invariant. We gain significant confidence in the implementation from the observation that the invariant $\mathfrak{I}_c(K)$ is the same in both cases.

A second approach to gaining confidence in the implementation is to represent the granny knot as a polyhedral complex such as the one in Figure 1.2.4. The `HAP` package has two types of polyhedral complex implemented—one in which each top-dimensional cell is an n-dimensional cube, and one in which each top-dimensional cell is an n-dimensional permutahedron. Figure 1.2.4 shows a 3-dimensional permutahedral complex. Euclidean space $\mathbb{R}^3$ has a tessellation by permutahedra and the knot K is represented as a subspace of $\mathbb{R}^3$ consisting of finitely many permutahedra.

Let S be any contractible union of finitely many of the tessellating permutahedra such that K lies in the interior of S. The space $X = S \setminus \mathring{K}$ is again a finite permutahedral complex and, as such, can be converted to the datatype of a regular CW-complex. The CW-complex X has 13228 cells of dimensions ≤ 3. The corresponding CW-complex X in `GAP` Session 1.2.1 has just 759 cells and illustrates an advantage to working with CW-complexes rather than polyhedral complexes (there are also advantages to working with permutahedral complexes when constructing cell complexes for topological data analysis since the Euclidean metric can be used to compute filtrations needed for persistent homology techniques). The regular CW-complex X with 13228 cells can now be used to recalculate the invariant $\mathfrak{I}_c(K)$, lending further confidence to the original computation in `GAP` Session 1.2.1.

1.3 Chapter summaries

- For the remainder of Chapter 1, we establish standard notation and recall fundamental definitions that will be used throughout.
- In Chapter 2 we introduce algorithms to construct the chain complex of the universal cover $C_*\widetilde{X}$ of a finite connected regular CW-complex X and use this to obtain the homology and cohomology of X with local coefficients in a finitely generated $\mathbb{Z}\pi_1 X$-module.
- In Chapter 3 we introduce an ambient isotopy invariant of knotted surfaces capable of proving the inequivalence between two surfaces with isomorphic integral homology and fundamental groups. This invariant is computed in the context of pure cubical complexes, highlighting the need for smaller cell structures.
- In Chapter 4 we describe a procedure for obtaining small regular CW-decompositions of knot complements. We introduce an ambient isotopy invariant of knots which is capable of exposing a classical knot inequivalence between the granny and reef knots.
- Chapter 5 concerns a variety of classical topological notions that necessitate computer implementation in order to extend our embeddings of knots into S^3 to embeddings of knotted surfaces into S^4.

- In Chapter 6 we describe our computer implementation of Satoh's Tube map that obtains regular CW-decompositions of ribbon torus-knots in B^4 from virtual knot diagrams.
- In Chapter 7 we illustrate an algorithmic implementation of Dehn surgery. This modifies our existing regular CW-decomposition of link complements in S^3 in order to obtain an arbitrary closed, orientable, connected 3-manifold. We illustrate Dehn surgery on the trivial knot in S^3 and produce some Lens spaces.

1.4 Background material review

1.4.1 Homological algebra

Definition 1.4.1.1. [16] Let R denote a unital ring. A *left R-module M* consists of an abelian group $(M, +)$ together with an operation $\cdot : R \times M \to M$ such that for all r, s in R and all x, y in M, we have

(i) $r \cdot (x + y) = r \cdot x + r \cdot y$

(ii) $(r + s) \cdot x = r \cdot x + s \cdot x$

(iii) $(rs) \cdot x = r \cdot (s \cdot x)$

(iv) $1 \cdot x = x$.

Definition 1.4.1.2. [16] Let R denote a unital ring. A *right R-module M* consists of an abelian group $(M, +)$ together with an operation $\cdot : M \times R \to M$ such that for all r, s in R and all x, y in M, we have

(i) $(x + y) \cdot r = x \cdot r + y \cdot r$

(ii) $x \cdot (r + s) = x \cdot r + x \cdot s$

(iii) $x \cdot (rs) = (x \cdot r) \cdot s$

(iv) $x \cdot 1 = x$.

Definition 1.4.1.3. [16] Let A denote a right R-module and let B denote a left R-module. The *tensor product* $A \otimes_R B$ of A and B is an R-module with generators $a \otimes b$ for $a \in A$ and $b \in B$ such that the following distributivity and associativity relations hold:

(i) $(a_1 + a_2) \otimes b = a_1 \otimes b + a_2 \otimes b$ and $a \otimes (b_1 + b_2) = a \otimes b_1 + a \otimes b_2$.

(ii) $ar \otimes b = a \otimes rb$ for $r \in R$.

Definition 1.4.1.4. [16] A *chain complex* (C_*, ∂_*) is a sequence of R-modules and R-module homomorphisms (called boundary homomorphisms)

$$\cdots \xrightarrow{\partial_{n+2}} C_{n+1} \xrightarrow{\partial_{n+1}} C_n \xrightarrow{\partial_n} C_{n-1} \xrightarrow{\partial_{n-1}} \cdots$$

such that $\partial_n \partial_{n+1} = 0$ for all n. A chain complex is of length n if $C_m = 0$ for $m > n$ and $C_n \neq 0$. Chain complexes of free R-modules are referred to as a *free chain complexes*.

Definition 1.4.1.5. [16] The n^{th} *homology module* of the chain complex (C_*, ∂_*) is the quotient module

$$H_n(C_n) = \ker \partial_n / \operatorname{im} \partial_{n+1}.$$

Definition 1.4.1.6. [16] A *cochain complex* (C^*, ∂^*) is a sequence of R-modules and R-module homomorphisms (called coboundary homomorphisms)

$$\cdots \xleftarrow{\partial^{n+1}} C^{n+1} \xleftarrow{\partial^n} C^n \xleftarrow{\partial^{n-1}} C^{n-1} \xleftarrow{\partial^{n-2}} \cdots$$

such that $\partial^{n+1} \partial^n = 0$ for all n.

Definition 1.4.1.7. [16] The n^{th} *cohomology module* of the cochain complex (C^*, ∂^*) is the quotient module

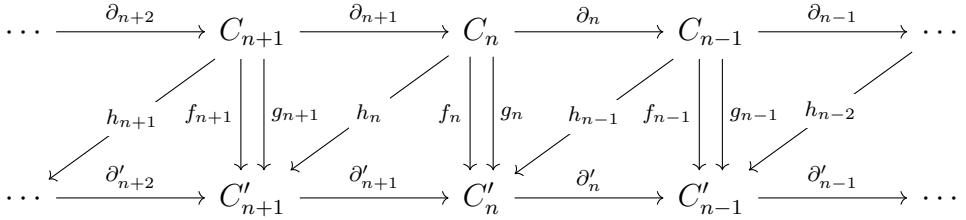
$$H^n(C^n) = \ker \partial^n / \operatorname{im} \partial^{n-1}.$$

Definition 1.4.1.8. [16] Let (C_*, ∂_*) and (C'_*, ∂'_*) be chain complexes of R-modules. A *chain map* $f_* : C_* \to C'_*$ is a sequence of R-module homomorphisms such that

$$\begin{array}{ccccccccc}
\cdots & \xrightarrow{\partial_{n+2}} & C_{n+1} & \xrightarrow{\partial_{n+1}} & C_n & \xrightarrow{\partial_n} & C_{n-1} & \xrightarrow{\partial_{n-1}} & \cdots \\
 & & \downarrow{\scriptstyle f_{n+1}} & & \downarrow{\scriptstyle f_n} & & \downarrow{\scriptstyle f_{n-1}} & & \\
\cdots & \xrightarrow{\partial'_{n+2}} & C'_{n+1} & \xrightarrow{\partial'_{n+1}} & C'_n & \xrightarrow{\partial'_n} & C'_{n-1} & \xrightarrow{\partial'_{n-1}} & \cdots
\end{array}$$

is a commutative diagram.

Definition 1.4.1.9. [8] Two chain maps $f_*, g_* : C_* \to C'_*$ are *chain homotopic* if there exists a sequence of R-linear homomorphisms $h_n : C_n \to C'_{n+1}$



satisfying

$$\partial'_{n+1}h_n + h_{n-1}\partial_n = f_n - g_n.$$

We write $f_* \simeq g_*$ to denote chain homotopic chain maps and call the collection of homomorphisms $\{h_n\}$ a *chain homotopy*.

Definition 1.4.1.10. [8] Two chain complexes (C_*, ∂_*) and (C'_*, ∂'_*) are *chain equivalent* if there are chain maps $f : C_* \to C'_*$ and $g : C'_* \to C_*$ such that $fg \simeq 1_{C'_*}$ and $gf \simeq 1_{C_*}$.

1.4.2 Cell complexes

Definition 1.4.2.1. [16] A *CW-complex* X can be constructed as follows:

- Begin with a discrete set of points X^0 whose elements are referred to as 0-cells.
- Obtain the space X^n from X^{n-1} by attaching n-cells e^n_α by way of continuous maps $\phi : \mathbb{D}^n \to X^n$ which restrict to $\phi|_{S^{n-1}} : S^{n-1} \to X^{n-1}$. Thus, $X^n = X^{n-1} \coprod_\alpha e^n_\alpha$, where each of the e^n_α are n-dimensional open balls (referred to as the *cells* of X).
- After finitely many steps, one can set $X = X^n$ for some finite n, or one can continue indefinitely, setting $X = \bigcup_{n\in\mathbb{N}} X^n$. The topology inherited in the latter case is the weak topology.

A CW-complex is called *regular* if each attaching map ϕ as described above is a homeomorphism onto the closure of the cell e^n_α.

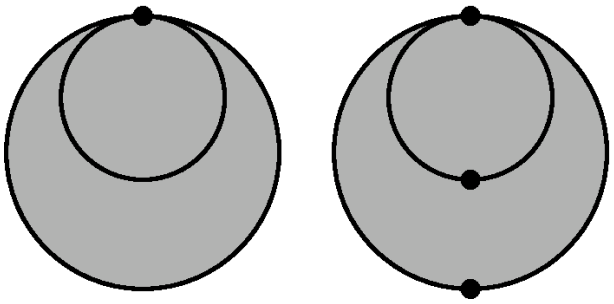


FIGURE 1.4.2.1: A non-regular CW-complex (left) and a regular CW-complex (right) of a contractible space.

Proposition 1.4.2.1. *[8] Let X be a regular CW-complex with some ordering on its cells and let C_n be the free abelian group with free basis consisting of one generator for each n-cell of X. There are unique integers $\epsilon^n_{i,j}$ and a unique series of homomorphisms $\partial_n : C_n \to C_{n-1}$ satisfying*

(i) $\partial_n \partial_{n+1} = 0$ *for* $n \geq 1$*;*

(ii) $\partial_n(e^n_\lambda) = \sum_{e^{n-1}_\mu \subset X} \epsilon^n_{\lambda,\mu} e^{n-1}_\mu$ *where* e^{n-1}_μ *ranges over all* $(n-1)$*-cells of* X *and*

$$\epsilon^n_{\lambda,\mu} = \begin{cases} \pm 1 & \text{if } e^{n-1}_\mu \text{ lies in the boundary of } e^n_\lambda, \\ 0 & \text{otherwise;} \end{cases}$$

(iii) $\epsilon^1_{\lambda,\mu}\epsilon^1_{\lambda,\mu'} = -1$ *whenever two vertices* e^0_μ, $e^0_{\mu'}$ *are incident with a common 1-cell* e^1_λ.

We refer to the chain complex (C_*, ∂_*) as the cellular chain complex of X.

Definition 1.4.2.2. [8] A *discrete vector field* on a CW-complex X is a collection of arrow symbols $s \to t$ where

- s, t are cells of X with $\dim(t) = \dim(s) + 1$ and with s lying in the boundary of t. We say that s and t are *involved* in the arrow, that s is the source of the arrow and that t is the target.
- Any cell is involved in at most one arrow.

A *chain* in a discrete vector field is a sequence of arrows

$$\ldots, s_1 \to t_1, s_2 \to t_2, s_3 \to t_3, \ldots$$

where s_{i+1} lies in the boundary of t_i for all i. A chain is a *circuit* if it is of finite length and source s_1 of the initial arrow lies in the boundary of the target t_n of the final arrow. An *admissible* discrete vector field contains no circuits nor any chains which extend infinitely to the right. A cell in a discrete vector field is said to be *critical* if it is not involved in any arrow.

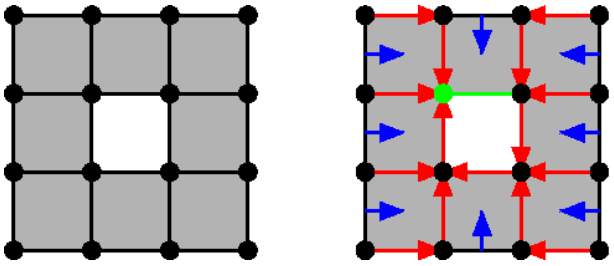


FIGURE 1.4.2.2: A regular CW-complex homotopy equivalent to the circle and an admissible discrete vector field on it with one critical 0-cell and one critical 1-cell (shown in green).

Theorem 1.4.2.2. *[8] If X is a regular CW-complex with admissible discrete vector field then there is a homotopy equivalence*

$$X \simeq Y$$

where Y is a CW-complex whose cells are in one-to-one correspondence with the critical cells of X.

Definition 1.4.2.3. [8] A *discrete vector field on a chain complex* (C_*, ∂_*) of free R-modules consists of a choice of basis $\underline{b}_n$ for each R-module C_n, $n \geq 0$, together with a collection of formal arrows $s \to t$ where

- $s \in \underline{b}_n$, $t \in \underline{b}_{n+1}$ for some $n \geq 0$ and

$$\partial_{n+1} t = \lambda_s s + \sum_{e \in \underline{b}_n \setminus \{s\}} \lambda_e e$$

  with $\lambda_s, \lambda_e \in R$ and with λ_s invertible. We say that s and t are *involved* in the arrow, that s is the *source* of the arrow, and that t is the *target* of the arrow.

- any basis element $e \in \underline{b}_n$, $n \geq 0$, is involved in at most one arrow.

A discrete vector field is said to be *finite* if it contains only finitely many arrows. For $w \in C_n$ and $e \in \underline{b}_n$ we define $w_e \in R$ by the expression $w = \sum_{e \in \underline{b}_n} w_e e$ and say

that w_e is the coefficient of e in w. A *chain* in a discrete vector field is a sequence of arrows

$$\ldots, s_1 \to t_1, s_2 \to t_2, s_3 \to t_3, \ldots$$

where s_{i+1} has an invertible coefficient in the boundary of t_i for each i. A chain is said to be a *circuit* if it is of finite length with source s_1 of the initial arrow $s_1 \to t_1$ having an invertible coefficient in the boundary of the target t_n of the final arrow $s_n \to t_n$. A discrete vector field is said to be *admissible* if it contains no circuits and no chains that extend indefinitely to the right. A basis element $e \in \underline{b}_n$ is said to be *critical* if it is not involved in any arrow.

Theorem 1.4.2.3. *[8] Let C_* be a free chain complex of a regular CW-complex endowed with a discrete vector field. There exists a homotopy equivalence*

$$C_* \simeq D_*$$

with D_* a chain complex in which D_n is the free module generated by critical elements of the basis $\underline{b}_n \subset C_n$, $n \geq 0$.

Definition 1.4.2.4. [16] A *covering space* of a space X is a space $\widetilde{X}$ together with a map $p : \widetilde{X} \to X$ satisfying the following condition: each point $x \in X$ has an open neighbourhood U in X such that $p^{-1}(U)$ is a union of disjoint open sets in $\widetilde{X}$, each of which is mapped homeomorphically onto U by p.

Definition 1.4.2.5. [16] The *universal cover* of a space X is a path-connected covering space of X with trivial fundamental group.

We assume that the reader is familiar with the notion of the fundamental group $\pi_1(X, x_0)$ of a topological space X. When X is a regular CW-complex, we assume some 0-cell x_0 has been chosen as its base point. We assume that the reader knows that the elements of $\pi_1 X$ can be represented as paths in the 1-skeleton X^1 and, in turn, that these paths can be represented as sequences of oriented edges. Lastly, we assume familiarity with the free and cellular action of the fundamental group of a space on its universal cover $\widetilde{X}$.

1.4.3 Knot theory

Definition 1.4.3.1. [17] A *knot* $\kappa : S^1 \to \mathbb{R}^3$ is an embedding of the circle into three-dimensional Euclidean space.

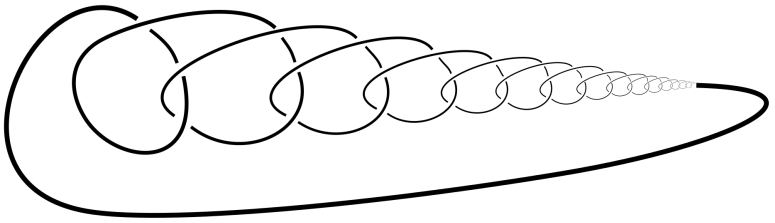


FIGURE 1.4.3.1: A wild knot with infinitely many crossings.

Definition 1.4.3.2. [17] A *link* $\kappa : \bigsqcup_\alpha S^1 \to \mathbb{R}^3$ is an embedding of a union of disjoint circles into three-dimensional Euclidean space. A link is *finite* if it is an embedding of finitely many copies of the circle. We use *link* to refer to finite links as well as knots.

Definition 1.4.3.3. [8] Two links κ, κ' are said to be *ambient isotopic* if there exists a continuous map $H : \mathbb{R}^3 \times [0,1] \to \mathbb{R}^3$ such that $H_t(x) = H(x,t)$ is a homeomorphism from $\mathbb{R}^3$ to itself for each $t \in [0,1]$, $H_0(x)$ is the identity, and $\kappa'(y) = H_1(\kappa(y))$ for $y \in S^1$.

Definition 1.4.3.4. [8] A link is *tame* if it is ambient isotopic to a piecewise linear embedding $\kappa : S^1 \to \mathbb{R}^3$ involving only finitely many linear pieces. We will only consider tame links in this thesis as the alternative, *wild links*, likely do not yield finite CW-decompositions.

Definition 1.4.3.5. [8] For any tame link κ we can choose an affine projection $\pi : \mathbb{R}^3 \to \mathbb{R}^2$ onto some hyperplane $\mathbb{R}^2 \subset \mathbb{R}^3$ not containing any point in the link image, such that the composite $\pi\kappa$ is one-one at all but finitely many points $x \in S^1$ called *crossing points*; this can always be done such that $\pi\kappa : S^1 \to \mathbb{R}^2$ is two-one on the set of crossing points. For any pair of crossing points with identical image in the hyperplane $\mathbb{R}^2$, one of the points will be nearer to the hyperplane than the other. The nearer point is called an *undercrossing point*, and the other is called an *overcrossing point*. The image in $\mathbb{R}^2$ of a crossing point is called a *crossing*. When sketching the planar image of $\pi\kappa$, we instead sketch the restriction $\pi\kappa : S^1_* \to \mathbb{R}^2$ to a subset S^1_* of S^1 obtained by removing small neighbourhoods of undercrossing points. We refer to the collection of non-intersecting simple arcs in the plane $\pi\kappa(S^1_*)$ as a *link diagram*.

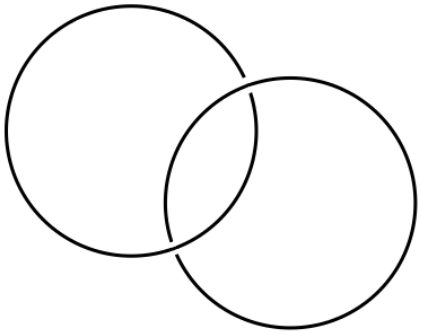


FIGURE 1.4.3.2: A link diagram of the Hopf link.

Definition 1.4.3.6. [17] A *welded link* is an immersion of a union of circles into three-dimensional Euclidean space. It differs from a traditional link in that there is the potential for self-intersections. The points at which these self-intersections occur are referred to as *welded crossings*. The corresponding notion of a *welded link diagram* is similarly defined. In sketching a welded link diagram, we illustrate welded crossings by a shaded circle. Alternatively, the welded crossings can be drawn as intersecting arcs in contrast to undercrossing and overcrossing points which are represented via the omission of a neighbourhood of points.

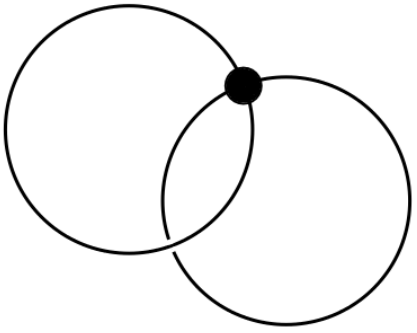


FIGURE 1.4.3.3: A welded link diagram of the welded Hopf link.

Definition 1.4.3.7. [8] An *arc presentation* is a link diagram such that:

(i) it is piecewise linear, consisting only of vertical and horizontal segments

(ii) no two vertical segments share a common x-coordinate nor do any two horizontal segments share a common y-coordinate

(ii) all overcrossing points belong to vertical segments and all undercrossing points belong to horizontal line segments.

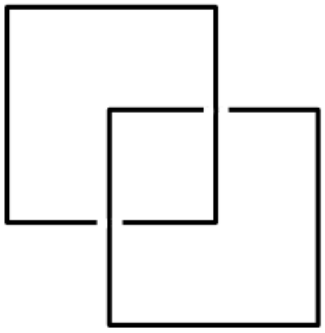


FIGURE 1.4.3.4: An arc presentation of the Hopf link.

Definition 1.4.3.8. [17] Let κ_1 and κ_2 denote two knots. We form their *knot sum* $\kappa_1 \# \kappa_2$ by removing an open ball from each of them and gluing the resulting manifolds with boundary together along the new spherical boundary components. Knots that do not arise as the knot sum of two other knots are referred to as *prime knots*.

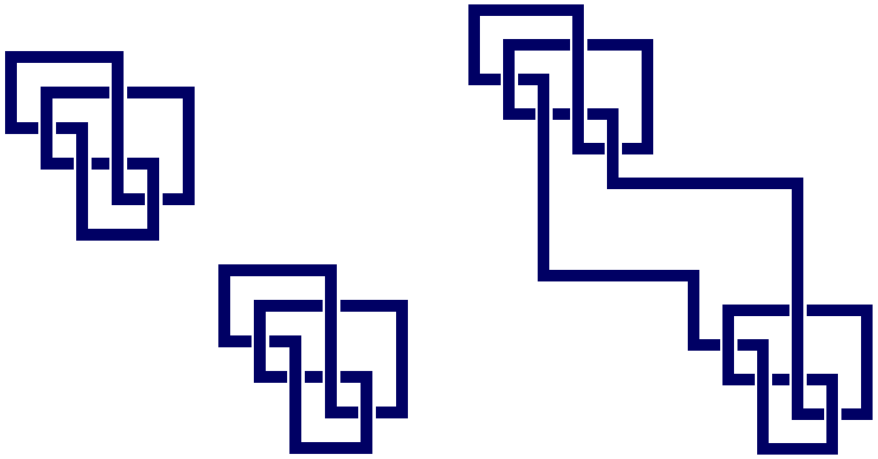


FIGURE 1.4.3.5: Forming the knot sum $4_1 \# 4_1$.

Definition 1.4.3.9. [16] An *n-manifold* is a topological space X such that for each point $x \in X$ there exists an open set U that contains x and is homeomorphic to $\mathbb{R}^n$. A 2-manifold is known as a *surface*.

Definition 1.4.3.10. [16] In an n-dimensional topological space X, a point $x \in X$ is *singular* if there are no open sets U with $x \in U$ and U homeomorphic to $\mathbb{R}^n$.

Definition 1.4.3.11. A *self-intersecting surface* is a 2-dimensional topological space that is otherwise a manifold but for some set of singular points that is either empty or a disjoint union of 1-spheres S^1.

Chapter 2

Computing local cohomology

Let $\iota : N \to M$ be a continuous cellular embedding of a finite CW-complex N into a finite CW-complex M. We are concerned with computing ambient invariants of such embeddings so that we may distinguish between their isotopy types. These invariants are of most interest in the case where N and M are manifolds. For embeddings of spheres $S^n \to S^{n+k}$ we focus on the case $k = 2$ thanks to a result of [18] stating that any two embeddings are ambient isotopic (in the piecewise linear setting) for $k \geq 3$, and a result of [19] implying that there is just one isotopy class of embeddings for $k = 1$. However, Zeeman also shows for instance that there is more than one embedding $S^n \to S^{2n}$, up to ambient isotopy, for $n \geq 1$ (again in the piecewise linear setting).

The invariants that we wish to calculate are the integral homology groups $H_n(\widetilde{X}_H, \mathbb{Z})$ of finite covers of the complement $X = M \setminus N$, where H denotes the finite index subgroup of $\pi_1 X$ arising as the image of the induced homomorphism of fundamental groups $p_H : \pi_1 \widetilde{X}_H \to \pi_1 X$. By considering an open tubular neighbourhood $N_\epsilon \supset N$ with boundary ∂N_ϵ, we obtain the inclusion map $\partial N_\epsilon \hookrightarrow X$. Letting $B = p_H^{-1}(\partial N_\epsilon)$ denote the preimage in $\widetilde{X}_H$ of the boundary, we wish to calculate the induced homology homomorphisms $H_n(B, \mathbb{Z}) \to H_n(\widetilde{X}_H, \mathbb{Z})$ so that we may distinguish between embeddings of manifolds.

It is worth noting that this use of homology of finite covers as an ambient isotopy invariant is well-documented in the literature. One of the earliest invariants in the study of knot embeddings $S^1 \hookrightarrow S^3$ was the first integral homology $H_1(\widetilde{(S^3 \setminus S^1)}_H, \mathbb{Z})$ of finite coverings of knot complements. Methods for computing these first homology groups directly from planar knot diagrams can be found in [20], [21], [22], and [23].

The usefulness of first homology of finite coverings in knot theory hints at an analogous role for higher homology of coverings in the study of higher dimensional embeddings, especially for those of codimension two. We will describe computer processes that aid in investigating the potential of this analogous role.

Let $C_*\widetilde{X}$ denote the cellular chain complex of the universal cover of a connected CW-complex X. This is a chain complex of free $\mathbb{Z}G$-modules and $\mathbb{Z}G$-equivariant module homomorphisms with $G = \pi_1 X$, the fundamental group of X. Given a $\mathbb{Z}G$-module A, the *homology* and *cohomology* of X *with local coefficients* in A is defined as

$$H_n(X, A) = H_n(C_*\widetilde{X} \otimes_{\mathbb{Z}G} A) \text{ and } H^n(X, A) = H^n(\mathrm{Hom}_{\mathbb{Z}G}(C_*\widetilde{X}, A)). \tag{2.1}$$

Note that $H_n(\widetilde{X}_H, \mathbb{Z}) = H_n(X, A)$ when $A = \mathbb{Z}G \otimes_{\mathbb{Z}H} \mathbb{Z}$. We present an algorithm for computing (2.1) in the case where A is a finitely generated $\mathbb{Z}G$-module. This algorithm takes as its input a CW-complex X and details of A. In order to apply this algorithm to embeddings of manifolds, we are required to provide algorithms for converting descriptions of such embeddings to embeddings of CW-complexes. More specifically, we present algorithms for constructing an embedding $N \hookrightarrow S^{n+2}$ of an n-dimensional manifold N into an $(n+2)$-sphere for $n \in \{1, 2\}$, namely where N is a link or knotted surface.

2.1 Representation of CW-complexes

Before tackling the computation of (co)homology with local coefficients it is necessary to establish how we will obtain a computer representation of the CW-complex X. As justification for our chosen implementation of CW-complexes, let us first consider three alternative methods. For example, consider the Klein bottle K with CW-structure involving one 0-cell, two 1-cells and one 2-cell. One could represent K as a free presentation of its fundamental group, involving two generators and one relator. This representation is readily adapted to 2-dimensional CW-complexes with more than one 0-cell, and is used by [24] in their algorithm for computing fundamental groups and finite covers of 2-dimensional cell complexes. However, this group presentation does not readily adapt to higher dimensional CW-complexes such as $K \times K$. A second potential representation arises from subdividing the CW-structure on K as to obtain a simplicial complex. We can triangulate K using 8 vertices, 24

edges, and 16 faces; the resulting simplicial complex can be stored as a collection of subsets of the vertex set. This method does not have the restriction of the prior

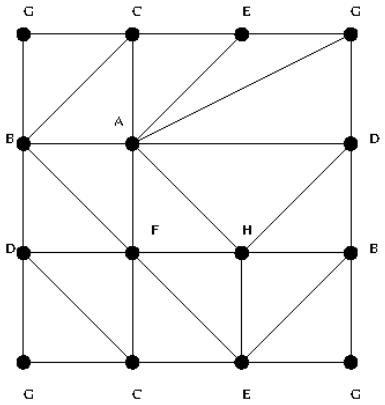


FIGURE 2.1.1: Simplicial complex homotopy equivalent to the Klein bottle K.

method and can easily be extended to CW-complexes of arbitrary dimension, but the number of simplices can become prohibitively large. For example, the n-sphere S^n admits CW-structure with two cells whereas any homotopy equivalent simplicial complex will have at least $2^{n+2} - 2$ simplices. Another possibility is to reduce the size of the simplicial cell structure by relaxing the requirements of a simplicial complex to those of a simplicial set. The n-sphere can be represented as a simplicial set with just two non-degenerate cells. A computer implementation of simplicial sets, their fundamental groups, and their (co)homology with trivial coefficients is available in the `SAGE`[25] language. Our reasoning for opting against using simplicial sets in our algorithms is that it seems to be a nontrivial task to find small simplicial set representations of the CW-complexes of interest to us, *i.e.*, CW-complexes arising as the complements of knotted surfaces in S^3. This leads us to our choice of a representation for CW-complexes: regular CW-complexes. We choose to represent a CW-complex X as a regular CW-complex Y together with a simple homotopy equivalence $Y \simeq X$. More specifically, we work with a regular CW-complex Y endowed with an admissible discrete vector field whose critical cells are in one-to-one correspondence with the cells of X. The arrows in a discrete vector field are elementary simple homotopies and can be viewed analogously to the degeneracy maps of a simplicial set. We will now exhibit how regular CW-complexes and discrete vector fields can be encoded on a computer.

Datatype 2.1.1. [8] A regular CW-complex can be represented as a component object `X` with the following components:

- `X!.boundaries[n+1][k]` is a list of integers $[t, a_1, a_2, \ldots, a_t]$ recording that the ${a_i}^{\text{th}}$ cell of dimension $n-1$ lies in the boundary of the k^{th} cell of dimension n.

- `X!.coboundaries[n][k]` is a list of integers $[t, a_1, a_2, \ldots, a_t]$ recording that the k^{th} cell of dimension n lies in the boundary of the ${a_i}^{\text{th}}$ cell of dimension $n+1$.

- `X!.nrCells(n)` is a function returning the number of cells of dimension n.

- `X!.properties` is a list of properties of the complex, each property stored as a pair such as `["dimension",4]`.

Datatype 2.1.2. [8] A discrete vector field on a regular CW-complex is represented as a regular CW-complex `X` with the following additional components, each of which is a 2-dimensional array:

- `X!.vectorField[n][k]` is equal to the integer j if there is an arrow from the k^{th} cell of dimension $n-1$ to the j^{th} cell of dimension n. Otherwise, it is unbound.

- `X!.inverseVectorField[n][j]` is equal to the integer k if there is an arrow from the k^{th} cell of dimension $n-1$ to the j^{th} cell of dimension n. Otherwise, it is unbound.

`GAP` Session 2.2.1 will endow a 2-dimensional regular CW-complex Y (Figure 2.1.2 (left)) with a discrete vector field. The critical cells of Y are shown to be one-to-one with the CW-complex $X = S^1 \vee S^1$ (Figure 2.1.2 (right)) consisting of one 0-cell and two 1-cells.

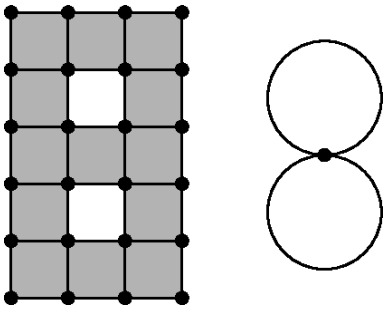


FIGURE 2.1.2: Regular CW-complex Y (left) and the homotopy equivalent CW-complex X (right).

```
gap> Y:=[[1,1,1],[1,0,1],[1,1,1],[1,0,1],[1,1,1]];;
gap> Y:=RegularCWComplex(PureCubicalComplex(Y));
Regular CW-complex of dimension 2

gap> pi1:=FundamentalGroupOfRegularCWComplex(Y);
<fp group of size infinity on the generators [ f1, f2 ]>
gap> RelatorsOfFpGroup(pi1);
[  ]
gap> CriticalCellsOfRegularCWComplex(Y);
[ [ 1, 13 ], [ 1, 15 ], [ 0, 17 ] ]
```

GAP Session 2.1.1: Computing the critical cells of a regular CW-complex homotopy equivalent to $S^1 \vee S^1$.
Runtime: 6ms.

2.2 The practical need for smaller cell structures

Now that we have established a representation for CW-complexes, we can begin to address the size of the computations involved in a naive implementation of cellular chain complexes of CW-complexes. Consider the CW-complex S (see Figure 2.2.1) consisting of 8 vertices, 12 edges, and 4 faces. It might be of interest to investigate the direct product $Y = S \times S \times S \times S \times S$ which naturally inherits regular CW-structure involving a total of $24^5 = 7962624$ cells. Its fundamental group $\pi_1 Y = \mathbb{Z}^5$ is free abelian on 5 generators a, b, c, d, e and has an index 14 subgroup $H = \langle a^7, b, c^2, d, e \rangle$. There exists a corresponding 14-fold covering map $p : \widetilde{Y}_H \to Y$. The integral homology groups $H_n(\widetilde{Y}_H, \mathbb{Z})$ might be of interest to calculate. General theory tells us that $\widetilde{Y}_H$ is homeomorphic to Y, but if we were ignorant to this fact we might try to compute the integral homology $H_n(\widetilde{Y}_H, \mathbb{Z}) = H_n(C_* \widetilde{Y}_H)$ directly from the chain complex of $\widetilde{Y}_H$. We can express the chain complex $C_* \widetilde{Y}_H$ of the 14-fold cover in terms of the cellular chain complex $C_* \widetilde{Y}$ of the universal cover:

$$C_* \widetilde{Y}_H = C_* \widetilde{Y} \otimes_{\mathbb{Z}G} A \tag{2.2.1}$$

with $G = \pi_1 Y, A = \mathbb{Z}G \otimes_{\mathbb{Z}H} \mathbb{Z}$. The desired homology $H_n(\widetilde{Y}_H, \mathbb{Z})$ can be viewed as the homology $H_n(Y, A)$ of Y with local coefficients in the module A. The chain complex $C_* \widetilde{Y}_H$ is a sequence of free abelian groups and group homomorphisms of

the following form:

$$\begin{aligned} 0 \longrightarrow \mathbb{Z}^{32768} \longrightarrow \mathbb{Z}^{245760} \longrightarrow \mathbb{Z}^{819200} \longrightarrow \mathbb{Z}^{1597440} \longrightarrow \mathbb{Z}^{2017280} \\ \hookrightarrow \mathbb{Z}^{1723392} \longrightarrow \mathbb{Z}^{1008640} \longrightarrow \mathbb{Z}^{399360} \longrightarrow \mathbb{Z}^{102400} \longrightarrow \mathbb{Z}^{15360} \longrightarrow \mathbb{Z}^{1024}. \end{aligned} \tag{2.2.2}$$

It is not practical to compute $H_n(\widetilde{Y}_H, \mathbb{Z})$ by applying the Smith Normal Form algorithm directly to 2.2.2. We aim to show how simple homotopy equivalences by way of discrete vector fields can be used to produce a smaller homotopy equivalent chain complex from which we can compute the homology. These techniques are implemented in the `HAP`[10] package for the `GAP`[6] language and can be seen working in `GAP` Session 2.2.1 which computes $H_0(\widetilde{Y}_H, \mathbb{Z}) = \mathbb{Z}$, $H_1(\widetilde{Y}_H, \mathbb{Z}) = \mathbb{Z}^5$, $H_2(\widetilde{Y}_H, \mathbb{Z}) = \mathbb{Z}^{10}$, $H_3(\widetilde{Y}_H, \mathbb{Z}) = \mathbb{Z}^{10}$, $H_4(\widetilde{Y}_H, \mathbb{Z}) = \mathbb{Z}^5$, $H_5(\widetilde{Y}_H, \mathbb{Z}) = \mathbb{Z}$ from the regular CW-complex $Y = S \times S \times S \times S \times S$. This computation involves a significantly smaller non-regular CW-complex $\widetilde{X}_H$ which is homotopy equivalent to $\widetilde{Y}_H$. More specifically, $\widetilde{X}_H$ is a finite cover of $X = \mathfrak{S} \times \mathfrak{S} \times \mathfrak{S} \times \mathfrak{S} \times \mathfrak{S}$ with $\mathfrak{S}$ the non-regular CW-complex seen in Figure 2.2.1 (right).

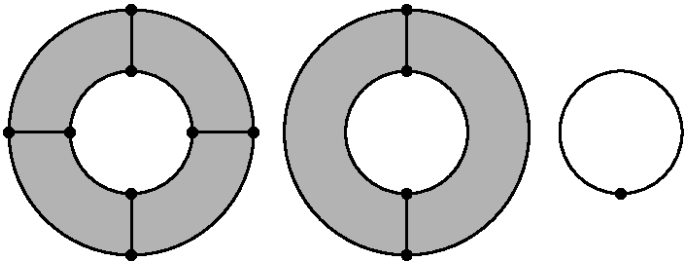


FIGURE 2.2.1: A regular CW-complex S with 24 cells (left), a regular CW-complex $\mathbb{S} \cong S$ with simplified CW-structure and 12 cells (centre), and a non-regular CW-complex $\mathfrak{S} \simeq S$ with just 2 cells (right).

In some cases, one may wish to work with a smaller regular CW-complex that is not just homotopy equivalent but homeomorphic to $\widetilde{Y}_H$. In the above example the number of cells in $\widetilde{Y}_H$ is quite prohibitive, placing its construction outside of the reach for computers with modest specifications. One can attempt to simplify the cell structure on $\widetilde{Y}_H$ so that it contains fewer cells, which we can approach by first simplifying the cell structure of S. In the above example, we have $Y = S \times S \times S \times S \times S$ with S the regular CW-complex of Figure 2.2.1 (left). This could be replaced by the homeomorphic regular CW-complex $\mathbb{S}$ of Figure 2.2.1 (centre). We could then work with the space $\mathbb{Y} = \mathbb{S} \times \mathbb{S} \times \mathbb{S} \times \mathbb{S} \times \mathbb{S}$ which again is homeomorphic to Y. Using an algorithm for simplifying the CW-structure on a regular CW-complex

```
gap> 0cells:=List([1..8],x->[1,0]);;
gap> 1cells:=[
[2,1,2],[2,1,5],[2,1,4],[2,2,3],[2,2,6],[2,3,4],
[2,3,7],[2,4,8],[2,5,6],[2,5,8],[2,6,7],[2,7,8]
];;
gap> 2cells:=[[4,1,2,5,9],[4,4,5,7,11],[4,6,7,8,12],[4,2,3,8,10]];;
gap> S:=RegularCWComplex([0cells,1cells,2cells,[]]);
Regular CW-complex of dimension 2

gap> Y:=DirectProduct(S,S,S,S,S);
Regular CW-complex of dimension 10

gap> Size(Y);
7962624
gap> C:=ChainComplexOfUniversalCover(Y);
Equivariant chain complex of dimension 5

gap> G:=C!.group;
<fp group on the generators [ f1, f2, f3, f4, f5 ]>
gap> H:=Group(G.1^7,G.2,G.3^2,G.4,G.5);;
gap> D:=TensorWithIntegersOverSubgroup(C,H);
Chain complex of length 5 in characteristic 0 .

gap> Homology(D,0);
[ 0 ]
gap> Homology(D,1);
[ 0, 0, 0, 0, 0 ]
gap> Homology(D,2);
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
gap> Homology(D,3);
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
gap> Homology(D,4);
[ 0, 0, 0, 0, 0 ]
gap> Homology(D,5);
[ 0 ]
```

GAP Session 2.2.1: Computing the homology of a 14-fold cover of a regular CW-complex Y with 7962624 cells.
Runtime: 4min 31s 938ms.

on a regular CW-complex, the code of GAP Session 2.2.2 computes the covering map $p : \widetilde{\mathbb{Y}}_H \to \mathbb{Y}$, mapping each cell of $\widetilde{\mathbb{Y}}_H$ homeomorphically to a cell of $\mathbb{Y}$.

The theory, algorithms and datatypes behind the above examples is explained in the next section.

```
gap> 0cells:=List([1..8],x->[1,0]);;
gap> 1cells:=[
[2,1,2],[2,1,5],[2,1,4],[2,2,3],[2,2,6],[2,3,4],
[2,3,7],[2,4,8],[2,5,6],[2,5,8],[2,6,7],[2,7,8]
];;
gap> 2cells:=[[4,1,2,5,9],[4,4,5,7,11],[4,6,7,8,12],[4,2,3,8,10]];;
gap> S:=RegularCWComplex([0cells,1cells,2cells,[]]);;
gap> S:=SimplifiedComplex(S);
Regular CW-complex of dimension 2

gap> Size(S);
12
gap> Y:=DirectProduct(S,S,S,S,S);;
gap> U:=UniversalCover(Y);
Equivariant CW-complex of dimension 10

gap> G:=U!.group;;
gap> H:=Group(G.1^7,G.2,G.3^2,G.4,G.5);;
gap> p:=EquivariantCWComplexToRegularCWMap(U,H);
Map of regular CW-complexes

gap> U_H:=Source(p);
Regular CW-complex of dimension 10

gap> Size(U_H);
3483648
```

GAP Session 2.2.2: Computing a 14-fold covering map $p : \widetilde{\mathbb{Y}}_H \to \mathbb{Y}$ with $\mathbb{Y}$ homeomorphic to a regular CW-complex with 7962624 cells.
Runtime: 16min 45s 755ms.

2.3 Computing chains on the universal cover

The goal of this section is as follows. Given a finite regular CW-complex X, how can we store $C_*(\widetilde{X})$, the chain complex of the universal cover of X, on a computer. The chain groups are free $\pi_1 X$-modules, and the boundary homomorphisms $\partial_n : C_n\widetilde{X} \to C_{n-1}\widetilde{X}$ can be represented as matrices with entries from the group ring $\mathbb{Z}\pi_1 X$. We represent the group $\pi_1 X = F/R$ as a finitely presented group (using the algorithm from [8] and [26]). Elements of $\mathbb{Z}\pi_1 X$ are represented as lists $[\epsilon_1 f_1, \epsilon_2 f_2, \ldots, \epsilon_n f_n]$ with $f_i \in F$ and $\epsilon_i = \pm 1$. We will use this implementation to access finite regular CW-covering spaces $\widetilde{X}_H \to X$ corresponding to finite index subgroups $H < \pi_1 X$.

We will now recall the details of this algorithm from [8] and [26]. As a disclaimer, the

well-known unsolvability of the word problem for finitely presented groups G (*i.e.* given two words s, t in the generators of G, can we determine whether or not $s = t$?) does not obstruct our calculations for this algorithm as will be demonstrated later on (see in particular equation 2.3.4 and the subsequent paragraph). Let $\widetilde{X}$ denote the universal cover of X which inherits a canonical CW-structure from X. The cellular chain group $C_n\widetilde{X}$ is a free $\mathbb{Z}G$-module whose free generators correspond to the n-cells of X. The module $C_n\widetilde{X}$ can be represented on a computer by specifying the number $\text{rank}_{\mathbb{Z}G}(C_n\widetilde{X})$ of free generators and specifying the free presentation for G. Given an n-cell e_i^n which lifts to some $\widetilde{e}_i^n$ in $\widetilde{X}$, denote by $g\widetilde{e}_i^n$ the cell obtained by the action of $g \in G$ on $\widetilde{e}_i^n$. The boundary homomorphisms $\partial_n : C_n\widetilde{X} \to C_{n-1}\widetilde{X}$ of the chain complex are equivariant under the action of G and can be expressed by considering the computation of $\partial_n(\widetilde{e}_i^n)$ for each of the free generators. We now describe an explicit expression for $\partial_n(\widetilde{e}_i^n)$ by induction on n.

Let T denote some choice of maximal subtree of the graph given by the 1-skeleton X^1 of X. Each edge e in this graph can be oriented in one of two ways, which is done arbitrarily. Every edge in $X^1 \setminus T$ lies in the boundary of some face f in the 2-skeleton X^2. The boundary of this face can be viewed as a circuit of oriented edges denoted by $\omega(e)$. By mapping every edge in $X^1 \setminus T$ to this circuit, we induce a function $\omega : \{ \text{ edges in } X^1 \setminus T \} \to G$ which can be extended to $\omega : \{ \text{ edges in } X^1 \} \to G$ by mapping all oriented edges in T to the trivial element.

Let F denote the free group generated by the symbols $\omega(e)$ where e loops through all edges of $X^1 \setminus T$. The 2-cells e^2 of X can be arbitrarily oriented in one of two ways similarly to the edge case; we fix an orientation for each 2-cell. Thus, the boundary of each 2-cell—when read from some point in a clockwise direction—will spell out a word in F corresponding to the trivial word in the fundamental group G. It is well-known (see, *e.g.*, Chapter 3, Section 7, Corollary 5 of [27]) that G admits a free presentation with one generator $\omega(e)$ for each edge $e \subset X^1 \setminus T$ and one relator for every 2-cell of X.

We now can consider the calculation of an expression for the boundary of the free generators $\partial_n(\widetilde{e}_i^n)$ of G by an inductive procedure beginning with the case $n = 1$. The 1-cell $\widetilde{e}_i^1$ maps to e_i^1 by the covering map $p : \widetilde{X} \to X$. Let $e_{i_1}^0, e_{i_2}^0$ be the 0-cells in the boundary of e_i^1 which are ordered according to its orientation which, say, starts at $e_{i_1}^0$ and ends at $e_{i_2}^0$. Suppose that in the chain complex of X, with the previous

choices of orientation, the boundary homomorphism satisfies

$$\partial_1(e_i^1) = e_{i_1}^0 - e_{i_2}^0. \tag{2.3.1}$$

We then set

$$\partial_1(\widetilde{e}_i^1) = \omega(\widetilde{e}_i^1)\widetilde{e}_{i_1}^0 - \widetilde{e}_{i_2}^0. \tag{2.3.2}$$

Computing $\partial_{n+1}(\widetilde{e}_i^{n+1})$ requires already having computed $\partial_1(\widetilde{e}_i^1), \partial_2(\widetilde{e}_i^2), \ldots, \partial_n(\widetilde{e}_i^n)$ for all free generators in degrees at most n. Once this has been done, begin by reading off

$$\partial_{n+1}(e_i^{n+1}) = \epsilon_1 e_{i_1}^n + \epsilon_2 e_{i_2}^n + \ldots + \epsilon_m e_{i_m}^n \tag{2.3.3}$$

where $\epsilon_i = \pm 1$ and $\dim X^n = m$. What remains is to determine group element representatives $g_{i_1}, \ldots, g_{i_m}$ in the finitely presented group G such that

$$\partial_{n+1}(\widetilde{e}_i^{n+1}) = \epsilon_1 g_{i_1}\widetilde{e}_{i_1}^n + \epsilon_2 g_{i_2}\widetilde{e}_{i_2}^n + \ldots + \epsilon_m g_{i_m}\widetilde{e}_{i_m}^n \tag{2.3.4}$$

satisfies $\partial_n(\partial_{n+1}\widetilde{e}_i^{n+1}) = 0$. We set $g_{i_1} = 1_G$ and suppose that $\partial_n(g_{i_1}\widetilde{e}_{i_1}^n)$ contains $\pm h\widetilde{e}_j^{n-1}$ as a summand for some $h \in G$. Then for some $i_j \neq i_1$ the boundary $\partial_n(\widetilde{e}_{i_j}^n)$ must contain $\mp h'\widetilde{e}_j^{n-1}$ as a summand for some $h' \in G$. In equation 2.3.4, we set $g_{i_j} = hh'^{-1}$ and continue with this method of matching summands in the boundaries $\partial_n(\widetilde{e}_{i_k}^n)$ in order to determine all g_{i_k}.

Suppose given a finitely generated group A and group homomorphism $\phi : G \to \mathrm{Aut}(A)$ specified on generators of G. This data comprises a $\mathbb{Z}G$-module A. Beginning with our computer representation of the chain complex $C_*\widetilde{X}$, it is routine to implement the cochain complex $\mathrm{Hom}_{\mathbb{Z}G}(C_*\widetilde{X}, A)$ of finitely generated abelian groups, and also the chain complex $C_*\widetilde{X} \otimes_{\mathbb{Z}G} A$. In particular, given a finite set of elements in G that generate a finite index subgroup $H < G$, we can use coset enumeration to construct the $\mathbb{Z}G$-module $A = \mathbb{Z}G \otimes_{\mathbb{Z}H} \mathbb{Z}$. In this case, the chain complex $C_*\widetilde{X} \otimes_{\mathbb{Z}G} A$ can be viewed as the chain complex $C_*\widetilde{X}_H$ of the finite covering space $p : \widetilde{X}_H \twoheadrightarrow X$ with p inducing an isomorphism $\pi_1\widetilde{X}_H \cong H$. Since $\widetilde{X}_H$ is a regular CW-complex, its CW-structure is completely determined by the chain complex $C_*\widetilde{X}_H$. Therefore, in principle, we have an algorithm for computing the finite cover $\widetilde{X}_H$. Furthermore, in principle, the Smith Normal Form algorithm can be used to compute the local homology $H_n(X, A)$ and cohomology $H^n(X, A)$ for any $\mathbb{Z}G$-module A which is finitely generated over $\mathbb{Z}$.

For practical computations, such as those in `GAP` Sessions 2.2.1 and 2.2.2, the above

Algorithm 2.3.1 Computing chains on the universal cover of a regular CW-complex.

Input:

A finite connected regular CW-complex X.

Output:

An equivariant chain complex $C_*\widetilde{X}$ corresponding to the chain complex of the universal cover of X.

1: Arbitrarily fix an orientation for all 1-cells and all 2-cells of X.
2: Choose some maximal subtree T of the graph given by the 1-skeleton X^1.
3: Assign to each edge in T the trivial element of $\pi_1 X$, and to each edge e_i^1 in $X^1 \setminus T$ a generator $\omega(e_i)$ of $\pi_1 X$.
4: Express the boundary of each orbit of 1-cells as

$$\partial_1(\widetilde{e}_i^1) = \omega(e_i^1)\widetilde{e}_{i_1}^0 - \widetilde{e}_{i_2}^0.$$

5: **for** $n = 1$ **to** $\dim X$ **do**
6: Express the boundary of each orbit of $(n+1)$-cells as

$$\partial_{n+1}(\widetilde{e}_i^{n+1}) = \epsilon_1 g_{i_1}\widetilde{e}_{i_1}^n + \epsilon_2 g_{i_2}\widetilde{e}_{i_2}^n + \ldots + \epsilon_m g_{i_m}\widetilde{e}_{i_m}^n.$$

7: Using the fact that $\partial_n(\partial_{n+1}\widetilde{e}_i^{n+1}) = 0$, express the above boundary as

$$0 = \partial_n(\epsilon_1 g_{i_1}\widetilde{e}_{i_1}^n) + \partial_n(\epsilon_2 g_{i_2}\widetilde{e}_{i_2}^n) + \ldots + \partial_n(\epsilon_m g_{i_m}\widetilde{e}_{i_m}^n).$$

8: Set $g_{i_1} = 1_{\pi_1 X}$.
9: The previously computed $\partial_n(g_{i_1}\widetilde{e}_{i_1}^n)$ will contain $\pm h\widetilde{e}_j^{n-1}$ as a summand for some $h \in \pi_1 X$, meaning that for some $i_j \neq i_1$, the boundary $\partial_n(\widetilde{e}_{i_j}^n)$ must contain $\mp h'\widetilde{e}_j^{n-1}$ for some $h' \in \pi_1 X$. Set $g_{i_j} = hh'^{-1}$.
10: Repeat the methodology of the previous step in order to determine all g_{i_k}.
11: **end for**
12: Return $C_*(\widetilde{X})$.

theoretical algorithm (Algorithm 2.3.1) presents two issues. Firstly, if we apply it directly then the finitely presented fundamental group $G = \pi_1 X$ will generally have excessive numbers of generators and relators. It is necessary to reduce these numbers in a way that retains the relationship between the finite presentation and the cellular structure of X if, for example, one wants to apply coset enumeration and the Reidemeister-Schreier algorithm to list free presentations of subgroups $H < G$ of given index k in order to enumerate all k-fold covers of X. Secondly, the number of cells of X will typically be large and as a result, so too will be the number of generators of chain groups $C_n\widetilde{X} \otimes_{\mathbb{Z}G} A$. This makes it impractical to apply the Smith Normal Form algorithm directly to the chain complex $C_*\widetilde{X} \otimes_{\mathbb{Z}G} A$ or the cochain complex $\mathrm{Hom}_{\mathbb{Z}G}(C_*\widetilde{X}, A)$. We need a method for reducing the number of cells in X and $\widetilde{X}$ in such a way that retains their homotopy types or, in some cases, their

homeomorphism types. The code in GAP Session 2.2.1 reduces the number of cells while retaining just the homotopy type of X while the code in GAP Session 2.2.2 retains the homeomorphism type of X. Our approach to addressing these issues will be described in Section 5.3.

Regarding a method for reducing the number of cells in X in such a way that retains only the homotopy type, we turn to admissible discrete vector fields. There are a variety of algorithms for constructing an admissible discrete vector field on a finite regular CW-complex X. The implementation found in HAP is shown in Algorithm 2.3.2 which we recall from [28], [8]. In the second line of the algorithm, the cells of X could be partially ordered in such a way that ensures any cell of dimension n is less than all cells of dimension $n+1$. This partial ordering guarantees that the resulting discrete vector field on a path-connected regular CW-complex X will have a unique critical 0-cell.

Algorithm 2.3.2 Discrete vector field on a regular CW-complex.

Input:
A finite regular CW-complex X.
Output:
A maximal admissible discrete vector field on X.

1: Partially order the cells of X in any fashion.
2: At every stage of the algorithm, each cell will have precisely one of the following three states: (i) critical, (ii) potentially critical, (iii) non-critical.
3: Initially deem all cells of X to be potentially critical.
4: **while** there exists a potentially critical cell **do**
5: **while** there exists a pair of potentially critical cells s, t such that: $\dim(t) = \dim(s) + 1$; s lies in the boundary of t; no other potentially critical cell of dimension $\dim(s)$ lies in the boundary of t **do**
6: Choose such a pair (s, t) with s minimal in the given partial ordering.
7: Add the arrow $s \to t$ and deem s and t to be non-critical.
8: **end while**
9: **if** there exists a potentially critical cell **then**
10: Choose a minimally potentially critical cell and deem it to be critical.
11: **end if**
12: **end while**

A *reduced* CW-complex has only one cell in dimension 0. There is a standard correspondence between the 2-skeleton of a reduced CW-complex and a presentation of its fundamental group $G = \pi_1 X$. This correspondence, Theorem 1.4.2.2 and Algorithm 2.3.2 constitute our algorithm for finding a presentation for the fundamental group of any connected CW-complex X. This presentation has one generator for

each critical 1-cell of X and one relator for each critical 2-cell. Each oriented critical 2-cell $e \subset X$ determines an oriented circuit $\omega(e)$ in X^1. The discrete vector field provides a deformation of this circuit $\omega(e)$ into a circuit $\omega'(e)$ each of whose oriented edges t is either critical or else the target of some arrow $s \to t$ with s a 0-cell. The subsequence of $\omega'(e)$ consisting of the oriented critical edges spell a word in the free group on the generators of the presentation. For illustrations and further details of this algorithm see [26], [8].

We are now equipped to address the reduction of the number of cells in the universal cover $\widetilde{X}$ of a connected regular CW-complex while preserving its homotopy type. It is sufficient to note that, any discrete vector field on X induces a discrete vector field on $\widetilde{X}$: there is an arrow $\widetilde{s} \to \widetilde{t}$ on $\widetilde{X}$ if and only if $s \to t$ is an arrow on X where $\widetilde{s}, \widetilde{t}$ are cells in the universal cover that map to s, t, and where $\widetilde{s}$ lies in the boundary of $\widetilde{t}$. It also follows that if the discrete vector field on X is admissible then so too is the induced vector field. Let $X \simeq Y$ be the homotopy equivalence of Theorem 1.4.2.2. This equivalence induces a homotopy equivalence of the respective universal covers $\widetilde{X} \simeq \widetilde{Y}$ and a chain homotopy equivalence of cellular chain complexes $C_*\widetilde{X} \simeq C_*\widetilde{Y}$, *i.e.*, the cells in $\widetilde{Y}$ are one-one with the critical cells in the induced discrete vector field on $\widetilde{X}$. The cellular chain complex $C_*\widetilde{Y}$ on the (typically non-regular) CW-complex Y is readily constructed directly from the cell structure and admissible discrete vector field of X. This construction has been implemented in `HAP` as the function `ChainComplexOfUniversalCover(X)` (A.2) which inputs a regular CW-complex X, constructs a maximal discrete vector field on X, uses this vector field to compute a presentation for $\pi_1 X \cong \pi_1 Y$, and returns the $\pi_1 X$-equivariant chain complex $C_*(\widetilde{X})$.

Chapter 3

Invariants of knotted manifolds

The machinery we have developed to calculate ambient isotopy invariants of embeddings of manifolds $\iota : N \to M$ necessitates procedures for endowing N and M with regular CW-structure. This need is addressed in Chapters 4 and 6 by providing some algorithms for obtaining CW-decompositions of links in $\mathbb{R}^3$ and knotted surfaces in $\mathbb{R}^4$. However, in this chapter, we opt not to represent these embeddings as embeddings of regular CW-complexes, but instead as embeddings of *pure cubical complexes*.

Definition 3.1. [8] Let $\mathbb{R}^n$ be endowed with CW-structure wherein each n-cell has closure equal to a unit cube with standard CW-structure and with centre an integer vector $c \in \mathbb{Z}^n$. A finite CW-subcomplex of $\mathbb{R}^n$ is said to be an n-dimensional *cubical complex*. Such a complex is said to be *pure* if each cell lies in the closure of some n-cell.

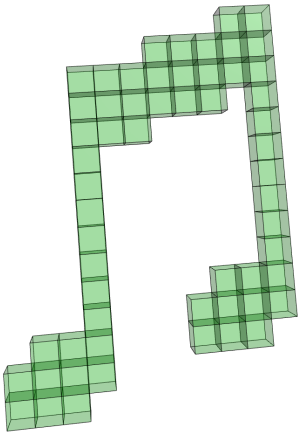


FIGURE 3.1: A 3-dimensional pure cubical complex with 933 cells.

Working with pure cubical complexes can be convenient in forming cell complexes given that all that is needed is to specify the coordinates of each top-dimensional cell. The downside however, is that the resulting complexes tend to have a large number of cells and so computing with them can be quite taxing. We momentarily forego more efficient cell structures to demonstrate that our algorithms can be practical even when working with large cell complexes.

Definition 3.2. Let N denote a compact submanifold of $\mathbb{R}^{n+2}$ with $n \in \{1, 2\}$ and suppose that $p : \mathbb{R}^{n+2} \to \mathbb{R}^{n+1}$ is a projection onto a hyperplane. Let $M \subset \mathbb{R}^{n+2}$ be a submanifold homeomorphic to the closed unit ball whose interior $\mathring{M}$ contains N. Set $D^{n+1} = p(M)$. We refer to the pair $p(N) \subset D^{n+1}$ as a *diagram* for N.

It is possible to embellish a diagram with additional information on the preimages $p^{-1}(y)$ of certain points $y \in N$ so that the ambient isotopy type of the embedding $N \hookrightarrow \mathbb{R}^{n+1}$ can be recovered. We use this embellished diagram to construct a regular CW-structure on M containing a subspace ambient isotopic to N.

3.1 A 3-manifold in 3-space and the complement of its interior

There are various representations for links in $\mathbb{R}^3$. Our chosen representation is that of an arc presentation. We represent an arc presentation on a computer as a list l of ordered pairs of positive integers. The i^{th} pair in the list specifies the starting and ending columns of the i^{th} horizontal line, where the bottom line is the 1^{st} horizontal line and the leftmost column is the 1^{st} column. For example, the list $l = [[2, 4], [1, 3], [2, 4], [1, 3]]$ specifies an arc presentation of the Hopf link (see Figure 3.1.2 (left)). Note that a choice must be made when using arc presentations: at each crossing, all horizontal lines will lie above all vertical lines or vice versa. This does not restrict the scope of knots and links that can be represented in this way.

Arc presentations lead to a very intuitive, albeit somewhat inefficient, representation of the corresponding link as a 3-dimensional CW-manifold with boundary. Consider the diagram $(D^2, p(N))$ with D^2 a 2-dimensional pure cubical complex homeomorphic to a disk. Let the real interval $[0, 5]$ be given a CW-structure with integers the 0-cells. The desired 3-dimensional pure cubical complex N is realised as a CW-subcomplex of the direct product $D^2 \times [0, 5]$.

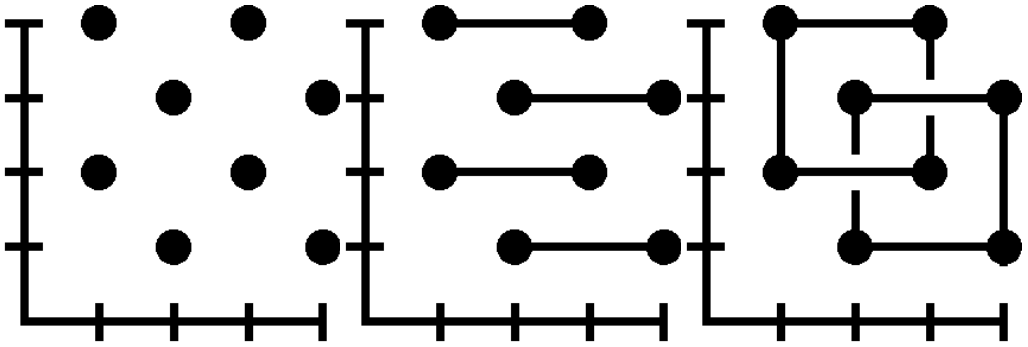


FIGURE 3.1.1: Reconstructing a link diagram from an arc presentation.

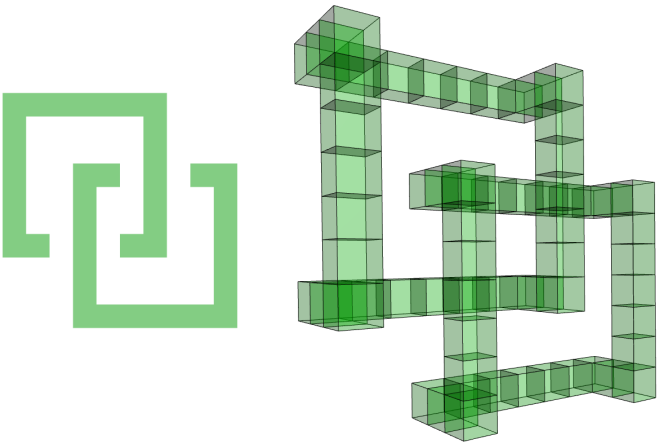


FIGURE 3.1.2: An arc presentation of the Hopf link (left) and the associated pure cubical complex (right).

The 3-dimensional pure cubical complex N_l corresponding to the example list for the Hopf link is shown in Figure 3.1.2 (right). Let M denote a contractible pure cubical complex in $\mathbb{R}^3$ whose interior contains N_l. Then the complement $X = M \setminus \mathring{N}_l$ is a pure cubical complex homotopy equivalent to the complement $\mathbb{R}^3 \setminus N_l$. The construction of the CW-complex X is implemented in `HAP`, with M the minimal solid rectangular pure cubical complex whose interior contains N_l. In the case of the Hopf link of Figure 3.1.2 (right), the implemented CW-complex X contains 9123 cells.

3.2 A closed 2-manifold in 4-space

In order to begin specifying an embedding $N \hookrightarrow \mathbb{R}^4$ of a closed surface N we start by considering a diagram $(D^3, p(N))$ with D^3 a regular CW-complex homeomorphic to a closed 3-ball, and with $p(N) \subset D^3$ a CW-subcomplex lying in the interior of D^3. We do not require the CW-subcomplex $p(N)$ to be a surface. We allow there to be

singular points $y \in p(N)$ for which there exists no open set of $p(N)$ containing y that is homeomorphic to a 2-ball. We do require that the collection of singular points is either empty or forms a closed 1-manifold. As an example, in the usual projection of the Klein bottle into $\mathbb{R}^3$, the singular points form a circle (see Figure 3.2.1). We say that the CW-complex $p(N)$ is a *self-intersecting surface*. In order to finish our specification of the embedding of the surface N we set $M = D^3 \times [0, k]$ where k is some positive integer and the interval is given a CW-structure with the 0-cells the integers. Let $p : M \to D^3$ be the projection. To specify a CW-manifold $N \subset M$, we just need to specify the cells in the preimage $p^{-1}(e)$ for each cell $e \subset p(N)$.

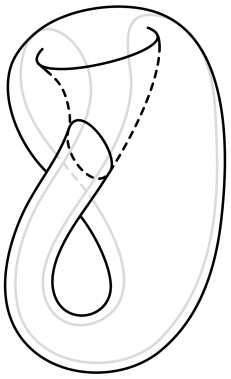


FIGURE 3.2.1: Klein bottle.

In Chapter 6, we describe a procedure for embedding self-intersecting tori into $\mathbb{R}^3$ by way of an embellished arc presentation. These embedded tori can be lifted in order to obtain an embedding of a closed surface N in $\mathbb{R}^4$. We then construct a small open tubular neighbourhood N_ϵ of N and extend the CW-structure on $M \setminus N$ to a CW-structure on $M \setminus N_\epsilon$.

3.3 The spun Hopf link and the Tube of the welded Hopf link

We now provide a computer proof of a variant of Theorem 10 from [29]. This theorem uncovers the ambient isotopy inequivalence between two *knotted surface* complements. These surfaces are the Tube of the welded Hopf link and the spun Hopf link, they each represent two methods of obtaining embeddings of knotted tori in $\mathbb{R}^4$. Our implementation of the Tube map in full generality comprises most of the content of Chapter 6, while our implementation of knot spinning is discussed in Section 5.4. For now, we begin by describing the construction of the Tube of the

welded Hopf link as a pure cubical complex. Figure 3.3.1 shows a 3-dimensional pure cubical complex Y formed from the union of two intersecting pure cubical subcomplexes each of which is homotopy equivalent to a torus. The space Y is a union of 1632 3-cubes. The four horizontal rectangular tubes of Y have a 3×3 cross section. The four vertical tubes of Y have a 7×7 cross section. The central axes of the horizontal tubes and vertical tubes lie in a common plane. We will use Y to construct two 4-dimensional pure cubical complexes S and T as follows.

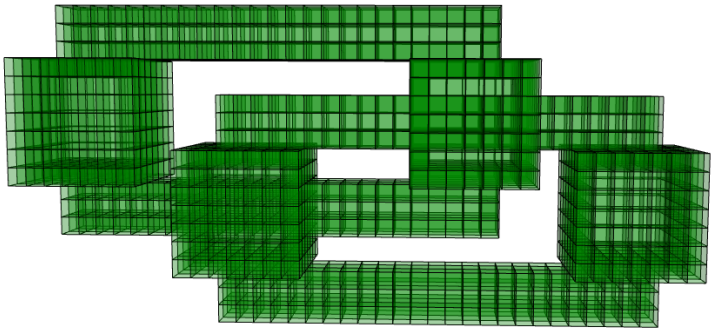


FIGURE 3.3.1: Union of two 3-dimensional pure cubical complexes, each of which is homotopy equivalent to a torus.

To construct S we first assign integers t, called *temperature*, to each 3-cube of Y. Most 3-cubes are assigned a single temperature t, but a few 3-cells are assigned three temperatures. All 3-cubes in the top and bottom horizontal rectangular tubes are assigned one temperature $t = 0$. All 3-cubes in the four vertical rectangular tubes are also assigned one temperature $t = 0$. The middle two horizontal rectangular tubes each have length 25. In these two tubes, the x-coordinate of a cube's centre determines its temperature(s) according to the profiles shown in Figure 3.3.2. Profile 1 is the cross sectional temperature profile of the upper-middle horizontal tube of Y while the lower-middle horizontal tube of Y is assigned Profile 2. The space S is the 4-dimensional pure cubical complex whose 4-cubes are centred on the integer vectors (x, y, z, t) where (x, y, z) is the centre of a 3-cube of Y with temperature t. The space S is homotopy equivalent to a disjoint union of two tori.

The space T is a 4-dimensional pure cubical complex constructed similarly to S but whose temperatures profiles are those of Figure 3.3.3. Again, the space T is homotopy equivalent to a disjoint union of two tori.

Let $\mathring{S}$ and $\mathring{T}$ denote the interiors of S and T respectively. The complements $\mathbb{R}^4 \backslash \mathring{S}$ and $\mathbb{R}^4 \setminus \mathring{T}$ are subcomplexes of $\mathbb{R}^4$, both with infinitely many cells. It is straightforward

Profile 1:

0	0	0	0	0	0	0 -1 -2	-2	-2	-2	-2	0 -1 -2	0	0 1 2	2	2	2	2	0 1 2	0	0	0	0	0	0

Profile 2:

0	0	0	0	0	0	0 1 2	2	2	2	2	0 1 2	0	0 -1 -2	-2	-2	-2	-2	0 -1 -2	0	0	0	0	0	0

FIGURE 3.3.2: Profiles of cube temperatures for S.

Profile 1:

0	0	0	0	0	0	0 1 2	2	2	2	2	2	2	2	2	2	2	2	0 1 2	0	0	0	0	0	0

Profile 2:

0	0	0	0	0	0	0 -1 -2	-2	-2	-2	-2	0 -1 -2	0	0 1 2	2	2	2	2	0 1 2	0	0	0	0	0	0

FIGURE 3.3.3: Profiles of cube temperatures for T.

to construct finite deformation retracts $X_S \subset \mathbb{R}^4 \setminus \mathring{S}$ and $X_T \subset \mathbb{R}^4 \setminus \mathring{T}$ where X_S and X_T are 4-dimensional pure cubical complexes. The code in `GAP` Session 3.3.1 computes a regular CW-complex X_T involving 4508573 cells, together with a list $\mathrm{Inv}(X_T)$ of all possible abelian invariants of the second homology groups $H_2(\widetilde{X}_H, \mathbb{Z})$ of 5-fold covering spaces X_H for X_T. The list $\mathrm{Inv}(X_T)$ is an invariant of the homotopy type of X_T, establishing that there exist 5-fold covers with $H_2(\widetilde{X}_H, \mathbb{Z}) = \mathbb{Z}^{12}$ and 5-fold covers with $H_2(\widetilde{X}_H, \mathbb{Z}) = \mathbb{Z}^{16}$. We can apply similar commands to X_S to find that all 5-fold covers X_H have $H_2(\widetilde{X}_H, \mathbb{Z}) = \mathbb{Z}^{12}$, thus establishing that X_T and X_S are homotopy inequivalent. For the sake of efficiency, the construction of X_S that is employed in `GAP` Session 3.3.2 uses techniques that will be explained in Chapter 5.

The additional `GAP` commands of `GAP` Session 3.3.3 establish that $H_0(X_T, \mathbb{Z}) = \mathbb{Z}$, $H_1(X_T, \mathbb{Z}) = \mathbb{Z}^2$, $H_2(X_T, \mathbb{Z}) = \mathbb{Z}^4$, $H_3(X_T, \mathbb{Z}) = \mathbb{Z}^2$, $H_n(X_T, \mathbb{Z}) = 0$ for $n \geq 4$, and that $\pi_1 X_T \cong \pi_1 X_S \cong \mathbb{Z} \times \mathbb{Z}$, $H_n(X_S, \mathbb{Z}) = H_n(X_T, \mathbb{Z})$ for $n \geq 0$. This tells us that this is an example where the fundamental group and homology with trivial coefficients fail to distinguish between two homotopy inequivalent spaces, while the second homology with twisted coefficients succeeds in doing so.

Figure 3.3.4 shows an example of a classical planar diagram of a link (the Hopf link) and an example of a welded diagram (the welded Hopf link). Such a classical or welded link diagram L gives rise to an embedding of a closed surface $\mathrm{Tube}(L)$ into

```
gap> X_T:=PureComplexComplement(HopfSatohSurface());;
gap> X_T:=RegularCWComplex(X_T);;
gap> C:=ChainComplexOfUniversalCover(X_T);;
gap> Inv_X_T:=LowIndexSubgroupsFpGroup(C!.group,5);;
gap> Inv_X_T:=Filtered(Inv_X_T,g->Index(C!.group,g)=5);;
gap> Inv_X_T:=Set(
>     Inv_X_T,
>     g->Homology(
>         TensorWithIntegersOverSubgroup(C,g),2
>     )
> );
[ [ 0,0,0,0,0,0,0,0,0,0,0,0 ], [ 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ] ]
```

GAP Session 3.3.1: Computing the regular CW-complex X_T and an invariant of its homotopy type.
Runtime: 17min 17s 803ms.

```
gap> X_S:=SpunLinkComplement([[2,4],[1,3],[2,4],[1,3]]);;
gap> C:=ChainComplexOfUniversalCover(X_S);;
gap> L:=LowIndexSubgroupsFpGroup(C!.group,5);;
gap> L:=Filtered(L,g->Index(C!.group,g)=5);;
gap> Set(
>     L,
>     g->Homology(
>         TensorWithIntegersOverSubgroup(C,g),2
>     )
> );
[ [ 0,0,0,0,0,0,0,0,0,0,0,0 ] ]
```

GAP Session 3.3.2: Computing a CW-structure on the complement of the Hopf link and a homotopy invariant of the spun link.
Runtime: 2s 650ms.

$\mathbb{R}^4$ via the *Tube map* of Satoh[30]. The surface Tube(L) contains one knotted torus Tube(K) for each component K of L. In Chapter 6, we will go into detail regarding the direct construction of the Tube map. Additionally, a good explanation and visualisation of the Tube map is given in [31]. Satoh showed that for a classical link diagram L, the knotted surface Tube(L) $\subset \mathbb{R}^4$ is the same as the surface obtained by spinning the link about a plane that does not intersect it. This spin construction is elaborated upon in Chapter 5. An algebraic invariant of the homotopy type of the complement $X_L = \mathbb{R}^4 \setminus \text{Tube}(L)$ is introduced in [32, 33] and used in [29] to show, for instance, that the classical Hopf link diagram L_1 and the welded Hopf link diagram L_2 (see Figure 3.3.4) yield spaces X_{L_1} and X_{L_2} with different homotopy

types. As a consequence of this, the knotted surfaces Tube(L_1) and Tube(L_2) are homotopy inequivalent.

FIGURE 3.3.4: Classical and welded link diagrams.

The space T can be viewed as a thickening of two knotted tori in $\mathbb{R}^4$. The complement $\mathbb{R}^4 \setminus T$ is homeomorphic to the space X_{L_2} associated to the welded Hopf diagram. The complement $\mathbb{R}^4 \setminus S$ is homeomorphic to the space X_{L_1} associated it the classical Hopf link diagram. Hence the computations of `GAP` Sessions 3.3.1 and 3.3.2 illustrate how the homotopy inequivalence of X_{L_1} and X_{L_2} can be recovered using a `GAP` computation of second homology with local coefficients. This computation yields an analogue of Theorem 10 of [29]. We define the invariant $\mathfrak{I}_c(\Sigma)$ of a knotted surface $\Sigma \subset \mathbb{R}^4$ to be:

$$\begin{aligned}\mathfrak{I}_c(\Sigma) = {} & \text{Set of isomorphism types of abelian groups} \\ & \text{arising as } H_2(\widetilde{X}_H, \mathbb{Z}) \text{ for some } c\text{-fold cover} \\ & \widetilde{X}_H \to X \text{ of } X = \mathbb{R}^4 \setminus \Sigma.\end{aligned}$$

Theorem 3.3.1. *The invariant $\mathfrak{I}_c$ is powerful enough to distinguish between knotted surfaces $\Sigma, \Sigma' \subset \mathbb{R}^4$, with Σ diffeomorphic to Σ' and whose complements have isomorphic fundamental groups and isomorphic integral homology, at least in one specific case.*

Proof. See `GAP` Sessions 3.3.1 and 3.3.2. □

```
gap> X_T:=PureComplexComplement(HopfSatohSurface());;
gap> X_T:=RegularCWComplex(X_T);;
gap> X_T:=ContractedComplex(X_T);;
gap> CriticalCells(X_T);;
gap> H0:=Homology(X_T,0);
[ 0 ]
gap> H1:=Homology(X_T,1);
[ 0, 0 ]
gap> H2:=Homology(X_T,2);
[ 0, 0, 0, 0 ]
gap> H3:=Homology(X_T,3);
[ 0, 0 ]
gap> H4:=Homology(X_T,4);
[  ]
gap> F:=FundamentalGroup(X_T);
#I  there are 2 generators and 1 relator of total length 4
<fp group of size infinity on the generators [ f1, f2 ]>
gap> RelatorsOfFpGroup(F);
[ f1^-1*f2^-1*f1*f2 ]
gap> X_S:=SpunLinkComplement([[2,4],[1,3],[2,4],[1,3]]);;
gap> X_S:=ContractedComplex(X_S);;
gap> CriticalCells(X_S);;
gap> h0:=Homology(X_S,0);
[ 0 ]
gap> h1:=Homology(X_S,1);
[ 0, 0 ]
gap> h2:=Homology(X_S,2);
[ 0, 0, 0, 0 ]
gap> h3:=Homology(X_S,3);
[ 0, 0 ]
gap> h4:=Homology(X_S,4);
[  ]
gap> f:=FundamentalGroup(X_S);
#I  there are 2 generators and 1 relator of total length 4
<fp group of size infinity on the generators [ f2, f3 ]>
gap> RelatorsOfFpGroup(f);
[ f3^-1*f2^-1*f3*f2 ]
```

GAP Session 3.3.3: Verifying that X_S and X_T have isomorphic integral homology and fundamental groups.
Runtime: 2min 33s 599ms.

Chapter 4

Small CW-structures for link complements

Pure cubical complexes are particularly useful as a tool for converting a range of experimental data into regular CW-complexes so that the underlying topological features can be analysed (see for example [8]). The resulting CW-complexes tend to be large and cumbersome to work with on a computer. From the viewpoint of theoretical knot theory, it is desirable to have algorithms which convert a symbolic representation of a link or knotted surface complement into a regular CW-complex with as few cells as possible so that computations of fundamental groups and local cohomology run efficiently. We now explain (in four steps) how to construct a smaller CW-structure on the complement $X = M \setminus \mathring{N}_l$ of a link N_l arising from an arc presentation l where M denotes a contractible 3-dimensional regular CW-complex homeomorphic to the 3-ball B^3.

4.1 A more efficient complement of a link in 3-space

Step 1. Let us suppose that the arc presentation for the list l involves precisely h horizontal lines (note that an arc presentation must be of even length and will have the same number of horizontal and vertical lines) and precisely k crossings. Let D^2_l denote the unit 2-disk with $2h$ non-overlapping subdisks removed. This is illustrated for the Hopf link in Figure 4.1.1. The CW-structure is made canonical by insisting

that the two 0-cells on the boundary of the disk are connected to the top left-most and bottom right-most vertices of the inner diagram. In general, the CW-structure has $V = 4h + 4k + 2$ vertices, $E = 8h + 8k + 4$ edges, and $E - V - 2h + 1$ cells of dimension 2. The formula for the number of 2-cells is derived from the Euler characteristic $\chi(D_l^2) = 1 - 2h$. The space D_l^2 has $14h + 16k + 9$ cells.

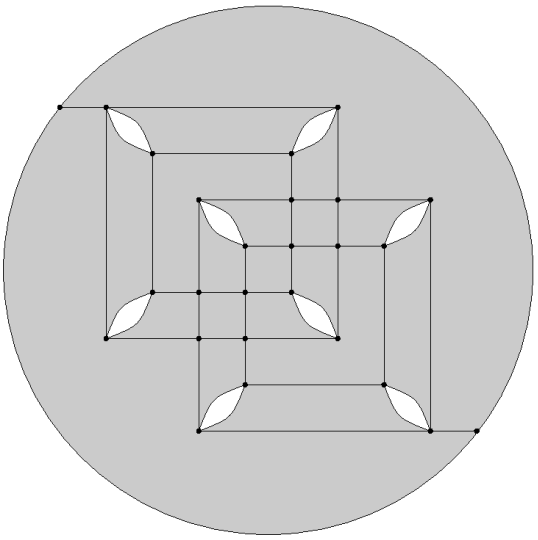


FIGURE 4.1.1: CW-structure on a 2-dimensional disk D_l^2 for $h = 4$ and $k = 2$.

A practical problem that arises in automating the construction of the space D_l^2 lies in determining the boundaries of each of the 2-cells. Recall that in Datatype 2.1.1, a regular CW-complex is encoded on a computer by ordering each of its cells and specifying which $(n-1)$-cells lie in the boundary of each n-cell e^n. It is not immediately obvious how one can keep track of the boundaries of all 2-cells e^2 of D_l^2 in a manner robust enough to work with any valid choice of arc presentation l. Note that as D_l^2 is a surface, all of its 1-cells have at most two 2-cells in their coboundaries. The 1-cells that have just one 2-cell in their coboundaries are those which bound the holes of the disk and those which bound the disk itself. Using this fact, we can work towards a solution to track the boundaries of all 2-cells. Firstly, for each 0-cell in D_l^2, we assign to all of the 1-cells in its coboundary an integer $t \in [1, 12]$ which we refer to as a *time*. These times relate the angles formed by each coboundary 1-cell in the CW-structure of Figure 4.1.1 to time on a 12-hour clock. In this way every 1-cell in D_l^2 has two times, one for each 0-cell in its boundary.

We will use these times to compute a clockwise walk along the 1-skeleton of the disk. This edge walk will systematically travel along the boundaries of each 2-cell of D_l^2. By recording the indexing of these edges, and ensuring that no edge is visited more than twice, we have a procedure for determining the boundaries of all 2-cells as lists

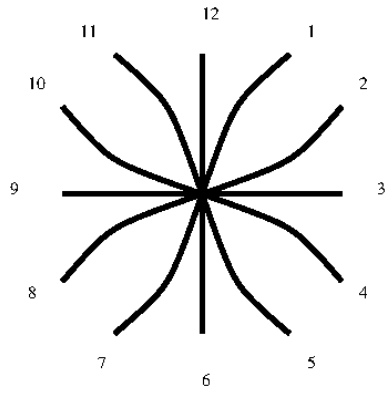


FIGURE 4.1.2: Orienting the 1-cells of D^2_l.

of integers $[t, a_1, a_2, \ldots, a_t]$ where a_i records that the $a_i{}^{\text{th}}$ 1-cell lies in the boundary of a given 2-cell.

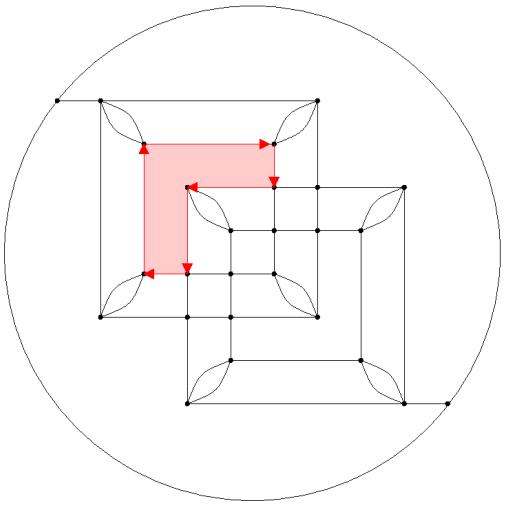


FIGURE 4.1.3: A clockwise walk along the 1-skeleton of D^2_l.

Step 2. Let I denote the unit interval with CW-structure involving two vertices and one edge. Now form the direct product $D^2_l \times I$. This is a CW-complex which we view as a solid cylinder from which $2h$ vertical tubes, running from the bottom to the top, have been removed. This 3-dimensional CW-complex involves a total of $3(14h + 16k + 9)$ cells.

Step 3. Corresponding to each horizontal line in the arc diagram glue one 2-cell to the bottom of D^2_l, and corresponding to each vertical line in the arc diagram glue one 2-cell to the top of D^2_l. In particular, these $2h$ 2-cells are glued so that the resulting CW-complex $W_l = D^2_l \times I \cup \bigcup_{i-1}^{2h} e^2$ is homeomorphic to the link complement $\mathbb{R}^3 \setminus N_l$.

Step 4. Glue one 3-cell and one 2-cell to the bottom of W_l, and glue one 3-cell and one 2-cell to the top of W_l in such a way that the resulting CW-complex $Y_l =$

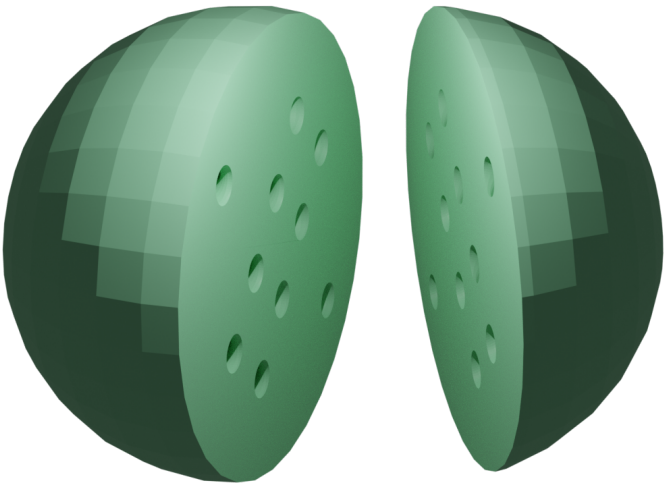


FIGURE 4.1.4: A cross-section of the complex Y_l for l the standard arc presentation of the trefoil.

$W_l \cup \bigcup_{i=1}^{2} e^2 \cup \bigcup_{i=1}^{2} e^3$ is homeomorphic to $M \setminus \mathring{N}_l$. The total number of cells in the CW-complex Y_l is $3(14h + 16k + 9) + 2h + 4$.

Two implementations of this four-step procedure are illustrated in `GAP` Session 4.1.1. The first implementation returns the regular CW-complex Y_l corresponding to link complement as specified by the arc presentation l. The second implementation computes the complex Y_l, identifies the subcomplex homeomorphic to the boundary of some small open tubular neighbourhood (see 5.1 for details) of the link N_l (which we denote by B'_l), then returns the inclusion of regular CW-complexes $f'_l : B'_l \hookrightarrow Y_l$.

4.2 The granny and reef knots

The granny knot $\kappa_1 = 3_1 \# 3_1^*$ occurs as the knot sum of the trefoil with its mirror image, while the reef knot $\kappa_2 = 3_1 \# 3_1$ occurs as the knot sum of two trefoils. Arc presentations for these knots are displayed in Figure 4.2.1. Regarding a knot as a 1-dimensional subspace of $\mathbb{R}^3$, we define $X_1 = \mathbb{R}^3 \setminus \kappa_1$ and $X_2 = \mathbb{R}^3 \setminus \kappa_2$. Using the method of Section 4.1, we can construct finite CW-complexes Y_i whose interiors are homeomorphic to X_i, $i = 1, 2$. The `FundamentalGroup(Y)` function in `HAP` can be used to compute presentations for $\pi_1 Y_i$, $i = 1, 2$. From these presentations, Tietze moves can be used to establish an isomorphism $\pi_1 Y_1 \cong \pi_1 Y_2$. Therefore knot invariants, such as the Alexander polynomial, which are based entirely on the knot group are unable to distinguish between the granny and reef knots. More precisely, such invariants fail to distinguish between the homeomorphism types of Y_1 and Y_2.

```
gap> N_l:=PureCubicalKnot(10,5);;
gap> l:=ArcPresentation(N_l);
[ [ 8, 12 ], [ 1, 10 ], [ 9, 11 ], [ 10, 12 ], [ 7, 11 ], [ 6, 8 ],
  [ 5, 7 ], [ 4, 6 ], [ 3, 5 ], [ 2, 4 ], [ 1, 3 ], [ 2, 9 ] ]
gap> Size(PureComplexComplement(N_l));
6926
gap> Y_l:=KnotComplement(l);
Regular CW-complex of dimension 3

gap> Size(Y_l);
1087
gap> pi_1:=FundamentalGroup(Y_l);
#I  there are 2 generators and 1 relator of total length 73
<fp group of size infinity on the generators [ f1, f2 ]>
gap> RelatorsOfFpGroup(pi_1);
[ f1*f2^-1*f1^-1*f2*f1*f2^-1*(f2^-1*f1*f2*f1^-1*f2*f1*f2^-1*f1^-1*\
f2^2*f1^-1)^3*f2^-1*f1*f2*(f1^-1*f2*f1*f2^-2*f1*f2*f1^-1*f2^-1*f1*\
f2^-1)^2*f1^-1*f2*f1*f2^-2*f1*f2*f1^-1*f2^-1 ]
gap> f_l:=KnotComplementWithBoundary(l);
Map of regular CW-complexes

gap> FundamentalGroup(f_l);
[ f1, f2 ] ->
[
  f2*(f2*f1^-1*f2^-1*f1*f2^-1*f1^-1*f2*f1*f2^-2*f1)^3*f2*f1^-1*f2^2*\
  (f1^-1*f2*f1*f2^-2*f1*f2*f1^-1*f2^-1*f1*f2^-1)^3*f1^-1*f2*f1*f2^-1\
  ,f2*f1^-1 ]
```

GAP Session 4.1.1: Computing the induced homomorphism of fundamental groups given by f'_l where l is an arc presentation corresponding to the fifth prime knot on ten crossings 10_5.
Runtime: 205ms.

In GAP Session 4.2.1, we see that the granny and reef knots have the same Alexander polynomial.

In GAP Session 4.2.2, the integral homology groups of Y_1 and Y_2 are computed which turn out to be the same.

Let us view X_i as the interior of Y_i and let B'_i denote the boundary of a 3-dimensional small open tubular neighbourhood of the knot κ_i. Then B'_i is a 2-dimensional CW-subspace of Y_i homeomorphic to a torus. The `BoundaryMap(Y)` function in HAP can be used to construct an inclusion $f_i : B_i \hookrightarrow Y_i$ of CW-subspaces where B_i is the smallest CW-subspace containing those 2-cells of Y_i that lie in the boundary of exactly one 3-cell. Thus B_i is a disjoint union of B'_i with the 2-sphere S^2. We are

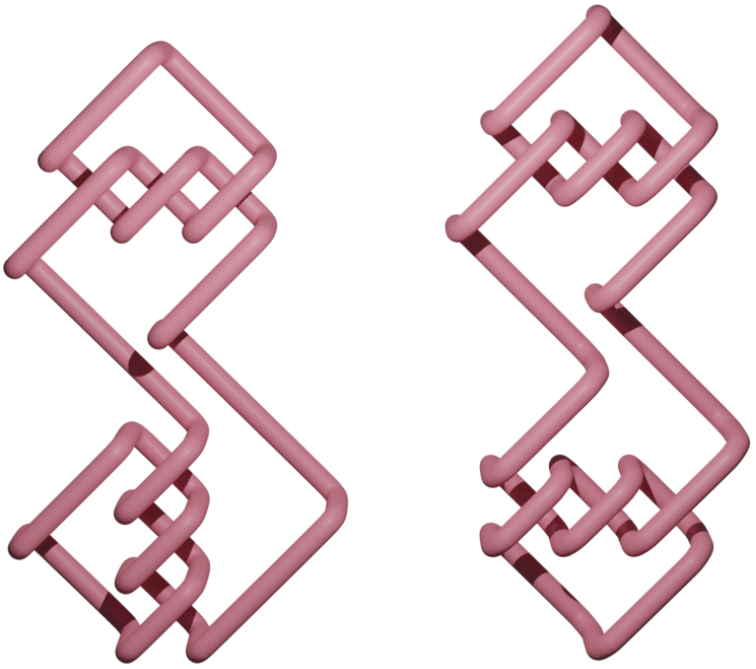


FIGURE 4.2.1: The granny knot (right) and the reef knot (left).

```
gap> trefoil:=PureCubicalKnot(3,1);;
gap> granny:=KnotSum(trefoil,KnotReflection(trefoil));;
gap> reef:=KnotSum(trefoil,trefoil);;
gap> AlexanderPolynomial(granny);
#I  there are 3 generators and 2 relators of total length 12
x_1^4-2*x_1^3+3*x_1^2-2*x_1+1
gap> AlexanderPolynomial(reef);
#I  there are 3 generators and 2 relators of total length 14
x_1^4-2*x_1^3+3*x_1^2-2*x_1+1
```

GAP SESSION 4.2.1: Calculating the Alexander polynomials of the granny and reef knots, both of which are $x^4 - 2x^3 + 3x^2 - 2x + 1$.
Runtime: 7s 419ms.

interested only in the first integral homology group $H_1(B_i, \mathbb{Z}) = H_1(B_i', \mathbb{Z})$ and so there is no difference in working with either B_i or with B_i', apart from the fact that B_i is a bit easier to construct.

For any finite index subgroup $H \leq \pi_1 Y_i$, the HAP functions `U:=UniversalCover(Y)` (A.1) and `p:=EquivariantCWComplexToRegularCWMap(U,H)` (A.4) can be used to construct the covering map $p_i : \widetilde{Y}_{i,H} \to Y_i$ which maps $\pi_1 \widetilde{Y}_{i,H}$ isomorphically onto the subgroup H. We can then use the function `LiftedRegularCWMap(f,p)` (A.5) to construct the subspace $p_i^{-1}(B_i) \subset \widetilde{Y}_{i,H}$ that maps onto B_i, together with the

```
gap> trefoil:=PureCubicalKnot(3,1);;
gap> granny:=KnotSum(trefoil,KnotReflection(trefoil));
prime knot 1 with 3 crossings + Reflected( prime knot 1 with 3 cros\
sings )

gap> reef:=KnotSum(trefoil,trefoil);
prime knot 1 with 3 crossings + prime knot 1 with 3 crossings

gap> l_1:=ArcPresentation(granny);;
gap> l_2:=ArcPresentation(reef);;
gap> Y_1:=KnotComplement(l_1);
Regular CW-complex of dimension 3

gap> Homology(Y_1,0);
[ 0 ]
gap> Homology(Y_1,1);
[ 0 ]
gap> Homology(Y_1,2);
[ 0 ]
gap> Homology(Y_1,3);
[  ]
gap> Y_2:=KnotComplement(l_2);
Regular CW-complex of dimension 3

gap> Homology(Y_2,0);
[ 0 ]
gap> Homology(Y_2,1);
[ 0 ]
gap> Homology(Y_2,2);
[ 0 ]
gap> Homology(Y_2,3);
[  ]
```

GAP Session 4.2.2: Computing the integral homology groups of the granny and square knots.
Runtime: 57ms.

inclusion mapping $\widetilde{f}_i : p_i^{-1}(B_i) \hookrightarrow \widetilde{Y}_{i,H}$. The induced homology homomorphism

$$\bar{f}_i : H_1(p_i^{-1}(B_i), \mathbb{Z}) \to H_1(\widetilde{Y}_{i,H}, \mathbb{Z})$$

is readily computed. The homomorphism $\bar{f}_i$ is a homeomorphism invariant of Y_i. However, in general, the space $p_i^{-1}(B_i)$ is a disjoint union $p_i^{-1}(B_i) = B_{i,1} \sqcup B_{i,2} \sqcup \cdots \sqcup B_{i,t}$ of path-connected CW-subspaces $B_{i,j}$. Therefore the homomorphism $\bar{f}_i$ is

a sum of homomorphisms

$$\bar{f}_{i,j} : H_1(B_{i,j}, \mathbb{Z}) \to H_1(\widetilde{Y}_{i,H}, \mathbb{Z}).$$

The set of all abelian groups $\mathrm{coker}(\bar{f}_{i,j})$ arising as the cokernel of $\bar{f}_{i,j}$ is a homeomorphism invariant of Y_i.

For the reef knot κ_2 with some choice of subgroup $H < \pi_1 Y_2$ of index 6 and some choice of path component $P_{2,j}$, we find that $\mathrm{coker}(\bar{f}_{2,j}) = \mathbb{Z} \oplus \mathbb{Z}_2 \oplus \mathbb{Z}_2 \oplus \mathbb{Z}_8$. However, for the granny knot κ_1 we find that for all subgroups $H < \pi_1 Y_1$ of index 6 and all path components $P_{1,j}$, $\mathrm{coker}(\bar{f}_{1,j}) \ncong \mathbb{Z} \oplus \mathbb{Z}_2 \oplus \mathbb{Z}_2 \oplus \mathbb{Z}_8$. We conclude that Y_1 is not homeomorphic to Y_2. The sequence of computations exposing this homeomorphism inequivalence is shown in `GAP` Session 4.2.3.

These computations motivate the definition of the ambient isotopy invariant $\mathfrak{J}_c(\kappa)$ of a link $\kappa \subset \mathbb{R}^3$:

$$\begin{aligned}\mathfrak{J}_c(\kappa) = {} & \text{Set of isomorphism types of abelian groups arising} \\ & \text{as } \mathrm{coker}(h_j : H_1(B_j, \mathbb{Z}) \to H_1(\widetilde{X}_H, \mathbb{Z})) \text{ for some} \\ & c\text{-fold cover } p : \widetilde{X}_H \to X \text{ of } X = \mathbb{R}^3 \setminus \kappa, \text{ and for } B_j \\ & \text{some path component of } p^{-1}(B), \text{ where } B \text{ is the} \\ & \text{boundary of a small tubular neighbourhood of } \kappa.\end{aligned}$$

This definition allows us to to formalise the computations of `GAP` Session 4.2.3 in the following theorem.

Theorem 4.2.1. *The invariant $\mathfrak{J}_c(\kappa)$ is powerful enough to distinguish between knots $\kappa, \kappa' \subset \mathbb{R}^3$ whose complements have isomorphic fundamental groups and isomorphic integral homology, at least in one particular case.*

Proof. See `GAP` Session 4.2.3. □

```
gap> K:=PureCubicalKnot(3,1);
prime knot 1 with 3 crossings

gap> granny:=ArcPresentation(KnotSum(K,ReflectedCubicalKnot(K)));;
gap> reef:=ArcPresentation(KnotSum(K,K));;
gap> f_1:=KnotComplementWithBoundary(granny);
Map of regular CW-complexes

gap> f_2:=KnotComplementWithBoundary(reef);
Map of regular CW-complexes

gap> J6granny:=Set(FirstHomologyCoveringCokernels(f_1,6));
[ [ 0, 0, 0, 0 ], [ 0, 0, 0, 2 ], [ 0, 0, 2, 2 ], [ 0, 0, 2, 2, 2 ],
  [ 0, 0, 2, 3 ], [ 0, 0, 3, 3 ], [ 0, 2 ], [ 0, 2, 2, 2, 3 ],
  [ 0, 2, 2, 3 ], [ 0, 2, 2, 3, 3 ], [ 0, 3 ], [ 0, 3, 3, 3 ],
  [ 0, 8 ], [ 2, 2 ], [ 2, 2, 2, 2, 2, 2 ], [ 2, 2, 2, 2, 2, 4 ],
  [ 2, 2, 2, 3, 3 ], [ 2, 2, 3, 3 ], [ 2, 2, 3, 5 ], [ 2, 2, 3, 8 ],
  [ 3, 3, 3, 9 ] ]
gap> J6reef:=Set(FirstHomologyCoveringCokernels(f_2,6));
[ [ 0, 0, 0, 0 ], [ 0, 0, 0, 2 ], [ 0, 0, 2, 2, 2 ], [ 0, 0, 2, 3 ],
  [ 0, 0, 3, 3 ], [ 0, 2 ], [ 0, 2, 2, 2, 3 ], [ 0, 2, 2, 3 ],
  [ 0, 2, 2, 3, 3 ], [ 0, 2, 2, 8 ], [ 0, 3 ], [ 0, 3, 3, 3 ],
  [ 0, 8 ], [ 2, 2 ], [ 2, 2, 2, 2, 2, 2 ], [ 2, 2, 2, 2, 2, 4 ],
  [ 2, 2, 2, 3, 3 ], [ 2, 2, 3, 3 ], [ 2, 2, 3, 5 ], [ 2, 2, 3, 8 ],
  [ 3, 3, 3, 9 ] ]
gap> Difference(J6granny,J6reef);
[ [ 0, 0, 2, 2 ] ]
gap> Difference(J6reef,J6granny);
[ [ 0, 2, 2, 8 ] ]
```

GAP Session 4.2.3: Computing the invariants $\mathfrak{J}_6(\kappa_1)$ and $\mathfrak{J}_6(\kappa_2)$.
Runtime: 1min 9s 9ms.

4.3 Alternative approaches to small CW-structures for link complements

4.3.1 Thickening a knotted 1-manifold in 3-space

Consider the trefoil knot $\kappa : S^1 \hookrightarrow \mathbb{R}^3$ whose embedding is illustrated in Figure 4.3.1.1 (left). We set N equal to the image of κ. A diagram $(D^2, p(N))$ for the trefoil knot is shown in Figure 4.3.1.1 (centre). A CW-structure on D^2 is shown in Figure 4.3.1.1 (right). This CW-structure is designed to reflect the undercrossing and overcrossing information of the knot. Consider the unit interval $I = [0, 1]$ given

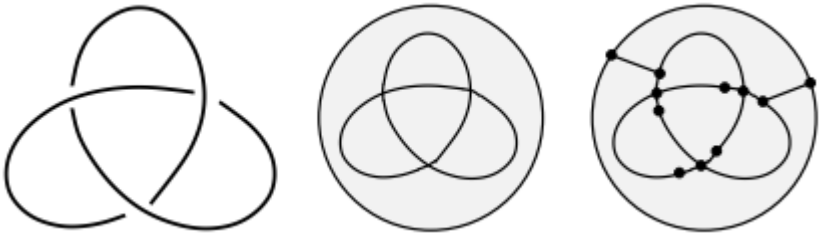


FIGURE 4.3.1.1: The trefoil knot 3_1.

CW-structure consisting of two 0-cells and one 1-cell. The CW-complex obtained from adjoining one 2-cell and one 3-cell to either end of the cylinder $D^2 \times I$ is homeomorphic to the 3-ball M and it contains, in its interior, a CW-subcomplex N ambient isotopic to the trefoil knot as shown in Figure 4.3.1.2.

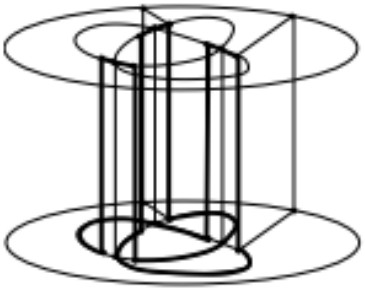


FIGURE 4.3.1.2: A CW-subcomplex of M ambient isotopic to the trefoil knot.

This procedure readily extends to an algorithm which inputs some symbolic representation of a knot or link and returns a regular CW-decomposition of the 3-ball M containing the specified knot or link N as a subcomplex of the 1-skeleton. This is implemented in `HAP` as the function `ArcPresentationToKnottedOneComplex` (A.9). However, we are interested in the complement $M \setminus N$, and the difference of CW-complexes is not a CW-complex. This barrier can be overcome by constructing a small open tubular neighbourhood N_ϵ of N and extending the cell structure of $M \setminus N$ to a CW-structure on $M \setminus N_\epsilon$. The exact details of this tubular neighbourhood construction are addressed in Section 5.1. The code of `GAP` Session 4.3.1.1 constructs a regular CW-complex homeomorphic to $M \setminus N_\epsilon$ for N the figure-eight knot.

4.3.2 Link complements in the 3-sphere

Often in theoretical knot theory, when one wants to investigate the complement of some link κ it is usually in reference to $S^3 \setminus \kappa$. The reason for this is typically to avail

```
gap> fig_eight:=ArcPresentation(PureCubicalKnot(4,1));
[ [ 3, 5 ], [ 4, 6 ], [ 2, 5 ], [ 1, 3 ], [ 2, 6 ], [ 1, 4 ] ]
gap> K:=ArcPresentationToKnottedOneComplex(fig_eight);
Map of regular CW-complexes

gap> K:=RegularCWComplexComplement(K);
Testing contractibility...
247 out of 247 cells tested.
The input is compatible with this algorithm.
Regular CW-complex of dimension 3

gap> Size(K);
355
```

GAP Session 4.3.1.1: Thickening the figure-eight knot.
Runtime: 272ms.

of the compactness of the S^3. While our choice to consider link complements in B^3 has not affected our calculations thus far, for consistency we have included in HAP the algorithm `SphericalKnotComplement` (A.10) for computing link complements in the 3-sphere. Furthermore, this function makes use of an algorithm for simplifying a given regular CW-complex in order to reduce the number of cells. This simplification process is detailed in Section 5.3.

The `S:=SphericalKnotComplement(arc)` function inputs an arc presentation and computes the complement of the tubular neighbourhood of a knotted 1-complex in the 3-ball exactly as detailed in the previous section. We then use `f:=BoundaryMap(S)` to obtain an inclusion map from the subspace of B^3 consisting of the closure of all of those 2-cells which have exactly one 3-cell in their coboundary into B^3. By employing the `PathComponentsCWSubcomplex` (A.20) function, we can identify the path component of this subcomplex corresponding to the boundary sphere of B^3. By adding to our subcomplex an additional 3-cell whose boundary is all of those 2-cells in this boundary sphere, we obtain our link complement in the 3-sphere as desired.

4.3.3 Relative efficiency of different link complement algorithms

This section compares various implementations of link complements on the basis of number of cells and computation time. We present a total of four different methods,

one of which returns a pure cubical complex while the rest return regular CW-complexes. As can be seen in Table 4.3.3.1, even in the simplest case, the method involving cubical complexes yields drastically larger cell complexes while taking the most time. For this reason, in our more extensive testing on all prime knots on fewer than twelve crossings, we opt to consider only those methods producing regular CW-complexes. Note that the arc presentations for these knots are pre-stored in `HAP`.

	Number of cells	Number of cells after simplification	Time to compute (ms)
`PureComplexComplement( PureCubicalKnot )`	13291	1837	26349
`KnotComplement`	395	141	85
`RegularCWComplexComplement( ArcPresentationToKnottedOneComplex )`	255	107	197
`SphericalKnotComplement`	256	128	108

TABLE 4.3.3.1: Constructing the trefoil with arc presentation $[[2,5],[1,3],[2,4],[3,5],[1,4]]$ for each algorithm.

`GAP` Session 4.3.3.2 employs the remaining three methods in order to:

(i) obtain regular CW-decompositions of all 801 prime knots on at most eleven crossings,

(ii) record the size of each complex before and after we simplify the CW-structure using Algorithm 5.3.1, and

(iii) record the time taken to both construct and simplify each complex.

The mean complex sizes and computation times are presented in Figure 4.3.3.1. We can see that, almost without exception, successively applying `ArcPresentationToKnottedOneComplex` and `RegularCWComplexComplement` yields the smallest complexes while `KnotComplement` is the fastest to compute.

```
gap> fn:=function(x,y,alg)
>     local
>         t, arc, comp, size, ssize;
>     t:=Runtime();
>     arc:=ArcPresentation(
>         PureCubicalKnot(x,y)
>     );
>     if alg=1 then
>         comp:=KnotComplement(arc);
>     elif alg=2 then
>         comp:=RegularCWComplexComplement(
>             ArcPresentationToKnottedOneComplex(
>                 arc
>             )
>         );
>     else
>         comp:=SphericalKnotComplement(arc);
>     fi;
>     size:=Size(comp);
>     ssize:=Size(SimplifiedComplex(comp));
>     return [size,ssize,Runtime()-t];
> end;
gap> prime:=[0,0,1,1,2,3,7,21,49,165,552];
gap> p:=[1..3];
gap> for i in [1..3] do
>     p[i]:=List([3..11],x->List([1..prime[x]],y->fn(x,y,i)));
> od;
gap> for i in [1..9] do # accounting for the simplification already
>     for j in [1..prime[i+2]] do # done in SphericalKnotComplement
>         p[3][i][j][1]:=p[2][i][j][1]+1;
>     od;
> od;
gap> data:=List(p,x->List(x,Average));
```

GAP Session 4.3.3.2: Computing the data for Figure 4.3.3.1.
Runtime: 36min 3s 178ms.

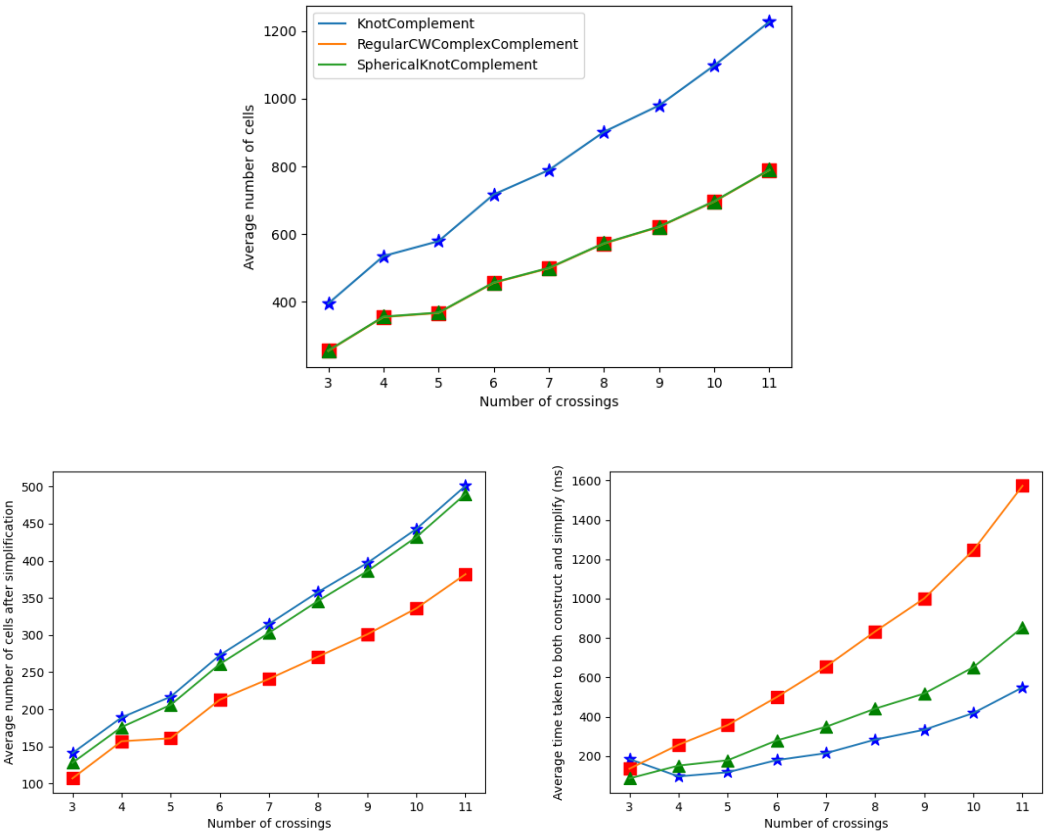


FIGURE 4.3.3.1: A comparison of link complement algorithms.

Chapter 5

Constructions on regular CW-complexes

There are a number of topological constructions that we have called upon thus far without mention of how they might be implemented on a computer. This chapter addresses implementations of these constructions in the case of regular CW-complexes. Of particular focus is the tubular neighbourhood which is pivotal in our goal of encoding knotted surfaces as regular CW-complexes (see Chapter 6). Our implementation of the tubular neighbourhood touches on the need for subdividing subcomplexes of regular CW-complexes which drastically increases the size of the resulting spaces. A procedure for reducing the number of cells of a regular CW-complex while preserving homeomorphism type is illustrated in Section 5.3 which can help combat the excess of cells in some way. Lastly, we need a procedure for performing Artin spinning on regular CW-complexes to give us access to a class of knotted surfaces. Theorem 3.3.1 requires a CW-decomposition of the spun Hopf link, for example.

5.1 Tubular neighbourhoods

Let X denote a finite regular CW-complex containing a CW-subspace $Y \subset X$. In this section we describe the construction of a finite CW-complex W that models the complement of $X \setminus N_\epsilon(Y)$ of a small open tubular neighbourhood of Y. Note that the difference of two regular CW-complexes is not a regular CW-complex, and that $N_\epsilon(Y)$ is not a regular CW-complex due to its openness. We opt not to give the

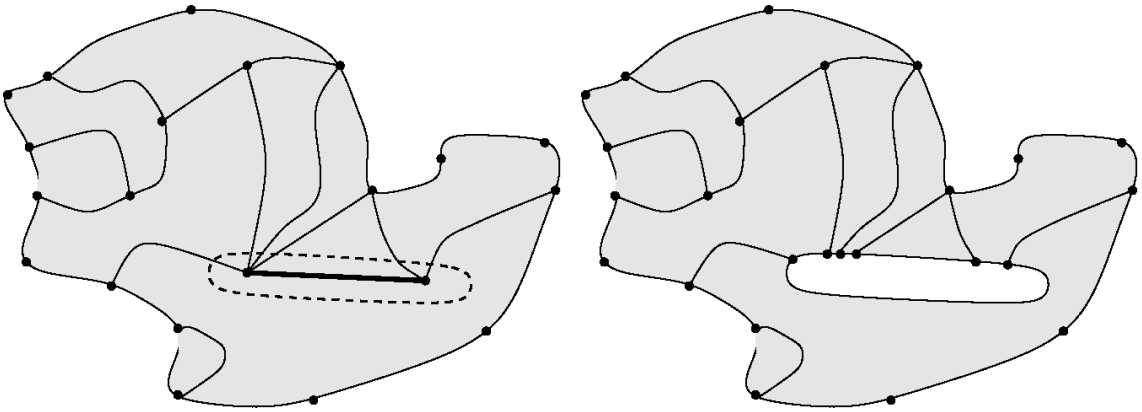


FIGURE 5.1.1: A CW-complex X with open neighbourhood of a subcomplex Y (left) and the CW-complex W for this choice of X and Y (right).

precise definition of the open subspace $N_\epsilon(Y) \subset X$ (which is routinely formulated under the additional assumption that X is piecewise Euclidean), and instead decide to focus on an exact description of the space W. The examples we have in mind are where X is some contractible region of $\mathbb{R}^n$ for $n = 3, 4$ and Y is an embedded circle in the case $n = 3$ (see Section 4.3.1) or an embedded surface in the case $n = 4$ (see Chapter 6).

To describe a procedure for constructing the space W we first introduce some terminology and notation for enumerating the cells of W and for describing their homological boundaries. The CW-complex W will consist of all of the cells $X \setminus Y$ together with some extra cells. We say that a cell of W is *internal* if it lies in $X \setminus Y$, and that it is *external* otherwise. The complement $X \setminus Y$ is a cell complex—a union of open cells—but it is not in general a CW-complex. The external cells ensure that W is a CW-complex.

Figure 5.1.1 (left) shows a contractible region X in the plane $\mathbb{R}^2$ endowed with a CW-structure involving 10 2-cells, 30 1-cells, and 21 0-cells. A 1-dimensional subcomplex $Y \subset X$, involving 1 1-cell and 2 0-cells, is shown in bold. An open tubular neighbourhood $N_\epsilon(Y) \subset X$ is shown enclosed by dotted lines. Figure 5.1.1 (right) shows the CW-complex W corresponding to this particular choice of X and Y.

The closure $\overline{e^n}$ of any n-cell $e^n \subset X$ is a finite CW-subcomplex of X. For an internal n-cell $e^n \subset X \setminus Y$ the intersection $\overline{e^n} \cap Y$ is also a CW-subcomplex of X, which we

can express as a union

$$\overline{e^n} \cap Y = A_1^{e^n} \cup A_2^{e^n} \cup \cdots \cup A_k^{e^n}$$

of its path components $A_i^{e^n}$. Each $A_i^{e^n}$ is a CW-subcomplex of Y.

We will make the following simplifying assumption: let us suppose that each $A_i^{e^n}$ is contractible. We will refer to this assumption as the contractible closure assumption. When a space fails to meet the contractible closure assumption it is sometimes possible to apply barycentric subdivision (or some less costly subdivision) to the offending top-dimensional cells in order to modify the CW-structure so that it does satisfy the contractible closure assumption.

Under the contractible closure assumption the CW-complex W has precisely one internal n-cell e^n for each cell $e^n \subset X \setminus Y$, and precisely one external $(n-1)$-cell $f_{A_i}^{e^n}$ for each path component A_i of $\overline{e^n} \cap Y$.

The homological boundary of an internal cell e^n in W is a sum of all the internal $(n-1)$-cells lying in $\overline{e^n}$ together with all external $(n-1)$-cells $f_{A_i}^{e^n}$. The homological boundary of an external $(n-1)$-cell $f_{A_i}^{e^n}$ is a sum of all those external $(n-1)$-cells $f_B^{e^{n-1}}$ with e^{n-1} an $(n-1)$-cell of $\overline{e^n}$ and path component B of $\overline{e^n} \cap Y$ with $B \subseteq A_i$. This procedure is formalised in Algorithm 5.1.1.

Algorithm 5.1.1 Excise an open tubular neighbourhood from a regular CW-complex.

Input:

A finite regular CW-complex X and pure subcomplex $Y \subset X$ satisfying the above contractible closure assumption.

Output:

A finite regular CW-complex W.

1: For each internal cell $e^n \subset X \setminus Y$ compute the CW-complex $\overline{e^n} \cap Y$ and express it as a union $\overline{e^n} \cap Y = A_1^{e^n} \cup A_2^{e^n} \cup \cdots \cup A_k^{e^n}$ of its path components $A_i^{e^n}$. This determines the number of cells of each dimension in W.

2: Create a list of empty lists $B = [[], [], \ldots, []]$ of length $1 + \dim X$.

3: For $0 \leq n \leq \dim X$ set $B[n+1] = [b_1, b_2, \ldots, b_{\alpha_n}]$ where α_n denotes the number of n-cells in W and b_1 is a list of integers determining the $(n-1)$-cells with non-zero coefficient in the homological boundary of the i^{th} n-cell of W. In the ordering of n-cells it is convenient to order all internal cells before any external cell.

4: Represent the data in B as a regular CW-complex W and return W.

This procedure is implemented in HAP as the RegularCWComplexComplement(f) (A.25) function, which inputs an inclusion of regular CW-complexes $f : Y \hookrightarrow X$ and returns the regular CW-complex W, modelling the space $X \setminus N_\epsilon(Y)$. The code of GAP Session 5.1.1 showcases this algorithm when applied to two different knots, one of which satisfies the contractible closure assumption and the other which does not and hence requires some subdivision.

```
gap> K:=SphericalKnotComplement([[1,2],[1,2]]);;
gap> Size(K);
28
gap> K:=BoundaryMap(K);;
gap> K:=RegularCWComplexComplement(K,"all","basic",true);
Testing contractibility...
28 out of 28 cells tested.
Subdividing 4 cell(s):
100% complete.
Testing contractibility...
100 out of 100 cells tested.
The input is compatible with this algorithm.
Regular CW-complex of dimension 3

gap> Size(K);
144
gap> K:=SphericalKnotComplement(
>   ArcPresentation(PureCubicalKnot(3,1))
>);;
gap> Size(K);
128
gap> K:=BoundaryMap(K);;
gap> K:=RegularCWComplexComplement(K);
Regular CW-complex of dimension 3

gap> Size(K);
90
```

GAP Session 5.1.1: Computing tubular neighbourhoods of the unknot and the trefoil knot in S^3. The former requires subdivision before it is compatible with Algorithm 5.1.1.
Runtime: 208ms.

5.1.1 Cell-by-cell construction of tubular neighbourhoods

A necessity of our implementation of the tubular neighbourhood is that the contractible closure assumption holds. We will see in Section 5.2 that our solution of

subdividing the top-dimensional cells that cause the contractible closure assumption to fail can result in immensely large cell complexes. To this end, it is desirable to implement a tubular neighbourhood algorithm that bypasses any potential need for subdivision.

As before, let X denote a finite n-dimensional regular CW-complex with m-dimensional subcomplex $Y \subset X$ for $m \leq n$. We begin by considering the top-dimensional cells of Y. Let $e^m_{i_1}$ be such a top-dimensional cell. We construct the regular CW-complex W_{i_1} which models the complement $X \setminus N_\epsilon(\overline{e^m_{i_1}})$ as per Algorithm 5.1.1. By restricting our scope to a subcomplex which arises as the closure of just one cell, we sometimes satisfy the contractible closure assumption.

For the next top dimensional cell, $e^m_{i_2}$, we construct the space W_{i_2} which is homotopy equivalent to $X \setminus N_\epsilon(\overline{e^m_{i_1}} \cup \overline{e^m_{i_2}})$. We repeat the application of Algorithm 5.1.1 for all k top-dimensional cells resulting in the space W_{i_k}, homotopy equivalent to $X \setminus N_\epsilon(\bigcup_{j=1}^k \overline{e^m_{i_j}})$. Thus, a necessary condition of our cell-by-cell formation of the tubular neighbourhood of a subcomplex Y is that Y must arise as the closure of the union of its top-dimensional cells, *i.e.*, $Y = \bigcup_{j=1}^k \overline{e^m_{i_j}}$. In other words, Y must be a *pure* regular CW-complex. For our purposes—in constructing link complements and knotted surface complements—the condition of Y being a pure regular CW-complex does not pose any problems as it is something that is automatically satisfied by algorithms A.7, A.8 and A.10 by design.

In Figure 5.1.1.1, we illustrate the advantages of this procedure by applying it to a 1-dimensional subcomplex homotopy equivalent to a circle (highlighted in bold) of a contractible 2-dimensional regular CW-complex. Note that Algorithm 5.1.1 applied to this example would require some subdivision. We formalise this sequential cell-by-cell excision of a tubular neighbourhood in Algorithm 5.1.1.1.

The cell-by-cell method is also implemented in `HAP` and can be accessed via the function `SequentialRegularCWComplexComplement` (A.26). It can be seen in use in `GAP` Session 5.1.1.1, being applied to the knot which previously required subdivision in `GAP` Session 5.1.1.

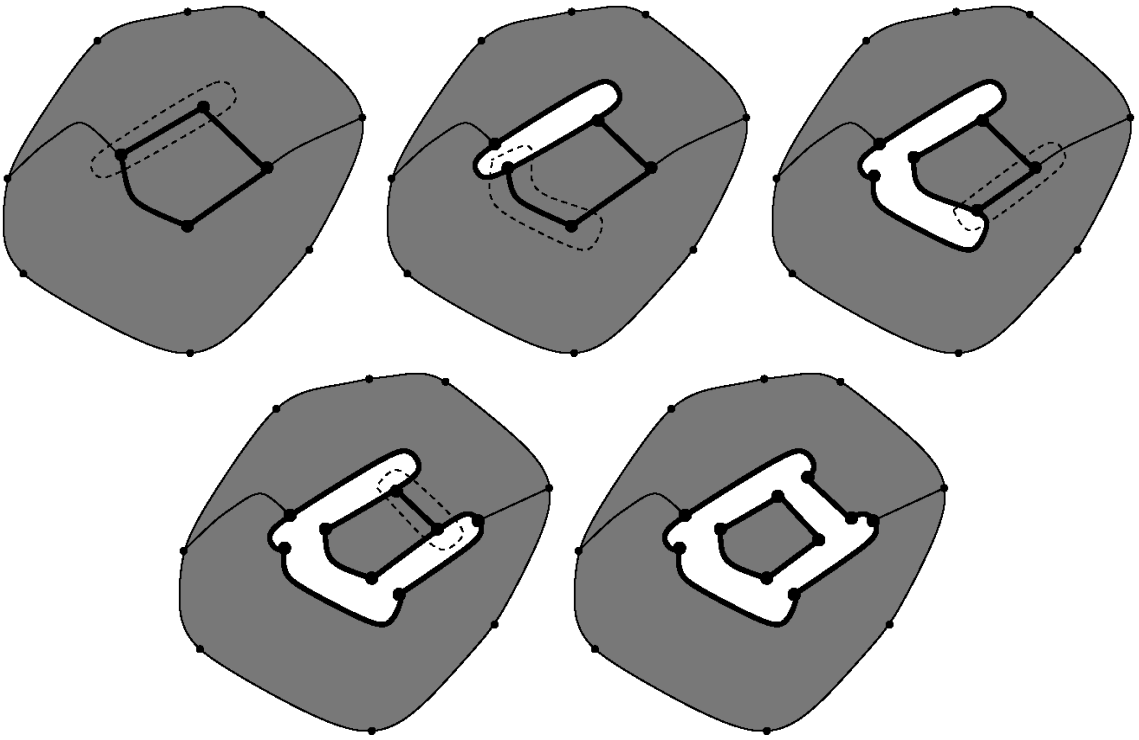


FIGURE 5.1.1.1: Cell-by-cell excision of a tubular neighbourhood.

```
gap> K:=SphericalKnotComplement([[1,2],[1,2]]);;
gap> Size(K);
28
gap> K:=BoundaryMap(K);;
gap> K:=SequentialRegularCWComplexComplement(K);
Regular CW-complex of dimension 3

gap> Size(K);
39
```

GAP SESSION 5.1.1.1: Computing a tubular neighbourhood of the unknot using the cell-by-cell method. Note that the use of this method bypasses the need for subdivision which was needed in GAP Session 5.1.1. Also of note is that the cell-by-cell method produces a complex of size 39 versus the subdivision method which produces a complex of size 144.
Runtime: 50ms.

5.2 Subdivision

5.2.1 Barycentric subdivision

Our implementation of the tubular neighbourhood construction requires us to consider procedures for subdividing cells of a regular CW-complex. There are many such

Algorithm 5.1.1.1 Sequentially excise an open tubular neighbourhood from a regular CW-complex.

Input:
A finite regular CW-complex X and pure subcomplex $Y \subset X$.

Output:
A finite regular CW-complex W.

1: Set $j = 1$.
2: Let m denote the dimension of Y. Compute the complex $Y_j = \overline{e_j^m}$.
3: For each internal cell $e^n \subset X \setminus Y_j$ compute the CW-complex $\overline{e^n} \cap Y_j$ and express it as a union $\overline{e^n} \cap Y_j = A_1^{e^n} \cup A_2^{e^n} \cup \cdots \cup A_k^{e^n}$ of its path components $A_i^{e^n}$. This determines the number of cells of each dimension in W.
4: Create a list of empty lists $B = [[], [], \ldots, []]$ of length $1 + \dim X$.
5: For $0 \leq n \leq \dim X$ set $B[n+1] = [b_1, b_2, \ldots, b_{\alpha_n}]$ where α_n denotes the number of n-cells in W and b_1 is a list of integers determining the $(n-1)$-cells with non-zero coefficient in the homological boundary of the i^{th} n-cell of W. In the ordering of n-cells it is convenient to order all internal cells before any external cell.
6: Represent the data in B as a regular CW-complex W.
7: Increase the value of j by 1.
8: Repeat steps 2 to 7 replacing X with W. Do this until j exceeds the number of top-dimensional cells of Y.
9: Return W.

subdivision procedures, the most well-known being the barycentric subdivision. Let $B(e^n)$ denote the regular CW-complex obtained from barycentrically subdividing some n-cell e^n where $n \geq 1$. In $B(e^n)$ there exists one k-cell for every sequence of length $k+1$ of cells $e^0 \subset e^1 \subset \ldots \subset e^k$, where $e^i \subset e^j$ denotes that e^i lies in the boundary of e^j. The boundaries of these k-cells are uniquely determined by their associated sequences. Our computer implementation of $B(e^n)$ applies to the closure of a given n-cell of some regular CW-complex. For this reason it is necessary to sequentially rewrite the boundaries and coboundaries of all cells which intersect e^n. The construction of $B(e^n)$ begins by letting $B(e^n) = \overline{e^n}$, the regular CW-complex arising as the closure of e^n.

Step 1. For each 1-cell e_α^1 of $B(e^n)$, we add a 0-cell to $B(e^n)$ denoted by b_α.

Step 2. Unbind each 1-cell e_α^1 of $B(e^n)$. Let ∂e_α^1 denote the boundary of e_α^1. For each 0-cell in ∂e_α^1, we add a 1-cell to $B(e^n)$ whose boundaries consist of a cell from ∂e_α^1 and the cell b_α from the previous step. Should there have been any 2-cells in the coboundary of e_α^1, glue the 1-cells added by this step to the boundary of these coboundary 2-cells in place of e_α^1.

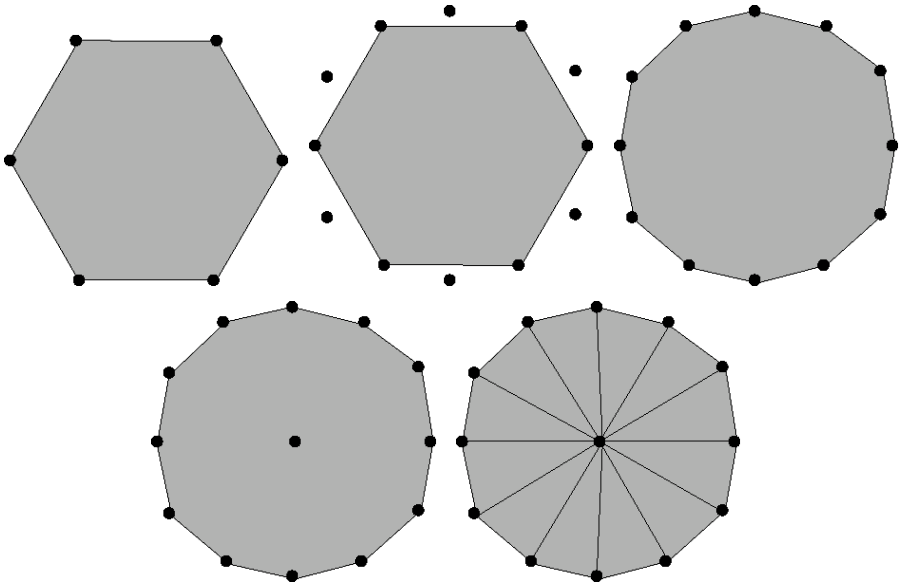


FIGURE 5.2.1.1: Barycentrically subdividing a 2-cell consisting of six 0-cells and six 1-cells.

Step 3. For each 2-cell e^2_α of $B(e^n)$, we add a 0-cell to $B(e^n)$ denoted by b_α.

Step 4. Unbind each 2-cell e^2_α of $B(e^n)$. Let ∂e^2_α denote the boundary of e^2_α. For each 0-cell in ∂e^2_α, we add a 1-cell to $B(e^n)$ whose boundaries consist of a cell from ∂e^2_α and the cell b_α from the previous step. For each 1-cell in ∂e^2_α, we add a 2-cell to $B(e^n)$ whose boundaries consist of a cell from ∂e^2_α, say $e^1_{\alpha_i}$, and the two 1-cells added earlier in this step associated to the 0-cells in $\partial e^1_{\alpha_i}$. Should there have been any 3-cells in the coboundary of e^2_α, glue the 2-cells added by this step to the boundary of these coboundary 3-cells in place of e^2_α.

$\vdots$

This procedure will continue for $2n$ steps. An example of this implementation of barycentric subdivision is illustrated in Figure 5.2.1.1. Furthermore, barycentric subdivision is implemented in `HAP` as the function `BarycentricallySubdivideCell` (A.23). The code in `GAP` Session 5.2.1.1 shows this algorithm running in a higher dimensional context.

Barycentric subdivision will drastically increase the number of cells in a regular CW-complex. In the case of simplicial complexes for example, the barycentric subdivision of an n-simplex will consist of $(n+1)!$ n-simplices. For this reason, our implementation subdivides only a single cell at a time as opposed to subdividing

```
gap> S1:=[[1,1,1],[1,0,1],[1,1,1]];;
gap> S1:=PureCubicalComplex(S1);;
gap> T:=DirectProduct(S1,S1);;
gap> T:=RegularCWComplex(T);
Regular CW-complex of dimension 4

gap> e3:=ClosureCWCell(T,3,200);;
gap> e3:=CWSubcomplexToRegularCWMap(e3);
Map of regular CW-complexes

gap> Be3:=BarycentricallySubdivideCell(e3,3,200);
Map of regular CW-complexes

gap> Size(Target(e3));
2304
gap> Size(Target(Be3));
2570
```

GAP Session 5.2.1.1: Barycentrically subdividing a 3-cell of the Clifford torus arising as the direct product of two 2-dimensional regular CW-complexes, each homeomorphic to S^1.
Runtime: 434ms.

an entire regular CW-complex. A downside to this individual cell approach can be seen, for example, if there were two cells with non-empty intersection that needed to be subdivided. In Figure 5.2.1.2, we can see an example of an ideal approach to barycentric subdivision that does not subdivide the intersection of any two cells more than once. However in Figure 5.2.1.3, we can see our implementation of barycentric subdivision applied to the same context which results in comparatively more cells. Of course, it is possible to implement the approach of Figure 5.2.1.2 but for our purposes, it is best to focus on methods that do not rely on barycentric subdivision whatsoever.

5.2.2 A basic subdivision

Our justification for implementing the barycentric subdivision arose from the issues posed by failing the contractible closure condition in Section 5.1. However, it is often the case that barycentric subdivision excessively subdivides cells where a much more basic subdivision would have been sufficient in helping us satisfy the contractible closure condition. In Figure 5.2.2.1, we can see a 1-dimensional subcomplex Y of a 2-dimensional regular CW-complex X containing a 2-cell that intersects Y in a

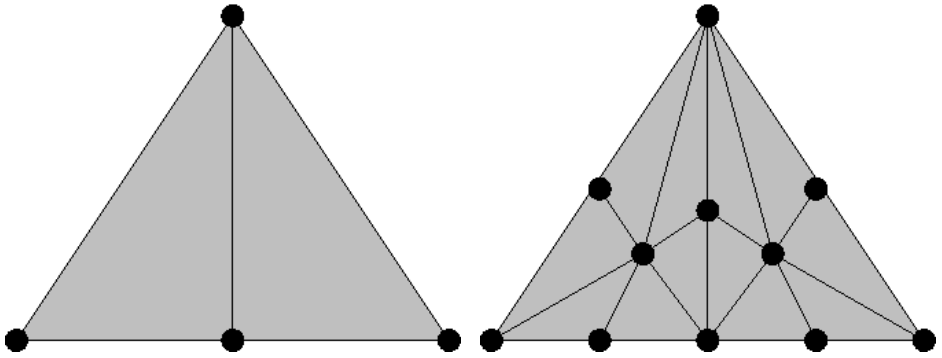


FIGURE 5.2.1.2: Barycentrically subdividing two 2-cells simultaneously in such a way that does not subdivide their intersection twice.

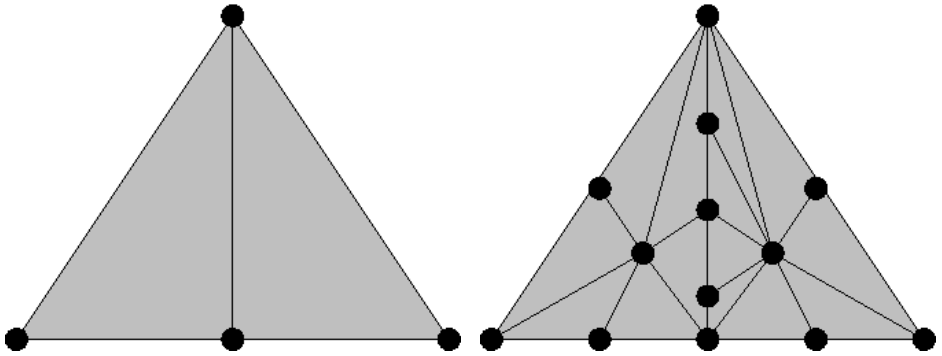


FIGURE 5.2.1.3: Barycentrically subdividing two 2-cells one after the other as per the function `BarycentricallySubdivideCell`.

non-contractible way. We can barycentrically subdivide this offending 2-cell yielding four 2-cells in its place, but for the purposes of satisfying the contractible closure condition it is sufficient to divide the 2-cell in two.

To this end we have implemented `SubdivideCell` (A.24), which divides an n-cell into as many n cells as there are $(n-1)$-cells in its boundary. It is essentially the procedure described in Section 5.2.1 without any of the odd-numbered steps. Let $B^*(e^n)$ denote the basic subdivision of e^n. Begin by letting $B^*(e^n)$ equal the interior of e^n, *i.e.*, $\overline{e^n} \setminus e^n$. Add to $B^*(e^n)$ a single 0-cell denoted by b. Each 0-cell of $B^*(e^n)$ yields a 1-cell whose boundaries consist of b and another 0-cell. For each m-cell of $B^*(e^n)$, $m \in [1, n-1]$, we add to $B^*(e^n)$ an $(m+1)$-cell whose boundary consists of said m-cell and the m-cells that arise from the $(m-1)$-cells of its boundary.

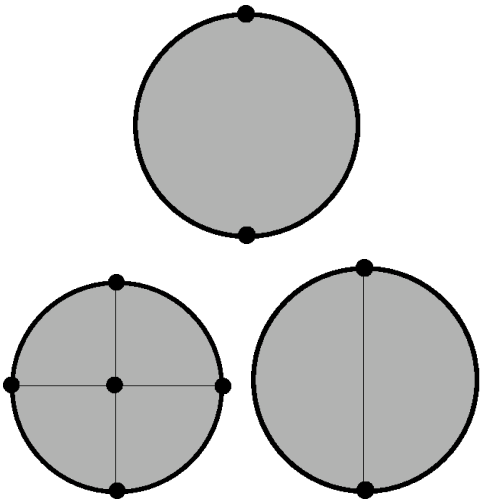


FIGURE 5.2.2.1: Two different subdivisions of a 2-cell with two 1-cells in its boundary.

```
gap> arc:=ArcPresentation(PureCubicalKnot(11,12));;
gap> K:=SphericalKnotComplement(arc);;
gap> K:=BoundaryMap(K);;
gap> Be3:=BarycentricallySubdivideCell(K,3,5);
Map of regular CW-complexes

gap> B_e3:=SubdivideCell(K,3,5);
Map of regular CW-complexes

gap> Size(Target(Be3));
1072
gap> Size(Target(B_e3));
512
```

GAP SESSION 5.2.2.1: Comparing the size of a complex after having (i) barycentrically subdivided one of its 3-cells and (ii) subdivided the same 3-cell using A.24. **Runtime: 4s 95ms.**

5.3 Simplification of regular CW-structure

Recall from [8] a simple method for reducing the number of cells in a regular CW-complex X. This simplification procedure is based on the observation that if a regular CW-complex X contains n-cell e^n lying in the boundary of precisely two $(n+1)$-cells e_1^{n+1}, e_2^{n+1} with identical coboundaries then these three cells can be removed and replaced by a single cell of dimension $n+1$. The topological space X is unchanged; only its CW-structure is altered. In general, the resulting CW-complex after this simplification will not be regular. The condition for it being regular is if

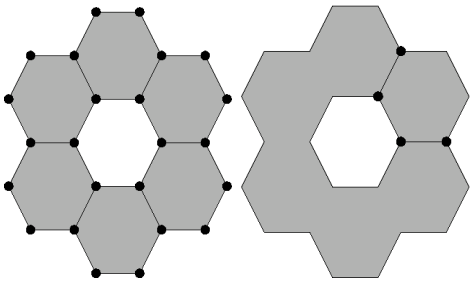


FIGURE 5.3.1: Illustration of Algorithm 5.3.1 on a regular CW-complex.

the sets V_0, V_1, V_2 of cells lying in the boundaries of $e^n, e_1^{n+1}, e_2^{n+1}$ respectively are such that $V_1 \cap V_2 = V_0 \cup \{e^n\}$. We formalise this procedure as Algorithm 5.3.1.

Algorithm 5.3.1 Simplification of a regular CW-complex.

Input:
A regular CW-complex Y.

Output:
A regular CW-complex X with $\text{Size}(X) \leq \text{Size}(Y)$ and with X homeomorphic to Y.

1: Let X be a copy of Y.
2: **while** there exists a cell e^n in X with precisely two cells e_1^{n+1}, e_2^{n+1} in its coboundary which have identical coboundaries **do**
3: Compute the sets V_0, V_1, V_2 of cells in the boundaries of $e^n, e_1^{n+1}, e_2^{n+1}$.
4: **if** $|V_1 \cap V_2| = 1 + |V_0|$ **then**
5: Remove the cells $e^n, e_1^{n+1}, e_2^{n+2}$ from X and add a new $(n+1)$-cell f^{n+1} to X whose boundary is the union of the boundaries of e_1^{n+1} and e_2^{n+1} minus the cell e^n. Adjust coboundaries accordingly.
6: **end if**
7: **end while**
8: Return X.

5.4 Artin spinning

We recall details on a spinning construction for links, the origins of which go back to Artin[34]. Artin used spinning to construct 4-dimensional knots from classical knots, but we shall give a more general topological description.

Let X be a topological space with subspace B. We define the space obtained by *spinning X about B* to be

$$S_B(X) = X \times [0,1]/\{(x,t) = (x,0) \text{ if } x \in B \text{ or } t = 1\}.$$

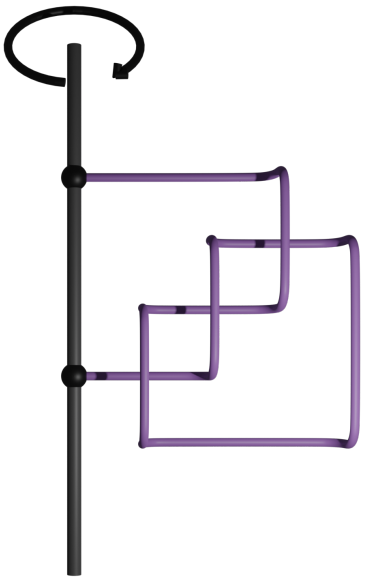


FIGURE 5.4.1: Spinning the trefoil about a hyperplane.

An alternative space $\hat{S}_B(X)$ can be formed from X using the closed 2-disk D^2 and its boundary circle S^1. We set

$$\hat{S}_B(X) = (B \times D^2) \cup (X \times S^1) \subset X \times D^2.$$

Theorem 5.4.1. *The space $\hat{S}_B(X)$ is homotopy equivalent to $S_B(X)$.*

The proof of this theorem is left to the reader. The space $\hat{S}_B(X)$ rather than $S_B(X)$ is implemented in `HAP`. More precisely, suppose that X is a regular CW-complex with subcomplex $B \subset X$. Let D^2 be given a regular CW-structure involving two 0-cells, two 1-cells and one 2-cell. The direct product $X \times D^2$ is naturally a regular CW-complex, and $\hat{S}_B(X)$ is a subcomplex.

Artin was interested in the case where $X = \{(x, y, z) \in \mathbb{R}^3 : z \geq 0\}$ and $B = \{(x, y, z) \in \mathbb{R}^3 : z = 0\}$. In this case $S_B(X)$ is homeomorphic to $\mathbb{R}^4$, and any knot κ embedded in the interior of X gives rise to a knotted torus $S_\emptyset(\kappa)$ in $\mathbb{R}^4$. More generally, a link κ gives rise to an embedded surface $S_\emptyset(\kappa) \subset \mathbb{R}^4$. The complement $\mathbb{R}^4 \setminus S_\emptyset(\kappa)$ is homeomorphic to $S_B(X \setminus \kappa)$.

The Artin spin construction is implemented in `HAP` as the function `Spin(f)` (A.13). which inputs an inclusion of regular CW-complexes $f : X \hookrightarrow Y$ and returns a regular CW-complex obtained by spinning Y about X. The functions `SpunAboutHyperplane` (A.14), `SpunKnotComplement` (A.15), and `SpunkLinkComplement` (A.16) also employ this spin construction.

```
gap> arc:=ArcPresentation(PureCubicalKnot(7,3));;
gap> f:=SphericalKnotComplement(arc);
Regular CW-complex of dimension 3

gap> f:=Spin(BoundaryMap(f));
Regular CW-complex of dimension 4
gap> S1:=[[0],[1],[0]];;
gap> S1:=PureCubicalComplex(S1);
Pure cubical complex of dimension 2.

gap> Homology(S1,0);
[ 0 ]
gap> Homology(S1,1);
[  ]
gap> S1:=SpunAboutHyperplane(S1);
Regular CW-complex of dimension 1

gap> Homology(S1,0);
[ 0 ]
gap> Homology(S1,1);
[ 0 ]
gap> SpunKnotComplement([3,1]);
Regular CW-complex of dimension 5

gap> SpunLinkComplement([[2,4],[1,3],[2,4],[1,3]]);
Regular CW-complex of dimension 3
```

GAP Session 5.4.1: (i) Spinning the complement of 7_3 about its boundary, (ii) spinning a cube about a hyperplane to form a complex homeomorphic to S^1, (iii) removing an unknotted segment of the trefoil knot 3_1 and spinning it about a hyperplane, and (iv) spinning the Hopf link about a hyperplane.
Runtime: 2s 877ms.

`SpunAboutHyperplane(K)` inputs a pure cubical complex K and returns a regular CW-complex $S(K)$ obtained from spinning K about a hyperplane. The `SpunKnotComplement([N,M])` inputs a list of two integers N and M corresponding to M_N (the N^{th} prime knot on M crossings) and returns a 5-dimensional regular CW-complex K^* obtained by removing an unknotted segment from the knot M_N and spinning it about a hyperplane. Lastly, the function `SpunkLinkComplement(l)` simply inputs an arc presentation corresponding to some link and spins it about a hyperplane, returning a regular CW-complex. In GAP Session 5.4.1, these four functions are shown in use.

Chapter 6

Small CW-structures for knotted surface complements

In this chapter we seek to describe our computational implementation of knotted surfaces, which we recall are surfaces that are embedded in $\mathbb{R}^4$. Our first course of action is to establish a method of representing knotted surfaces. We do this by way of *colourings* which generalise the notion of temperatures from Section 3.3.

Definition 6.1. Let M be some surface embedded in $\mathbb{R}^4$ and let $p : M \to H$ denote a projection of M onto a hyperplane H disjoint from M such that $p(M)$ is a self-intersecting surface. A *colouring* is a pair $(p(M), c)$ where $c : M \to \mathbb{R}, m \mapsto d(m, p(m))$ is a map that sends a point $m \in M$ to the distance from that point to its image in the hyperplane.

By associating the range of values given by c to some colour gradient, we can illustrate a colouring by sketching $p(M)$ and assigning colours to each $s \in p(M)$ according to $d(p^{-1}(s), s)$. Should s be a singular point, *i.e.* should $|p^{-1}(s)| = 2$, we colour it according to a value of $p^{-1}(s)$ that maximises the distance from s.

For the class of knotted surfaces we consider in this chapter—the ribbon surface-links—it is unnecessary to allow for each point in our self-intersecting surface to be assigned a colour from a continuum of colours. We can instead limit our range to only finitely many colours (moreover just three colours) in order to have enough information to recover the homeomorphism type of M from a colouring of M (see Section 6.3 for details).

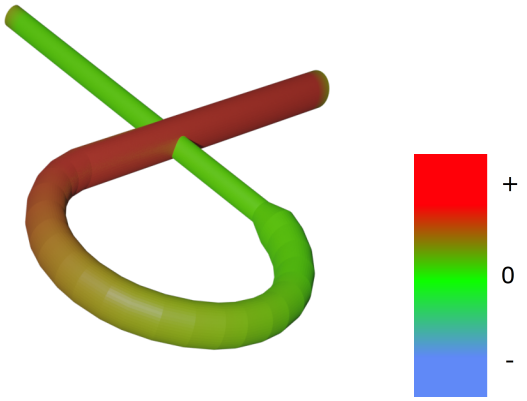


FIGURE 6.1: A colouring corresponding to an unknotted 2-sphere in $\mathbb{R}^4$ with a colour gradient indicating the height in the fourth spatial dimension.

Definition 6.2. Let M be some surface embedded in $\mathbb{R}^4$ and let $p : M \to H$ denote a projection of M onto a hyperplane H disjoint from M such that $p(M)$ is a self-intersecting surface. A *discrete colouring* is a pair $(p(M), C)$ where C is a finite partition of the range of values given by $c : M \to \mathbb{R}, m \mapsto d(m, p(m))$, a map that sends a point $m \in M$ to the distance from that point to its image in the hyperplane. We refer to the sets $U \in C$ as *temperatures*.

The convention we adopt in sketching discrete colourings is to colour those points of $p(M)$ lying in the temperature of points furthest away from their preimage in M red; those closest, blue; and those halfway between, green. Our reasoning for using just these three temperatures comes from an existing and widely used method of illustrating knotted surfaces, *broken surface diagrams*. In a broken surface diagram, we represent a knotted surface as a self-intersecting surface $p(M)$ with singular points s and illustrate it by excising open tubular neighbourhoods of points about each s from $p(M)$ such that only points closer to M are removed (see Figure 6.2). This procedure requires that we are able to capture, for two points $x, y \in p(M)$, any of the following:

(i) $d(p^{-1}(x), x) < d(p^{-1}(y), y)$

(ii) $d(p^{-1}(x), x) > d(p^{-1}(y), y)$

(iii) $d(p^{-1}(x), x) = d(p^{-1}(y), y)$.

Hence, we need only use three temperatures in our discrete colourings in order to have an analogous method of representing knotted surfaces.

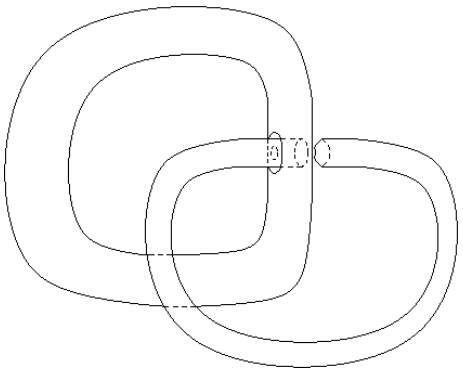


FIGURE 6.2: A broken surface diagram depicting the Tube of the Hopf link with one welded crossing.

6.1 Satoh's Tube map

In Section 3.3, we briefly mentioned the *Tube map* of Satoh[30] which we employed to gain access to a class of knotted surfaces referred to as the *ribbon torus-links*, a specific type of *ribbon surface-link*. Each of the italicised terms will now be defined.

Definition 6.1.1. [30] Let $\mathbb{S} = \bigsqcup_{i=1}^{r} S_i^2$ denote a disjoint union of 2-spheres embedded in $\mathbb{R}^3$. Let $h_j : D^2 \times I \hookrightarrow \mathbb{R}^3$ $(1 \leq j \leq s)$ denote a family of embeddings of solid cylinders into $\mathbb{R}^3$. These embedded cylinders $h_j(D^2 \times I)$ obey the following criteria:

- $h_j(D^2 \times I) \cap h_k(D^2 \times I) = \emptyset$ if $j \neq k$,
- $h_j(D^2 \times I) \cap \mathbb{S}$ is a disjoint union of 2-disks including $h_j(D^2 \times \partial I)$, and
- each $h_j(D^2 \times I)$ is attached to the exterior of $\mathbb{S}$.

A *self-intersecting ribbon surface-link* is defined as:

$$\mathfrak{S} = \left(\mathbb{S} \setminus \bigcup_{j=1}^{s} h_j(D^2 \times \partial I) \right) \cup \bigcup_{j=1}^{s} h_j(\partial D^2 \times I).$$

A *ribbon surface-link*, denoted by $\mathfrak{R}$, is a surface embedded in $\mathbb{R}^4$ whose projection onto a hyperplane is $\mathfrak{S}$. A *ribbon torus-link* is a ribbon surface-link, each of whose connected components are homeomorphic to a torus.

Satoh's *Tube map* is a construction which inputs an oriented welded link diagram L and outputs an embedding of a disjoint union of tori into $\mathbb{R}^4$:

$$\tau : (S^1 \times S^1) \cup (S^1 \times S^1) \cup \cdots \cup (S^1 \times S^1) \hookrightarrow \mathbb{R}^4,$$

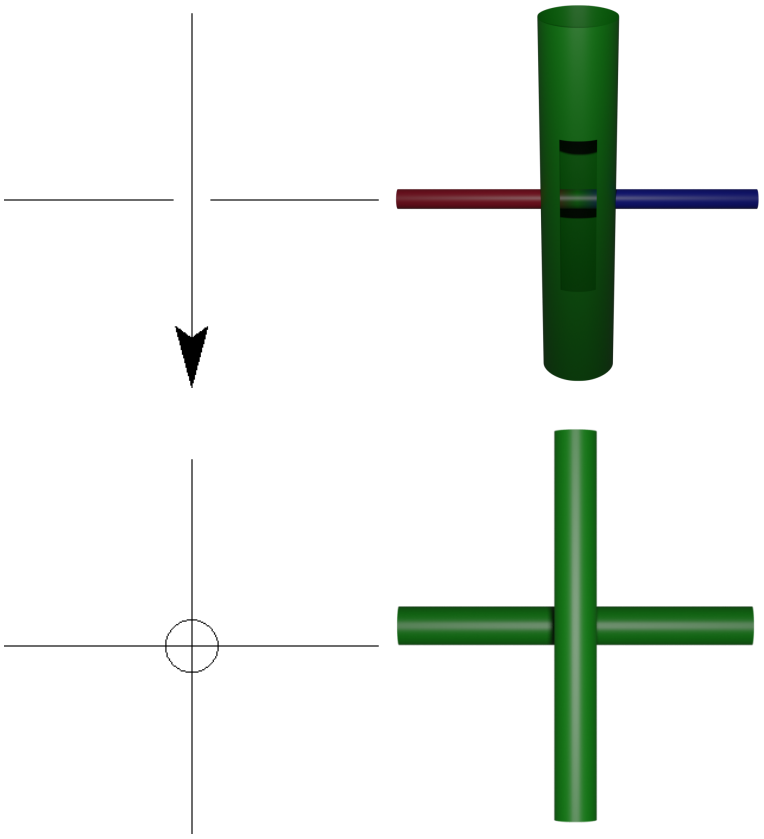


FIGURE 6.1.1: Regions of ribbon torus-links that the Tube map associates to classical and welded crossings respectively (a portion of the tube associated to the overcrossings at classical crossings has been removed for visualisation purposes).

one torus ($S^1 \times S^1$) for each component K of L. The image of this embedding is a ribbon torus-link. We denote the embedding τ by Tube(L). Ignoring the crossings, the construction of Tube(L) is essentially a thickening of the welded link associated to L that is then coloured in order to obtain an embedding in $\mathbb{R}^4$. The regions of Tube(L) associated to welded and classical crossings are illustrated in Figure 6.1.1. We associate to welded crossings two disjoint tubes, the choice of which lies above the other is irrelevant up to ambient isotopy. We associate to classical crossings two intersecting tubes, the larger tube corresponding to the overcrossing is coloured green, while the smaller tube corresponding to the undercrossing enters the larger tube while coloured red, is coloured green while inside the larger tube and leaves while coloured blue. The colour change is with respect to the orientation such that the portion of the smaller tube to the right of the arrow is always red, and the portion to the left of the arrow is coloured blue. In Theorem 3.1 of Satoh's paper, it was shown that all ribbon torus-knots arise as Tube(L) for some welded link diagram L.

```
gap> L:=ArcPresentation(PureCubicalKnot(3,1));;
gap> spun:=SpunLinkComplement(L);;
gap> C1:=ChainComplexOfUniversalCover(spun);;
gap> tube:=Tube([L,[0,0,0],[2,2,2]]);;
gap> C2:=ChainComplexOfUniversalCover(tube);;
gap> bool:=true;;
gap> for c in [1..10] do
>    Ic_spun:=LowIndexSubgroupsFpGroup(C1!.group,c);;
>    Ic_spun:=Filtered(Ic_spun,g->Index(C1!.group,g)=c);;
>    Ic_spun:=Set(
>        Ic_spun,
>        g->Homology(
>            TensorWithIntegersOverSubgroup(C1,g),
>            2
>        )
>    );;
>    Ic_tube:=LowIndexSubgroupsFpGroup(C2!.group,c);;
>    Ic_tube:=Filtered(Ic_tube,g->Index(C2!.group,g)=c);;
>    Ic_tube:=Set(
>        Ic_tube,
>        g->Homology(
>            TensorWithIntegersOverSubgroup(C2,g),
>            2
>        )
>    );;
>    if Ic_spun<>Ic_tube then
>        bool:=false;;
>        break;
>    fi;
> od;
gap> bool;
true
```

GAP Session 6.1.1: Testing the homeomorphism equivalence between Tube(L) and $S(\kappa)$ for the trefoil knot.
Runtime: 1h 17min 4s 828ms.

For a classical knot diagram L associated to a knot κ, Tube(L) $\cong S(\kappa)$ where $S(\kappa)$ is the complex obtained from spinning the knot κ about a hyperplane (as per A.14). This is Theorem 4.3 of Satoh's paper[30]. GAP Session 6.1.1 illustrates a specific case of this theorem using our local homology invariant $\mathfrak{J}_c$ of Section 3.3 for $c \leq 10$.

6.1.1 Arc 2-presentations

Towards making the Tube map algorithmic, we first need to establish a datatype for the input. We desire that this datatype be robust enough to describe any ambient isotopy class of ribbon surface-links. A reasonable starting point is to consider our existing datatype for arc presentations.

Recall that an arc presentation is encoded in `HAP` as a list of positive integer pairs (*cf.* Section 3.1)

$$l = [[a_{1,1}, a_{1,2}], \ldots, [a_{n,1}, a_{n,2}]]$$

specifying the coordinates of the endpoints of a series of horizontal line segments in the plane, *i.e.*, the i^{th} entry $[a_{i,1}, a_{i,2}]$ denotes that the endpoints of the $(n-i+1)^{\text{th}}$ horizontal bar are $(a_{i,1}, i)$ and $(a_{i,2}, i)$. We join any two points that share the same x-coordinate via a vertical line segment and excise a neighbourhood of points about any intersections of horizontal and vertical line segments belonging solely to horizontal line segments. The result is a piecewise linear link diagram corresponding to some link L where all vertical line segments lie above all horizontal line segments. We seek to broaden this datatype so that it can account for welded crossings. We would also like to remove the requirement that all vertical line segments lie above all horizontal line segments. To this end, we introduce an *arc 2-presentation* which begins as an embellishment of our arc presentation with an additional list whose length, denoted by k, is the number of crossings in our associated link L. The i^{th} entry of this list corresponds to the i^{th} crossing of L as the crossings are ordered top-to-bottom, left-to-right. These entries can be either $-1, 0$ or 1 denoting an undercrossing (*i.e.* a horizontal bar beneath a vertical bar), a welded crossing (i.e. a horizontal bar intersecting a vertical bar), or an overcrossing (*i.e.* a horizontal bar above a vertical bar).

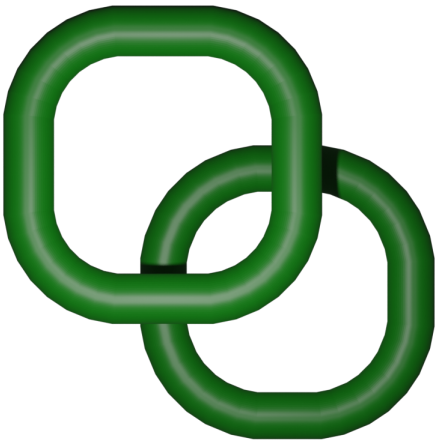
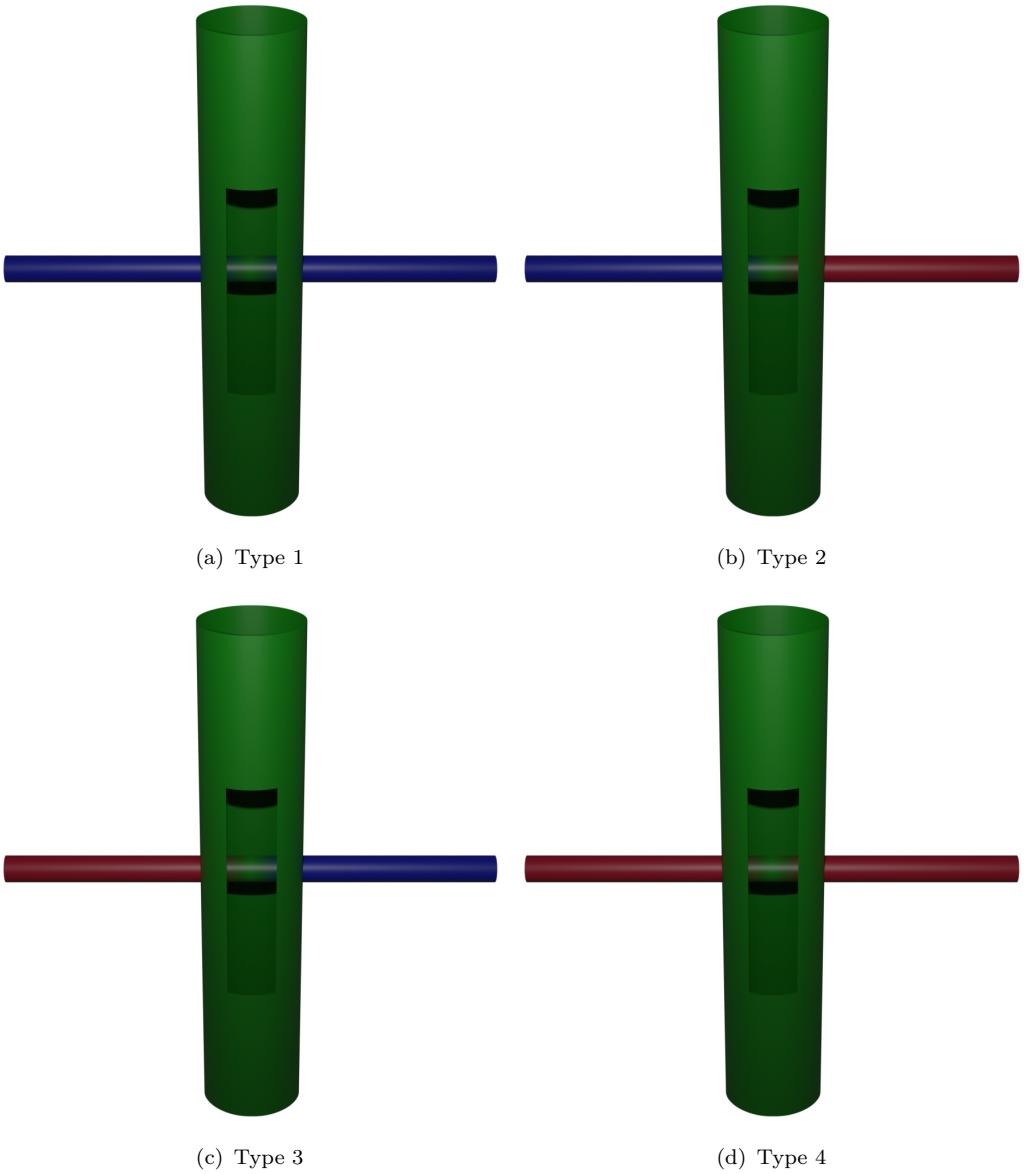


FIGURE 6.1.1.1: Two unknotted circles with arc 2-presentation $[[[2,4],[1,3],[2,4],[1,3]],[-1,1]]$.

At this stage, the arc 2-presentation is essentially just a welded arc presentation and it can be associated to self-intersecting ribbon torus-links as per the Tube map. However, we have not yet addressed any way of describing discrete colourings of ribbon torus-links nor have we addressed any way of describing ribbon-surface links whose connected components are not all tori. Regarding the colourings, we impose that the ribbon surface-links associated to our arc 2-presentations are coloured almost entirely green but for small tubular neighbourhoods of points about the singular points. This differs slightly from Satoh's description of the Tube map in that we associate non-intersecting tubes to classical crossings, and we associate intersecting tubes which undergo a change in colour to welded crossings. In reversing which parts of a ribbon surface-link are associated to which crossings, and in opting to not orient our arc 2-presentation, we lose the method of colouring the intersecting tubes that we previously established. To remedy this, we simply embellish our arc 2-presentation with another list whose length is the number of welded crossings (*i.e.* 0 entries) in the previous list. The entries of this list are integers from 1 to 4. These integers correspond to four distinct shifts in temperature that occur as a smaller horizontal tube intersects a larger vertical tube. These temperature shifts correspond to the horizontal tube (going from left to right): entering as blue, turning green, then exiting as blue; entering as blue, turning green, then exiting as red; entering as red, turning green, then exiting as blue, and; entering as red, turning green, then exiting as red. These temperature shifts are illustrated in Figure 6.1.1.2. Note that the assumption that all temperature shifts occur in the horizontal tubes does not restrict the class of ribbon surface-links we can represent with an arc 2-presentation,



(a) Type 1 (b) Type 2

(c) Type 3 (d) Type 4

FIGURE 6.1.1.2: The four distinct shifts in temperature about the intersection of two hollow cylinders in the region of a ribbon-torus knot the Tube map associates to a welded crossing.

just as in an arc presentation assuming that all crossings arise as horizontal bars below vertical bars does not restrict the class of knots that can be represented.

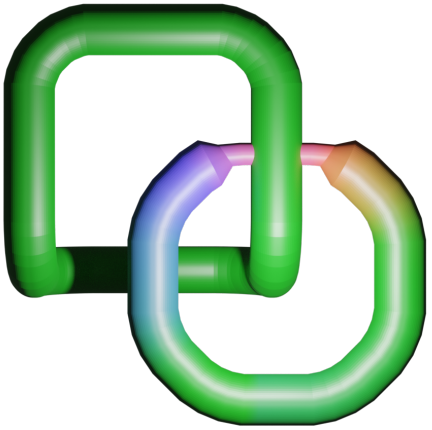


FIGURE 6.1.1.3: A colouring of a ribbon torus-link with arc 2-presentation $[[[2,4],[1,3],[2,4],[1,3]]],[0,-1],[2]]$.

The final port of call in defining the arc 2-presentation is to enable it to describe embeddings of not just knotted tori, but knotted spheres. We do this by omitting a vertical bar that is not involved in any crossings from a given component K of our arc 2-presentation. We encode this on a computer by multiplying all x-coordinates of any such vertical bar in our arc 2-presentation by -1.

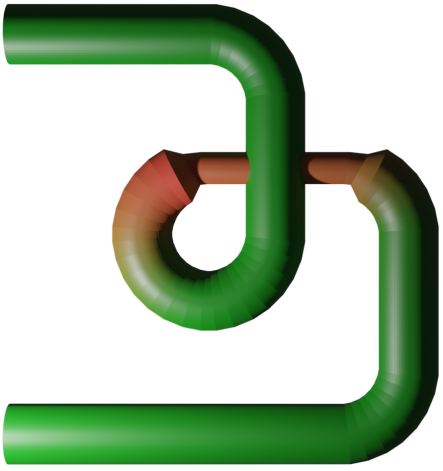
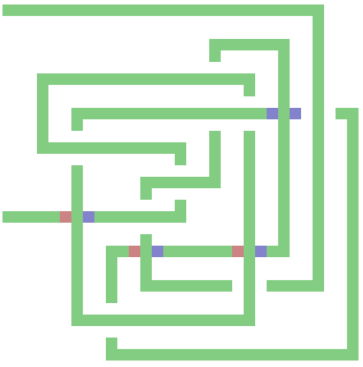


FIGURE 6.1.1.4: An unknotted sphere with arc 2-presentation $[[[-1,4],[2,3],[2,4],[-1,3]],[0],[4]]$.

With these modifications established, we define an arc 2-presentation.

Datatype 6.1.1.1. An *arc 2-presentation* $A = [l_1, l_2, l_3]$ is a list of up to three lists encoding a planar representation of a colouring of a ribbon-surface link.

```
gap> l1:=ArcPresentation(PureCubicalKnot(11,12));
[ [ 4, 11 ], [ 3, 8 ], [ 5, 10 ], [ 4, 9 ], [ 1, 6 ], [ 5, 7 ],
  [ 2, 6 ], [ 3, 11 ], [ 2, 8 ], [ 7, 9 ], [ 1, 10 ] ]
gap> arc[5]:=[-1,6];; arc[11]:=[-1,10];;
gap> l2:=List([1..13],x->Random([0,-1,1]));
[ 1, 1, 1, 0, -1, 1, 1, 0, 1, 0, 0, -1, 1 ]
gap> l3:=List([1..Length(Positions(l2,0))],x->Random([1,2,3,4]));
[ 1, 3, 3, 3 ]
gap> arc2:=[l1,l2,l3];;
gap> ViewArc2Presentation(arc2);
convert-im6.q16: pixels are not authentic '/tmp/HAPtmpImage.txt' @
error/cache.c/QueueAuthenticPixelCacheNexus/4381.
```



GAP Session 6.1.1.1: Computing a PNG file of a planar representation of a random arc 2-presentation.
Runtime: 17ms.

- The mandatory list $l_1 = [[a_{1,1}, a_{1,2}], \ldots, [a_{n,1}, a_{n,2}]]$ is an arc presentation. Any negative integers $a_{i,j} < 0$ that occur in the arc presentation denote the omission of any vertical line segment in our arc 2-presentation with x-coordinate $-a_{i,j}$.

- The optional list $l_2 = [b_1, \ldots, b_k]$ is of length the number of crossings in l_1. The entries b_i of l_2 are either $-1, 0$ or 1 denoting that the i^{th} crossing in our arc presentation is an undercrossing, a welded crossing or an overcrossing respectively.

- The optional list $l_3 = [c_1, \ldots, c_m]$ is of length the number of 0 entries in l_2. The entries are integers between 1 and 4 denoting the four distinct temperature shifts that occur at singular points in our colouring as per Figure 6.1.1.2.

We can obtain a visualisation of an arc 2-presentation via the `ViewArc2Presentation` (A.28) function. GAP Session 6.1.1.1 displays a visualisation of a randomly generated arc 2-presentation arising from an arc presentation of 11_{12}.

6.2 Small CW-structures for self-intersecting surfaces

The association between arc 2-presentations and ribbon surface-links is made concrete in these next few sections via our three-step implementation of Satoh's Tube map. Our first step is to obtain a regular CW-decomposition of a discrete colouring of a self-intersecting surface in the 3-ball. We then lift this surface according to its colouring, resulting in a 2-dimensional knotted surface embedded in $\mathbb{R}^4$. Lastly, we construct the tubular neighbourhood of this surface and return an inclusion map from its boundary into the 4-ball.

In this section, we describe an algorithm that inputs an arc 2-presentation and returns an inclusion of regular CW-complexes $f : \partial M \hookrightarrow B^3$ where ∂M denotes a discrete colouring whose self-intersecting surface arises as the boundary of some self-intersecting ribbon surface-link M and where B^3 denotes the 3-ball. Recall that in Section 4.1, we described a procedure for endowing the complement of some thickened 1-dimensional knotted manifold with a regular CW-structure. The process by which we endow self-intersecting surface complements with a regular CW-structure is very similar to this, albeit more complex as we need to account for both the colouring as well as up to three different types of crossings. This process comprises three steps which we label α, β, and γ.

Step (α). Let D^2_a denote a regular CW-complex homeomorphic to the solid 2-disk and let M_a denote a 1-dimensional subcomplex of D^2_a homeomorphic to $2h$ disjoint copies of S^1, where h is the number of horizontal bars in the arc 2-presentation a (see Figure 6.2.1). This CW-complex is constructed as in Section 4.1 wherein we perform a clockwise walk along the 1-skeleton of the complex in order to trace the boundary of its 2-cells.

Step (β). We endow the unit interval I with CW-structure consisting of two 0-cells and one 1-cell and form the direct products $M_a \times I \subset D^2_a \times I$. The 2-dimensional subcomplex $M_a \times I$ consisting of $2h$ hollow cylinders is contained in the 3-dimensional contractible space $D^2_a \times I$. We now glue a number of 2-cells to the top and bottom of $D^2_a \times I$ so that the CW-subcomplex $M_a \times I$ is homeomorphic to ∂M. As arc 2-presentations allow for the specification of the crossing type at every crossing, we have complicated the CW-structure so that the simple procedure of gluing one 2-cell at the bottom of $D^2_a \times I$ for each horizontal bar, and gluing one 2-cell at the top of

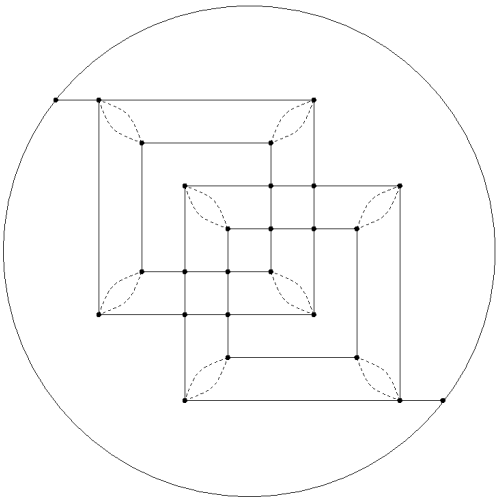


FIGURE 6.2.1: The CW-complexes D^2_a and M_a (dotted lines) as of Step (α) for arc 2-presentation $a = [[[2,4],[1,3],[2,4],[1,3]],[0,-1],[2]]$.

$D^2_a \times I$ for each vertical bar (as is done in Section 4.1) is no longer possible. The gluing of 2-cells to either end of $D^2_a \times I$ is instead determined as follows:

(i) For all bars—be they horizontal or vertical—that are not involved in any crossings, we glue one 2-cell e^2 to the top of $D^2_a \times I$.

(ii) If the 2-cells that are glued to $D^2_a \times I$ intersect in any way, we introduce an additional 1-cell e^1 at their intersection in order to preserve the regularity of the CW-structure (refer to Figure 6.2.2).

(iii) At each undercrossing, we glue one 2-cell along the horizontal bar on the bottom of $D^2_a \times I$ and one 2-cell along the vertical bar on the top of $D^2_a \times I$.

(iv) At each overcrossing, we glue one 2-cell along the horizontal bar on the top of $D^2_a \times I$ and one 2-cell along the vertical bar on the bottom of $D^2_a \times I$.

(v) At each welded crossing, we glue a regular CW-complex W consisting of a total of 8 0-cells, 22 1-cells and 19 2-cells to the top of $D^2_a \times I$ as outlined in Figure 6.2.3.

(vi) At the end of this gluing process, for each of the remaining holes on either side of $D^2_a \times I$, we glue a 2-cell along their boundaries.

All cells that are glued to $D^2_a \times I$ are present in the subcomplex $M_a \times I$ except for those 2-cells with two 1-cells in their boundaries that were added in step (vi) of the gluing process. Were these 2-cells to be included, it would result in an extruding cylindrical

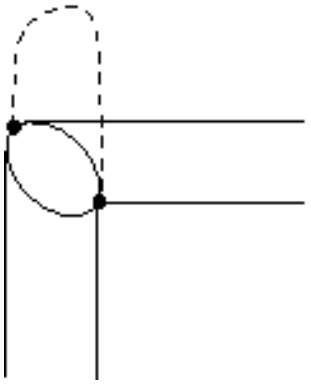


FIGURE 6.2.2: Adding a 1-cell where two 2-cells are to intersect.

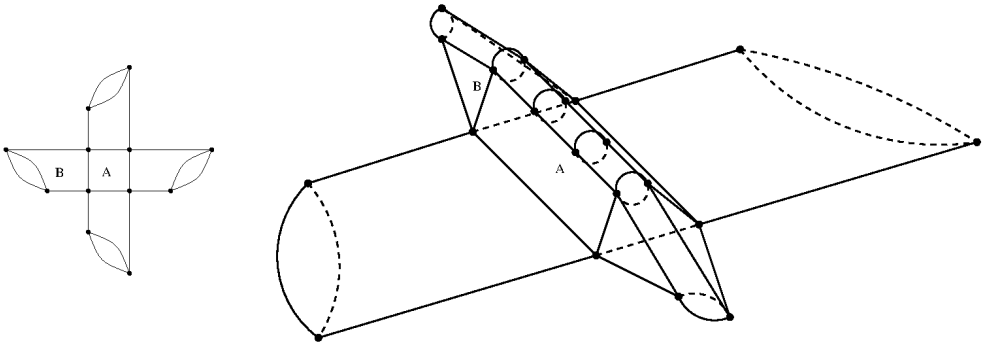


FIGURE 6.2.3: The CW-structure given to the welded crossings of our self-intersecting surface.

portion of our subcomplex that contributes nothing to the homeomorphism type of ∂M. Thus, we remove from $M_a \times I$ any 2-cells that have non-empty intersection with the 2-cells added in step (vi) and in their place, we add a single 2-cell to $M_a \times I$ (see Figure 6.2.4). After the gluing process, the subcomplex is homeomorphic to ∂M while $B^* = D^2_a \times I \cup_i e^2 \cup_j W$ needs an additional two 2-cells and several 3-cells in order to be homeomorphic to B^3.

Step (γ). The gluing procedure in the previous step introduces a number of homology cycles that prevent B^* from being contractible. We introduce a number of 3-cells to the complex whose boundaries are those 2-cells which were glued in the previous step as well as the 2-cells which share boundaries with the glued cells. Additionally, we glue a 3-cell to either end of B^* such that when we eventually take the tubular neighbourhood of a lifted ∂M, it is entirely contained in the 4-ball. At this stage, we have the desired inclusion of regular CW-complexes $\partial M \hookrightarrow B^3$, however we have yet to address the colouring. As explained earlier, most all of the cells of our regular CW-complex will be assigned a temperature associated to the colour green. Only a few cells of dimension two or fewer will be coloured otherwise. These are exactly

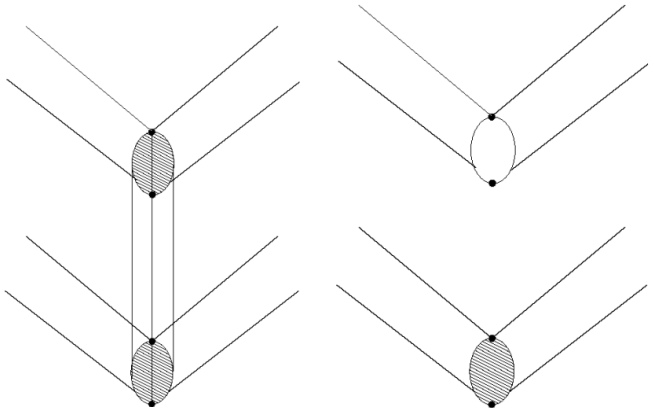


FIGURE 6.2.4: Simplifying the CW-structure of ∂M by omitting two 1-cells and three 2-cells.

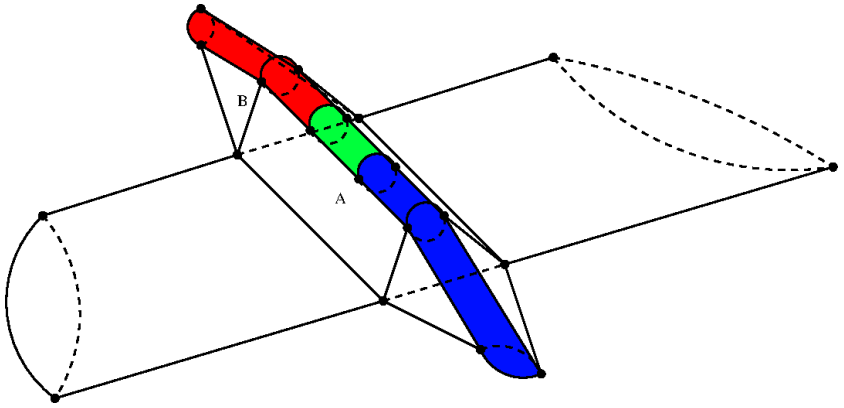


FIGURE 6.2.5: Colouring the 2-cells of ∂M that are associated to the welded crossings of an arc 2-presentation.

eight 2-cells (and the cells in their closures) found in the regions of ∂M we associate to welded crossings; they are shown in Figure 6.2.5. We encode the colour of a given cell e^n_k as a component function `X!.colour(n,k)` which returns a list of integers corresponding to all colours assigned to the k^{th} n-cell of X. It suffices to specify one single colour for each of the top-dimensional cells (*i.e.* 2-cells) of ∂M. In this way, any 1-cells or 0-cells will inherit a colour from the temperature of the 2-cells in their coboundaries.

These steps are encoded in `HAP` as the function `ArcDiagramToTubularSurface(a)` (A.11). `GAP` Session 6.2.1 produces an inclusion of regular CW-complexes arising from the arc 2-presentation `[[[2,4],[1,3],[2,4],[1,3]],[0,-1],[3]]` and returns a list of the temperatures assigned to each 2-cell in the subcomplex ∂M corresponding to the self-intersecting surface M.

```
gap> a:=[[[2,4],[1,3],[2,4],[1,3]],[0,-1],[2]];;
gap> M:=ArcDiagramToTubularSurface(a);
Map of regular CW-complexes

gap> c:=List(
>     [1..Length(Target(M)!.boundaries[3])],
>     x->M!.colour(2,x)
> );
[ [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ],
  [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ],
  [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ],
  [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ],
  [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ],
  [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ],
  [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ],
  [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ],
  [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ],
  [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ],
  [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ],
  [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ],
  [ 0 ], [ 0 ], [ 0 ], [ -1 ], [ -1 ], [ 0 ], [ 0 ], [ 1 ], [ 1 ],
  [ -1 ], [ 1 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ],
  [ 0 ], [ 0 ], [ 0 ], [ -1 ], [ 1 ], [ 0 ], [ 0 ], [ 0 ], [ 0 ] ]
```

GAP Session 6.2.1: The temperatures of each 2-cell of a discretely coloured self-intersecting surface of the welded Hopf link.
Runtime: 25ms.

6.2.1 Kinking arc 2-presentations

Note that in Step β of Section 6.2, there is some choice involved in the gluing procedure due to the fact that an arc 2-presentation a does not give us enough information to determine the exact positions of all horizontal and vertical 2-cells. This indeterminacy presents an issue should there be any horizontal or vertical bars in a that are involved in multiple crossings that are not all of the same type. For example, consider the arc 2-presentation:

$$a = [[[2,3],[4,5],[1,5],[1,2],[3,4]],[1,-1,0],[4]]$$

(Figure 6.2.1.1). All three crossings occurring along the same horizontal bar provide conflicting information about the 2-cells which are to be glued to them as per the procedure we described above in Step (β). For instance, the overcrossing instructs that a 2-cell is to be glued to the top of $D^2_a \times I$ while the ensuing undercrossing

FIGURE 6.2.1.1: An arc 2-presentation involving three crossings along the same horizontal component.

instructs that the same 2-cell be glued to the bottom of $D^2_a \times I$. This issue can be addressed by increasing the size of the arc 2-presentation. A procedure for ensuring that an arc 2-presentation does not have multiple crossings occurring along the same horizontal/vertical bar is encoded in `HAP` as the algorithm `KinkArc2Presentation` (A.12) which we now detail.

Consider the arc 2-presentation:

$$[[[3,4],[-1,-7],[5,6],[2,6],[2,3],[-1,-7],[4,5]],[-1,0,1,-1,0,0,1],[2,3,1]]$$

consisting of a total of seven crossings (Figure 6.2.1.2). There are three horizontal bars and three vertical bars in this arc 2-presentation that are involved in two or more crossings of differing types. We begin by constructing an $n \times n$ zero matrix, denoted by $A = (a_{ij})$, where n is the largest absolute value of any integer in any sublist of the arc presentation. Let $[x_1, x_2]$ denote the i^{th} integer pair of the arc presentation. We set $a_{n-i+1,|x_1|} = 3$ and $a_{n-i+1,|x_2|} = 3$. For each $|x_1| < j < |x_2|$, we set $a_{n-i+1,j} = 1$. At this stage, each column of A will have exactly two entries equal to 3. For $1 \leq j \leq n$, let $|x| < i < |y|$ where $a_{|x|,j} = a_{|y|,j} = 3$, and increase the value of $a_{i,j}$ by 1 unless x or y are negative. This procedure produces a matrix that models the first list of our arc 2-presentation: each 3 entry denotes an endpoint of a horizontal and/or vertical bar, each 2 entry corresponds to a crossing in our arc 2-presentation, and each 1 entry denotes an unknotted segment of the arc 2-presentation (see Equation 6.2.1.1 (left)).

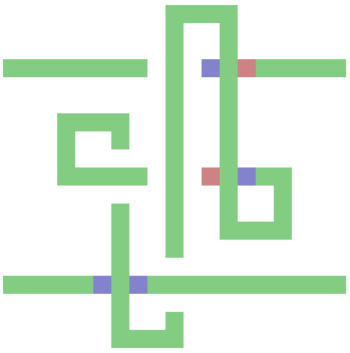


FIGURE 6.2.1.2: An arc 2-presentation with multiple crossings that lie on the same horizontal and vertical bars.

$$A = \left(\begin{array}{cccc|ccc} 0 & 0 & 0 & 3 & 3 & 0 & 0 \\ 3 & 1 & 1 & 2 & 2 & 1 & 3 \\ \hline 0 & 3 & 3 & 1 & 1 & 0 & 0 \\ 0 & 3 & 2 & 2 & 2 & 3 & 0 \\ 0 & 0 & 1 & 1 & 3 & 3 & 0 \\ 3 & 1 & 2 & 2 & 1 & 1 & 3 \\ 0 & 0 & 3 & 3 & 0 & 0 & 0 \end{array}\right) \longrightarrow \left(\begin{array}{cccc|c|ccc} 0 & 0 & 0 & 3 & 1 & 3 & 0 & 0 \\ 3 & 1 & 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 3 & 2 & 1 & 3 \\ \hline 0 & 3 & 3 & 1 & 0 & 1 & 0 & 0 \\ 0 & 3 & 2 & 2 & 1 & 2 & 3 & 0 \\ 0 & 0 & 1 & 1 & 0 & 3 & 3 & 0 \\ 3 & 1 & 2 & 2 & 1 & 1 & 1 & 3 \\ 0 & 0 & 3 & 3 & 0 & 0 & 0 & 0 \end{array}\right) \tag{6.2.1.1}$$

We are now equipped to deal with the task of lengthening our arc 2-presentation so that no two crossings occur on the same horizontal or vertical bar. Let $a_{k,l} = 2$ denote the first crossing of a row of A in which there are multiple crossings, and let $a_{k,m} = 2$ denote the immediately preceding crossing on the same row. Consider the four submatrices $A_1 = (a_{ij})_{i \leq k, j \leq l}$, $A_2 = (a_{ij})_{i<k, j \geq m}$, $A_3 = (a_{ij})_{i>k, j \leq l}$, and $A_4 = (a_{ij})_{i \geq k, j \geq m}$. We construct a new matrix with one more row and column than A by arranging these submatrices as in Equation 6.2.1.1. We set $a_{i,m-1} = 3$ and $a_{i+1,m-1} = 3$. The remaining entries are determined as in the initial construction of A, by setting any entries lying between two 3 entries to 1 unless the arc 2-presentation specifies that said column consist of zeroes and setting them to 0 otherwise.

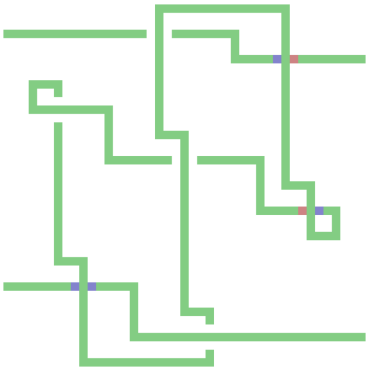


FIGURE 6.2.1.3: The arc 2-presentation of Figure 6.2.1.2 after having applied to it Algorithm A.12. There are no two crossings that occur on the same horizontal or vertical bar.

$$\mathrm{Kink}(A) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 3 & 1 & 1 & 3 & 0 & 0 \\ 3 & 1 & 1 & 1 & 1 & 2 & 3 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 3 & 1 & 2 & 1 & 3 \\ 0 & 3 & 3 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 3 & 2 & 3 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 3 & 1 & 2 & 1 & 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 3 & 2 & 3 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 3 & 3 & 0 \\ 3 & 1 & 2 & 1 & 3 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 3 & 2 & 1 & 1 & 1 & 1 & 3 \\ 0 & 0 & 3 & 1 & 1 & 3 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{6.2.1.2}$$

By repeating this process for all rows of A we obtain the matrix $\mathrm{Kink}(A)$ (see Equation 6.2.1.2). This matrix has the property that each row has at most one entry equal to 2. However, there could still be columns containing multiple entries equal to 2. To resolve this, we construct $B = \mathrm{Kink}(\mathrm{Kink}(A)^T)$ using the same techniques we used to construct $\mathrm{Kink}(A)$. The final step is to reconstruct our arc 2-presentation from B^T. We did not introduce any new crossings in constructing B^T, nor did we change the ordering of our crossings, so the latter two lists of our arc 2-presentation remain the same. We obtain the first list of our arc 2-presentation by noting the columns that each 3 entry lies in. Starting from the final row, we can express this as a list of integer pairs concluding the process of ensuring that no two crossings occur in the same row or column.

$$B^T = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 3 & 1 & 1 & 1 & 1 & 3 & 0 & 0 & 0 \\ 3 & 1 & 1 & 1 & 1 & 1 & 2 & 1 & 1 & 3 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 3 & 1 & 2 & 1 & 1 & 3 \\ 0 & 3 & 3 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 3 & 2 & 1 & 3 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 3 & 3 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 3 & 1 & 1 & 2 & 1 & 1 & 3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 3 & 3 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 3 & 1 & 2 & 3 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 3 & 3 & 0 \\ 0 & 0 & 3 & 3 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 1 & 1 & 2 & 1 & 3 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 3 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 3 & 1 & 1 & 2 & 1 & 1 & 1 & 1 & 1 & 3 \\ 0 & 0 & 0 & 3 & 1 & 1 & 1 & 1 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{6.2.1.3}$$

The resulting arc 2-presentation will consist of $n + m$ integer pairs, where n is the original length of the arc presentation and m is the sum of the number of crossings that occur per row or column -1. For the arc 2-presentation illustrated in Figure 6.2.1.2, Algorithm A.12 will return:

$$[[[4, 9], [6, -15], [8, 9], [-1, 6], [3, 4], [13, 14], [11, 14], [12, 13], [5, 11], [7, 8],$$
$$[2, 5], [2, 3], [10, -15], [-1, 10], [7, 12]], [-1, 0, 1, -1, 0, 0, 1], [2, 3, 1]].$$

6.3 Lifting coloured regular CW-complexes

Suppose that the inclusion of regular CW-complexes $f : \partial M \hookrightarrow B^3$ that we constructed in Section 6.2 arises as a discrete colouring with three temperatures. We endow the interval $I = [0, 4]$ with a regular CW-structure consisting of five 0-cells and four 1-cells. We can construct the 4-ball by forming the direct product $B^3 \times I$. What remains is to use the colouring information of f to identify the subcomplex of $B^3 \times I$ homeomorphic to $\widetilde{\partial M}$, the boundary of some ribbon surface-link.

We associate colours to each of the four vertices of I; we let 0 denote black, 1 denote blue, 2 denote green, 3 denote red, and 4 denote white. This permits the

interpretation of the product:

$$B^3 \times I = B^3 \times \{0\} \cup B^3 \times (0,1) \cup B^3 \times \{1\} \cup B^3 \times (1,2) \cup B^3 \times \{2\} \cup$$
$$B^3 \times (2,3) \cup B^3 \times \{3\} \cup B^3 \times (3,4) \cup B^3 \times \{4\}$$

as five differently coloured copies of B^3 joined together by cells that are coloured on a continuous spectrum. The reasoning for the additional two extremal colours is to ensure that when we lift ∂M, it lies in the interior of $B^3 \times I$.

Let e^n_k be an n-cell of the subcomplex ∂M. We use the component function from the previous section `colour(n,k)` to obtain an ordered list of integers corresponding to each colour associated to e^n_k. This list tells us that $e^n_k \times \{i\}$, $e^n_k \times \{j\}$ and $e^n_k \times (i,j)$ all occur in the complex $\widetilde{\partial M}$ for two consecutive integers i and j in `colour(n,k)`. We repeat this process for every cell of ∂M. The one exception to this are those 1-cells of ∂M occurring as singular points. They are identified by having exactly four 2-cells in their coboundary. The direct product of these 1-cells and any interval does not occur in $\widetilde{\partial M}$.

This procedure is encoded in `HAP` as the function `LiftColouredSurface(f)`. It is compatible with any discrete colouring of regular CW-complexes provided that the complexes are manifolds or at most self-intersecting surfaces. Additionally, the temperatures of the colouring must be represented by consecutive integers, or else the resulting inclusion of regular CW-complexes will contain drastically more cells than necessary.

With a method for lifting coloured surfaces, we can at last present a computational implementation of Satoh's Tube map. The function `Tube(a)` inputs an arc 2-presentation of some ribbon surface-link and successively applies the algorithms `ArcDiagramToTubularSurface(a)`, `LiftColouredSurface(f)`, and `RegularCWComplexComplement(f)` in order to obtain a regular CW-complex $B^4 \backslash \partial K$ homeomorphic to the complement of a thickened ribbon surface-link embedded in $\mathbb{R}^4$. The code of `GAP` Session 6.3.1 constructs this complex for an arc 2-presentation of the welded Hopf link (see, *e.g.*, Figure 6.1.1.3), compares the size of the output with the manually constructed pure cubical complex `HopfSatohSurface()`, and computes its homology.

```
gap> a:=[[[2,4],[1,3],[2,4],[1,3]],[0,-1],[2]];;
gap> t:=Tube(a);
Regular CW-complex of dimension 4

gap> t_:=HopfSatohSurface();;
gap> t_:=PureComplexComplement(t_);;
gap> t_:=RegularCWComplex(t_);
Regular CW-complex of dimension 4

gap> Size(t);
9685
gap> Size(t_);
4508573
gap> hom:=[0..4];;
gap> thom:=List(hom,x->Homology(t,x));
[ [ 0 ], [ 0, 0 ], [ 0, 0, 0, 0 ], [ 0, 0 ], [ ] ]
gap> time;
31
gap> t_hom:=List(hom,x->Homology(t_,x));
[ [ 0 ], [ 0, 0 ], [ 0, 0, 0, 0 ], [ 0, 0 ], [ ] ]
gap> time;
351607
```

GAP Session 6.3.1: Endowing the Tube of the welded Hopf link with a small regular CW-structure.
Runtime: 15min 18s 48ms.

Chapter 7

Dehn surgery on links in the 3-sphere

Let κ denote a link embedded in a 3-manifold M. *Dehn surgery* is a construction that modifies M, it can be summarised in two steps:

(i) *Drilling* along κ consists of removing an open tubular neighbourhood of κ from M. After this step, the boundary of $M \setminus N_\epsilon(\kappa)$ consists of n tori $T_1 \cup \ldots \cup T_n$, one for each connected component of κ.

(ii) *Dehn filling* involves gluing one solid torus to $M \setminus N_\epsilon(\kappa)$ for each connected component in its boundary. This gluing is done with respect to homeomorphisms $h_i : \partial(D^2 \times S^1)_i \hookrightarrow T_i$.

The *Lickorish-Wallace* theorem states that all closed, orientable, connected 3-manifolds X arise from performing Dehn surgery on links in the 3-sphere S^3:

$$X = (S^3 \setminus \kappa) \quad \bigcup_{h_i} \quad \left\{ (D^2 \times S^1) \right\}_{i=1}^{n}.$$

This chapter will describe our computer implementation of Dehn surgery. Using our algorithms to endow link complements with regular CW-structure (see Section 4.1), Dehn surgery will allow us to obtain a regular CW-decomposition of all closed, orientable, connected 3-manifolds.

7.1 Regular CW-surgery

Throughout this section we will refer to Dehn surgery only in the case of knots. This is done for ease of explanation, all of the methods we show readily extend to links. Note that at the time of writing, `HAP` can only handle knots as input.

We seek to implement Dehn surgery in such a way as to produce regular CW-complexes. Recall that the algorithms of section 4.3.2 provide us with regular CW-decompositions of knot complements in the 3-sphere, which we denote by K, so all that needs to be done to complete our implementation is to glue a solid torus to the boundary of K. In the context of non-regular CW-complexes, Dehn filling is simply the addition of a single 2-cell e^2 and a single 3-cell e^3 to K, *i.e.*, the resulting 3-manifold can be expressed as

$$X = K \cup e^2 \cup e^3.$$

However, the restriction of regularity significantly complicates things. The condition that no cell can be present more than once in the boundary of e^2 means that some subdivision is required, and the requirement that e^3 must be a homotopically a solid 3-ball means that we must thicken e^2. Before detailing this gluing, it is necessary to consider the input to a Dehn surgery algorithm.

The homeomorphism $(S^1 \times S^1) \hookrightarrow K$ induces the following group homomorphisms

$$\begin{array}{ccccc}
\pi_1(S^1 \times S^1) & \longrightarrow & \pi_1(K) & \twoheadrightarrow & \pi_1(K)_{\text{ab}} \\
\| & & & & \| \\
H_1(S^1 \times S^1, \mathbb{Z}) & & \longrightarrow & & H_1(K, \mathbb{Z}) \\
\| & & & & \| \\
A & & \xrightarrow{\phi} & & G_{\text{ab}} \\
\| & & & & \| \\
\mathbb{Z} \oplus \mathbb{Z} & & \twoheadrightarrow & & \mathbb{Z}.
\end{array}$$

Let $z \in \ker(\phi)$ denote a generator for the kernel of ϕ. We refer to $l = \pi_1(z) \in \pi_1(K)$ as a *longitudinal element*. Let $y \in A$ be any element such that $\{y, z\}$ generates A. For such an element, we refer to $m = \pi_1(y) \in \pi_1(K)$ as a *meridional element*. Let p and q denote two coprime integers. We can attach e^2 to K such that its boundary corresponds to $l^p m^q$. The quotient of these integers, p/q, is referred to as the *surgery*

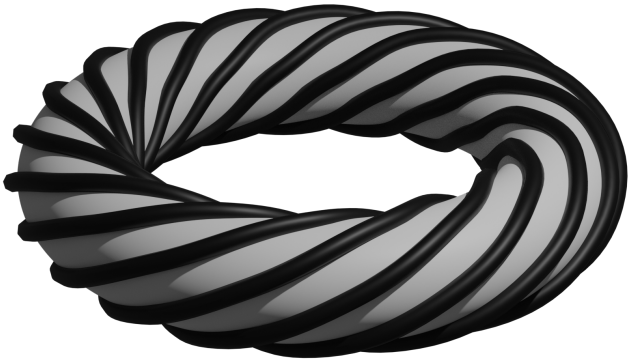


FIGURE 7.1.1: A curve that wraps around the torus 7 times longitudinally and 17 times meridionally.

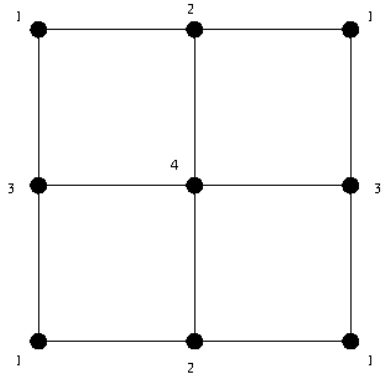


FIGURE 7.1.2: The minimal regular CW-structure of a torus.

coefficient, and it is the only piece of information we need in order to specify Dehn surgery.

We seek to alter the knot complement K such that the regular CW-structure of its boundary contains a path of 1-cells that wrap around the torus p times longitudinally and q times meridionally. It is to this path that we glue e^2. Our first step is to ensure that the boundary of K contains as few cells as possible; we desire for it to have the minimal regular CW-structure for a torus, *i.e.*, four 0-cells, eight 1-cells and four 2-cells (see Figure 7.1.2). In many cases, this can be achieved by repeatedly simplifying (via Algorithm 5.3.1) and barycentrically subdividing K (see `GAP` Session 7.1.1). Allowing for up to twenty repeats of this simplification/subdivision procedure produces minimal regular CW-structure on all but only ten prime knots: 9_{17}, 9_{43}, 10_{29}, 10_{47}, 11_{16}, 11_{40}, 11_{326}, 11_{365}, 11_{410}, and 11_{442}.

At this point we may begin to add more cells to K. We do so by adding cells to a square regular CW-complex $K_\square$ which upon identifying opposite sides yields a

```
gap> prime:=[0,0,1,1,2,3,7,21,49,165,552];;
gap> min:=[];;
gap> for i in [3..Length(prime)] do
>        for j in [1..prime[i]] do
>            K:=ArcPresentation(PureCubicalKnot(i,j));
>            K:=SphericalKnotComplement(K);
>            s:=Size(BoundaryOfPureRegularCWComplex(K));
>            count:=0;
>            repeat
>                K:=RegularCWComplex(BarycentricSubdivision(K));;
>                K:=SimplifiedComplex(K);;
>                s:=Size(BoundaryOfPureRegularCWComplex(K));;
>                count:=count+1;
>            until
>                s=16 or count=20;
>            if s=16 then
>                Add(min,[i,j]);
>            fi;
>        od;
> od;
gap> Print(100*Float(Length(min)/Sum(prime)),"%");
98.7516%
```

GAP Session 7.1.1: Simplifying and barycentrically subdividing all prime knots with fewer than twelve crossings until, within twenty attempts, their toroidal boundaries have minimal regular CW-structure.
Runtime: 47min 35s 363ms.

regular CW-decomposition of ∂K. For $p, q \neq 0$, we arrange $|p| + |q|$ 2-dimensional strips onto $K_{\square}$. Each strip (with the exception of two) consists of six 0-cells, seven 1-cells and two 2-cells. There will be $p-1$ strips with half of their cells located in the upper portion of $K_{\square}$, and $q-1$ strips with half of their cells located on the right-hand side of $K_{\square}$, these strips correspond to one complete longitudinal/meridional traversal of ∂K respectively. The remaining two strips are located in the upper left-hand side and along the off-diagonal of $K_{\square}$, the first consists of three 0-cells, three 1-cells and a 2-cell while the other consists of eight 0-cells, nine 1-cells and three 2-cells. These strips join together to form thickened path along K (please see Figure 7.1.3 for an example with $p/q = 7/4$). The resulting decomposition of ∂K consists of $4(p+q)+2$ vertices, $8(p+q)+3$ edges, and $4(p+q)+1$ faces. In order to complete the gluing of e^2 to K, we must add a number of cells of dimension ≤ 3 to K such that our strips bound a thickened 3-dimensional 'bulkhead', denoted by b, homotopy equivalent to e^2.

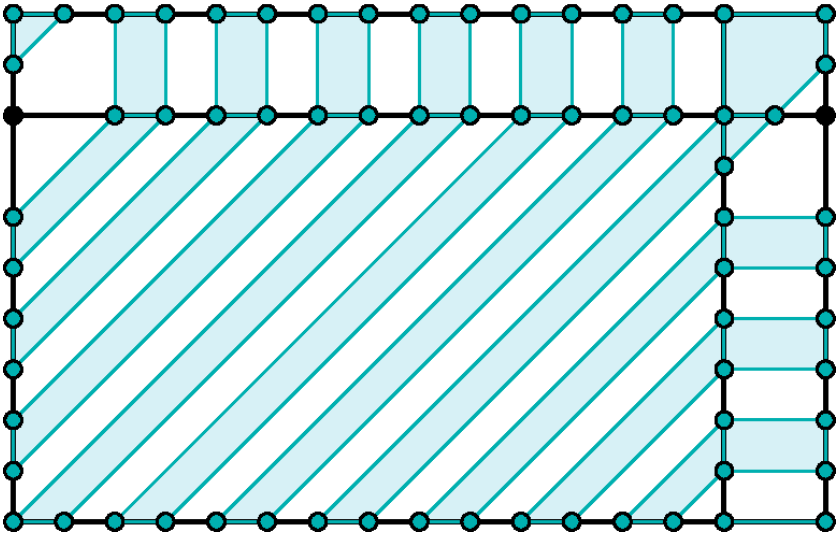


FIGURE 7.1.3: The regular CW-structure of $K_\square$ for $p = 7$ and $q = 4$.

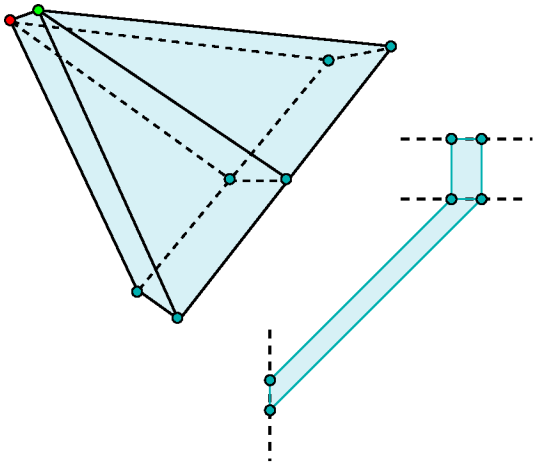


FIGURE 7.1.4: The regular CW-structure of each triangular prism segment of the thickened b (note that ∞_L is depicted as the red vertex and ∞_R as the green).

Let ∞_L and ∞_R denote the 0-cells in the centre of b and let d denote a diagonal strip that was added to $K_\square$. For each 0-cell in d, we add a 1-cell to K whose boundary contains either ∞_L or ∞_R. Similarly, each 1-cell of d contributes a 2-cell to K (which is either on the left or right-hand side of b) and each 2-cell of d contributes a 3-cell. The result is a triangular prism (see Figure 7.1.4) which when repeated for each d, yields a regular CW-decomposition of b.

The extra effort in adding the thickened 2-dimensional bulkhead b to K in place of a 2-dimensional disk greatly pays off in simplifying the gluing of e^3 to K. The boundary of this 3-cell is simply all 2-cells on the left and right sides of b together with all remaining 2-cells of ∂K (not contained in any d). Note that if we did not

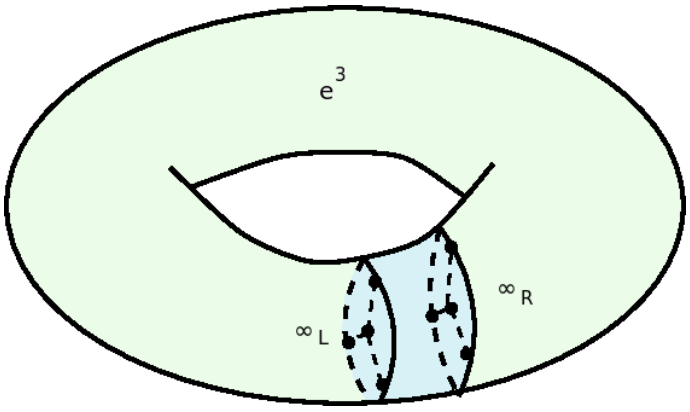


FIGURE 7.1.5: The necessity of thickening e^2.

```
gap> arc:=[[1,2],[1,2]];;
gap> L:=ThreeManifoldViaDehnSurgery(arc,22,17);
Regular CW-complex of dimension 3

gap> List([0..4],x->Homology(L,x));
[ [ 0 ], [ 22 ], [ ], [ 0 ] ]
gap> CriticalCells(L);
[ [ 3, 1 ], [ 2, 2 ], [ 1, 35 ], [ 0, 76 ] ]
gap> IsPureRegularCWComplex(L);
true
```

GAP SESSION 7.1.2: Constructing the Lens space $L(22, 17)$ via Dehn surgery.
Runtime: 1s 80ms.

thicken e^2, then the addition of e^3 in this way would not yield a regular CW-complex (see Figure 7.1.5), rather, e^3 would contain e^2 in its boundary twice.

These procedures are encoded in the algorithm `ThreeManifoldViaDehnSurgery(arc,p,q)` which inputs an arc presentation and two coprime integers and outputs a regular CW-decompostion of a closed, orientable, connected 3-manifold obtained by performing Dehn surgery with coefficient p/q on the complement of the link given by `arc` in S^3. The code of GAP Session 7.1.2 computes such a manifold for the unknot with surgery coefficient $22/17$ producing the Lens space $L(22, 17)$. In general, we define a *Lens space* $L(p, q)$ as the 3-manifold obtained by performing p/q-surgery on the unknot in S^3.

Bibliography

[1] H. R. Brahana. Systems of circuits on two-dimensional manifolds. *Ann. of Math. (2)*, 23(2):144–168, 1921. ISSN 0003-486X. doi: 10.2307/1968030. URL https://doi-org.nuigalway.idm.oclc.org/10.2307/1968030.

[2] Gabor Wiese. Computational arithmetic of modular forms (course notes), 2018. URL https://arxiv.org/abs/1809.04645.

[3] F. Waldhausen. On irreducible 3-manifolds which are sufficiently large. *Ann. of Math. (2)*, 87:56–88, 1968. ISSN 0003-486X.

[4] Gunnar Carlsson, Tigran Ishkhanov, Vin de Silva, and Afra Zomorodian. On the local behavior of spaces of natural images. *International Journal of Computer Vision*, 76(1):1–12, 2008. ISSN 0920-5691. doi: 10.1007/s11263-007-0056-x. URL http://dx.doi.org/10.1007/s11263-007-0056-x.

[5] K. Alexander, A. Taylor, and M Dennis. Proteins analysed as virtual knots. *Nature, Scientific Reports*, 7, 42300, 2017.

[6] GAP. *GAP – Groups, Algorithms, and Programming, Version 4.10.2*. The GAP Group, 2019. URL https://www.gap-system.org.

[7] David De Wit. Automatic evaluation of the Links-Gould invariant for all prime knots of up to 10 crossings. *J. Knot Theory Ramifications*, 9(3):311–339, 2000. ISSN 0218-2165. doi: 10.1142/S0218216500000153. URL https://doi-org.nuigalway.idm.oclc.org/10.1142/S0218216500000153.

[8] Graham Ellis. *An invitation to computational homotopy*. Oxford University Press, UK, 2019. ISBN 9780198832980. URL https://global.oup.com/academic/product/an-invitation-to-computational-homotopy-9780198832980.

[9] J. R. Links and M. D. Gould. Two variable link polynomials from quantum supergroups. *Lett. Math. Phys.*, 26(3):187–198, 1992. ISSN 0377-9017. doi: 10.1007/BF00420752. URL https://doi-org.nuigalway.idm.oclc.org/10.1007/BF00420752.

[10] G. Ellis. *HAP – Homological Algebra Programming, Version 1.47*, August 2022. (http://www.gap-system.org/Packages/hap.html).

[11] Joshua Evan Greene. The lens space realization problem. *Ann. of Math. (2)*, 177(2):449–511, 2013. ISSN 0003-486X. doi: 10.4007/annals.2013.177.2.3. URL https://doi-org.nuigalway.idm.oclc.org/10.4007/annals.2013.177.2.3.

[12] S. Rees and L.H. Soicher. An algorithmic approach to fundamental groups and covers of combinatorial cell complexes. *J. Symbolic Comput.*, 29(1):59–77, 2000. ISSN 0747-7171. doi: 10.1006/jsco.1999.0292. URL http://dx.doi.org/10.1006/jsco.1999.0292.

[13] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.7.0)*, 2019. URL https://doc.sagemath.org/html/en/reference/categories/sage/categories/simplicial_sets.html.

[14] N Culler, N Dunfield, and M Goerner. *SnapPy, a program for studying the topology and geometry of 3-manifolds, 3.0.3)*, 2021. https://snappy.math.uic.edu/.

[15] Vladimir Voevodsky, 2014. URL https://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/2014_IAS.pdf.

[16] Allen Hatcher. *Algebraic Topology*. Cambridge University Press, 2002.

[17] Colin Adams, Erica Flapan, Allison Henrich, Louis H. Kauffman, Lewis D. Ludwig, and Sam Nelson, editors. *Encyclopedia of Knot Theory*. CRC Press, 2020. URL https:/www.doi.org/10.1201/9781138298217.

[18] E. C. Zeeman. Knotting manifolds. *Bulletin of the American Mathematical Society*, 67(1):117 – 119, 1961. URL https://doi.org/10.1090/S0002-9904-1961-10529-6.

[19] B. Mazur. On embeddings of spheres. *Bulletin of the American Mathematical Society*, 65(2):59 – 65, 1959. URL https://doi.org/10.1090/S0002-9904-1959-10274-3.

[20] R. H. Fox. Free differential calculus. ii. the isomorphism problem of groups. *Annals of Mathematics*, 59(2):196 – 210, 1954. URL https://doi-org.libgate.library.nuigalway.ie/10.2307/1969686.

[21] H. F. Trotter. Homology of group systems with applications to knot theory. *Annals of Mathematics*, 76(2):464 – 498, 1962. URL https://doi.org/10.2307/1970369.

[22] J. Hempel. Homology of coverings. *Pacific Journal of Mathematics*, 112(1):83 – 113, 1984. URL http://projecteuclid.org.libgate.library.nuigalway.ie/euclid.pjm/1102710101.

[23] S. Ocken. Homology of branched cyclic covers of knots. *Proceedings of the American Mathematical Society*, 110(4):1063 – 1067, 1990. URL https://doi-org.libgate.library.nuigalway.ie/10.2307/2047757.

[24] S. Rees and L. Soicher. An algorithmic approach to fundamental groups and covers of combinatorial cell complexes. *Journal of Symbolic Computation*, 29 (1):59 – 77, 2000. URL http://dx.doi.org/10.1006/jsco.1999.0292.

[25] The Sage Developers. SageMath, the Sage Mathematics Software System (Version 8.7.0), 2019. URL https://doc.sagemath.org/html/en/reference/categories/sage/categories/simplicial_sets.html.

[26] P. Brendel, P. Dłotko, G. Ellis, M. Juda, and M. Mrozek. Computing fundamental groups from point clouds. *Applicable Algebra in Engineering, Communication and Computing*, 26(1 - 2):27 – 48, 2015. URL http://dx.doi.org/10.1007/s00200-014-0244-1.

[27] Edwin H. Spanier. *Algebraic Topology*. McGraw-Hill, 1966.

[28] G. Ellis and F. Hegarty. Computational homotopy of finite regular cw-spaces. *Journal of Homotopy and Related Structures*, 9(1):25 – 54, 2014. URL http://dx.doi.org/10.1007/s40062-013-0029-4.

[29] J. Faria Martins L. Kauffman. Invariants of welded virtual knots via crossed module invariants of knotted surfaces. *Compositio Mathematica*, 144(4):1046 – 1080, 2008. URL https://doi.org/10.1112/S0010437X07003429.

[30] S. Satoh. Virtual knot presentation of ribbon torus-knots. *Journal of Knot Theory and Its Ramifications*, 9(4):531 – 542, 2000. doi: https://doi.org/10.1142/

S0218216500000293. URL https://www.worldscientific.com/doi/abs/10.1142/S0218216500000293.

[31] E. Dalvit. visKO - Visualization of Knotted Objects, 2016. URL https://www.youtube.com/watch?v=Lx85kIGjIoQ&list=PLyxHTRWELFBr9TP8jx400bPGP-ECsBufz.

[32] J. Faria Martins. Categorical groups, knots and knotted surfaces. *Journal of Knot Theory and Its Ramifications*, 16(9):1181 – 1217, 2007. URL https://doi-org.libgate.library.nuigalway.ie/10.1142/S0218216507005713.

[33] J. Faria Martins. The fundamental crossed module of the complement of a knotted surface. *Transactions of the American Mathematical Society*, 361(9):4593 – 4630, 2009. URL http://dx.doi.org/10.1090/S0002-9947-09-04576-0.

[34] E. Artin. Zur isotopie zweidimensionaler flächen im $\mathbb{R}^4$. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 4(1):174 – 177, 1925. URL https://doi-org.libgate.library.nuigalway.ie/10.1007/BF02950724.

Index

Appendix A

GAP manual

A.1 UniversalCover

Function call	:	`UniversalCover(X)`.
Input	:	A regular CW-complex X.
Output	:	An equivariant CW-complex $\widetilde{X}$ corresponding to the universal cover of X.

A.2 ChainComplexOfUniversalCover

Function call	:	`ChainComplexOfUniversalCover(X)`.
Input	:	A regular CW-complex X.
Output	:	An equivariant chain complex $C_*\widetilde{X}$ corresponding to the chain complex of the universal cover of X.

A.3 EquivariantCWComplexToRegularCWComplex

Function call	:	`EquivariantCWComplexToRegularCWComplex(X,H)`.
Input	:	An equivariant CW-complex $\widetilde{X}$ corresponding to the universal cover of X and some finite index subgroup H of the fundamental group $\pi_1(X)$.
Output	:	A regular CW-complex corresponding to a $[\pi_1 X : H]$-fold cover of X.

A.4 EquivariantCWComplexToRegularCWMap

Function call	:	`EquivariantCWComplexToRegularCWMap(X,H)`.
Input	:	An equivariant CW-complex $\widetilde{X}$ corresponding to the universal cover of X and some finite index subgroup H of the fundamental group $\pi_1(X)$.
Output	:	A regular CW-map $f : X_H \to X$ which maps the $[\pi_1 X : H]$-fold cover X_H to its base space X.

A.5 LiftedRegularCWMap

Function call	:	`LiftedRegularCWMap(f,p)`.
Input	:	A regular CW-map $f : B \hookrightarrow Y$ and a covering map $p : \widetilde{Y} \to Y$.
Output	:	A regular CW-map $\widetilde{f} : \widetilde{B} \hookrightarrow \widetilde{Y}$ corresponding to the lift of f.

A.6 FirstHomologyCoveringCokernels

Function call	:	`FirstHomologyCoveringCokernels(f,n)`.
Input	:	A regular CW-map $f : B \hookrightarrow Y$ and a positive integer n.
Output	:	The invariant $\mathfrak{J}_n(f)$ (see 4.2).

A.7 KnotComplement

Function call	:	`KnotComplement(l)`, `KnotComplement(l,"rand")`.
Input	:	A list of integer pairs $l = [[a_{1,1}, a_{1,2}], \ldots, [a_{n,1}, a_{n,2}]]$ corresponding to an arc presentation of some knot or link κ. The optional string argument of `"rand"` may be provided.
Output	:	A 3-dimensional regular CW-complex $X = B^3 \setminus \kappa$ homeomorphic to the complement of κ in the 3-ball. If `"rand"` was included in the input the ordering of the 2-cells will be randomised.

A.8 KnotComplementWithBoundary

Function call	:	`KnotComplementWithBoundary(l)`.
Input	:	A list of integer pairs $l = [[a_{1,1}, a_{1,2}], \ldots, [a_{n,1}, a_{n,2}]]$ corresponding to an arc presentation of some knot or link κ.
Output	:	An inclusion of regular CW-complexes $f : X \hookrightarrow Y$ where X is a 2-dimensional complex homeomorphic to the boundary of the thickened knot or link κ and where $Y = B^3 \setminus \kappa$ is a 3-dimensional complex homeomorphic to the complement of κ in the 3-ball.

A.9 ArcPresentationToKnottedOneComplex

Function call	:	`ArcPresentationToKnottedOneComplex(l)`.
Input	:	A list of integer pairs $l = [[a_{1,1}, a_{1,2}], \ldots, [a_{n,1}, a_{n,2}]]$ corresponding to an arc presentation of some knot or link κ.
Output	:	An inclusion of regular CW-complexes $f : X \hookrightarrow Y$ where X is a 1-dimensional complex homeomorphic to κ and where Y is a contractible 3-dimensional complex.

A.10 SphericalKnotComplement

Function call	:	`SphericalKnotComplement(l)`.
Input	:	A list of integer pairs $l = [[a_{1,1}, a_{1,2}], \ldots, [a_{n,1}, a_{n,2}]]$ corresponding to an arc presentation of some knot or link κ.
Output	:	An inclusion of regular CW-complexes $f : X \hookrightarrow Y$ where X is a 1-dimensional complex homeomorphic to the boundary of the thickened knot or link κ and where Y is a contractible 3-dimensional complex homeomorphic to the 3-sphere S^3.

A.11 ArcDiagramToTubularSurface

Function call	:	`ArcDiagramToTubularSurface(a)`.
Input	:	An arc 2-presentation $a = [l_1, l_2, l_3]$ corresponding to some ribbon surface-link K.
Output	:	An inclusion of regular CW-complexes $f : K \hookrightarrow B^3$ where K is a 2-dimensional complex corresponding to the boundary of a (potentially self-intersecting) thickened knot or link, and where B^3 is the 3-ball. If l_3 was bound, f will have as a component object a function `f!.colour(2,k)` which returns the colour of the k^{th} 2-cell of K.

A.12 KinkArc2Presentation

Function call	:	`KinkArc2Presentation(a)`.
Input	:	An arc 2-presentation $a = [l_1, l_2, l_3]$ corresponding to some ribbon surface-link K.
Output	:	An arc 2-presentation corresponding which does not have more than one crossing occurring per horizontal/vertical segment in its planar representation.

A.13 Spin

Function call	:	`Spin(f)`.
Input	:	An inclusion of regular CW-complexes $f : X \hookrightarrow Y$.
Output	:	A regular CW-complex $S_X(Y)$ corresponding to the Artin spinning of Y about the subcomplex X.

A.14 SpunAboutHyperplane

Function call	:	`SpunAboutHyperplane(K)`.
Input	:	A pure cubical complex K.
Output	:	A pure cubical complex $S(K)$ corresponding to the Artin spinning of K about a hyperplane.

A.15 SpunKnotComplement

Function call	:	`SpunKnotComplement([N,M])`.
Input	:	A list of integers $[N, M]$ denoting the M_N, the N^{th} prime knot on M crossings.
Output	:	A 5-dimensional regular CW-complex $S(M_N)$ corresponding to the space obtained by removing an open neighbourhood of points from an unknotted arc of M_N and spinning the resulting space about a hyperplane.

A.16 SpunLinkComplement

Function call	:	`SpunLinkComplement(l)`.
Input	:	A list of integer pairs $l = [[a_{1,1}, a_{1,2}], \ldots, [a_{n,1}, a_{n,2}]]$ corresponding to an arc presentation of some link κ.
Output	:	A 3-dimensional regular CW-complex obtained by Artin spinning κ about a hyperplane.

A.17 RegularCWMapToCWComplex

Function call	:	`RegularCWMapToCWComplex(f)`.
Input	:	An inclusion of regular CW-complexes $f : X \to Y$.
Output	:	A list $[Y, S]$ where S is a list of n lists of integers recording the indexing of each cell of $f(X)$. Let z denote the i^{th} entry of the j^{th} sublist of S. Then we know that the i^{th} $(j-1)$-cell of X maps to the z^{th} $(j-1)$-cell of Y.

A.18 CWSubcomplexToRegularCWMap

Function call	:	`CWSubcomplexToRegularCWMap([Y,S])`.
Input	:	A CW-subcomplex encoded as a list $[Y, S]$ where Y denotes a regular CW-complex and S is a list of lists of positive integers S.
Output	:	A inclusion of regular CW-complexes $f : X \to Y$ where X is the subcomplex of Y as specified by S.

A.19 IntersectionCWSubcomplex

Function call	:	`IntersectionCWSubcomplex([Y1,S1],[Y2,S2])`.
Input	:	Two CW-subcomplexes $[Y, S_1]$ and $[Y, S_2]$.
Output	:	A CW-subcomplex $[Y, S_3]$ corresponding to the intersection of the input CW-subcomplexes.

A.20 PathComponentsCWSubcomplex

Function call	:	`PathComponentsCWSubcomplex([Y,S])`.
Input	:	A CW-subcomplex $[Y, S]$.
Output	:	A list of CW-subcomplexes $[[Y, S_1], \ldots, [Y, S_n]]$ corresponding to each of the n path components of $[Y, S]$.

A.21 ClosureCWCell

Function call	:	`ClosureCWCell(Y,k,i)`.
Input	:	A CW-complex Y and two integers $k \geq 0$, $i \geq 1$.
Output	:	A CW-subcomplex $[Y, S]$ corresponding to the closure of the i^{th} k-cell of Y.

A.22 HAP_KK_AddCell

Function call	:	`HAP_KK_AddCell(B,k,b,c)`.
Input	:	The boundary list of a cell complex B, an integer $k \geq 0$, and two lists of positive integers b and c.
Output	:	This function modifies B by adding to it a k-cell whose boundary $(k-1)$-cells are determined by b, and whose coboundary $(k+1)$-cells are determined by c.

A.23 BarcentricallySubdivideCell

Function call	:	`BarycentricallySubdivideCell(f,k,i)`.
Input	:	An inclusion of regular CW-complexes $f : X \hookrightarrow Y$, and two integers $k \geq 0$, $i \geq 1$.
Output	:	An inclusion of regular CW-complexes f' where f' denotes the map f after having barycentrically subdivided the the i^{th} k-cell of Y.

A.24 SubdivideCell

Function call	:	`SubdivideCell(f,k,i)`.
Input	:	An inclusion of regular CW-complexes $f : X \hookrightarrow Y$, and two integers $k \geq 0$, $i \geq 1$.
Output	:	An inclusion of regular CW-complexes f' where f' denotes the map f after subdividing the i^{th} k-cell of Y into as many k-cells as there are $(k-1)$-cells in its boundary.

A.25 RegularCWComplexComplement

Function call	:	`RegularCWComplexComplement(f)` `RegularCWComplexComplement(f, "all", "barycentric", true)`, `RegularCWComplexComplement(f, "some", "basic", false)`, `RegularCWComplexComplement(f, "some", "none", false)`, etc.
Input	:	An inclusion of regular CW-complexes $f : X \hookrightarrow Y$ with $Y \subset X$ a pure subcomplex of X, and three optional arguments.
Output	:	A regular CW-complex $Y \setminus N_\epsilon(X)$ homeomorphic to the complement of some thickened tubular neighbourhood of X in Y as in Algorithm 5.1.1. The arguments `"all"` and `"some"` determine how many cells are checked for satisfying the contractible closure condition. The arguments `"barycentric"`, `"basic"` and `"none"` refer to methods of subdivision. Note that `"none"` should be used only when it is known that no subdivision will be needed in forming the complement. The final boolean argument prints/does not print a progress bar indicating how far along the algorithm is in calculating the tubular neighbourhood. The omission of the optional arguments assumes the input `RegularCWComplexComplement(f, "all", "basic", false)`.

A.26 SequentialRegularCWComplexComplement

Function call	:	`SequentialRegularCWComplexComplement(f)` `SequentialRegularCWComplexComplement(f, "all", "barycentric", true)`, `SequentialRegularCWComplexComplement(f, "some", "basic", false)`, `SequentialRegularCWComplexComplement(f, "some", "none", false)`, etc.
Input	:	An inclusion of regular CW-complexes $f : X \hookrightarrow Y$ with $Y \subset X$ a pure subcomplex of X, and three optional arguments.
Output	:	Precisely as above in A.25 but using the methods as described in Algorithm 5.1.1.1 to form the tubular neighbourhood.

A.27 LiftColouredSurface

Function call	:	`LiftColouredSurface(f)`.
Input	:	An inclusion of regular CW-complexes $f : X \hookrightarrow Y$ with component object `f!.colour` bound, ideally as the output of the `ArcDiagramToTubularSurface` algorithm.
Output	:	An inclusion of regular CW-complexes $\widetilde{f} : \widetilde{X} \hookrightarrow Y \times I$ corresponding to the lift of X as specified by `f!.colour`.

A.28 ViewArc2Presentation

Function call	:	`ViewArc2Presentation(a)`.
Input	:	An arc 2-presentation $a = [l_1, l_2, l_3]$ of some ribbon surface-link K.
Output	:	A PNG file will be displayed illustrating the arc 2-presentation.

A.29 NumberOfCrossingsInArc2Presentation

Function call	:	`NumberOfCrossingsInArc2Presentation(a)`.
Input	:	An arc 2-presentation $a = [l_1, l_2, l_3]$ (neither l_2 nor l_3 need to be bound).
Output	:	A non-negative integer n corresponding to the number of crossings in a.

A.30 RandomArc2Presentation

Function call	:	`RandomArc2Presentation()`.
Input	:	Nothing.
Output	:	A randomised arc 2-presentation arising as the knot sum of prime knots.

A.31 Tube

Function call	:	`Tube(a)`.
Input	:	An arc 2-presentation $a = [l_1, l_2, l_3]$ of some ribbon surface-link K.
Output	:	A 4-dimensional regular CW-complex corresponding to the Tube map applied to a.

A.32 ThreeManifoldViaDehnSurgery

Function call	:	`ThreeManifoldViaDehnSurgery(arc,p,q)`.
Input	:	An arc presentation `arc` and two coprime integers p and q.
Output	:	The closed, connected, orientable 3-manifold obtained by performing p/q-surgery on the complement of the link determined by `arc` in the 3-sphere.

Appendix B

GAP code

This final appendix contains all code that was written during the course of this thesis either by the author or in collaboration with the supervisor. Those sections containing code that was written by—and that is maintained by—the supervisor are marked with the symbol †.

B.1 UniversalCover†

```
InstallGlobalFunction(
    UniversalCover,
    function(X)
    local
        U, G, dim, Elts, FreeElts, Boundary,
        PseudoBoundary, gamma, epi, n, i;

    if not IsHapRegularCWComplex(X) then
        Print("The function applies only to regular CW complexes.\n");
        return fail;
    fi;
    if Length(PiZero(X)[1])>1 then
        Print(
    "The function applies only to path connected regular CW complexes.\n"
        );
        return fail;
    fi;
```

```
OrientRegularCWComplex(X);
dim:=EvaluateProperty(X,"dimension");
G:=FundamentalGroupOfRegularCWComplex(X,"nosimplify");
epi:=EpimorphismFromFreeGroup(G);
gamma:=G!.edgeToWord;
Elts:=[One(G)];
FreeElts:=[PreImagesRepresentative(epi,One(G))];
PseudoBoundary:=List([1..dim],i->[]);

#####################################################################
Boundary:=function(n,k)
    local
        vts, g, fg, ng, B, bb, bbb, xx, ii, j,
        kk, pos, bool, indx, bnd, ornt;

    if n=0 then
        return [];
    fi;
    if IsBound(PseudoBoundary[n][k]) then
        return 1*PseudoBoundary[n][k];
    fi;

    if n=1 then
        vts:=X!.boundaries[2][k];
        ornt:=X!.orientation[2][k];
        g:=gamma(k);
        fg:=PreImagesRepresentative(epi,g);
        ng:=Position(FreeElts,fg);
        if ng=fail then
            Add(Elts,g);
            Add(FreeElts,fg);
            ng:=Length(Elts);
        fi;
        PseudoBoundary[n][k]:=[
            [ornt[1]*vts[2],1],
            [ornt[2]*vts[3],ng]
        ];
        return 1*PseudoBoundary[n][k];
```

```
fi;

B:=1*X!.boundaries[n+1][k];
B:=1*B{[2..Length(B)]};
ornt:=X!.orientation[n+1][k];
bnd:=List([1..Length(B)],j->[ornt[j]*B[j],1]);
indx:=1*B{[2..Length(B)]};

while Length(indx)>0 do
    bool:=false;
    for ii in indx do
        bb:=1*Boundary(n-1,ii);
        bbb:=1*List(bb,x->AbsInt(x[1]));
        for j in Difference(B,indx) do
            xx:=Boundary(n-1,j);
            for kk in [1..Length(xx)] do
                pos:= Position(bbb, AbsInt(xx[kk][1]));
                if not pos=fail then
                    #g:=(Elts[xx[kk][2]]*Elts[bb[pos][2]]^-1);
                    g:=Elts[bnd[Position(B,j)][2]]*
                    Elts[xx[kk][2]]*Elts[bb[pos][2]]^-1;
                    fg:=PreImagesRepresentative(epi,g);
                    ng:=Position(FreeElts,fg);
                    if ng=fail then
                        Add(Elts,g);
                        Add(FreeElts,fg);
                        ng:=Length(Elts);
                    fi;
                    bnd[Position(B,ii)][2]:=ng;
                    indx:=Filtered(indx,a->not a=ii);
                    bool:=true;
                    break;
                fi;
                if bool then
                    break;
                fi;
            od;
            if bool then
                break;
```

```
                    fi;
                od;
                if bool then
                    break;
                fi;
            od;
        od;
        PseudoBoundary[n][k]:= bnd;

        return 1*PseudoBoundary[n][k];
    end;
    ##########################################################################

    U:=Objectify(
        HapEquivariantCWComplex,
        rec(
            dimension:=X!.nrCells,
            boundary:=Boundary,
            elts:=Elts,
            group:=G,
            stabilizer:=Group(One(G)),
            baseSpace:=X,
            properties:=
            [
                ["dimension",dim],
            ]
        )
    );

    for n in [1..dim] do
        for i in [1..U!.dimension(n)] do
            U!.boundary(n,i);
        od;
    od;

    return U;
end);
```

B.2 ChainComplexOfUniversalCover†

```
InstallGlobalFunction(
    ChainComplexOfUniversalCover,
    function(arg)
    local
        X, Y, dim, bool, nrCriticalCells, C, Boundary, critical,
        BASIS, BIJ, DEFORM, DEFORMrec, f, bnd, sn, def, def1, def2,
        mult, inv, BOUNDARY, BNDrec;

    X:=ContractedComplex(arg[1]);
    dim:=Dimension(X);
    X!.allcocriticalcells:=dim;
    Y:=UniversalCover(X);
    bool:=true;

    if Length(arg)>1 then
        bool:=arg[2];
    fi;
    if not bool then
        return Objectify(
            HapEquivariantChainComplex,
            rec(
                dimension:=Y!.dimension,
                boundary:=Y!.boundary,
                elts:=Y!.elts,
                group:=Y!.group,
                properties:=[
                    ["dimension",dim],
                    ["characteristic",0],
                    ["length",dim]
                ]
            )
        );
    fi;

    ################################
    mult:=function(g,h)
```

```
    local
        m, pos;

    m:=Y!.elts[g]*Y!.elts[h];
    #pos:=Position(Y!.elts,m);
    pos:=fail;
    if pos=fail then
        pos:=Length(Y!.elts)+1;
        Add(Y!.elts,m);
    fi;

    return pos;
end;
################################


###################################
inv:=function(g)
    local
        m, pos;

        m:=Y!.elts[g]^-1;
        # pos:=Position(Y!.elts,m);
        pos:=fail;
        if pos=fail then
            pos:=Length(Y!.elts)+1;
            Add(Y!.elts,m);
        fi;

        return pos;
    end;
###################################


critical:=CriticalCells(X);
C:=ChainComplexOfRegularCWComplexWithVectorField(X);
BASIS:=C!.basis;
BIJ:=C!.bij;
dim:=Maximum(List(CriticalCells(X),x->x[1]));
```

```
DEFORMrec:=List([1..dim+1],i->[]);;

############################################################
DEFORM:=function(n,c)
    local
        k, kk, kkk, sgnk, sgnn, g, r, x;



    kk:=c[1];
    k:=AbsInt(kk);
    sgnk:=SignInt(kk);
    g:=c[2];

    if [n,k] in critical then
        return [c];
    fi;
    if n>0 then
        if IsBound(X!.vectorField[n][k]) then
            return [];
        fi;
    fi;
    if IsBound(DEFORMrec[n+1][k]) then
        r:=List(DEFORMrec[n+1][k],a->[a[1],mult(g,a[2])]);
        if sgnk=1 then
            return r;
        else
            return List(r,a->[-a[1],a[2]]);
        fi;
    fi;

    f:=X!.inverseVectorField[n+1][k];
    bnd:=Y!.boundary(n+1,f);
    kkk:=Position(List(bnd,a->AbsInt(a[1])),k);
    kkk:=bnd[kkk][2];
    kkk:=inv(kkk);
    #Apply(bnd,a->[a[1],mult(g,mult(kkk,a[2]))]);
    Apply(bnd,a->[a[1], mult(kkk,a[2]) ]);
    sn:=X!.orientation[n+2][f];
```

```
    def:=[]; def1:=[]; def2:=[];
    for x in [1..Length(bnd)] do
        if not AbsInt(bnd[x][1])=k then
            Add(def1,[bnd[x][1],bnd[x][2]]);
        else
            sgnn:=sn[x];
            break;
        fi;
    od;
    cnt:=x+1;

    for x in [cnt..Length(bnd)] do
        Add(def2,[bnd[x][1],bnd[x][2]]);
    od;

    if sgnn=1 then
        def:=Concatenation(def1,def2);
        def:=List(def,a->[-a[1],a[2]]);
    else
        def:=Concatenation(def2,def1);
    fi;
    def:=List(def,x->DEFORM(n,x));
    def:=Concatenation(def);
    def:=Filtered(def,x->Length(x)>0);
    def:=AlgebraicReduction(def);
    DEFORMrec[n+1][k]:=def;
    def:=List(def,a->[a[1],mult(g,a[2])]);
    if sgnk=1 then
        return def;
    fi;
    def:=List(def,a->[-a[1],a[2]]);
    return def;
end;
############################################################

# outputs the number of critical n-cells in C_n
############################################################
nrCriticalCells:=function(n)
    return Length(Filtered(X!.criticalCells, x->x[1]=n));
```

```
end;
############################################################

# boundary of critical cells -- reindexed
####################################################################
Boundary:=function(n,k)
local
    bnd, BND, x, y, L;

    bnd:=Y!.boundary(n,BASIS[n+1][k]);
    BND:=[];
    for x in bnd do
        L:=DEFORM(n-1,x);
        for y in L do
            Add(BND,[SignInt(y[1])*BIJ[n][AbsInt(y[1])],y[2]]);
        od;
    od;

    return BND;
end;
####################################################################

BNDrec:=List([1..dim],i->[]);

#####################################################
BOUNDARY:=function(n,k)
    local
        kk, sk;

    if n<1 or n>dim then
        return [];
    fi;
    kk:=AbsInt(k); sk:=SignInt(k);

    if not IsBound(BNDrec[n][kk]) then
        BNDrec[n][kk]:=Boundary(n,kk);
    fi;

    if sk=1 then
```

```
            return BNDrec[n][kk];
        else
            return List(BNDrec[n][kk],a->[-a[1],a[2]]);
        fi;
    end;
    ######################################################

    return Objectify(
        HapEquivariantChainComplex,
        rec(
            dimension:=nrCriticalCells,
            boundary:=BOUNDARY,
            elts:=Y!.elts,
            group:=Y!.group,
            properties:=[
                ["dimension",dim],
                ["characteristic",0],
                ["length",dim]
            ]
        )
    );
end);
```

B.3 EquivariantCWComplexToRegularCWComplex†

```
InstallGlobalFunction(
    EquivariantCWComplexToRegularCWComplex,
    function(U,H)
    local
        boundaries, bnd, gbnd, ind, trans,
        pair2int, dimU, n, k, g, W;

    if not IsHapEquivariantCWComplex(U)  then
        Print("This function applies only to G-CW-complexes.\n");
        return fail;
    fi;
```

```
if not IsSubgroup(U!.group,H) then
    Print(
        "The provided group is not a subgroup of the fundamental group
        of the G-CW-complex.\n"
    );
    return fail;
fi;

# Apply(U!.elts,x->x^-1); # So now we have a right action!
trans:=RightCosets(U!.group,H);
ind:=Length(trans);
if not ind < infinity then
    Print("The provided subgroup is not of finite index.\n");
    return fail;
fi;
dimU:=EvaluateProperty(U,"dimension");


############################
pair2int:=function(e,gH);
    return (e-1)*ind + gH;
end;
############################

boundaries:=[];
boundaries[1]:=List([1..ind*U!.dimension(0)],i->[1,0]);
for n in [1..dimU] do
    boundaries[n+1]:=[];
    for k in [1..U!.dimension(n)] do
        bnd:=U!.boundary(n,k);
        for g in trans do
            gbnd:=List(bnd,x->[x[1],
            Position(trans, g*U!.elts[x[2]])]);
            gbnd:=List(gbnd,x->pair2int(AbsInt(x[1]),x[2]));
            gbnd:=Concatenation([Length(gbnd)],gbnd);
            Add(boundaries[n+1],gbnd);
        od;
    od;
od;
```

```
    boundaries[dimU+2]:=[];

    W:=HAPRegularCWComplex(boundaries);
    W!.index:=ind;

    return W;
end);
```

B.4 EquivariantCWComplexToRegularCWMap†

```
InstallGlobalFunction(
    EquivariantCWComplexToRegularCWMap,
    function(U,H)
    local
        YH, Y, map, ind;

    YH:=EquivariantCWComplexToRegularCWComplex(U,H);
    Y:=U!.baseSpace;
    ind:=YH!.index;

    ####################
    map:=function(n,k)
        local
            m, a;

        m:=k mod ind;
        a:=Int(k/ind);
        if m=0 then
            return a;
        else
            return a+1;
        fi;
    end;
    ####################

    return Objectify(
        HapRegularCWMap,
```

```
        rec(
            source:=YH,
            target:=Y,
            mapping:=map
        )
    );
end);
```

B.5 LiftedRegularCWMap†

```
InstallGlobalFunction(
    LiftedRegularCWMap,
    function(f,p)
    local
        Y, B, W, WB, WBcells, WBcellsinv, Bcells, WBbnd,
        WBorient, cnt, ff, n, i;

    #       ff
    #    WB ---> W                ff is the lift of f.
    #            |
    #            | p
    #       f    v
    #    B ----> Y

    Y:=Target(f);
    B:=Source(f);
    W:=Source(p);
    if not Y=Target(p) then
        return fail;
    fi;

    Bcells:=[];
    for n in [1..Length(B!.boundaries)] do
        Bcells[n]:=SSortedList(
            List([1..B!.nrCells(n-1)],
            i->f!.mapping(n-1,i))
        );
```

```
od;

WBcells:=[];
WBcellsinv:=[];
for n in [1..Length(W!.boundaries)] do
    WBcells[n]:=[];
    WBcellsinv[n]:=[];
    cnt:=0;
    for i in [1..Length(W!.boundaries[n])] do
        if p!.mapping(n-1,i) in Bcells[n] then
            Add(WBcells[n], i);
            cnt:=cnt+1;WBcellsinv[n][i]:=cnt;
        fi;
    od;
od;

############################
ff:=function(n,i);
    return WBcells[n+1][i];
end;
############################

WBbnd:=[];
WBbnd[1]:=List(WBcells[1],i->[1,0]);
for n in [2..Length(WBcells)] do
    WBbnd[n]:=List(WBcells[n],i->1*W!.boundaries[n][i]);
    WBbnd[n]:=List(
        WBbnd[n],
        x->Concatenation(
            [x[1]],
            List(x{[2..1+x[1]]},i->WBcellsinv[n-1][i]
            )
        )
    );
od;

WB:=RegularCWComplex(WBbnd);
OrientRegularCWComplex(W);
WBorient:=[];
```

```
    for n in [1..1+Dimension(WB)] do
        WBorient[n]:=List(
            [1..WB!.nrCells(n-1)],
            i->W!.orientation[n][ff(n-1,i)]
        );
    od;
    WB!.orientation:=WBorient;

    return Objectify(
        HapRegularCWMap,
        rec(
            source:=WB,
            target:=W,
            mapping:=ff
        )
    );

end);
```

B.6 FirstHomologyCoveringCokernels†

```
InstallGlobalFunction(
    FirstHomologyCoveringCokernels,
    function(f,n)
    local
        Y, U, G, L, M, p, x, y;

    Y:=Target(f);
    U:=UniversalCover(Y);
    G:=U!.group;
    L:=LowIndexSubgroupsFpGroup(G,n);;
    L:=Filtered(L,H->Index(G,H)=n);;
    L:=List(L,H->EquivariantCWComplexToRegularCWMap(U,H));
    L:=List(L,p->LiftedRegularCWMap(f,p));

    M:=[];
    for p in L do
```

```
        CocriticalCellsOfRegularCWComplex(
            Source(p),
            Dimension(Source(p))
        );
        for y in CocriticalCellsOfRegularCWComplex(Source(p),0) do
            Add(M,[p,y]);
        od;
    od;

    M:=List(M,x->FundamentalGroup(x[1],x[2][2]));
    M:=List(
        M,
        h->AbelianInvariants(
            Target(h)/GeneratorsOfGroup(Image(h))
        )
    );

    return SortedList(M);
end);
```

B.7 KnotComplement

```
InstallGlobalFunction(
    KnotComplement,
    function(arg...)
    local
        rand, D, len, signless, PuncturedDisk,
        P, grid, loop_correction, PuncturedTube, i;

    if Length(arg)>1
        then
        rand:=true;
        Print("Random 2-cell selection is enabled.\n");
    else
        rand:=false;
    fi;
```

```
D:=arg[1];
len:=Length(D);
signless:=List(D,x->[AbsInt(x[1]),AbsInt(x[2])]);

PuncturedDisk:=function(D)
    local
        grid, i, IsIntersection,
        CornerConfiguration, bound,
        bigGrid, GridFill, j, tick,
        hslice, vslice, k, 0c, Orient,
        path, FaceTrace, traced_bound, cgrid;

    grid:=List([1..len],x->List([1..len],y->0));
    for i in [1..len]
        do
        grid[len-i+1][D[i][1]]:=1;
        grid[len-i+1][D[i][2]]:=1;
    od;

    IsIntersection:=function(i,j)

        if grid[i][j]=0
            then
            if 1 in grid[i]{[1..j]}
                then
                if 1 in grid[i]{[j..len]}
                    then
                    if 1 in List([1..i],x->grid[x][j])
                        then
                        if 1 in List([i..len],x->grid[x][j])
                            then
                            return true;
                        fi;
                    fi;
                fi;
            fi;
        fi;

    return false;
```

```
        end;

        CornerConfiguration:=function(i,j);

            if grid[i][j]=1
                then
                if Size(Positions(grid[i]{[j..len]},1))=2
                    then
                    if Size(Positions(List([i..len],x->grid[x][j]),1))=2
                        then # Corner type 1, i.e : __
                        return 1; #                |
                    elif Size(Positions(List([1..i],x->grid[x][j]),1))=2
                        then # Corner type 3, i.e :
                        return 3; #                |__
                    fi;
                elif Size(Positions(grid[i]{[1..j]},1))=2
                    then
                    if Size(Positions(List([i..len],x->grid[x][j]),1))=2
                        then # Corner type 2, i.e : __
                        return 2; #                  |
                    elif Size(Positions(List([1..i],x->grid[x][j]),1))=2
                        then # Corner type 4, i.e :
                        return 4; #                 __|
                    fi;
                fi;
            fi;

            return 0;
        end;

        bound:=[[],[],[],[],[]];
        bigGrid:=List([1..2*len],x->List([1..2*len],y->0));

        GridFill:=function(c,i,j);
# places an * at each point where a 0-cell is to be added to bigGrid
            if c=1 or c=4
                then
                bigGrid[(2*i)-1][(2*j)-1]:='*';
                bigGrid[2*i][2*j]:='*';
```

```
        elif c=2 or c=3
            then
            bigGrid[(2*i)-1][2*j]:='*';
            bigGrid[2*i][(2*j)-1]:='*';
        fi;
    end;

    for i in [1..len]
        do # loop through bigGrid and add temporary *s
        for j in [1..len]
            do
            if IsIntersection(i,j)
                then # four 0-cells at an intersection
                bigGrid[(2*i)-1][(2*j)-1]:='*';
                bigGrid[(2*i)-1][2*j]:='*';
                bigGrid[2*i][(2*j)-1]:='*';
                bigGrid[2*i][2*j]:='*';
            elif grid[i][j]=1
                then # two 0-cells at the ends of each horizontal bar
                GridFill(CornerConfiguration(i,j),i,j);
            fi;
        od;
    od;

    tick:=2;
    for i in [1..2*len]
        do # number the 0-cells row-by-row
        for j in [1..2*len]
            do
            if bigGrid[i][j]='*'
                then
                bigGrid[i][j]:=tick;
                tick:=tick+1;
            fi;
        od;
    od;

    for i in [1..2*len]
        do # connect all 0-cells that lie in the same
```

```
        hslice:=[]; # horizontal/vertical 'slice'
        vslice:=[];
        for j in [1..2*len]
            do
            if not bigGrid[i][j]=0
                then
                Add(hslice,bigGrid[i][j]);
            fi;
            if not bigGrid[j][i]=0
                then
                Add(vslice,bigGrid[j][i]);
            fi;
        od;
        for k in [1..Length(hslice)]
            do
            if Length(hslice)>k
                then
                Add(bound[2],[2,hslice[k],hslice[k+1]]);
            fi;
        od;
        for k in [1..Length(vslice)]
            do
            if Length(vslice)>k
                then
                Add(bound[2],[2,vslice[k],vslice[k+1]]);
            fi;
        od;
    od;

    for i in [1..len]
        do # add the looping 1-cells to the 1-skeleton
        for j in [1..len]
            do
            if CornerConfiguration(i,j) in [1,4]
                then
                Add(bound[2],[
                    2,
                    bigGrid[(2*i)-1][(2*j)-1],
                    bigGrid[2*i][2*j]
```

```
                    ]
                );
                Add(bound[2],[
                    2,
                    bigGrid[(2*i)-1][(2*j)-1],
                    bigGrid[2*i][2*j]
                    ]
                );
            elif CornerConfiguration(i,j) in [2,3]
                then
                Add(bound[2],[
                    2,
                    bigGrid[(2*i)-1][2*j],
                    bigGrid[2*i][(2*j)-1]
                    ]
                );
                Add(bound[2],[
                    2,
                    bigGrid[(2*i)-1][2*j],
                    bigGrid[2*i][(2*j)-1]
                    ]
                );
            fi;
        od;
    od;

    0c:=Maximum(List(bigGrid,x->Maximum(x)));
    for i in [1..0c+1]
        do
        Add(bound[1],[1,0]);
    od;

    Add(bound[2],[2,1,2]); # connect the central component to the
    Add(bound[2],[2,Length(bound[1])-1,Length(bound[1])]); # circumf.
    Add(bound[2],[2,1,Length(bound[1])]); # of the disk
    Add(bound[2],[2,1,Length(bound[1])]);

    bigGrid:=FrameArray(bigGrid);
    bigGrid[1][2]:=1; # Adds the first and last 0-cells to bigGrid
```

```
        bigGrid[Length(bigGrid)][Length(bigGrid[1])-1]:=0c+1;

        Orient:=function(bound)
# traces the 1-skeleton in a clockwise walk to yield the 2-cells
            local
                unchosen, neighbours, i, j,
                Clockwise;

            unchosen:=List(ShallowCopy(bound[2]),x->[x[2],x[3]]);
            neighbours:=List(ShallowCopy(bound[1]),x->[]);

            for i in [1..Length(bound[1])]
                do
                for j in [1..Length(unchosen)]
                    do
                    if i in unchosen[j]
                        then
                        Add(neighbours[i],j);
                    fi;
                od;
            od;

            Clockwise:=function(neighbours)
                local # orders clockwise the neighbours of each 0-cell
                    oriented, first0, last0,
                    i, j, x, k, l, posi, posx;

                oriented:=List(neighbours,x->List([1..12],y->"pass"));
                first0:=SortedList(neighbours[1]);
                last0:=SortedList(neighbours[Length(neighbours)]);

                oriented[1][7]:=first0[1];
                oriented[1][6]:=first0[3];
                oriented[1][8]:=first0[2];
            # these two orderings are always fixed;
            # they correspond to the circumferential edges
                oriented[Length(oriented)][1]:=last0[1];
                oriented[Length(oriented)][2]:=last0[3];
                oriented[Length(oriented)][12]:=last0[2];
```

```
for i in [2..Length(neighbours)-1]
    do # excludes the 1st and last 0-cells
    for j in [1..Length(neighbours[i])]
        do # x is a neighbouring 0-cell to i
        x:=bound[2][neighbours[i][j]];
        x:=Filtered(x{[2,3]},y->y<>i)[1];
        for k in [1..Length(bigGrid)]
            do
            for l in [1..Length(bigGrid[1])]
                do
                if i=bigGrid[k][l]
                    then
                    posi:=[k,l];
                fi;
                if x=bigGrid[k][l]
                    then
                    posx:=[k,l];
                fi;
            od;
        od;
        # below are the checks for orientation,
        # there are 12 in total (two for each diagonal):
        # _\\|//_
        #  //|\\
        if posi[1]>posx[1]
            then
            if posi[2]=posx[2]
                then
                oriented[i][1]:=neighbours[i][j];
            elif posi[2]<posx[2]
                then
                if oriented[i][2]="pass"
                    then # *assign the upper loop first*
                    oriented[i][2]:=neighbours[i][j];
                else
                    oriented[i][3]:=neighbours[i][j];
                fi;
            elif posi[2]>posx[2]
```

```
                then
                if oriented[i][12]="pass"
                    then
                    oriented[i][12]:=neighbours[i][j];
                else
                    oriented[i][11]:=neighbours[i][j];
                fi;
            fi;
        elif posi[1]=posx[1]
            then
            if posi[2]<posx[2]
                then
                oriented[i][4]:=neighbours[i][j];
            elif posi[2]>posx[2]
                then
                oriented[i][10]:=neighbours[i][j];
            fi;
        elif posi[1]<posx[1]
            then
            if posi[2]=posx[2]
                then
                oriented[i][7]:=neighbours[i][j];
            elif posi[2]<posx[2]
                then
                if oriented[i][5]="pass"
                    then
                    oriented[i][5]:=neighbours[i][j];
                else
                    oriented[i][6]:=neighbours[i][j];
                fi;
            elif posi[2]>posx[2]
                then
                if oriented[i][9]="pass"
                    then
                    oriented[i][9]:=neighbours[i][j];
                else
                    oriented[i][8]:=neighbours[i][j];
                fi;
            fi;
```

```
                    fi;
                od;
            od;

            return oriented;
        end;

        return Clockwise(neighbours);
    end;

    path:=Orient(bound);
    # this is an ordered list of the neighbours of each 1-cell

    FaceTrace:=function(path)
        local
            unselectedEdges, sourceORtarget, faceloops,
            x, ClockwiseTurn, IsLoop, loop_correction, edge,
            2nd_loop, 2cell, sORt, ori, e1, e0, i;

        unselectedEdges:=List([1..Length(bound[2])-2]);
        unselectedEdges:=Concatenation(
            unselectedEdges,
            unselectedEdges
        );
        Add(unselectedEdges,Length(bound[2])-1);
        Add(unselectedEdges,Length(bound[2]));
# list of two of each edge except for the circumferential edges

        ClockwiseTurn:=function(p,e)
# inputs the orientation list of a node and the number of an edge in that
# list, outputs the next edge after a clockwise turn
            local
                f;

            f:=(Position(p,e) mod 12)+1;
            while p[f]="pass"
                do
                f:=(f mod 12)+1;
            od;
```

```
    return p[f];
end;

############ ADDED 15/10/19 ############
IsLoop:=function(n)

    if Length(Positions(bound[2],bound[2][n]))=2 then
        return true;
    else
        return false;
    fi;

end;

loop_correction:=List(bound[2],x->0);
########################################

sourceORtarget:=List([1..Length(bound[2])],y->[3,2]);
x:=1;
while unselectedEdges<>[]
    do # main loop, locates all 2-cells
    if rand
        then
        x:=Random([1..Length(bound[2])]); # random edge
    fi;
    while (not x in unselectedEdges) and
        (not e1 in unselectedEdges)
        do # reselect edge if it already has two
        if rand # 2-cells in its coboundary
            then
            x:=Random([1..Length(bound[2])]);
        else
            x:=x+1;
        fi;
    od;

    2cell:=[x];# the 2-cell begins with just x in its boundary
    if rand
```

```
    then
    sORt:=Random([2,3]);
else
    sORt:=sourceORtarget[x][Length(sourceORtarget[x])];
    Unbind(sourceORtarget[x][Length(sourceORtarget[x])]);
fi;
ori:=path[bound[2][x][sORt]]; # the orient. of x's target
e0:=bound[2][x][sORt];
e1:=ClockwiseTurn(ori,x); # next edge to travel along
while e1<>x
    do
    Add(2cell,e1);
    e0:=Filtered(bound[2][e1]{[2,3]},y->y<>e0)[1];
    # e1's target
    ori:=path[e0];
    e1:=ClockwiseTurn(ori,e1);
od;
Add(2cell,Length(2cell),1);
if (not Set(2cell) in List(bound[3],x->Set(x)))
    then
    for i in Filtered(2cell{[2..Length(2cell)]},
    y->y in unselectedEdges)
        do
        Unbind(
            unselectedEdges[Position(unselectedEdges,i)]
        );
    od;
    Add(bound[3],2cell);
fi;

############ ADDED 15/10/19 ############
# orders any loops that are present in the 2cell by the
# order in which they were selected (doesn't include
# redundant 2cells which are filtered out after the main
# while loop)
if 2cell[1]<>2 then
    faceloops:=Filtered(2cell{[2..Length(2cell)]},IsLoop);
    if faceloops<>[] then
        for edge in faceloops do
```

```
                        if loop_correction[edge]=0 then
                            loop_correction[edge]:=1;
                            2nd_loop:=Filtered(
                                Positions(
                                    bound[2],
                                    bound[2][edge]
                                ),
                                y->y<>edge
                            )[1];
                            loop_correction[2nd_loop]:=2;
                        fi;
                    od;
                fi;
            fi;
            ########################################

        od;

        bound[3]:=Filtered(bound[3],y->y[1]<>2);
        return [bound,loop_correction];
    end;

    cgrid:=grid*0; # this is needed at the very end when
    for i in [1..Length(grid)] # patching the tubes together
        do
        for j in [1..Length(grid)]
            do
            cgrid[i][j]:=CornerConfiguration(i,j);
        od;
    od;

    traced_bound:=FaceTrace(path);

    return [traced_bound[1],cgrid,traced_bound[2]];
end;

P:=PuncturedDisk(D);
grid:=P[2];
loop_correction:=P[3];
```

```
    P:=P[1];

    PuncturedTube:=function(bound)
        local
            l0, l1, l2, DuplicateDisk,
            JoinDisks, Patch, prepatch,
            postpatch, Cap;

        l0:=Length(bound[1]);
        l1:=Length(bound[2]);
        l2:=Length(bound[3]);

        DuplicateDisk:=function(bound)
            local # creates a disjoint copy of the punctured
                i, edges2, faces2; # disk and concatenates the two

            for i in [1..l0]
                do
                Add(bound[1],[1,0]);
            od;

            edges2:=List(ShallowCopy(bound[2]),x->[2,x[2]+l0,x[3]+l0]);
            bound[2]:=Concatenation(bound[2],edges2);

            faces2:=List(ShallowCopy(bound[3]),
            x->Concatenation([x[1]],x{[2..Length(x)]}+l1));
            bound[3]:=Concatenation(bound[3],faces2);

            return bound;
        end;

        bound:=DuplicateDisk(bound);
        loop_correction:=Concatenation(loop_correction,loop_correction);

        JoinDisks:=function(bound)
# patch together the two disks via 1-cells, 2-cells & 3-cells
# i.e., form the space P x I where I is the unit interval
            local
                i, x, y, 3cell;
```

```
    for i in [1..l0]
        do # connect the 2 disks by 1-cells
        Add(bound[2],[2,i,l0+i]);
    od;

    for i in [1..l1]
        do # for each base 1-cell, form a 2-cell
        x:=bound[2][i][2];
        y:=bound[2][i][3];
        Add(bound[3],[4,i,l1+i,(l1*2)+x,(l1*2)+y]);
    od;

    for i in [1..l2]
        do # form a 3-cell from each 2-cell in the base disk
       x:=List(bound[3][i]{[2..Length(bound[3][i])]},y->y+(2*l2));
        3cell:=Concatenation([i,l2+i],x);
        Add(3cell,Length(3cell),1);
        Add(bound[4],3cell);
    od;

    return bound;
end;

bound:=JoinDisks(bound);

prepatch:=Length(bound[3]);
postpatch:=0;

Patch:=function(bound)
    local # close the tubes to complete the construction
        loops, horizontals, verticals, i,
        lst, htube, h1, h2, vtube, x,
        cycle, loopless;

    loops:=Filtered(
        [1..l1-4],
        x->Length(Positions(bound[2],bound[2][x]))>1 and
        bound[2][x][2]<>1
```

```
);

horizontals:=Filtered(
    [1..l1-4],
    x->bound[2][x][2]=bound[2][x][3]-1
);

verticals:=Filtered(
    [1..l1-4],
    x->not (x in loops or x in horizontals)
);
verticals:=verticals+l1;

for i in [1..Length(loops)/4]
    do
    lst:=[0,0];
    if 1 in grid[i]
        then # check for corner configuration
        lst[1]:=2
    fi;
    if 2 in grid[i]
        then
        lst[2]:=4;
    fi;
    if 3 in grid[i]
        then
        lst[1]:=1;
    fi;
    if 4 in grid[i]
        then
        lst[2]:=3;
    fi;

    htube:=loops{lst+4*(i-1)};
    h1:=Filtered(
        horizontals,
        x->bound[2][x][2] in
        [bound[2][htube[1]][2]..bound[2][htube[2]][2]]
    );
```

```
    h2:=Filtered(
        horizontals,
        x->bound[2][x][2] in
        [bound[2][htube[1]][3]..bound[2][htube[2]][3]]
    );
    htube:=Concatenation(htube,h1,h2);
    Add(htube,Length(htube),1);
    Add(bound[3],htube);
od;

postpatch:=Length(bound[3]);

loops:=loops+l1;

vtube:=[];
Add(vtube,verticals[1]);
x:=bound[2][verticals[1]][3];
cycle:=0;
loopless:=[];
for i in [2..Length(verticals)]
    do
    if bound[2][verticals[i]][2]=x
        then
        Add(vtube,verticals[i]);
        x:=bound[2][verticals[i]][3];
    else
        cycle:=cycle+1;
        if cycle=2
            then
            cycle:=0;
            Add(loopless,vtube);
            vtube:=[];
        fi;
        x:=bound[2][verticals[i]][3];
        Add(vtube,verticals[i]);
    fi;
od;
Add(loopless,vtube);
```

```
        for i in loopless
            do
            Add(i,Filtered(
                loops,
                y->bound[2][i[1]][2] in bound[2][y]{[2,3]})[1]
            );
            Add(i,Filtered(
                loops,
                y->bound[2][i[Length(i)-1]][3] in
                bound[2][y]{[2,3]})[2]
            );
            Add(i,Length(i),1);
            Add(bound[3],i);
        od;

        return bound;
    end;

    bound:=Patch(bound);

    Cap:=function(bound)
        local
            bottom, btm, top, tp,
            i, x, j, k;

        Add(bound[3],[2,l1-1,l1]); # the upper and lower
        Add(bound[3],[2,(2*l1)-1,2*l1]); # domes

        bottom:=[1..l2];
        btm:=[];
        for i in bound[3]{[prepatch+1..postpatch]}
            do
            x:=(Length(i)-3)/2;
            for j in [4..3+x]
                do
                for k in bottom
                    do
                    if
                    i[j] in bound[3][k]{[2..Length(bound[3][k])]} and
```

```
            i[j+x] in bound[3][k]{[2..Length(bound[3][k])]}
                then
                Add(btm,k);
            fi;
        od;
    od;
od;

bottom:=Difference(bottom,btm);
bottom:=Concatenation(
    bottom, # all base 2-cells without the overlap
    [prepatch+1..postpatch], # the 2-cells enclosing the tubes
    [Length(bound[3])-1] # the dome
);
Add(bottom,Length(bottom),1);

top:=[l2+1..2*l2];
tp:=[];
for i in bound[3]{[postpatch+1..Length(bound[3])-2]}
    do
    x:=(Length(i)-3)/2;
    for j in [2..1+x]
        do
        for k in top
            do
            if
            i[j] in bound[3][k]{[2..Length(bound[3][k])]} and
            i[j+x] in bound[3][k]{[2..Length(bound[3][k])]}
                then
                Add(tp,k);
            fi;
        od;
    od;
od;

top:=Difference(top,tp);
top:=Concatenation(
    top,
    [postpatch+1..Length(bound[3])-2],
```

```
                [Length(bound[3])]
            );
            Add(top,Length(top),1);

            Add(bound[4],bottom);
            Add(bound[4],top);

            return bound;
        end;

        return Cap(bound);
    end;

    P:=PuncturedTube(P);
    P:=RegularCWComplex(P);

    for i in [1..Length(P!.boundaries[2])-Length(loop_correction)] do
        Add(loop_correction,0);
    od;

    P!.loopCorrection:=loop_correction;

    return P;
end);
```

B.8 KnotComplementWithBoundary

```
InstallGlobalFunction(
    KnotComplementWithBoundary,
    function(arc)
    local
        comp, RegularCWKnot, knot, hcorrection,
        threshold, inclusion, iota, inv_mapping;

    comp:=KnotComplement(arc);

    RegularCWKnot:=function(arc)
```

```
local
    D, len, signless, HollowTubes, max,
    bigGrid, correction, threshold, hcorrection, TubeJoiner;

D:=arc;
len:=Length(D);
signless:=List(D,x->[AbsInt(x[1]),AbsInt(x[2])]);

HollowTubes:=function(D)
    local
        grid, i, IsIntersection,
        CornerConfiguration, bound,
        bigGrid, GridFill, j, tick, correction, hcorrection,
        hslice1, hslice2, l1, l2, l3,
        max, vslice1, vslice2, threshold;

    grid:=List([1..len],x->List([1..len],y->0));
    for i in [1..len]
        do
        grid[len-i+1][D[i][1]]:=1;
        grid[len-i+1][D[i][2]]:=1;
    od;

    IsIntersection:=function(i,j)

        if grid[i][j]=0
            then
            if 1 in grid[i]{[1..j]}
                then
                if 1 in grid[i]{[j..len]}
                    then
                    if 1 in List([1..i],x->grid[x][j])
                        then
                        if 1 in List([i..len],x->grid[x][j])
                            then
                            return true;
                        fi;
                    fi;
                fi;
```

```
        fi;
    fi;

return false;
end;

CornerConfiguration:=function(i,j);

    if grid[i][j]=1
        then
        if Size(Positions(grid[i]{[j..len]},1))=2
            then
            if Size(Positions(List([i..len],
                x->grid[x][j]),1))=2
                then # Corner type 1, i.e : __
                return 1; #                |
            elif Size(Positions(List([1..i],
                x->grid[x][j]),1))=2
                then # Corner type 3, i.e :
                return 3; #                |__
            fi;
        elif Size(Positions(grid[i]{[1..j]},1))=2
            then
            if Size(Positions(List([i..len],
                x->grid[x][j]),1))=2
                then # Corner type 2, i.e : __
                return 2; #                  |
            elif Size(Positions(List([1..i],
                x->grid[x][j]),1))=2
                then # Corner type 4, i.e :
                return 4; #                __|
            fi;
        fi;
    fi;

    return 0;
end;

bound:=[[],[],[],[],[]];
```

```
bigGrid:=List([1..2*len],x->List([1..2*len],y->0));

GridFill:=function(c,i,j);
    if c=1 or c=4
        then
        bigGrid[(2*i)-1][(2*j)-1]:='*';
        bigGrid[2*i][2*j]:='*';
    elif c=2 or c=3
        then
        bigGrid[(2*i)-1][2*j]:='*';
        bigGrid[2*i][(2*j)-1]:='*';
    fi;
end;

for i in [1..len]
    do
    for j in [1..len]
        do
        if IsIntersection(i,j)
            then
            bigGrid[(2*i)-1][(2*j)-1]:='*';
            bigGrid[(2*i)-1][2*j]:='*';
            bigGrid[2*i][(2*j)-1]:='*';
            bigGrid[2*i][2*j]:='*';
        elif grid[i][j]=1
            then
            GridFill(CornerConfiguration(i,j),i,j);
        fi;
    od;
od;

tick:=1;
for i in [1..2*len]
    do
    for j in [1..2*len]
        do
        if bigGrid[i][j]='*'
            then
            bigGrid[i][j]:=tick;
```

```
                    tick:=tick+1;
                fi;
            od;
        od;

# UPDATE: needed to account for configuration of corners at the end
# (i.e. when matching the loops of one layer to the other).
# There are sometimes disparities in the ordering on 1-cells from
# left-to-right vs. when ordering from top-to-bottom.
        correction:=[];
        hcorrection:=[];
        for i in [1..len] do
            for j in [1..len] do
                if CornerConfiguration(i,j)<>0 then
                    if CornerConfiguration(i,j) in [1,4] then
                        Add(correction,1);
                        Add(correction,-1);
                        Add(hcorrection,2);
                        Add(hcorrection,1);
                    else
                        Add(correction,0);
                        Add(correction,0);
                        Add(hcorrection,1);
                        Add(hcorrection,2);
                    fi;
                fi;
            od;
        od;

        ### add the 0, 1 & 2-cells ###
        ########## to bound ##########
        for i in [1..2*Maximum(bigGrid[Length(bigGrid)])] do
            Add(bound[1],[1,0]);
        od;

        for i in [1..len] do # add the 'horizontal' 2-cells
            hslice1:=Filtered(bigGrid[2*i-1],x->x<>0);
            hslice2:=Filtered(bigGrid[2*i],x->x<>0);
            l2:=Length(bound[2]);
```

```
for j in [1..Length(hslice1)-1] do
    Add(
        bound[2],
        [2,hslice1[j],hslice2[j]]
    );
    if j=1 then
        Add(
            bound[2],
            [2,hslice1[j],hslice2[j]]
        );
    fi;
    if j<>1 then
        l1:=Length(bound[2]);
        Add(
            bound[3],
            [4,l1-3,l1-2,l1-1,l1]
        );
    fi;
    Add(
        bound[2],
        Concatenation([2],hslice1{[j,j+1]})
    );
    Add(
        bound[2],
        Concatenation([2],hslice2{[j,j+1]})
    );
    if j=Length(hslice1)-1 then
        Add(
            bound[2],
            [2,hslice1[j+1],hslice2[j+1]]
        );
        l1:=Length(bound[2]);
        Add(
            bound[2],
            [2,hslice1[j+1],hslice2[j+1]]
        );
        Add(
            bound[3],
            [4,l1-3,l1-2,l1-1,l1]
```

```
            );
        fi;
    od;
    l3:=Concatenation(
        [l2+1],
        Filtered(
            [l2+3..Length(bound[2])-2],
            x->AbsInt(bound[2][x][2]-bound[2][x][3])=1
        ),
        [Length(bound[2])]
    );
    Add(l3,Length(l3),1);
    Add(bound[3],l3);
od;

max:=Maximum(bigGrid[Length(bigGrid)]);
for i in [1..2*len] do
    for j in [1..2*len] do
        if bigGrid[i][j]<>0 then
            bigGrid[i][j]:=bigGrid[i][j]+max;
        fi;
    od;
od;

for i in [1..len] do # add the 'vertical' 2-cells
    vslice1:=Filtered(
        List([1..2*len],
        x->bigGrid[x][2*i-1]),x->x<>0
    );
    vslice2:=Filtered(
        List([1..2*len],
        x->bigGrid[x][2*i]),x->x<>0
    );
    l2:=Length(bound[2]);
    for j in [1..Length(vslice1)-1] do
        Add(
            bound[2],
            Concatenation(
                [2],
```

```
            Set(
                [vslice1[j],
                vslice2[j]]
            )
        )
    );
    if j=1 then
        Add(
            bound[2],
            Concatenation(
                [2],
                Set(
                    [vslice1[j],
                    vslice2[j]]
                )
            )
        );
    fi;
    if j<>1 then
        l1:=Length(bound[2]);
        Add(
            bound[3],
            [4,l1-3,l1-2,l1-1,l1]
        );
    fi;
    Add(
        bound[2],
        Concatenation([2],Set(vslice1{[j,j+1]}))
    );
    Add(
        bound[2],
        Concatenation([2],Set(vslice2{[j,j+1]}))
    );
    if j=Length(vslice1)-1 then
        Add(
            bound[2],
            Concatenation(
                [2],
                Set(
```

```
                            [vslice1[j+1],
                            vslice2[j+1]]
                        )
                    )
                );
                l1:=Length(bound[2]);
                Add(
                    bound[2],
                    Concatenation(
                        [2],
                        Set(
                            [vslice1[j+1],
                            vslice2[j+1]]
                        )
                    )
                );
                Add(
                    bound[3],
                    [4,l1-3,l1-2,l1-1,l1]
                );
            fi;
        od;
        l3:=Concatenation(
            [l2+1],
            Filtered(
                [l2+3..Length(bound[2])-2],
                x-> not
        (bound[2][x][2] in vslice1 and bound[2][x][3] in vslice2)
                and
                not
        (bound[2][x][2] in vslice2 and bound[2][x][3] in vslice1)
            ),
            [Length(bound[2])]
        );
        Add(l3,Length(l3),1);
        Add(bound[3],l3);
    od;

    threshold:=Length(bound[2])/2;
```

```
    ##############################

    return [max,bound,bigGrid,correction,threshold,hcorrection];
end;

D:=HollowTubes(D);
max:=D[1];
bigGrid:=D[3];
correction:=D[4];
threshold:=D[5];
hcorrection:=D[6];
D:=D[2];

TubeJoiner:=function(D)
    local
        loops, size, hloops, vloops,
        i, l;

    loops:=Filtered(
        [1..Length(D[2])],
        x->Length(Positions(D[2],D[2][x]))=2
    );
    size:=Length(loops)/2;
    hloops:=loops{[1..size]};
    vloops:=List(
        [1..size],
        x->Position(D[2],D[2][hloops[x]]+[0,max,max])
    );
    for i in [1..size] do
        if i mod 2 = 0 then
            vloops[i]:=vloops[i]+1;
        fi;
    od;
    vloops:=vloops+correction;

    for i in [1..size/2] do
        Add(D[2],[2,D[2][hloops[2*i]][2],D[2][vloops[2*i]][2]]);
        Add(D[2],[2,D[2][hloops[2*i]][3],D[2][vloops[2*i]][3]]);
        l:=Length(D[2]);
```

```
            Add(D[3],[4,hloops[2*i-1],vloops[2*i-1],l-1,l]);
            Add(D[3],[4,hloops[2*i],vloops[2*i],l-1,l]);
        od;

        return D;
    end;

    D:=TubeJoiner(D);
    D:=RegularCWComplex(D);
    D!.grid:=bigGrid;
    D!.arcPresentation:=arc;

    return [D,hcorrection,threshold];
end;

knot:=RegularCWKnot(arc);
hcorrection:=knot[2];
threshold:=knot[3];
knot:=knot[1];

inclusion:=function(bound)
    local
        bound1, bound2, len,
        1c1, 1c2, 2c2, HorizontalIndex, inc;

    bound1:=bound[1];
    bound2:=bound[2];

    len:=Length(bound1[1])/2;

    1c1:=bound1[2]*1;
    1c1:=List(
        1c1,
        x->Concatenation(
            [2],
            [x[2]+1+2*Int((x[2]-1)/len),
            x[3]+1+2*Int((x[3]-1)/len)]
        )
```

```
);

1c2:=bound2[2]*1;
1c2:=List(
    1c2,
    x->Concatenation(
        [2],
        Set(
            [x[2],x[3]]
        )
    )
);

2c2:=List(
    bound2[3],
    x->Concatenation([x[1]],Set(x{[2..Length(x)]}))
);

HorizontalIndex:=function(n)

    return hcorrection[
        Position(
            Filtered(
                [1..threshold],
                x->Length(
                    Positions(
                        bound1[2],
                        bound1[2][x]
                    )
                )=2
            ),n
        )
    ];
end;

inc:=function(n,k)
    local
        ind, 2cell;
```

```
            if n=0 then
                return k+1+2*Int((k-1)/len);
            elif n=1 then
                ind:=1;
                if k in [1..threshold] then
                    if k>1 and 1c1[k-1]=1c1[k] then
                        ind:=HorizontalIndex(k);
                    elif 1c1[k]=1c1[k+1] then
                        ind:=HorizontalIndex(k);
                    fi;
                elif k>threshold then
                    if 1c1[k-1]=1c1[k] then
                        ind:=2;
                    elif k<Length(1c1) and 1c1[k]=1c1[k+1] then
                        ind:=1;
                    fi;
                fi;
                return Positions(1c2,1c1[k])[ind];
            elif n=2 then
                2cell:=List(
                    bound1[3][k]{[2..Length(bound1[3][k])]},
                    x->inc(1,x)
                );
                2cell:=Concatenation([Length(2cell)],Set(2cell));
                return Position(2c2,2cell);
            else
                return fail;
            fi;
        end;

        return inc;
    end;

    iota:=inclusion([knot!.boundaries,comp!.boundaries]);

    inv_mapping:=[[],[],[]];
    inv_mapping[1]:=List([1..knot!.nrCells(0)],x->iota(0,x));
    inv_mapping[2]:=List([1..knot!.nrCells(1)],x->iota(1,x));
    inv_mapping[3]:=List([1..knot!.nrCells(2)],x->iota(2,x));
```

```
    return Objectify(
        HapRegularCWMap,
        rec(
            source:=knot,
            target:=comp,
            mapping:=iota,
            properties:=[
                ["image",inv_mapping]
            ]
        )
    );
end);
```

B.9 ArcPresentationToKnottedOneComplex

```
InstallGlobalFunction(
ArcPresentationToKnottedOneComplex,
function(arc)
    local
        gn, grid, i, kbnd, bnd, map, imap,
        IsIntersection, ints, ext_ints, j, 0c,
        B2Decomposition, B3Decomposition, embed;

    gn:=Length(arc); # the grid number

    grid:=List([1..5*gn],x->[1..5*gn]*0); # form a (3*gn) x gn
    for i in [0..gn-1] do # matrix from the arc presentation
        grid[5*(gn-i)-2][5*arc[i+1][1]-2]:=1;
        grid[5*(gn-i)-2][5*arc[i+1][2]-2]:=1;
    od;
    grid:=FrameArray(grid);

    kbnd:=List([1..3],x->[]); # boundary list of the knot
    bnd:=List([1..5],x->[]); # boundary list of the complement of the knot
    map:=List([1..2],x->[]); # inclusion map from kbnd to bnd
    imap:=List([1..4],x->[]); # inverse image of the above inclusion map
```

```
IsIntersection:=function(i,j) # finds where crossings occur in grid
    if grid[i][j]=0 then
        if 1 in grid[i]{[1..j]} then
            if 1 in grid[i]{[j..5*gn+2]} then
                if 1 in List([1..i],x->grid[x][j]) then
                    if 1 in List([i..5*gn+2],x->grid[x][j]) then
                        return true;
                    fi;
                fi;
            fi;
        fi;
    fi;

    return false;
end;

ints:=[]; # records the coordinates of each crossing
ext_ints:=[]; # records those 0-c which arise in intersection points
for i in [1..5*gn+2] do
    for j in [1..5*gn+2] do
        if IsIntersection(i,j) then
            Add(ext_ints,[i-1,j]); Add(ext_ints,[i+1,j]);
            Add(ints,[i,j]);
            grid[i-1][j]:='*';
            grid[i][j]:='*';
            grid[i+1][j]:='*';
        fi;
    od;
od;

0c:=4; # label the entries of grid so that it models the 0-skeleton
for i in [1..5*gn+2] do
    for j in [1..5*gn+2] do
        if grid[i][j]<>0 then
            grid[i][j]:=0c;
            0c:=0c+1;
        fi;
    od;
```

```
    od;
    0c:=0c+2;

    ints:=List(ints,x->grid[x[1],x[2]]);

    B2Decomposition:=function()
# takes what we have so far and uses it to form a regular CW-decomposition
# of the 2-ball with the appropriate inclusion map
        local i, j, hslice, vslice, 2SkeletonOfDisk, DuplicateDisk;

        kbnd[1]:=List([1..0c-6],x->[1,0]);
        bnd[1]:=List([1..0c],x->[1,0]);
        map[1]:=[1..0c-6]+3;
        imap[1]:=Concatenation([0,0,0],map[1]-3,[0,0,0]);

        Add(bnd[2],[2,1,2]); Add(bnd[2],[2,1,3]); Add(bnd[2],[2,2,0c-2]);
        Add(bnd[2],[2,3,0c-1]); Add(bnd[2],[2,0c-2,0c]);
        Add(bnd[2],[2,0c-1,0c]);
        Add(bnd[2],[2,3,4]); Add(bnd[2],[2,0c-3,0c-2]);
        # add some 1-cells to just the complement to stay regular
        # these act as a frame to the knot

        for i in [1..8] do
            Add(imap[2],0);
        od;

        for i in [1..5*gn] do # add the horizontal arcs of the knot first
            hslice:=[];
            for j in [1..5*gn] do
                if grid[i][j]<>0 and not [i,j] in ext_ints then
                    Add(hslice,grid[i][j]);
                fi;
            od;
            for j in [1..Length(hslice)-1] do
                Add(kbnd[2],[2,hslice[j]-3,hslice[j+1]-3]);
                Add(bnd[2],[2,hslice[j],hslice[j+1]]);
                Add(map[2],Length(bnd[2]));
                Add(imap[2],Length(kbnd[2]));
            od;
```

```
od;
for j in [1..5*gn] do # now add the vertical arcs
    vslice:=[];
    for i in [1..5*gn] do
        if grid[i][j]<>0 then
            Add(vslice,grid[i][j]);
        fi;
    od;
    for i in [1..Length(vslice)-1] do
        Add(bnd[2],[2,vslice[i],vslice[i+1]]);
        if not(vslice[i] in ints or vslice[i+1] in ints) then
            Add(kbnd[2],[2,vslice[i]-3,vslice[i+1]-3]);
            Add(map[2],Length(bnd[2]));
            Add(imap[2],Length(kbnd[2]));
        else
            Add(imap[2],0);
        fi;
    od;
od;

2SkeletonOfDisk:=function(bnd)
    local
        ori, i, j, cell, top, rgt, btm, lft,
        Clockwise, path, FaceTrace;

    grid[1][1]:=1; grid[1][5*gn+2]:=2; grid[4][1]:=3;
    grid[5*gn-1][5*gn+2]:=0c-2; grid[5*gn+2][1]:=0c-1;
    grid[5*gn+2][5*gn+2]:=0c;

    ori:=List([1..0c],x->[1..4]*0);
    # each 0-cell will have the 0-cells N/E/S/W of it
    # listed in that order

    for i in [1..5*gn+2] do
        for j in [1..5*gn+2] do
            cell:=grid[i][j];
            if cell<>0 then
                top:=List([1..i-1],x->grid[x][j]);
                top:=Filtered(top,x->x<>0);
```

```
if top<>[] then
    ori[cell][1]:=Position
    (
        bnd[2],
        Concatenation(
            [2],
            Set([cell,top[Length(top)]])
        )
    );
fi;
rgt:=grid[i]{[j+1..5*gn+2]};
rgt:=Filtered(rgt,x->x<>0);
if rgt<>[] then
    ori[cell][2]:=Position
    (
        bnd[2],
        Concatenation([2],Set([cell,rgt[1]]))
    );
fi;
btm:=List([i+1..5*gn+2],x->grid[x][j]);
btm:=Filtered(btm,x->x<>0);
if btm<>[] then
    ori[cell][3]:=Position(
        bnd[2],
        Concatenation([2],Set([cell,btm[1]]))
    );
fi;
lft:=grid[i]{[1..j-1]};
lft:=Filtered(lft,x->x<>0);
if lft<>[] then
    ori[cell][4]:=Position(
        bnd[2],
        Concatenation(
            [2],
            Set([cell,lft[Length(lft)]])
        )
    );
fi;
if [i,j] in ext_ints then # 0-cells in ext_ints
```

```
                        ori[cell][2]:=0; # never have edges from the
                        ori[cell][4]:=0; # left or right of them
                    fi;
                fi;
            od;
        od;


##########################################################################
# repurposed from KnotComplement and KnotComplementWithBoundary
        FaceTrace:=function(path)
            local
                unselected, sourceORtarget, x, ClockwiseTurn,
                2cell, sORt, dir, e1, e0, i, bool;

            unselected:=Concatenation
            (
                [1..Length(bnd[2])],
                [7..Length(bnd[2])]
                # the first 6 1-cells occur just once
            );

            ClockwiseTurn:=function(p,e)
                local f;

                f:=(Position(p,e) mod 4)+1;
                while p[f]=0 do
                    f:=(f mod 4)+1;
                od;

                return p[f];
            end;

            bool:=false;
            sourceORtarget:=List([1..Length(bnd[2])],y->[3,2]);
            x:=1;
            while unselected<>[] do
                while (not x in unselected) and (not e1 in unselected)
                    do
```

```
                    x:=x+1;
                od;
                2cell:=[x];
                sORt:=sourceORtarget[x][Length(sourceORtarget[x])];
                Unbind(sourceORtarget[x][Length(sourceORtarget[x])]);

                dir:=path[bnd[2][x][sORt]];
                e0:=bnd[2][x][sORt];
                e1:=ClockwiseTurn(dir,x);
                while e1<>x do
                    Add(2cell,e1);
                    e0:=Filtered(bnd[2][e1]{[2,3]},y->y<>e0)[1];
                    dir:=path[e0];
                    e1:=ClockwiseTurn(dir,e1);
                od;
                Add(2cell,Length(2cell),1);
                if (not Set(2cell) in List(bnd[3],x->Set(x))) then
                    for i in Filtered(
                        2cell{[2..Length(2cell)]},
                        y->y in unselected
                    ) do
                        Unbind(unselected[Position(unselected,i)]);
                    od;
                    if not bool then # to save some checks
                        if Set(2cell)=[1,2,3,4,5,6] then
                            bool:=true;
                        else
                            Add(bnd[3],2cell);
                            Add(imap[3],0);
                        fi;
                    else
                        Add(bnd[3],2cell);
                        Add(imap[3],0);
                    fi;
                fi;
            od;
##############################################################################
            bnd[3]:=List(
                bnd[3],
```

```
                x->Concatenation(
                    [x[1]],
                    Set(x{[2..Length(x)]})
                )
            ); # order the 2-cells nicely
        end;
        FaceTrace(ori);
    end;
    2SkeletonOfDisk(bnd);
end;
B2Decomposition();

B3Decomposition:=function()
    local
        k0, b0, k1, b1, b2, DuplicateDisk, b22, CrossI, 3c;

    k0:=Length(kbnd[1]); b0:=Length(bnd[1]);
    k1:=Length(kbnd[2]); b1:=Length(bnd[2]);
    b2:=Length(bnd[3]);

    DuplicateDisk:=function() # make a duplicate of everything
        local i, n, mult;

        for i in [1..Length(ext_ints)/2] do
            Add(kbnd[1],[1,0]);
            Add(kbnd[1],[1,0]);
            Add(kbnd[1],[1,0]);
        od;

        bnd[1]:=Concatenation(bnd[1],bnd[1]);

        imap[1]:=Concatenation(imap[1],imap[1]+Length(kbnd[1])-k0);
        for i in [b0+1..2*b0] do
            if imap[1][i]=Length(kbnd[1])-k0 then
                imap[1][i]:=0;
            fi;
        od;

        n:=k0+1;
```

```
mult:=1;
for i in [1..b1] do
    Add(bnd[2],bnd[2][i]+[0,b0,b0]);
    if i>8 and not (bnd[2][i]-[0,3,3] in kbnd[2]) then
        Add(
            kbnd[2],
            Concatenation
            (
                [2],
                [n,n+1]
            )
        );
        n:=n+1+Int(mult/2);

        Add(map[1],bnd[2][i][2]+b0);
        if Int(mult/2)=1 then
            mult:=1;
            Add(map[1],bnd[2][i][3]+b0);
        else
            mult:=2;
        fi;

        Add(map[2],Length(bnd[2]));
        Add(imap[2],Length(map[2]));
    else
        Add(imap[2],0);
    fi;
od;

for i in [1..b2] do
    Add(
        bnd[3],
        Concatenation(
            [bnd[3][i][1]],
            bnd[3][i]{[2..Length(bnd[3][i])]}+b1
        )
    );
    Add(imap[3],0);
od;
```

```
    return Length(bnd[3]);
end;
b22:=DuplicateDisk();

CrossI:=function() # connect the two disks
    local l, i, j, n, bool, 3cell;
    # each n-cell gives rise to an (n+1)-cell

    l:=[];
    for i in [1..5*gn+2] do
        for j in [1..5*gn+2] do
            if grid[i][j]<>0 then
                Add(l,grid[i][j]);
            fi;
        od;
    od;

    n:=k0+1;
    bool:=false;
    for i in l do
        Add(bnd[2],[2,i,i+b0]);
        if i in List(ext_ints,x->grid[x[1]][x[2]]) then
            Add(kbnd[2],[2,i-3,n]);
            if not bool then
                n:=n+2; bool:=true;
            else
                n:=n+1; bool:=false;
            fi;
            Add(map[2],Length(bnd[2]));
            Add(imap[2],Length(map[2]));
        else
            Add(imap[2],0);
        fi;
    od;

    for i in [1..b1] do
        Add(
            bnd[3],
```

```
                [
                    4,
                    i,
                    i+b1,
                    Position(
                        bnd[2],
                        [
                            2,
                            bnd[2][i][2],
                            bnd[2][i+b1][2]
                        ]
                    ),
                    Position(
                        bnd[2],
                        [
                            2,
                            bnd[2][i][3],
                            bnd[2][i+b1][3]
                        ]
                    )
                ]
            );
            Add(imap[4],[i,Length(bnd[3])]); # not exactly the inverse
        od; # image any more, but this list is used directly below

        for i in [1..b2] do
            3cell:=Concatenation(
                [
                    i,
                    i+b2
                ],
                List(
                    bnd[3][i]{[2..Length(bnd[3][i])]},
                    x->imap[4]
                        [
                            Position(
                                List(imap[4],y->y[1]),
                                x
                            )
```

```
                            ][2]
                    )
                );
                Add(3cell,Length(3cell),1);
                Add(bnd[4],3cell);
            od;
        end;
        CrossI();

        # add a cap to B3 /// not sure if necessary
        Add(bnd[3],[6,1,2,3,4,5,6]);
        Add(bnd[3],[6,b1+1,b1+2,b1+3,b1+4,b1+5,b1+6]);
        3c:=Concatenation([Length(bnd[3])-1],[1..b2]);
        Add(3c,Length(3c),1);
        Add(bnd[4],3c);
        3c:=Concatenation([Length(bnd[3])],[b2+1..b22]);
        Add(3c,Length(3c),1);
        Add(bnd[4],3c);

    end;
    B3Decomposition();

    embed:={n,k}->map[n+1][k];

    #return [map,grid,kbnd,bnd];
    return Objectify(
        HapRegularCWMap,
        rec(
            source:=RegularCWComplex(kbnd),
            target:=RegularCWComplex(bnd),
            mapping:=embed,
grid:=grid
        )
    );
end);
```

B.10 SphericalKnotComplement

```
InstallGlobalFunction(
    SphericalKnotComplement,
    function(arg)
    local
        arc, K, bm, lst, paths, Smap, S, B, bnds;

    arc:=arg[1];
    if Length(arg)=1 then
        K:=KnotComplement(arc);
    else
        Print("Using alternative method. \n");
        K:=ArcPresentationToKnottedOneComplex(arc);
        K:=RegularCWComplexComplement(K);
    fi;
    K:=SimplifiedComplex(K);
    bm:=BoundaryMap(K);
    lst:=RegularCWMapToCWSubcomplex(bm);
    paths:=PathComponentsCWSubcomplex(lst);
    paths:=List(paths,CWSubcomplexToRegularCWMap);

    for Smap in paths do
        if Homology(Source(Smap),1)=[] then
            break;
        fi;
    od;

    # Smap is a map from the sphere to the
    # boundary of the knot complement K

    S:=Source(Smap);
    B:=List([1..S!.nrCells(2)], i->Smap!.mapping(2,i));
    B:=Concatenation([Length(B)],B);

    bnds:=1*K!.boundaries;
    Add(bnds[4],B);
```

```
    return SimplifiedComplex(RegularCWComplex(bnds));
end);
```

B.11 ArcDiagramToTubularSurface

```
InstallGlobalFunction(
    ArcDiagramToTubularSurface,
    function(arc)
    local
        prs, crs, clr, grd, i, IsIntersection,
        CornerConfiguration, GRD, crossings, j,
        k, nr0cells, bnd, sub, hbars, hslice, cell,
        int, max, vbars, vslice, loops1, loops2,
        unchosen, neighbours, Clockwise,
        path, unselectedEdges, sourceORtarget,
        faceloops, x, ClockwiseTurn, 2cell, sORt,
        ori, e1, e0, loops, present_loops, vertices,
        check, l0, l1, l2, l1_, l2_, IsEdgeInDuplicate,
        copy1, hbars2, vbars2, copy2, 3cell, colour, lcap, reg,
        closure, ucap, l1__, l2__, path_comp, pipes, HorizontalOrVertical,
        l, AboveBelow0Cell, IntersectingCylinders, pos, colour_;

    if IsList(arc[1][1]) then
        prs:=arc[1]*1;
        crs:=arc[2]*1;
#                -1 |     +1 |      0 |
# crossing types: --|-- or ----- or --+--
#                   |        |        |
        if Length(arc)=3 then
            clr:=arc[3]*1;
# colours: 1; bgb, 2; bgr, 3; rgb, 4; rgr
        fi;
    else
        prs:=arc*1;
    fi;

# (i) the 0-skeleton of the disk
```

```
##########################################################################
    grd:=List([1..Length(prs)],x->[1..Length(prs)]*0);
    for i in [0..Length(prs)-1] do
        grd[Length(prs)-i][prs[i+1][1]]:=1;
        grd[Length(prs)-i][prs[i+1][2]]:=1;
    od;

    IsIntersection:=function(i,j)
        if grd[i][j]=0 and
            1 in grd[i]{[1..j]} and
                1 in grd[i]{[j..Length(prs)]} and
                    1 in List([1..i],x->grd[x][j]) and
                        1 in List([i..Length(prs)],x->grd[x][j]) then
            return true;
        fi;
        return false;
    end;
    CornerConfiguration:=function(i,j);
        if grd[i][j]=1 then
            if Size(Positions(grd[i]{[j..Length(prs)]},1))=2 then
                if Size(Positions(List([i..Length(prs)],
                x->grd[x][j]),1))=2 then
                        # Corner type 1, i.e :  __
                    return 1; #                |
                elif Size(Positions(List([1..i],x->grd[x][j]),1))=2 then
                        # Corner type 3, i.e :
                    return 3; #                |__
                fi;
            elif Size(Positions(grd[i]{[1..j]},1))=2 then
                if Size(Positions(List([i..Length(prs)],
                x->grd[x][j]),1))=2 then
                        # Corner type 2, i.e :  __
                    return 2; #                  |
                elif Size(Positions(List([1..i],
                x->grd[x][j]),1))=2 then
                        # Corner type 4, i.e :
                    return 4; #                __|
                fi;
            fi;
```

```
        fi;
        return 0;
    end;

    GRD:=List([1..4*Length(prs)],x->[1..4*Length(prs)]*0);
    crossings:=[];
# quadruple the size of grd to allow for the 0-skeleton
# to be displayed nicely without overlap in an array
    for i in [1..Length(prs)] do
        for j in [1..Length(prs)] do
            if CornerConfiguration(i,j) in [1,4] then
                GRD[4*i-3][4*j-3]:=1;
                GRD[4*i][4*j]:=1;
            elif CornerConfiguration(i,j) in [2,3] then
                GRD[4*i-3][4*j]:=1;
                GRD[4*i][4*j-3]:=1;
            elif IsIntersection(i,j) then
                for k in [0,3] do
                    GRD[4*i-3][4*j-3+k]:=1;
                    GRD[4*i][4*j-3+k]:=1;
                    Add(crossings,[4*i-3,4*j-3+k]);
                    Add(crossings,[4*i,4*j-3+k]);
                od;
            fi;
        od;
    od;
# label the 0-cells row by row
    nr0cells:=2;
    for i in [1..4*Length(prs)] do
        for j in [1..4*Length(prs)] do
            if GRD[i][j]=1 then
                GRD[i][j]:=nr0cells;
                nr0cells:=nr0cells+1;
            fi;
        od;
    od;
    crossings:=List(crossings,x->GRD[x[1]][x[2]]);
    crossings:=List(
        [1..Length(crossings)/4],
```

```
        x->List([1..4]+4*x-4,y->crossings[y])
    );
    GRD:=FrameArray(GRD);
    GRD[1][1]:=1;
    GRD[4*Length(prs)+2][4*Length(prs)+2]:=nr0cells;

    bnd:=List([1..5],x->[]); # eventual boundary list of the 3-ball
    # containing
    sub:=List([[],[],[]]); # the boundary of a knotted surface as a
    # subcomplex
    bnd[1]:=List([1..nr0cells],x->[1,0]);
    sub[1]:=[2..nr0cells-1];
    if IsBound(crs) then
        for i in [1..Length(crs)] do
            if crs[i]=0 then
                sub[1]:=Difference(sub[1],crossings[i]);
            fi;
        od;
    fi;
##############################################################################

# (ii) the 1-skeleton of the disk
##############################################################################
    # add the horizontal 1-cells
    hbars:=[];
    for i in [2..Length(GRD)-1] do
        hslice:=Filtered(GRD[i],x->x<>0);
        if hslice<>hslice*0 then
            Add(hbars,hslice);
        fi;
        for j in [1..Length(hslice)-1] do
            cell:=[2,hslice[j],hslice[j+1]];
            Add(bnd[2],cell);
            int:=List(crossings,x->Length(Intersection(cell{[2,3]},x)));
            max:=PositionMaximum(int);
            if IsBound(crs) then
                if int[max]=1 and
                    crs[max]=-1 and
                        not Length(bnd[2]) in sub[2] then
```

```
                        Add(sub[2],Length(bnd[2]));
            elif int[max]=2 and
                crs[max]<>0 and
                    not Length(bnd[2]) in sub[2] then
                        Add(sub[2],Length(bnd[2]));
            fi;
        else
            if max<>fail then
                if int[max]=2 then #  horizontal 1-cells default to
                    Add(sub[2],Length(bnd[2])); # the top so they're
                    # not all included here
                fi;
            fi;
        fi;
    od;
od;
hbars:=List(
    [1..Length(hbars)/2],x->Concatenation(hbars[2*x-1],
    hbars[2*x])
);

# add the vertical 1-cells
vbars:=[];
for i in TransposedMat(GRD){[2..Length(GRD)-1]} do
    vslice:=Filtered(i,x->x<>0);
    if vslice<>vslice*0 then
        Add(vbars,vslice);
    fi;
    for j in [1..Length(vslice)-1] do
        cell:=[2,vslice[j],vslice[j+1]];
        Add(bnd[2],cell);
        int:=List(crossings,x->Length(Intersection(cell{[2,3]},x)));
        max:=PositionMaximum(int);
        if IsBound(crs) then
            if int[max]=1 then
                if crs[max]=1 and
                    not Length(bnd[2]) in sub[2] then
                        Add(sub[2],Length(bnd[2]));
                fi;
```

```
                elif int[max]=2 then
                    if crs[max]<>0 and
                        not Length(bnd[2]) in sub[2] then
                            Add(sub[2],Length(bnd[2]));
                    fi;
                else
                    Add(sub[2],Length(bnd[2]));
                fi;
            else
                Add(sub[2],Length(bnd[2]));
                # vertical 1-cells default to the bottom
            fi;
        od;
    od;
    vbars:=List(
        [1..Length(vbars)/2],x->Concatenation(vbars[2*x-1],vbars[2*x])
    );

    # add the loops
    for i in [2,6..Length(GRD)-4] do
        loops1:=Filtered(GRD[i],x->x<>0);
        loops2:=Filtered(GRD[i+3],x->x<>0);
        for j in [1,2] do
            Add(bnd[2],[2,loops1[1],loops2[1]]);
            Add(sub[2],Length(bnd[2])); # loops always in subcomplex...
            Add(bnd[2],[2,loops1[Length(loops1)],loops2[Length(loops2)]]);
            Add(sub[2],Length(bnd[2]));
        od;
    od;

    # add the remaining four 1-cells to keep things regular
    Add(bnd[2],[2,1,2]); Add(bnd[2],[2,nr0cells-1,nr0cells]);
    Add(bnd[2],[2,1,nr0cells]); Add(bnd[2],[2,1,nr0cells]);
##############################################################################

# (iii) the 2-skeleton of the disk
##############################################################################
    unchosen:=List(bnd[2],x->x{[2,3]});
    neighbours:=List(bnd[1],x->[]);
```

```
for i in [1..Length(bnd[1])] do
    for j in [1..Length(unchosen)] do
        if i in unchosen[j] then
            Add(neighbours[i],j);
        fi;
    od;
od;

Clockwise:=function(neighbours)
    local
        oriented, first0, last0,
        i, j, x, k, l, posi, posx;

    oriented:=List(neighbours,x->List([1..12],y->"pass"));
    first0:=SortedList(neighbours[1]);
    last0:=SortedList(neighbours[Length(neighbours)]);

    oriented[1][7]:=first0[1];
    oriented[1][6]:=first0[3];
    oriented[1][8]:=first0[2];
    oriented[Length(oriented)][1]:=last0[1];
    oriented[Length(oriented)][2]:=last0[3];
    oriented[Length(oriented)][12]:=last0[2];

    for i in [2..Length(neighbours)-1] do
        for j in [1..Length(neighbours[i])] do
            x:=bnd[2][neighbours[i][j]];
            x:=Filtered(x{[2,3]},y->y<>i)[1];
            for k in [1..Length(GRD)] do
                for l in [1..Length(GRD[1])] do
                    if i=GRD[k][l] then
                        posi:=[k,l];
                    fi;
                    if x=GRD[k][l]then
                        posx:=[k,l];
                    fi;
                od;
            od;
```

```
# _\\|//_
#  //|\\
if posi[1]>posx[1] then
    if posi[2]=posx[2] then
        oriented[i][1]:=neighbours[i][j];
    elif posi[2]<posx[2] then
        if oriented[i][2]="pass" then
            oriented[i][2]:=neighbours[i][j];
        else
            oriented[i][3]:=neighbours[i][j];
        fi;
    elif posi[2]>posx[2] then
        if oriented[i][12]="pass" then
            oriented[i][12]:=neighbours[i][j];
        else
            oriented[i][11]:=neighbours[i][j];
        fi;
    fi;
elif posi[1]=posx[1] then
    if posi[2]<posx[2] then
        oriented[i][4]:=neighbours[i][j];
    elif posi[2]>posx[2] then
        oriented[i][10]:=neighbours[i][j];
    fi;
elif posi[1]<posx[1] then
    if posi[2]=posx[2] then
        oriented[i][7]:=neighbours[i][j];
    elif posi[2]<posx[2] then
        if oriented[i][5]="pass" then
            oriented[i][5]:=neighbours[i][j];
        else
            oriented[i][6]:=neighbours[i][j];
        fi;
    elif posi[2]>posx[2] then
        if oriented[i][9]="pass" then
            oriented[i][9]:=neighbours[i][j];
        else
            oriented[i][8]:=neighbours[i][j];
        fi;
```

```
                fi;
            fi;
        od;
    od;

    return oriented;
end;

path:=Clockwise(neighbours);

unselectedEdges:=List([1..Length(bnd[2])-2]);
unselectedEdges:=Concatenation(unselectedEdges,unselectedEdges);
Add(unselectedEdges,Length(bnd[2])-1);
Add(unselectedEdges,Length(bnd[2]));

ClockwiseTurn:=function(p,e)
    local f;
    f:=(Position(p,e) mod 12)+1;
    while p[f]="pass" do
        f:=(f mod 12)+1;
    od;
    return p[f];
end;

sourceORtarget:=List([1..Length(bnd[2])],y->[3,2]);
x:=1;
while unselectedEdges<>[] do
    while (not x in unselectedEdges) and (not e1 in unselectedEdges)
        do
        x:=x+1;
    od;
    2cell:=[x];
    sORt:=sourceORtarget[x][Length(sourceORtarget[x])];
    Unbind(sourceORtarget[x][Length(sourceORtarget[x])]);

    ori:=path[bnd[2][x][sORt]];
    e0:=bnd[2][x][sORt];
    e1:=ClockwiseTurn(ori,x);
    while e1<>x do
```

```
            Add(2cell,e1);
            e0:=Filtered(bnd[2][e1]{[2,3]},y->y<>e0)[1];
            ori:=path[e0];
            e1:=ClockwiseTurn(ori,e1);
        od;
        Add(2cell,Length(2cell),1);
        if not Set(2cell) in List(bnd[3],Set) then
            for i in
            Filtered(2cell{[2..Length(2cell)]},y->y in unselectedEdges) do
                Unbind(unselectedEdges[Position(unselectedEdges,i)]);
            od;
            Add(bnd[3],2cell);
           if Length(Intersection(2cell{[2..2cell[1]+1]},sub[2]))=2cell[1]
                and
                2cell[1]<>2 then
                    Add(sub[3],Length(bnd[3]));
            fi;
        fi;
    od;
    bnd[3]:=List(bnd[3],x->Concatenation([x[1]],Set(x{[2..Length(x)]})));
##############################################################################

# (iv) the direct product of the disk with [0,1]
##############################################################################
# make a duplicate of the disk
    l0:=Length(bnd[1]);
    l1:=Length(bnd[2]); l1_:=Length(sub[2]);
    l2:=Length(bnd[3]); l2_:=Length(sub[3]);

    IsEdgeInDuplicate:=function(k)
        local x;

        if Length(Positions(bnd[2],bnd[2][k]))=2 and
            bnd[2][k]<>[2,l0+1,2*l0] then
                return true;
        elif not Position(bnd[2],bnd[2][k]-[0,l0,l0]) in sub[2] and
            not bnd[2][k]-[0,l0,l0] in [[2,1,2],[2,1,l0],[2,l0-1,l0]] then
                return true;
        else
```

```
            x:=List(
                List(crossings,y->y+l0),
                z->Length(Intersection(z,bnd[2][k]{[2,3]}))=2
            );
            if true in x then
                if IsBound(crs) then
                    if crs[Position(x,true)] in [1,-1] then
                        return true;
                    fi;
                else
                    return true;
                fi;
            fi;
        fi;
        return false;
    end;

    loops:=Filtered(bnd[2]*1,x->Length(Positions(bnd[2],x))=2);
    loops:=Set(Concatenation(List(loops,x->x{[2,3]})));
    bnd[1]:=Concatenation(bnd[1],bnd[1]);
    sub[1]:=Concatenation(sub[1],[l0+2..2*l0-1]);
# sub contains all duplicate 0-cells except for those in the frame...
# for now

    copy1:=List(bnd[2],x->x+[0,l0,l0]);
    bnd[2]:=Concatenation(bnd[2],copy1);
    for i in [l1+1..2*l1] do
        if IsEdgeInDuplicate(i) then
            Add(sub[2],i);
        fi;
    od;

    hbars2:=List(hbars,x->x+l0);
    vbars2:=List(vbars,x->x+l0);
    copy2:=List(
        bnd[3],
        x->Concatenation([x[1]],List(x{[2..Length(x)]},y->y+l1))
    );
    bnd[3]:=Concatenation(bnd[3],copy2);
```

```
    for i in [l2..2*l2] do
        cell:=bnd[3][i]*1;
        x:=[]; # all 0-cells in a given 2-cell
        for j in [2..Length(cell)] do
            for k in [2,3] do
                Add(x,bnd[2][cell[j]][k]);
            od;
        od;
        x:=Set(x);
        if Length(Intersection(cell{[2..cell[1]+1]},sub[2]))=cell[1] and
            (
                true in List(hbars2,y->IsSubset(y,x)) or
                true in List(vbars2,y->IsSubset(y,x))
            ) and
                cell[1]<>2 then
                    Add(sub[3],i);
        fi;
    od;
    l1__:=Length(sub[2]); l2__:=Length(sub[3]);

# 13-02-21 remove any path components of sub that consist of two 0-cells
# and two 1-cells only
    path_comp:=PathComponentsCWSubcomplex([RegularCWComplex(bnd),sub]);
    for i in [1..Length(path_comp)] do
        if Length(path_comp[i][2][1])=2 and
            Length(path_comp[i][2][2])=2 then
                for j in [1,2] do
                    sub[j]:=Difference(sub[j],path_comp[i][2][j]);
                od;
        fi;
    od;

# join the original disk to the copy
# each n-cell of the disk yields an (n+1)-cell which connects it
# to its duplicate
    for i in [1..l0] do
        Add(bnd[2],[2,i,i+l0]);
        if i in loops and
            i in sub[1] and
```

```
            i+l0 in sub[1] and
                not i in [1,l0] then
                    Add(sub[2],Length(bnd[2]));
    fi;
od;

for i in [1..l1] do
    Add(
        bnd[3],
        [
            4,
            i,
            Position(bnd[2],[2,bnd[2][i][2],bnd[2][i+l1][2]]),
            Position(bnd[2],[2,bnd[2][i][3],bnd[2][i+l1][3]]),
            i+l1
        ]
    );
    if i in sub[2] and
        Length(Positions(bnd[2],bnd[2][i]))=2 and
            i+l1 in sub[2] then
            Add(sub[3],Length(bnd[3]));
    fi;
od;

for i in [1..l2] do
    3cell:=[];
    # 13-02-21 plug the corner holes
        if bnd[3][i][1]=2 and
            bnd[3][i][2] in sub[2] and
                not bnd[3][i][2]+l1 in sub[2] then
                    Add(sub[3],i);
        fi;
        if bnd[3][i][1]=2 and
            not bnd[3][i][2] in sub[2] and
                bnd[3][i][2]+l1 in sub[2] then
                    Add(sub[3],i+l2);
        fi;
    for j in bnd[3][i]{[2..Length(bnd[3][i])]} do
        Add(
```

```
                3cell,
                Position(
                    bnd[3],
                    [
                        4,
                        j,
                        Position(bnd[2],[2,bnd[2][j][2],bnd[2][j+l1][2]]),
                        Position(bnd[2],[2,bnd[2][j][3],bnd[2][j+l1][3]]),
                        j+l1
                    ]
                )
            );
        od;
        Add(3cell,i);
        Add(3cell,i+l2);
        Add(bnd[4],Concatenation([bnd[3][i][1]+2],3cell));
    od;

    for i in [3,4] do
     bnd[i]:=List(bnd[i],x->Concatenation([x[1]],Set(x{[2..Length(x)]})));
    od;
##############################################################################

# (v) join the open ended 'pipes' at either end of B^2xI according to crs
##############################################################################
    colour:=List([1..4],x->[]);

    if not IsBound(crs) then
        crs:=List([1..Length(crossings)],x->1);
    fi;

# (a) start with the lower caps, they're more straight forward ###########
    lcap:=[];
    reg:=RegularCWComplex(bnd);
    for i in [1..l2] do
        if bnd[3][i][1]<>2 and not i in sub[3] then
            Add(lcap,i);
        elif bnd[3][i][1]=2 and not i in sub[3] then
            closure:=ClosureCWCell(reg,2,i);
```

```
        if not IsSubset(sub[2],closure[2][2]) then
            Add(lcap,i);
        fi;
    fi;
od;

pipes:=[]; # the 0,1 and 2-cells of each 'pipe' in the lower dome
# which will join together to form the (intersecting) tubular surface
for i in [1..l2] do
    if i in sub[3] then
        Add(
            pipes,
            [
                [i],
                bnd[3][i]{[2..bnd[3][i][1]+1]},
                Set(
                    Concatenation(
                        List(
                            bnd[3][i]{[2..bnd[3][i][1]+1]},
                            x->bnd[2][x]{[2,3]}
                        )
                    )
                )
            ]
        );
    fi;
od;
pipes:=Filtered(pipes,x->Length(x[3])>2);
for i in [1..Length(pipes)] do
    for j in [i+1..Length(pipes)] do
        if Intersection(
            Intersection(
                pipes[i][3],pipes[j][3]
            ),
            Concatenation(crossings)
        )<>[] then
            Append(pipes[i][1],pipes[j][1]);
            Append(pipes[i][2],pipes[j][2]);
            Append(pipes[i][3],pipes[j][3]);
```

```
            pipes[j][3]:=[];
        fi;
    od;
od;
Apply(pipes,x->[Set(x[1]),Set(x[2]), Set(x[3])]);
pipes:=Filtered(pipes,x->x[3]<>[]);
for i in [1..Length(pipes)] do # either end of each pipe has two
# 2-cells which are currently absent from pipe[i][1],
# this step will add them
    for j in Filtered(bnd[3],x->x[1]=2) do
        int:=Intersection(pipes[i][2],j{[2,3]});
        if int<>[] then
            Add(pipes[i][1],Position(bnd[3],j));
        fi;
    od;
od;
for i in [1..Length(pipes)] do # if two pipes' 0-skeletons intersect,
    for j in [i+1..Length(pipes)] do # add a new 1-cell whose boundary
    # is their intersection
        int:=Intersection(pipes[i][3],pipes[j][3]);
        if int<>[] then
            Add(int,2,1);
            Add(bnd[2],int);
            Add(sub[2],Length(bnd[2]));
        fi;
    od;
od;
for i in [1..Length(pipes)] do # replace the 1-cells that occur twice
    for j in [1..Length(pipes[i][2])] do # with either the new cell we
    # just added or the other occurance of that cell which is not in
    # pipes[i][2]
        pos:=Positions(
            bnd[2],
            [
                2,
                bnd[2][pipes[i][2][j]][2],
                bnd[2][pipes[i][2][j]][3]
            ]
        );
```

```
        if Length(pos)>=2 then
            cell:=Last(Filtered(pos,x->x<>pipes[i][2][j]));
            pipes[i][2][j]:=cell;
        fi;
    od;
od;
HorizontalOrVertical:=function(l)
    if l in hbars then
        return "horizontal";
    fi;
    return "vertical";
end;
for i in [1..Length(pipes)] do # if a pipe contains 1-cells from
# crossings then filter out the horizontal/vertical 1-cells
# should said pipe be vertical/horizontal itself
    if Intersection(pipes[i][3],Concatenation(crossings))<>[] then
        ori:=HorizontalOrVertical(pipes[i][3]);
        l:=Filtered(crossings,x->Intersection(x,pipes[i][3])<>[])[1];
        if ori="horizontal" then
            pipes[i][2]:=Difference(
                pipes[i][2],
                [
                    Position(bnd[2],[2,l[1],l[2]]),
                    Position(bnd[2],[2,l[3],l[4]])
                ]
            );
        else
            pipes[i][2]:=Difference(
                pipes[i][2],
                [
                    Position(bnd[2],[2,l[1],l[3]]),
                    Position(bnd[2],[2,l[2],l[4]])
                ]
            );
        fi;
    fi;
od;
for i in [1..Length(pipes)] do # add new 2-cells to bnd[3] whose
# boundary is the pipes[i][2] 1-cells
```

```
        cell:=Set(pipes[i][2]);
        Add(cell,Length(cell),1);
        Add(bnd[3],cell);
        Add(sub[3],Length(bnd[3]));
        Add(lcap,Length(bnd[3]));
        Add(pipes[i][1],Length(bnd[3]));
    od;
    for i in [1..Length(pipes)] do # find where pipes intersect in their
    # 2-skeleta
    # use this to form the 3-cells comprising the interiors of the pipes
        for j in [i+1..Length(pipes)] do
            if Intersection(pipes[i][1],pipes[j][1])<>[] then
                Append(pipes[i][1],pipes[j][1]);
                pipes[j][1]:=[];
            fi;
        od;
    od;
    for i in [1..Length(pipes)] do
        if pipes[i][1]<>[] then
            cell:=Set(pipes[i][1]);
            Add(cell,Length(cell),1);
            Add(bnd[4],cell);
        fi;
    od;

# (b) now for the upper caps, 0 in crs leads to a very elaborate
# CW-structure
# 13-02-21 this function was added to help alter the intersection
# structure below after discovering it was incorrect (lol)
    AboveBelow0Cell:=function(n,str)
        local n_, i, j, l;

        if n>l0 then
            n_:=n-l0;
        else
            n_:=n*1;
        fi;

        i:=Position(List(GRD,x->n_ in x),true);
```

```
    j:=Position(GRD[i],n_);

    if str="above" then
        l:=Reversed([1..i-1]);
    else
        l:=[i+1..Length(GRD)];
    fi;

    for k in l do
        if GRD[k][j]<>0 then
            if n>l0 then
                return GRD[k][j]+l0;
            else
                return GRD[k][j];
            fi;
        fi;
    od;
end;

ucap:=[];
reg:=RegularCWComplex(bnd);
for i in [l2+1..2*l2] do
    if bnd[3][i][1]<>2 and not i in sub[3] then
        Add(ucap,i);
    elif bnd[3][i][1]=2 and not i in sub[3] then
        closure:=ClosureCWCell(reg,2,i);
        if not IsSubset(sub[2],closure[2][2]) then
            Add(ucap,i);
        fi;
    fi;
od;

pipes:=[]; # the 0,1 and 2-cells of each 'pipe' in the upper dome
# which will join together to form the (intersecting) tubular surface
for i in [l2+1..2*l2] do
    if i in sub[3] then
        Add(
            pipes,
            [
```

```
                    [i],
                    bnd[3][i]{[2..bnd[3][i][1]+1]},
                    Set(
                        Concatenation(
                            List(
                                bnd[3][i]{[2..bnd[3][i][1]+1]},
                                x->bnd[2][x]{[2,3]}
                            )
                        )
                    )
                ]
            );
        fi;
    od;
    pipes:=Filtered(pipes,x->Length(x[3])>2);

    IntersectingCylinders:=function(a,b,c,d) ##############################
        local
            n, i, m, j, l, a_abv,
            b_abv, c_blw, d_blw,
            del_2_cell_1, del_2_cell_2,
            del_2_cell_1_i, del_2_cell_2_i,
            dif, f_clr, f_clr1, f_clr2;
# attaches to a 0 crossing some additional regular CW-structure
# to allow for a self-intersection to occur
        n:=1*Length(bnd[1])+1;
        for i in [1..8] do # 0-skeleton of intersection
            Add(bnd[1],[1,0]);
            Add(sub[1],Length(bnd[1]));
        od;
        # 1-skeleton of intersection
        m:=1*Length(bnd[2])+1;
        Add(bnd[2],[2,a,n]); Add(sub[2],Length(bnd[2])); # m
        Add(bnd[2],[2,b,n+1]); Add(sub[2],Length(bnd[2])); # m+1
        Add(bnd[2],[2,c,n+6]); Add(sub[2],Length(bnd[2])); # m+2
        Add(bnd[2],[2,d,n+7]); Add(sub[2],Length(bnd[2])); # m+3
        for i in [0..3] do
            for j in [1,2] do
                Add(bnd[2],[2,n+2*i,n+1+2*i]); # m+4, m+5, m+6, m+7, m+10,
```

```
        # m+11, m+14, m+15
        # top first, then bottom (refer to drawing)
        Add(sub[2],Length(bnd[2]));
    od;
    if i>0 then
        Add(bnd[2],[2,n+2*i-2,n+2*i]); Add(sub[2],Length(bnd[2]));
        # m+8, m+12, m+16
      Add(bnd[2],[2,n+2*i-1,n+2*i+1]); Add(sub[2],Length(bnd[2]));
        # m+9, m+13, m+17
    fi;
od;

# 13-02-21 need to remove cells from sub to accommodate the
# new self-intersection structure
a_abv:=AboveBelow0Cell(a,"above");
b_abv:=AboveBelow0Cell(b,"above");
c_blw:=AboveBelow0Cell(c,"below");
d_blw:=AboveBelow0Cell(d,"below");
del_2_cell_1:=[
    Position(bnd[2],[2,a_abv,a]),
    Position(bnd[2],[2,b_abv,b]),
    Position(bnd[2],[2,a,b])
];
del_2_cell_1_i:=Position(
    List(
        bnd[3],
        x->Length(
            Intersection(
                x{[2..x[1]+1]},
                del_2_cell_1
            )
        )=3
    ),
    true
);
del_2_cell_2:=[
    Position(bnd[2],[2,c,c_blw]),
    Position(bnd[2],[2,d,d_blw]),
    Position(bnd[2],[2,c,d])
```

```
];
del_2_cell_2_i:=Position(
    List(
        bnd[3],
        x->Length(
            Intersection(
                x{[2..x[1]+1]},
                del_2_cell_2
            )
        )=3
    ),
    true
);
sub[3]:=Difference(sub[3],[del_2_cell_1_i,del_2_cell_2_i]);
sub[2]:=Difference(
    sub[2],
    [
        Position(bnd[2],[2,a_abv,a]),
        Position(bnd[2],[2,b_abv,b]),
        Position(bnd[2],[2,c,c_blw]),
        Position(bnd[2],[2,d,d_blw])
    ]
);

# 14-02-21 this new structure has 4 additional 1-cells
Add(bnd[2],[2,a_abv,n]); Add(sub[2],Length(bnd[2])); # m+18
Add(bnd[2],[2,b_abv,n+1]); Add(sub[2],Length(bnd[2])); # m+19
Add(bnd[2],[2,n+6,c_blw]); Add(sub[2],Length(bnd[2])); # m+20
Add(bnd[2],[2,n+7,d_blw]); Add(sub[2],Length(bnd[2])); # m+21

# 2-skeleton of intersection
l:=1*Length(bnd[3])+1;
Add( # l
    bnd[3],
    [
        4,
        Position(bnd[2],[2,a,b]),
        m,
        m+1,
```

```
        m+5
    ]
);
Add(sub[3],Length(bnd[3]));
Add( # l+1
    bnd[3],
    [
        4,
        Position(bnd[2],[2,c,d]),
        m+2,
        m+3,
        m+15
    ]
);
Add(sub[3],Length(bnd[3]));
# these 2-cells are those which should be coloured ###############
if IsBound(clr) then                                              ##
    pos:=Position(List(crossings,x->Set(x)+l0),Set([a,b,c,d])); ##
    pos:=pos-Length(Filtered(crs{[1..pos-1]},x->x<>0));          ##
fi;                                                               ##
Add(bnd[3],[4,m+4,m+6,m+8,m+9]); # l+2                            ##
Add(sub[3],Length(bnd[3]));                                       ##
Add(bnd[3],[4,m+5,m+7,m+8,m+9]); # l+3                            ##
Add(sub[3],Length(bnd[3]));                                       ##
for i in [0,1] do                                                 ##
    Add(bnd[3],[4,m+6+4*i,m+10+4*i,m+12+4*i,m+13+4*i]); # l+4, l+6
    Add(sub[3],Length(bnd[3]));                                   ##
    Add(bnd[3],[4,m+7+4*i,m+11+4*i,m+12+4*i,m+13+4*i]); # l+5, l+7
    Add(sub[3],Length(bnd[3]));                                   ##
od;                                                               ##
if IsBound(clr) then                                              ##
    if clr[pos]=1 then                                            ##
        colour[3][Length(bnd[3])-5]:=[-1]; # blue                 ##
        colour[3][Length(bnd[3])-4]:=[-1]; # blue                 ##
        colour[3][Length(bnd[3])-1]:=[-1]; # blue                 ##
        colour[3][Length(bnd[3])]:=[-1]; # blue                   ##
    elif clr[pos]=2 then                                          ##
        colour[3][Length(bnd[3])-5]:=[-1]; # blue                 ##
        colour[3][Length(bnd[3])-4]:=[-1]; # blue                 ##
```

```
            colour[3][Length(bnd[3])-1]:=[1]; # red                ##
            colour[3][Length(bnd[3])]:=[1]; # red                  ##
        elif clr[pos]=3 then                                       ##
            colour[3][Length(bnd[3])-5]:=[1]; # red                ##
            colour[3][Length(bnd[3])-4]:=[1]; # red                ##
            colour[3][Length(bnd[3])-1]:=[-1]; # blue              ##
            colour[3][Length(bnd[3])]:=[-1]; # blue                ##
        else                                                       ##
            colour[3][Length(bnd[3])-5]:=[1]; # red                ##
            colour[3][Length(bnd[3])-4]:=[1]; # red                ##
            colour[3][Length(bnd[3])-1]:=[1]; # red                ##
            colour[3][Length(bnd[3])]:=[1]; # red                  ##
        fi;                                                        ##
    fi;                                                            ##
    # 14-02-21 there are four more 2-cells                        ##
    dif:=Difference(bnd[3][del_2_cell_1_i]{[2..5]},del_2_cell_1)[1];##
    Add( # l+8 bottom                                              ##
        bnd[3],                                                    ##
        [                                                          ##
            4,                                                     ##
            dif,                                                   ##
            m+5,                                                   ##
            m+18,                                                  ##
            m+19                                                   ##
        ]                                                          ##
    );                                                             ##
    Add(sub[3],Length(bnd[3]));                                    ##
    dif:=Difference(bnd[3][del_2_cell_2_i]{[2..5]},del_2_cell_2)[1];##
    Add( # l+9 bottom                                              ##
        bnd[3],                                                    ##
        [                                                          ##
            4,                                                     ##
            dif,                                                   ##
            m+15,                                                  ##
            m+20,                                                  ##
            m+21                                                   ##
        ]                                                          ##
    );                                                             ##
    Add(sub[3],Length(bnd[3]));                                    ##
```

```
if IsBound(clr) then                                                    ##
    if clr[pos]=1 then                                                  ##
        colour[3][Length(bnd[3])-1]:=[-1]; # blue                       ##
        colour[3][Length(bnd[3])]:=[-1]; # blue                         ##
    elif clr[pos]=2 then                                                ##
        colour[3][Length(bnd[3])-1]:=[-1]; # blue                       ##
        colour[3][Length(bnd[3])]:=[1]; # red                           ##
    elif clr[pos]=3 then                                                ##
        colour[3][Length(bnd[3])-1]:=[1]; # red                         ##
        colour[3][Length(bnd[3])]:=[-1]; # blue                         ##
    else                                                                ##
        colour[3][Length(bnd[3])-1]:=[1]; # red                         ##
        colour[3][Length(bnd[3])]:=[1]; # red                           ##
    fi;                                                                 ##
fi;                                                                     ##
#########################################################################
Add(bnd[3],[2,m+4,m+5]); # l+10
Add(sub[3],Length(bnd[3]));
Add(bnd[3],[2,m+14,m+15]); # l+11
Add(sub[3],Length(bnd[3]));
# from this point onwards, cells added are only present in bnd,
# not sub
Add( # l+12
    bnd[3],
    [
        6,
        Position(bnd[2],[2,a,c]),
        m,
        m+2,
        m+8,
        m+12,
        m+16
    ]
);
Add( # l+13
    bnd[3],
    [
        6,
        Position(bnd[2],[2,b,d]),
```

```
            m+1,
            m+3,
            m+9,
            m+13,
            m+17
        ]
    );
    # 14-02-21 another few 2-cells to bnd only
    Add( # l+14
        bnd[3],
        [
            3,
            Position(bnd[2],[2,a_abv,a]),
            m,
            m+18
        ]
    );
    Add( # l+15
        bnd[3],
        [
            3,
            Position(bnd[2],[2,b_abv,b]),
            m+1,
            m+19
        ]
    );
    Add( # l+16
        bnd[3],
        [
            3,
            Position(bnd[2],[2,c,c_blw]),
            m+2,
            m+20
        ]
    );
    Add( # l+17
        bnd[3],
        [
            3,
```

```
        Position(bnd[2],[2,d,d_blw]),
        m+3,
        m+21
    ]
);
# 19/04/21 found some 2-cells missing from ucap causing the chain
# complex boundary matrices to detect that this wasn't a regular
# cw-complex
for i in [0..3] do Add(ucap,l+14+i); od;
# 3-skeleton of intersection
Add(
    bnd[4],
    [
        8,
        l+2,
        l+3,
        l+4,
        l+5,
        l+6,
        l+7,
        l+10,
        l+11
    ]
);
Add(
    bnd[4],
    [
        8,
        Position(
            List(bnd[3],Set),
            Set(
                [
                    4,
                    Position(bnd[2],[2,a,b]),
                    Position(bnd[2],[2,c,d]),
                    Position(bnd[2],[2,a,c]),
                    Position(bnd[2],[2,b,d])
                ]
            )
```

```
        ),
        l,
        l+1,
        l+3,
        l+5,
        l+7,
        l+12,
        l+13
    ]
);
# 14-02-21 need two more 3-cells to finish off the structure
Add(
    bnd[4],
    [
        5,
        del_2_cell_1_i,
        l,
        l+8,
        l+14,
        l+15
    ]
);
Add(
    bnd[4],
    [
        5,
        del_2_cell_2_i,
        l+1,
        l+9,
        l+16,
        l+17
    ]
);

if IsBound(clr) then
    f_clr:=clr[pos]*1;
    if f_clr=1 then
        f_clr1:=-1;
        f_clr2:=-1;
```

```
        elif f_clr=2 then
            f_clr1:=-1;
            f_clr2:=1;
        elif f_clr=3 then
            f_clr1:=1;
            f_clr2:=-1;
        else
            f_clr1:=1;
            f_clr2:=1;
        fi;
    else
        f_clr1:=0;
        f_clr2:=0;
    fi;

    for i in [1..Length(pipes)] do
        if IsBound(pipes[i][1][1]) then
            if pipes[i][1][1]=del_2_cell_1_i then
                pipes[i]:=[
                    [
                        Position(
                            bnd[3],
                            Concatenation(
                                [2],
                                Set(
                                    Positions(
                                        bnd[2],
                                        Concatenation(
                                            [2],
                                            Set(
                                                [a_abv,b_abv]
                                            )
                                        )
                                    )
                                )
                            )
                        ),
                        l+8,
                        l+10
```

```
                    ],
                    [
                        Filtered(
                            pipes[i][2],
                            x->Length(
                                Positions(
                                    bnd[2],
                                    bnd[2][x]
                                )
                            )=2
                        )[1],
                        m+4,
                        m+18,
                        m+19
                    ],
                    [
                        a_abv,
                        b_abv,
                        n,
                        n+1
                    ],
                    '*',
# just to mark this pipe as the vertical part of an intersection
                    f_clr1
                ];
            elif pipes[i][1][1]=del_2_cell_2_i then
                pipes[i]:=[
                    [
                        Position(
                            bnd[3],
                            Concatenation(
                                [2],
                                Set(
                                    Positions(
                                        bnd[2],
                                        Concatenation(
                                            [2],
                                            Set(
                                                [c_blw,d_blw]
```

```
                                        )
                                    )
                                )
                            )
                        )
                    ),
                    l+9,
                    l+11
                ],
                [
                    Filtered(
                        pipes[i][2],
                        x->Length(
                            Positions(
                                bnd[2],
                                bnd[2][x]
                            )
                        )=2
                    )[1],
                    m+14,
                    m+20,
                    m+21
                ],
                [
                    c_blw,
                    d_blw,
                    n+6,
                    n+7
                ],
                '*',
                f_clr2
            ];
        fi;
    fi;
od;
for i in [1..Length(pipes)] do
    if not IsBound(pipes[i][4]) then
        int:=Set(
            Intersection(
```

```
                        pipes[i][3],
                        [a,b,c,d]
                    )
                );
                if Length(int)=4 then
                    if crs[Position(List(crossings+l0,Set),int)]=0 then
                        pipes[i]:=[
                            [
                                l+2,
                                l+4,
                                l+6,
                                l+12,
                                l+13
                            ],
                            [
                                m,
                                m+1,
                                m+2,
                                m+3,
                                m+4,
                                m+14
                            ],
                            [
                                a,
                                b,
                                c,
                                d,
                                n,
                                n+1,
                                n+6,
                                n+7
                            ],
                        ];
                    fi;
                fi;
            fi;
        od;
    end; ##########################################################################
```

```
for i in [1..Length(crs)] do
    if crs[i]=0 then
        IntersectingCylinders(
            crossings[i][1]+l0,
            crossings[i][3]+l0,
            crossings[i][2]+l0,
            crossings[i][4]+l0
        );
    fi;
od;
for i in [1..Length(pipes)] do # ignoring all self-intersections, join
# all pipes which intersect at a common crossing point
    for j in [i+1..Length(pipes)] do
        if not IsBound(pipes[i][4]) and not IsBound(pipes[j][4]) then
            int:=Intersection(pipes[i][3],pipes[j][3]);
            if Length(int)>=2 and
                Intersection(int,Concatenation(crossings)+l0)=int
                then
                Append(pipes[i][1],pipes[j][1]);
                pipes[i][1]:=Set(pipes[i][1]); pipes[j][1]:=[];
                Append(pipes[i][2],pipes[j][2]);
                pipes[i][2]:=Set(pipes[i][2]); pipes[j][2]:=[];
                Append(pipes[i][3],pipes[j][3]);
                pipes[i][3]:=Set(pipes[i][3]); pipes[j][3]:=[];
            fi;
        fi;
    od;
od;
pipes:=Filtered(pipes,x->x<>[[],[],[]]);
for i in [1..Length(pipes)] do # add the degree two 2-cells to the
# ends of each pipe except for the vertical pipes at each intersection
# (they already have them)
    if not IsBound(pipes[i][4]) then
        for j in Filtered(bnd[3]{[l2+1..2*l2]},x->x[1]=2) do
            int:=Intersection(pipes[i][2],j{[2,3]});
            if int<>[] then
                Add(pipes[i][1],Position(bnd[3],j));
            fi;
        od;
```

```
    fi;
od;
for i in [1..Length(pipes)] do # if two pipes' 0-skeletons intersect
# (at somewhere other than a crossing), add a new 1-cell whose
# boundary is their intersection
    for j in [i+1..Length(pipes)] do
        int:=Intersection(pipes[i][3],pipes[j][3]);
        if int<>[] then
            if Intersection(int,Concatenation(crossings)+l0)=[] and
                int<[2*l0,2*l0] then
                    Add(int,2,1);
                    Add(bnd[2],int);
                    Add(sub[2],Length(bnd[2]));
            fi;
        fi;
    od;
od;
for i in [1..Length(pipes)] do # replace the 1-cells (added pre
# IntersectingCylinders) that occur twice with either the new cell we
# just added or the other occurance of that cell which is not in
# pipes[i][2]
    for j in [1..Length(pipes[i][2])] do
        pos:=Positions(
            bnd[2],
            [
                2,
                bnd[2][pipes[i][2][j]][2],
                bnd[2][pipes[i][2][j]][3]
            ]
        );
        if Length(pos)>=2 and
            [
                bnd[2][pipes[i][2][j]][2],
                bnd[2][pipes[i][2][j]][3]
            ]<[2*l0,2*l0] then
                cell:=Last(Filtered(pos,x->x<>pipes[i][2][j]));
                pipes[i][2][j]:=cell;
        fi;
    od;
```

```
od;
HorizontalOrVertical:=function(l)
    if l in hbars+l0 then
        return "horizontal";
    fi;
    return "vertical";
end;
for i in [1..Length(pipes)] do # if a pipe contains 1-cells from
# crossings then filter out the horizontal/vertical 1-cells should
# said pipe be vertical/horizontal itself however if a pipe contains
# a self intersection remove both the horizontal AND vertical 1-cells
    int:=Intersection(pipes[i][3],Concatenation(crossings)+l0);
    if Length(int)=4 then
        pos:=Position(List(crossings,Set)+l0,Set(int));
        if crs[pos]=0 then
            pipes[i][2]:=Difference(
                pipes[i][2],
                [
                    Position(
                        bnd[2],
                        [2,crossings[pos][1]+l0,crossings[pos][2]+l0]
                    ),
                    Position(
                        bnd[2],
                        [2,crossings[pos][3]+l0,crossings[pos][4]+l0]
                    ),
                    Position(
                        bnd[2],
                        [2,crossings[pos][1]+l0,crossings[pos][3]+l0]
                    ),
                    Position(
                        bnd[2],
                        [2,crossings[pos][2]+l0,crossings[pos][4]+l0]
                    )
                ]
            );
        else
            ori:=HorizontalOrVertical(pipes[i][3]);
            if ori="horizontal" then
```

```
                pipes[i][2]:=Difference(
                pipes[i][2],
                [
                    Position(
                        bnd[2],
                        [2,crossings[pos][1]+l0,crossings[pos][2]+l0]
                    ),
                    Position(
                        bnd[2],
                        [2,crossings[pos][3]+l0,crossings[pos][4]+l0]
                    ),
                ]
            );
            else
                pipes[i][2]:=Difference(
                pipes[i][2],
                [
                    Position(
                        bnd[2],
                        [2,crossings[pos][1]+l0,crossings[pos][3]+l0]
                    ),
                    Position(
                        bnd[2],
                        [2,crossings[pos][2]+l0,crossings[pos][4]+l0]
                    )
                ]
            );
            fi;
        fi;
    fi;
od;
for i in [1..Length(pipes)] do # add new 2-cells to bnd[3] whose
# boundary is the pipes[i][2] 1-cells
    cell:=Set(pipes[i][2]);
    Add(cell,Length(cell),1);
    Add(bnd[3],cell);
    Add(sub[3],Length(bnd[3]));
    Add(ucap,Length(bnd[3]));
    Add(pipes[i][1],Length(bnd[3]));
```

```
        if IsBound(pipes[i][5]) then
            if IsBound(clr) then
                colour[3][Length(bnd[3])]:=[pipes[i][5]];
            fi;
        fi;
    od;
    for i in [1..Length(pipes)] do # find where pipes intersect in their
    # 2-skeleta use this to form the 3-cells comprising the
    # interiors of the pipes
        for j in [i+1..Length(pipes)] do
            if Intersection(pipes[i][1],pipes[j][1])<>[] then
                Append(pipes[i][1],pipes[j][1]);
                pipes[j][1]:=[];
            fi;
        od;
    od;
    for i in [1..Length(pipes)] do
        if pipes[i][1]<>[] then
            cell:=Set(pipes[i][1]);
            Add(cell,Length(cell),1);
            Add(bnd[4],cell);
        fi;
    od;
##############################################################################

# add a cap to both ends of D x [0,1]
##############################################################################
    Add(
        bnd[3],
        [
            2,
            Positions(bnd[2],[2,1,10])[1],
            Positions(bnd[2],[2,1,10])[2]
        ]
    );
    Add(lcap,Length(bnd[3]));
    lcap:=Set(lcap);
    Add(lcap,Length(lcap),1);
    Add(bnd[4],lcap);
```

```
    Add(
        bnd[3],
        [
            2,
            Positions(bnd[2],[2,l0+1,2*l0])[1],
            Positions(bnd[2],[2,l0+1,2*l0])[2]
        ]
    );
    Add(ucap,Length(bnd[3]));
    ucap:=Set(ucap);
    Add(ucap,Length(ucap),1);
    Add(bnd[4],ucap);
##############################################################################

# add colour
##############################################################################
    for i in [1..Length(bnd[3])] do
        if not IsBound(colour[3][i]) then
            colour[3][i]:=[0];
        fi;
    od;
    colour_:={n,k}->colour[n+1][k];
##############################################################################
    x:=CWSubcomplexToRegularCWMap([RegularCWComplex(bnd),sub]);
    x!.colour:=colour_;
    return x;
end);
```

B.12 KinkArc2Presentation

```
InstallGlobalFunction(
    KinkArc2Presentation,
    function(arc)
    local
        mat, i, x, j, Twist, z, coord, keep, arc_;

    mat:=List([1..Length(arc[1])],x->[1..Length(arc[1])]*0);
```

```
for i in [1..Length(arc[1])] do
    mat[Length(mat)-i+1][AbsoluteValue(arc[1][i][1])]:=3;
    mat[Length(mat)-i+1][AbsoluteValue(arc[1][i][2])]:=3;
od;
for x in [1,2] do
    if x=2 then
        mat:=MutableTransposedMat(mat);
    fi;
    for i in [1..Length(mat)] do
        for j in [2..Length(mat)-1] do
            if 3 in mat[i]{[1..j-1]} and
            3 in mat[i]{[j+1..Length(mat)]} then
                mat[i][j]:=mat[i][j]+1;
            fi;
        od;
    od;
od;
mat:=MutableTransposedMat(mat);
for i in Set(Concatenation(arc[1])) do
    if i<0 then
        for j in [2..Length(mat)-1] do
            if mat[j][-i] in [1,2] then
                mat[j][-i]:=0;
            fi;
        od;
    fi;
od;
# mat models the arc 2-presentation, even should it correspond to
# (a) knotted sphere(s)

Twist:=function(i,j)
    local
        x, k, l, left, right, up, down;

    # first create a new row and column in mat
    for x in [1..Length(mat)] do
        Add(mat[x],4,j);
    od;
    Add(
```

```
        mat,
        Concatenation(
            List([1..j-1],y->4),
            [3],
            mat[i]{[j+1..Length(mat)+1]}
        ),
        i+1
    );
    mat[i]:=Concatenation(
        mat[i]{[1..j-1]},
        [3],
        List([j+1..Length(mat)],y->4)
    );

    # now determine the values of the 2n+1 new entries
    # where n is the length of mat before Twist
    for k in [1..Length(mat)] do
        for l in [1..Length(mat[k])] do
            if mat[k][l]=4 then
                if k in [1,Length(mat)] then
                # the entry is either in the top/bottom row
                # so there is no need to check above/below it
                    left:=mat[k]{[1..l-1]};
                    right:=mat[k]{[l+1..Length(mat)]};
                    if 3 in left and 3 in right then
                        if 0 in mat[k]{[l-1,l+1]} then
                        # this is to check if this row corresponds
                        # to an omitted row in the arc 2-pres.
                            mat[k][l]:=0;
                        else
                            mat[k][l]:=1;
                        fi;
                    else
                        mat[k][l]:=0;
                    fi;
                elif l in [1,Length(mat)] then
                # the entry is in the first/last column and
                # we don't need to check to the left/right of it
                    up:=mat{[1..k-1]}[l];
```

```
                    down:=mat{[k+1..Length(mat)]}[l];
                    if 3 in up and 3 in down then
                        if 0 in mat{[k,k+1]}[l] then
                            mat[k][l]:=0;
                        else
                            mat[k][l]:=1;
                        fi;
                    else
                        mat[k][l]:=0;
                    fi;
                else
                # we are required to check above/below/left/right
                # of the entry to determine its value
                    left:=mat[k]{[1..l-1]};
                    right:=mat[k]{[l+1..Length(mat)]};
                    up:=mat{[1..k-1]}[l];
                    down:=mat{[k+1..Length(mat)]}[l];
                    if 3 in left and 3 in right then
                        if 0 in mat[k]{[l-1,l+1]} then
                            mat[k][l]:=0;
                        else
                            mat[k][l]:=1;
                        fi;
                    elif 3 in up and 3 in down then
                        if 0 in mat{[k,k+1]}[l] then
                            mat[k][l]:=0;
                        else
                            mat[k][l]:=1;
                        fi;
                    else
                        mat[k][l]:=0;
                    fi;
                fi;
            fi;
        od;
    od;

    return mat;
end;
```

```
for z in [1,2] do
    if z=2 then
        mat:=MutableTransposedMat(mat);
    fi;
    for i in
        [1..Sum(List(mat,y->Maximum(0,Length(Positions(y,2))-1)))] do
    # number of times we need to perform Twist on the rows of mat
        coord:=Filtered(
            Concatenation(
                List(
                    [1..Length(mat)],
                    x->List(
                        [1..Length(mat)],
                        y->[x,y,mat[x][y]]
                    )
                )
            ),
            x->x[3]=2
        );
        keep:=[];
        for j in [1..Length(coord)] do
            if coord[j][1] in List(coord,x->x[1]){[1..j-1]} then
                Add(keep,j);
            fi;
        od;
        coord:=First(coord{keep});
        # the coordinate of the first instance of a crossing that
        # lies in a row where multiple crossings occur
        mat:=Twist(coord[1],coord[2]);
    od;
od;
mat:=MutableTransposedMat(mat);

# before output we need to see if there are any columns that were
# omitted. if so we need to find their new positions and reflect
# that with a negative entry in arc[1]
```

```
    # now to recover the new arc 2-presentation from mat
    arc_:=ShallowCopy(arc);
    arc_[1]:=[];
    for i in Reversed([1..Length(mat)]) do
        Add(
            arc_[1],
            Filtered([1..Length(mat)],y->mat[i][y]=3)
        );
    od;

    return arc_;
end);
```

B.13 Spin†

```
InstallGlobalFunction(
    Spin,
    function(inc)
    local
        X, Y, U, map, bndX, bndY, bnd, orient, quad2pair, pair2quad,
        indx, bndd, p, pr, INDX, pos, i, j, ij, n, x, y, a, b, ia, ib,
        count, BND, ORIEN, M;

    X:=Target(inc);
    U:=Source(inc);
    indx:=U!.boundaries;;
    map:=inc!.mapping;
    Y:=[ [[1,0],[1,0]], [[2,1,2],[2,1,2]], [[2,1,2]], [] ];
    Y:=RegularCWComplex(Y);
    #Y is the unit disk

    bndX:=X!.boundaries;
    bndY:=Y!.boundaries;
    bnd:=List([0..Dimension(X)+Dimension(Y)],i->[]);
    orient:=List([0..Dimension(X)+Dimension(Y)],i->[]);

    quad2pair:=[];
```

```
pair2quad:=List([1..1+Dimension(X)+Dimension(Y)],i->[]);;
count:=List([1..1+Dimension(X)+Dimension(Y)],i->0);

for i in [1..1+Dimension(X)] do
    quad2pair[i]:=[];
    for j in [1..1+Dimension(Y)] do
        quad2pair[i][j]:=[];
        ij:=i-1+j;
        for x in [1..Length(bndX[i])] do
            quad2pair[i][j][x]:=[];
            for y in [1..Length(bndY[j])] do
                count[ij]:=count[ij]+1;
                quad2pair[i][j][x][y]:=[ij,count[ij]];
                pair2quad[ij][count[ij]]:=[i,j,x,y];
            od;
        od;
    od;
od;

for i in [1..1+Dimension(X)] do
    for j in [1..1+Dimension(Y)] do
        for x in [1..Length(bndX[i])] do
            for y in [1..Length(bndY[j])] do
                BND:=[0];
                ORIEN:=[];
                if i>1 then
                    a:=bndX[i][x];
                    for ia in [2..Length(a)] do #a{[2..Length(a)]} do
                        Add(BND,quad2pair[i-1][j][a[ia]][y][2]);
                        BND[1]:=BND[1]+1;
                    od;
                fi;
                if j>1 then
                    b:=bndY[j][y];
                    for ib in [2..Length(b)] do #b{[2..Length(b)]} do
                        Add(BND,quad2pair[i][j-1][x][b[ib]][2]);
                        BND[1]:=BND[1]+1;
                    od;
                fi;
```

```
                bnd[i-1+j][quad2pair[i][j][x][y][2]]:=BND;
                orient[i-1+j][quad2pair[i][j][x][y][2]]:=ORIEN;
            od;
        od;
    od;
od;
bnd[1]:=List(bnd[1],i->[1,0]);
Add(bnd,[]);

#Let's now work on removing those cells of the
#form f x e with e the unique 2-cell in Y and f NOT in U.
indx:=List(
    [1..Length(X!.boundaries)],
    a-> [1..Length(X!.boundaries[a])]
);
for n in [1..Length(indx)-1] do
    for j in [1..Length(U!.boundaries[n])] do
        indx[n][map(n-1,j)]:=0;
    od;
od;

#INDX[n] is a list of n-1-cells
#bnd[n] is a list of boundaries of n-1-cells
#map(n,i) is the image of the ith n-cell in U
INDX:=List([1..Length(bnd)],a->[1..Length(bnd[a])]);
for p in [1..1+Dimension(X)] do
    for x in [1..X!.nrCells(p-1)] do
        if indx[p][x]>0 then
            pr:=quad2pair[p][3][x][1];
            INDX[pr[1]][pr[2]]:=0;
        fi;
    od;
od;

for n in [1..Length(INDX)] do
    INDX[n]:=Filtered(INDX[n],a->a>0);
od;

for n in [1..Length(bnd)] do
```

```
        bnd[n]:=bnd[n]{INDX[n]};
    od;

    ################################
    pos:=function(n,i);
        return Position(INDX[n],i);
    end;
    ################################

    #For safety, we'll be inefficient and create a copy of bnd rather
    #than modify bnd. This gets around silly mistakes that could arise
    #with pointers.

    bndd:=List([1..Length(bnd)],i->[]);

    bndd[1]:=bnd[1];
    for n in [1..Length(bnd)-1] do
        for x in bnd[n+1] do
            Add(
                bndd[n+1],
                Concatenation(
                    [x[1]],
                    List(x{[2..Length(x)]},i->pos(n,i))
                )
            );
        od;
    od;

    M:=RegularCWComplex(1*bndd);
    OrientRegularCWComplex(M);

    return M;
end);
```

B.14 SpunAboutHyperplane†

```
InstallGlobalFunction(
```

```
    SpunAboutHyperplane,
    function(K)
    local
        U, i, inc;

    U:=1*K!.binaryArray;
    for i in [2..Length(U)] do
        U[i]:=0*U[i];
    od;
    U:=PureCubicalComplex(U);
    inc:=RegularCWMap(K,U);

    return ContractedComplex(Spin(inc));
end);
```

B.15 SpunKnotComplement

```
InstallGlobalFunction(
    SpunKnotComplement,
    function(k)
    local
        d, K, DeletedPair, pair, C, U, C2, omicron;

    d:=[0,0,1,1,2,3,7,21,49,165,552];

    if IsList(k)
        then
        if not (IsInt(k[1]) and
            IsInt(k[2]) and
                SignInt(k[1])=1 and
                    SignInt(k[2])=1)
            then
            Error("the input must be a pair of positive integers.\n");
        elif k[1]>11
            then
    Error("only knots with less than 12 crossings are stored in HAP.\n");
        elif k[2]>d[k[1]]
```

```
        then
        Error("no such prime knot exists.\n");
    fi;
    K:=PureCubicalKnot(k[1],k[2]);
elif IsPureComplex(k)
    then
    K:=ShallowCopy(k);
else
    Error("input must be an integer pair or a knot.\n");
fi;

DeletedPair:=function(K)
    # inputs a cubical knot
    # outputs a list of cubical complexes [C,U]
    # C being the complement of K with a certain line removed
    # (except for its end-points)
    # U is a plane intersecting C at just these endpoints
    local
          C, array, 0row, i, pos, a, b, subarray, j;

    C:=PureComplexComplement(K);
    array:=ShallowCopy(C!.binaryArray);
    0row:=0; # location of the line to be altered

    while 0row=0
        do
        for i in [1..Length(array[2])]
            do
            if 0 in array[2][i]
                then
                0row:=0row+i;
                break;
            fi;
        od;
    od;

    pos:=Positions(array[2][0row],0);
    a:=pos[1]; # the endpoints of 0row
    b:=pos[Length(pos)];
```

```
        array[2][0row]:=0*ShallowCopy(array[2][0row])+1;
        array[2][0row][a]:=0;
        array[2][0row][b]:=0;
        array[1][0row]:=ShallowCopy(array[2][0row]);

        subarray:=0*ShallowCopy(array); # the plane about which the knot
        subarray[1]:=ShallowCopy(array[1]); # complement will be spun

        return [PureCubicalComplex(array), PureCubicalComplex(subarray)];

    end;

    pair:=DeletedPair(K);
    C:=ShallowCopy(pair[1]);
    U:=ShallowCopy(pair[2]);
    C2:=ContractedComplex(C,U);

    omicron:=RegularCWMap(C2,U);

    return Spin(omicron);

end);
```

B.16 SpunLinkComplement†

```
InstallGlobalFunction(
    SpunLinkComplement,
    function(K)
    local
        C, Y;

    Y:=PureCubicalKnot(K);
    Y:=PureComplexComplement(Y);
    Y:=SpunAboutHyperplane(Y);
    Y:=ContractedComplex(Y);
```

```
    return Y;
end);
```

B.17 RegularCWMapToCWSubcomplex

```
InstallGlobalFunction(
    RegularCWMapToCWSubcomplex,
    function(f)
    local
        dim, src_bnd, S, i, j;

    dim:=EvaluateProperty(f!.source,"dimension")*1;
    src_bnd:=f!.source!.boundaries*1;
    S:=List([0..dim],x->[]);

    for i in [1..dim+1] do
        for j in [1..Length(src_bnd[i])] do
            Add(S[i],f!.mapping(i-1,j));
        od;
    od;

    return [ShallowCopy(f!.target),S];
end);
```

B.18 CWSubcomplexToRegularCWMap

```
InstallGlobalFunction(
    CWSubcomplexToRegularCWMap,
    function(YS)
    local
        map, src, i, j, trg_cell;

    map:={i,j}->YS[2][i+1][j];

    src:=List([1..Length(Filtered(YS[2],y->y<>[]))+1],x->[]);
    src[1]:=List(YS[2][1],x->[1,0]);
```

```
    for i in [2..Length(src)-1] do
        for j in [1..Length(YS[2][i])] do
            trg_cell:=YS[1]!.boundaries[i][YS[2][i][j]]*1;
            trg_cell:=trg_cell{[2..trg_cell[1]+1]};
            trg_cell:=List(trg_cell,x->Position(YS[2][i-1],x));
            Add(trg_cell,Length(trg_cell),1);
            Add(src[i],trg_cell*1);
        od;
    od;

    return Objectify(
        HapRegularCWMap,
        rec(
            source:=RegularCWComplex(src),
            target:=ShallowCopy(YS[1]),
            mapping:=map
        )
    );
end);
```

B.19 IntersectionCWSubcomplex

```
InstallGlobalFunction(
    IntersectionCWSubcomplex,
    function(XS_1,XS_2)
    local
        max, xs_1, xs_2, i, j;

    max:=Maximum(Length(XS_1[2]),Length(XS_2[2]));
    xs_1:=ShallowCopy(XS_1[2]);
    xs_2:=ShallowCopy(XS_2[2]);

    for i in [xs_1,xs_2] do
        if Length(i)<max then
            for j in [Length(i)+1..max] do
                Add(i,[]);
```

```
            od;
        fi;
    od;

    return[
        ShallowCopy(XS_1[1]),
        List(
            [1..max],
            x->Intersection(xs_1[x],xs_2[x])
        )
    ];
end);
```

B.20 PathComponentsCWSubcomplex

```
InstallGlobalFunction(
    PathComponentsCWSubcomplex,
    function(XS)
    local
        ccs, i, j, k, cell, int, l;

    ccs:=List(
        XS[2][1]*1,
        x->Concatenation(
            [[x]],
            List([2..Length(XS[2])],y->[])
        )
    );
    for i in [1..Length(ccs)] do
        for j in [2..Length(ccs[i])] do
            for k in [1..Length(XS[2][j])] do
                cell:=XS[1]!.boundaries[j][XS[2][j][k]]*1;
                cell:=cell{[2..cell[1]+1]};
                int:=Intersection(cell,ccs[i][j-1]);
                if int<>[] then
                    for l in [1..Length(cell)] do
                        Add(ccs[i][j-1],cell[l]);
```

```
                    od;
                    Add(ccs[i][j],XS[2][j][k]);
                fi;
            od;
        od;
    od;
    for i in [1..Length(ccs)] do
        for j in Filtered([1..Length(ccs)],x->x<>i) do
            if ccs[i]<>[] and ccs[j]<>[] then
                for k in [1..Length(ccs[i])] do
                    if ccs[j]<>[] then
                        if Intersection(ccs[i][k],ccs[j][k])<>[] then
                            for l in [1..Length(ccs[i])] do
                                ccs[i][l]:=Set(
                                    Concatenation(
                                        ccs[i][l],
                                        ccs[j][l]
                                    )
                                );
                            od;
                            ccs[j]:=[];
                        fi;
                    fi;
                od;
            fi;
        od;
    od;
    ccs:=Filtered(List(ccs,x->List(x,y->Set(y))),z->z<>[]);

    return List(ccs,x->[ShallowCopy(XS[1]),x]);
end);
```

B.21 ClosureCWCell

```
InstallGlobalFunction(
    ClosureCWCell,
    function(Y,k,i)
```

```
    local
        complex, clsr, l, m, bnd, n;

    complex:=Y!.boundaries*1;

    clsr:=List([1..k+1],x->[]);
    Add(clsr[k+1],i);

    for l in Reversed([2..k+1]) do
        for m in [1..Length(clsr[l])] do
            bnd:=[];
            for n in [2..Length(complex[l][clsr[l][m]])] do
                Add(bnd,complex[l][clsr[l][m]][n]);
            od;
            clsr[l-1]:=Union(clsr[l-1],bnd);
        od;
    od;

    return [RegularCWComplex(complex),clsr];
end);
```

B.22 HAP_KK_AddCell

```
InstallGlobalFunction(
    HAP_KK_AddCell,
    function(B,k,b,c)
    local
        i;

    Add(b,Length(b),1);
    Add(B[k+1],b);

    for i in [1..Length(c)] do
        Add(B[k+2][c[i]],Length(B[k+1]));
        B[k+2][c[i]][1]:=B[k+2][c[i]][1]+1;
    od;
end);
```

B.23 BarycentricallySubdivideCell

```
InstallGlobalFunction(
    BarycentricallySubdivideCell,
    function(f,n,k)

    local
        inc, Y, clsr, complex,
        Subdivide, i, j;

    inc:=RegularCWMapToCWSubcomplex(ShallowCopy(f));
    Y:=inc[1];

    clsr:=ClosureCWCell(Y,n,k);
    complex:=Y!.boundaries*1;
    clsr:=clsr[2]*1;

    Subdivide:=function(a,b)
    # subdivides the bth a-cell into as many a-cells
    # as there are in its barycentric subdivision
        local
            sub_clsr, i, j, l, ints, bnd_ints,
            bnd, pre_len, m, pre_len_bnd;

        sub_clsr:=ClosureCWCell(RegularCWComplex(complex),a,b)[2];

        Add(complex[1],[1,0]); # the barycentre
        if Length(inc[2])>=a+1 then
            if b in inc[2][a+1] then
                Add(inc[2][1],Length(complex[1]));
            fi;
        fi;

        pre_len_bnd:=List([1..a],x->Length(complex[x]));

        for i in [1..a] do
            pre_len:=Length(complex[i+1])*1;
            for j in [1..Length(sub_clsr[i])] do
```

```
            if i=1 then # edge case is dealt with separately
                if a=1 and j=1 then # we overwrite the bth a-cell with
                # the first a-cell of its barycentric subdivision
                    complex[2][b]:=[
                        2,
                        sub_clsr[1][1],
                        Length(complex[1])
                    ];
                else
                    Add(
                        complex[2],
                        [
                            2,
                            sub_clsr[1][j],
                            Length(complex[1])
                        ]
                    );
                    if Length(inc[2])>=a+1 then
                        if b in inc[2][a+1] then
                            Add(inc[2][2],Length(complex[2]));
                        fi;
                    fi;
                fi;
            else
                bnd:=[];
                ints:=List([pre_len_bnd[i]+1..Length(complex[i])]);
                bnd_ints:=List(
                    ints,
                    x->complex[i][x]{[2..complex[i][x][1]+1]}
                );
                bnd_ints:=List(
                    bnd_ints,
                    x->Intersection(
                        x,
                        complex[i][sub_clsr[i][j]]{
                            [2..complex[i][sub_clsr[i][j]][1]+1]
                            }
                    )<>[]
                );
```

```
                    bnd:=Concatenation(
                        [sub_clsr[i][j]],
                        Filtered(ints,x->bnd_ints[Position(ints,x)]=true)
                    );
                    Add(bnd,Length(bnd),1);
                    if i=a and j=1 then # overwrite as before
                        complex[a+1][b]:=bnd;
                    else
                        Add(complex[i+1],bnd);

                        if Length(inc[2])>=a+1 then
                            if b in inc[2][a+1] then
                                Add(inc[2][i+1],Length(complex[i+1]));
                            fi;
                        fi;
                    fi;
                fi;

                if i=a and a<Dimension(Y) then # rewrite the coboundary of
                # the replaced a-cell to contain all the a-cells in its
                # barycentric subdivision
                    for l in [2..Length(Y!.coboundaries[a+1][b])] do
                        for m in [pre_len+1..Length(complex[a+1])] do
                            Add(
                                complex[a+2][Y!.coboundaries[a+1][b][l]],
                                m
                            );
                        od;
                    complex[a+2][Y!.coboundaries[a+1][b][l]][1]:=Length(
                        complex[a+2][Y!.coboundaries[a+1][b][l]]
                    )-1;
                    od;
                fi;
            od;
        od;
    end;

    for i in [2..Length(clsr)] do # inductively apply Subdivide to obtain
        for j in [1..Length(clsr[i])] do # the bary. subdivision of the
```

```
            Subdivide(i-1,clsr[i][j]); # kth n-cell of Y as required
        od;
    od;

    return CWSubcomplexToRegularCWMap(
        [
            RegularCWComplex(complex),
            inc[2]
        ]
    );
end);
```

B.24 SubdivideCell

```
InstallGlobalFunction(
    SubdivideCell,
    function(f,n,k)
    local
        sub, Y, closure, plus1,
        i, j, bnd;

    sub:=RegularCWMapToCWSubcomplex(ShallowCopy(f));
    Y:=sub[1]; # the actual CW-complex
    sub:=sub[2]*1; # the indexing of the subcomplex
    closure:=ClosureCWCell(Y,n,k)[2];
    Y:=Y!.boundaries*1;

    Add(Y[1],[1,0]); # the barycentre of the kth n-cell
    plus1:=List([1..Length(closure)],x->[]);
    # this will associate an x-cell in the closure to
    # the resulting (x+1)-cell in the subdivision

    for i in [1..Length(closure)-1] do
        for j in [1..Length(closure[i])] do
            if i=1 then
                Add(Y[2],[2,closure[i][j],Length(Y[1])]);
                Add(plus1[1],Length(Y[2]));
```

```
            else
                bnd:=Y[i][closure[i][j]];
                bnd:=bnd{[2..bnd[1]+1]};
                bnd:=List(bnd,x->plus1[i-1][Position(closure[i-1],x)]);
                Add(bnd,closure[i][j],1);
                Add(bnd,Length(bnd),1);
                if i=Length(closure)-1 and j=1 then
                    Y[i+1][closure[i+1][1]]:=bnd;
                    Add(plus1[i],closure[i+1][1]);
                else
                    Add(Y[i+1],bnd);
                    Add(plus1[i],Length(Y[i+1]));
                fi;
            fi;
        od;
    od;
    for i in [1..Length(Y[Length(closure)+1])] do
        if Last(closure)[1] in
            Y[Length(closure)+1][i]{[2..Y[Length(closure)+1][i][1]+1]}
            then
                Append(Y[Length(closure)+1][i],plus1[Length(closure)]);
                Unbind(Y[Length(closure)+1][i][1]);
                Y[Length(closure)+1][i]:=Set(Y[Length(closure)+1][i]);
                Add(
                    Y[Length(closure)+1][i],
                    Length(Y[Length(closure)+1][i]),
                    1
                );
        fi;
    od;
    return CWSubcomplexToRegularCWMap(
        [
            RegularCWComplex(Y),
            sub
        ]
    );
end);
```

B.25 RegularCWComplexComplement

```
InstallGlobalFunction(
    RegularCWComplexComplement,
    function(arg...)
    local
        f, check, subdiv, details, Y, B, IsInternal, count, total,
        path_comp, cobound_subcomplex, cbnd, i, j, clsr, int, crit,
        bary, IsSubpathComponent, ext_cell_2_f_notation,
        f_notation_2_ext_cell, ext_cells, k, ext_cell_bnd, e_n_bar,
        ext_cell_cbnd;

    if Length(arg)=1 then
     arg:=[arg[1],"some","basic",false];
    fi;
    f:=ShallowCopy(arg[1]);
    check:=arg[2]; # which cells we check for the contractible closures:
    # "some" or "all"
    subdiv:=arg[3]; # the subdivision we use:
    # "basic", "barycentric" or "none"
    details:=arg[4]; # do/don't suppress progress bars and other output

    Y:=RegularCWMapToCWSubcomplex(f);
    for i in [1..Dimension(Y[1])-Length(Y[2])+2] do
        Add(Y[2],[]);
    od;
    B:=List([1..Dimension(Y[1])+1],x->[]);

    IsInternal:=function(n,k) # is the kth n-cell of Y in Y\Z?
        if n+1>Length(Y[2]) then
            return true;
        elif k in Y[2][n+1] then
            return false;
        fi;
        return true;
    end;

    if details then
```

```
    Print("Testing contractibility...\n");
fi;
bary:=[];
path_comp:=List([1..Length(Y[1]!.boundaries)-1],x->[]);

if check="some" then
# we check only those cells that are ‘close’ to the subcomplex
# i.e. those cells lying within the coboundaries of the coboundaries
# of (...) the cells of the subcomplex
    cobound_subcomplex:=List(
        [1..Length(Y[2])],
        x->List(
            [1..Length(Y[2][x])],
            y->Y[1]!.coboundaries[x][Y[2][x][y]]{
                [2..Y[1]!.coboundaries[x][Y[2][x][y]][1]+1]
            }
        )
    );
    cobound_subcomplex:=List(cobound_subcomplex,Concatenation);
    cobound_subcomplex:=List(cobound_subcomplex,Set);
    cobound_subcomplex:=List(
        [1..Length(cobound_subcomplex)],
        x->Filtered(cobound_subcomplex[x],y->not y in Y[2][x+1])
    );
    Add(cobound_subcomplex,[],1); # keep indexing w/out 0-cells
    for i in [1..Length(cobound_subcomplex)-1] do
        for j in [1..Length(cobound_subcomplex[i])] do
            cbnd:=Y[1]!.coboundaries[i][cobound_subcomplex[i][j]];
            for k in [2..cbnd[1]+1] do
                Add(cobound_subcomplex[i+1],cbnd[k]);
            od;
        od;
    od;
    cobound_subcomplex:=List(cobound_subcomplex,Set);
    for i in [1..Length(cobound_subcomplex)-1] do
        cobound_subcomplex[i]:=Filtered(
            cobound_subcomplex[i],
            x->not x in Y[2][i+1]
        );
```

```
    od;
fi;

count:=0;
total:=Sum(
    List(Y[1]!.boundaries{[1..Length(Y[1]!.boundaries)]},Length)
);

# list of all of the path components of the intersection between
# the closure of each internal cell and the subcomplex Z < Y
# note: entries may be an empty list if they correspond to an
#       empty intersection or they may not be assigned a
#       value at all if the associated cell lies in Z
for i in [1..Length(Y[1]!.boundaries)] do
    for j in [1..Length(Y[1]!.boundaries[i])] do
        count:=count+1;
        if IsInternal(i-1,j) then
            Add(B[i],Y[1]!.boundaries[i][j]);
            # output will contain all internal cells
            if check="some" then
                if j in cobound_subcomplex[i] then
                    clsr:=ClosureCWCell(Y[1],i-1,j);
                    int:=IntersectionCWSubcomplex(clsr,Y);
                    path_comp[i][j]:=PathComponentsCWSubcomplex(int);
                else
                    path_comp[i][j]:=[];
                fi;
            else
                clsr:=ClosureCWCell(Y[1],i-1,j);
                int:=IntersectionCWSubcomplex(clsr,Y);
                path_comp[i][j]:=PathComponentsCWSubcomplex(int);
            fi;
            # CONTRACTIBILITY TEST
            # must be a non-empty subcomplex of dimension > 0
            if details then
                Print(count," out of ",total," cells tested.","\r");
            fi;
            if subdiv<>"none" then
                # test the contractibility of each path component
```

```
                # if we find anything non-contractible,
                # subdivide the problematic cells and restart
                for k in [1..Length(path_comp[i][j])] do
                    if path_comp[i][j][k][2]<>
                    List(path_comp[i][j][k][2],x->[])
                    and path_comp[i][j][k][2][2]<>[] then
                        crit:=CWSubcomplexToRegularCWMap(
                            path_comp[i][j][k]
                        );
                        crit:=Source(crit);
                        crit:=CriticalCellsOfRegularCWComplex(crit);
                        if not Length(crit)=1 then
                            Add(bary,[i-1,j]);
                        fi;
                    fi;
                od;
            fi;
        else
            Add(B[i],"*"); # temporary entry to keep correct indexing
        fi;
    od;
od;
bary:=Set(bary);

if bary=[] then
    if details then
        Print("\nThe input is compatible with this algorithm.\n");
    fi;
else
    if details then
        Print("\nSubdividing ",Length(bary)," cell(s):\n");
    fi;
    for i in [1..Length(bary)] do
        if subdiv="basic" then
            f:=SubdivideCell(
                f,
                bary[i][1],
                bary[i][2]
            );
```

```
            elif subdiv="barycentric" then
                f:=BarycentricallySubdivideCell(
                    f,
                    bary[i][1],
                    bary[i][2]
                );
            fi;
            if details then
                Print(Int(100*i/Length(bary)),"\% complete. \r");
            fi;
        od;
        Print("\n");
        return RegularCWComplexComplement(f,check,subdiv,details); # bad
    fi;

    for i in [1..Length(B)] do
        for j in [1..Length(B[i])] do
            if i>1 and B[i][j]<>"*" then
                B[i][j]:=Filtered(
                        B[i][j]{[2..B[i][j][1]+1]},
                        x->"*"<>B[i-1][x]
                    );
                Add(B[i][j],Length(B[i][j]),1);
            fi;
        od;
    od;
    # at this point, B corresponds to the cell complex Y\Z

    IsSubpathComponent:=function(super,sub)
        local
            i;

        for i in [1..Length(sub[2])-1] do
            if not IsSubset(super[2][i],sub[2][i]) then
                return false;
            fi;
        od;

        return true;
```

```
end;

ext_cell_2_f_notation:=NewDictionary([],true);
f_notation_2_ext_cell:=NewDictionary([],true);
# takes a pair [n,k] corresponding to the kth n-cell (external) of B
# and returns its associated "f notation" i.e. a triple [n+1,k',A]
# corresponding to the k'th (n+1)-cell (internal) whose closure
# intersected with Z in the path component A

ext_cells:=List([1..Length(B)],x->[]);

for i in [2..Length(path_comp)] do
    for j in [1..Length(path_comp[i])] do
        if IsBound(path_comp[i][j]) then
            if path_comp[i][j]<>[] then
                # we add as many external (i-1)-cells to B as
                # there are path components in path_comp[i][j]
                for k in [1..Length(path_comp[i][j])] do
                    if i=2 then
                        ext_cell_bnd:=[0];
                    else
                        e_n_bar:=ClosureCWCell(Y[1],i-1,j)[2];
                        ext_cell_bnd:=List(
                            ext_cells[i-2],
                            x->LookupDictionary(
                                ext_cell_2_f_notation,
                                [i-3,x]
                            )
                        );
                        ext_cell_bnd:=Filtered(
                            ext_cell_bnd,
                            x->
                                x[2] in e_n_bar[i-1]
                                and
                                IsSubpathComponent(
                                    path_comp[i][j][k],x[3]
                                )
                        );
                        ext_cell_bnd:=List(
```

```
                            ext_cell_bnd,
                            x->LookupDictionary(
                                f_notation_2_ext_cell,
                                x
                            )[2]
                        );
                    fi;
                    ext_cell_cbnd:=[j];

                    HAP_KK_AddCell(
                        B,
                        i-2,
                        ext_cell_bnd,
                        ext_cell_cbnd
                    );
                    Add(ext_cells[i-1],Length(B[i-1]));
                    AddDictionary(
                        ext_cell_2_f_notation,
                        [i-2,Length(B[i-1])],
                        [i-1,j,path_comp[i][j][k]]
                    );
                    AddDictionary(
                        f_notation_2_ext_cell,
                        [i-1,j,path_comp[i][j][k]],
                        [i-2,Length(B[i-1])]
                    );
                od;
            fi;
        fi;
    od;
od;

# a final reindexing of B and removal of "*"
# entries that once corresponded to cells of Z
for i in [2..Length(B)] do
    for j in [1..Length(B[i])] do
        for k in [2..Length(B[i][j])] do
            B[i][j][k]:=
            B[i][j][k]-
```

```
                Length(
                    Filtered(
                        B[i-1]{[1..B[i][j][k]]},
                        x->x="*"
                    )
                );
            od;
        od;
    od;
    B:=List(B,x->Filtered(x,y->y<>"*"));
    Add(B,[]);

    return RegularCWComplex(B);
end);
```

B.26 SequentialRegularCWComplexComplement

```
InstallGlobalFunction(
    SequentialRegularCWComplexComplement,
    function(arg...)
    local
        subdiv, method, details, map, sub, clsr, seq, tub, i;

    if Length(arg)=1 then
        arg:=[arg[1],"some","basic",false];
    fi;
    method:=arg[2];
    subdiv:=arg[3];
    details:=arg[4];

    map:=RegularCWMapToCWSubcomplex(arg[1]);
    sub:=SortedList(map[2][Length(map[2])]);
    sub:=List(sub,x->x-Position(sub,x)+1);
    clsr:=ClosureCWCell(map[1],2,sub[1])[2];;
    seq:=CWSubcomplexToRegularCWMap([map[1],clsr]);;
    tub:=RegularCWComplexComplement(seq,method,subdiv,details);
    for i in [2..Length(sub)] do
```

```
        clsr:=ClosureCWCell(tub,2,sub[i])[2];;
        seq:=CWSubcomplexToRegularCWMap([tub,clsr]);;
        tub:=RegularCWComplexComplement(seq,method,subdiv,details);
    od;

    return tub;
end);
```

B.27 LiftColouredSurface

```
InstallGlobalFunction(
    LiftColouredSurface,
    function(f)
    local
        src, trg, lens, colours,
        cbnd4_1cells, cbnd4_1cells_bnd,
        i, j, clr, closure, k,
        min, max, l_colours,
        cell, col1, col2, int,
        bnd, bbnd, l, cobnd4,
        l_src, prods;

    trg:=RegularCWMapToCWSubcomplex(f);
    src:=trg[2]*1;
    trg:=trg[1];
    lens:=List(
        [1..Length(trg!.boundaries)-1],
        x->Length(trg!.boundaries[x])
    );
    colours:=List([1..Length(src)],x->List([1..trg!.nrCells(x-1)],y->[]));
    cbnd4_1cells:=[]; # list of all 1-cells of src whose coboundary
    # consists of 4 2-cells
    cbnd4_1cells_bnd:=[]; # the boundary of the above 1-cells
    for i in [1..Source(f)!.nrCells(1)] do
        if Source(f)!.coboundaries[2][i][1]=4 then
            Add(cbnd4_1cells,src[2][i]);
            for j in [2,3] do
```

```
            Add(
                cbnd4_1cells_bnd,
                src[1][Source(f)!.boundaries[2][i][j]]
            );
        od;
    fi;
od;
cbnd4_1cells_bnd:=Set(cbnd4_1cells_bnd);
# we want to compute the inclusion src* -> trg x I where
# I is the interval [min,max] and where min and max are the
# smallest/largest integers that f!.colour assigns to 2-cells
#       we begin by associating to each cell e^n of src a list
# of integers e.g. [-1,2] indicating that e^n x {-1} and
# e^n x {2} will be in src x I
for i in [1..Length(src[Length(src)])] do
    clr:=f!.colour(Length(src)-1,src[Length(src)][i]);
    closure:=ClosureCWCell(trg,Length(src)-1,src[Length(src)][i])[2];
    for j in [1..Length(closure)] do
        for k in [1..Length(closure[j])] do
            colours[j][closure[j][k]]:=Set(
                Concatenation(
                    colours[j][closure[j][k]],
                    clr
                )
            );
        od;
    od;
od;
min:=Minimum(
    List(Filtered(colours[Length(colours)],x->x<>[]),Minimum)
);
# ^smallest & \/largest 'colours'
max:=Maximum(
    List(Filtered(colours[Length(colours)],x->x<>[]),Maximum)
);
# make a copy of trg for each colour in [min-1,min+1]
trg:=trg!.boundaries*1; Add(trg,[]);
l_colours:=List(
    [1..Length(trg)],
```

```
        x->List([1..Length(trg[x])],
            y->[[x-1,y],min-1]
        )
    );
    for i in [min..max+1] do
        for j in [1..lens[1]] do
            Add(trg[1],[1,0]);
            Add(l_colours[1],[[0,j],i]);
        od;
    od;
    for i in [1..Length([min..max+1])] do
        for j in [1..Length(lens)-1] do
            for k in [1..lens[j+1]] do
                cell:=trg[j+1][k]*1;
                cell:=Concatenation(
                    [cell[1]],
                    cell{[2..Length(cell)]}+lens[j]*i
                );
                Add(trg[j+1],cell);
                Add(l_colours[j+1],[[j,k],[min..max+1][i]]);
            od;
        od;
    od;
    # 'join' each Target(f) x {i-1} to Target(f) x {i}
    for i in [1..Length(lens)] do
        for j in [1..lens[i]] do
            cell:=[i-1,j];
            for k in [1..Length([min-1..max])] do
                col1:=[min-1..max][k];
                col2:=[min-1..max+1][k+1];
                int:=[col1,col2];
                bnd:=[]; # boundary of cell x int
                Add(
                    bnd,
                    Position(l_colours[i],[cell,col1])
                );
                Add(
                    bnd,
                    Position(l_colours[i],[cell,col2])
```

```
            );
            if i>1 then
                bbnd:=trg[i][j]*1;
                bbnd:=bbnd{[2..bbnd[1]+1]};
                bbnd:=List(
                    bbnd,
                    x->[i-2,x]
                );
                for l in [1..Length(bbnd)] do
                    Add(
                        bnd,
                        Position(l_colours[i],[bbnd[l],int])
                    );
                od;
            fi;
            bnd:=Set(bnd); Add(bnd,Length(bnd),1);
            Add(trg[i+1],bnd);
            Add(l_colours[i+1],[cell,int]);
        od;
    od;
od;
# identify the correct subcomplex of X x [min-1,max+1]
cobnd4:=[]; # there are some 1-cells which shouldn't be lifted
l_src:=List(src,x->[]);
for i in [1..Length(colours)] do
    for j in [1..Length(colours[i])] do
        if colours[i][j]<>[] then
            prods:=[];
            if Length(colours[i][j])=1 then
                Add(prods,colours[i][j][1]);
            else
                int:=[Minimum(colours[i][j]),Maximum(colours[i][j])];
                for k in [2..Length(int)] do
                    Add(prods,int[k-1]);
                    if not (i=2 and j in cbnd4_1cells) and
                        not (i=1 and j in cbnd4_1cells_bnd) then
                        Add(prods,[int[k-1],int[k]]);
                    fi;
                    Add(prods,int[k]);
```

```
                    od;
                    prods:=Set(prods);
                fi;
                for k in [1..Length(prods)] do
                    if IsInt(prods[k]) then
                        Add(
                            l_src[i],
                            Position(
                                l_colours[i],
                                [[i-1,j],prods[k]]
                            )
                        );
                    else
                        Add(
                            l_src[i+1],
                            Position(
                                l_colours[i+1],
                                [[i-1,j],prods[k]]
                            )
                        );
                    fi;
                od;
            fi;
        od;
    od;
    return CWSubcomplexToRegularCWMap(
        [
            RegularCWComplex(trg),
            l_src
        ]
    );
end);
```

B.28 ViewArc2Presentation

```
InstallGlobalFunction(
    ViewArc2Presentation,
```

```
function(l)
local
    arc, cross, cols, AppendTo, PrintTo, file, filetxt,
    bin_arr, coord, res, colours, i, j, x, k, y, z, clr;

arc:=List(l[1],x->[SignInt(x[1])*x[1],SignInt(x[2])*x[2]]);
cross:=l[2]*1;
cols:=l[3]*1;
AppendTo:=HAP_AppendTo;
PrintTo:=HAP_PrintTo;
file:="/tmp/HAPtmpImage";
filetxt:="/tmp/HAPtmpImage.txt";

# create a binary array from the arc presentation
bin_arr:=Sum(PureCubicalKnot(arc)!.binaryArray);
bin_arr:=TransposedMat(bin_arr); # graham has this backwards!
bin_arr:=List(
    bin_arr{[2..Length(bin_arr)-3]},
    x->x{[2..Length(bin_arr[1])-3]}
);
coord:=Concatenation(
    List(
        [1..Length(bin_arr)],
        x->List(
            [1..Length(bin_arr)],
            y->[x,y]
        )
    )
);
coord:=Filtered(coord,x->bin_arr[x[1]][x[2]]=2);

res:=String(Minimum(50*Length(bin_arr),950));

PrintTo(
    filetxt,
    "# ImageMagick pixel enumeration: ",
    Length(bin_arr)+2,
    ",",
    Length(bin_arr[1])+2,
```

```
    ",255,RGB\n"
);

colours:=NewDictionary([0,""],true);
colours!.entries:=[
    [0,"(255,255,255)"], # white
    [1,"(131,205,131)"], # green
    [2,"(131,205,131)"], # green
    [3,"(131,205,131)"], # green
    [4,"(205,131,131)"], # red
    [5,"(131,131,205)"], # blue
];

for i in [1..Length(bin_arr)] do
    for j in [1..Length(bin_arr)] do
        if bin_arr[i][j]=3 then
        # remove vertical bars that have -entry in l[1]
            x:=-1*(Int(j/3)+1);
            if x in Concatenation(l[1]) then
                for k in [i+1..Length(bin_arr)] do
                    if bin_arr[k][j]=1 then
                        bin_arr[k][j]:=0;
                    fi;
                od;
            fi;
        elif bin_arr[i][j]=2 then
        # check crossing type and adjust surrounding pixels
            x:=Position(coord,[i,j]);
            y:=cross[x];
            if y=-1 then
                bin_arr[i][j-1]:=0;
                bin_arr[i][j+1]:=0;
            elif y=1 then
                bin_arr[i-1][j]:=0;
                bin_arr[i+1][j]:=0;
            elif y=0 then
                z:=cols[Position(Positions(cross,0),x)];
                if z=1 then
                    bin_arr[i][j-1]:=5;
```

```
                    bin_arr[i][j+1]:=5;
                elif z=2 then
                    bin_arr[i][j-1]:=5;
                    bin_arr[i][j+1]:=4;
                elif z=3 then
                    bin_arr[i][j-1]:=4;
                    bin_arr[i][j+1]:=5;
                elif z=4 then
                    bin_arr[i][j-1]:=4;
                    bin_arr[i][j+1]:=4;
                fi;
            fi;
        fi;
    od;
od;
# image gets cropped if i dont do this
for i in [1..2] do
    Add(bin_arr,bin_arr[1]*0);
od;
for j in [1..Length(bin_arr)] do
    bin_arr[j]:=Concatenation(bin_arr[j],[0,0]);
od;
#return bin_arr;
# swap each entry of the binary area with an rgb value
for i in [1..Length(bin_arr)] do
    for j in [1..Length(bin_arr[i])] do
        clr:=LookupDictionary(colours,bin_arr[i][j]);
        AppendTo(filetxt,j,",",i,": ",clr,"\n");
    od;
od;

# convert the binary array to an upscaled 500x500 png
Exec(
    Concatenation(
        "convert ",filetxt," ","-scale ",res,"x",res," ",file,".png"
    )
);
# delete the old txt file
Exec(
```

```
        Concatenation("rm ",filetxt)
    );
    # display the image
    Exec(
        Concatenation(DISPLAY_PATH," ","/tmp/HAPtmpImage.png")
    );
    # delete the image on window close
    Exec(
        Concatenation("rm  ","/tmp/HAPtmpImage.png")
    );
end);
```

B.29 NumberOfCrossingsInArc2Presentation

```
InstallGlobalFunction(
    NumberOfCrossingsInArc2Presentation,
    function(arc)
    local
        mat, i, x, j;

    mat:=List([1..Length(arc[1])],x->[1..Length(arc[1])]*0);
    for i in [1..Length(arc[1])] do
        mat[Length(mat)-i+1][AbsoluteValue(arc[1][i][1])]:=3;
        mat[Length(mat)-i+1][AbsoluteValue(arc[1][i][2])]:=3;
    od;
    for x in [1,2] do
        if x=2 then
            mat:=MutableTransposedMat(mat);
        fi;
        for i in [1..Length(mat)] do
            for j in [2..Length(mat)-1] do
                if 3 in mat[i]{[1..j-1]} and
                3 in mat[i]{[j+1..Length(mat)]} then
                    mat[i][j]:=mat[i][j]+1;
                fi;
            od;
        od;
```

```
    od;

    return Sum(List(mat,y->Length(Filtered(y,z->z=2))));
end);
```

B.30 RandomArc2Presentation

```
InstallGlobalFunction(
    RandomArc2Presentation,
    function()
    local
        len, rand, prime, cross, final_arc, i, arc;

    len:=1;
    rand:=Random([0,1]);

    while rand=1 do
        len:=len+1;
        rand:=Random([0,1]);
    od;

    prime:=[0,0,1,1,2,3,7,21,49,165,552];

    final_arc:=Random([3..11]);
    final_arc:=[final_arc,Random([1..prime[final_arc]])];
    final_arc:=ArcPresentation(
        PureCubicalKnot(final_arc[1],final_arc[2])
    );

    for i in [2..len] do
        arc:=Random([3..11]);
        arc:=[arc,Random([1..prime[arc]])];
        final_arc:=ArcPresentation(
            KnotSum(
                PureCubicalKnot(final_arc),
                PureCubicalKnot(arc[1],arc[2])
            )
```

```
        );
    od;

    cross:=List(
        [1..NumberOfCrossingsInArc2Presentation([final_arc,[],[]])],
        x->Random([-1,0,1])
    );

    final_arc:=[
        final_arc,
        cross,
        List([1..Length(Filtered(cross,x->x=0))],y->Random([1..4]))
    ];

    return final_arc;
end);
```

B.31 Tube

```
InstallGlobalFunction(
    Tube,
    function(arc)
    local
        ribbon;

    arc:=KinkArc2Presentation(arc);

    ribbon:=ArcDiagramToTubularSurface(arc);
    ribbon:=LiftColouredSurface(ribbon);
    ribbon:=SequentialRegularCWComplexComplement(
        ribbon,"some","none",false
    );

    return ribbon;
end);
```

B.32 ThreeManifoldViaDehnSurgery†

```
InstallGlobalFunction(
    ThreeManifoldViaDehnSurgery,
    function(K,pp,qq)
    local
        start, edges, vertices, faces, mins, fn, TORUS, ffmod, fffmod,
        fmod, mymod, loop, B3, diff, BALL, MiddleEdge, LeftVert,
        RightVert, bool, lloop, pos, G1, G, F1, F, loop2, loopvertices,
        Bvertices, LB, Lvertices,Tvertices, Rvertices, cdba, dcab, bacd,
        abdc, CDBA, DCAB, BACD, ABDC, DB, BD, CA, AC, db, bd, ac, dc, cd,
        ca, DC, CD, BA, AB, ab, ba, Vd, VD, VC, VB, Vc, Vb, Va, VA, T3,
        Pdcab, Pcdba, Pbacd, Pabdc, Pbd, Pca, Pac, Pcd, Pba, Pdb, Pdc,
        Pab, PVd, PVc, PVb, PVa, Vabdc, vabdc, sqboundary, BR1, BR0, BL1,
        BL0, TR1, TL0, TL1, TR0, xtrballs, xtrfaces, xtrloop, xtrFaces,
        xtrvertices, Faces, SGN, A, B, e, ee, f, i, k, L, LL, m, M, MB, p,
        P, q, Q, s, ss, S, T, v, W, x, Y;

    p:=qq; q:=pp; # Better stick to conventions!
    SGN:=SignInt(p*q)<0;
    p:=AbsInt(p); q:=AbsInt(q);
    p:=p/Gcd(p,q); q:=q/Gcd(p,q);

    m:=p*q;
    L:=[];
    x:=[0,0];
    start:=[0,0];
    edges:=[];

    while not x in L do
        Add(L,x);
        x:=[
            (x[1]+p) mod m,
            (x[2]+q) mod m
        ];
        if x[1]=0 and x[2]>0 then
            Add(L,1*[m,x[2]]);
            Add(edges,SortedList([start,1*[m,x[2]]]));
```

```
        start:=1*x;
    fi;
    if x[2]=0 and x[1]>0 then
        Add(L,[x[1],m]);
        Add(edges,SortedList([start,1*[x[1],m]]));
        start:=1*x;
    fi;
od;
Add(L,[m,m]);
Add(edges,SortedList([start,1*[m,m]]));


#GenLoop:=1*edges;
#This is the loop determining the torus homeomorphism
edges:=SSortedList(edges);

vertices:=[];
# This will be the ordered list of vertices in the torus fd
faces:=[];
for x in edges do
    AddSet(vertices,x[1]);
    AddSet(vertices,x[2]);
od;
AddSet(vertices,[0,m]);
AddSet(vertices,[m,0]);
AddSet(vertices,[m,m]);


Bvertices:=Filtered(vertices,x->x[2]=0);
Tvertices:=Filtered(vertices,x->x[2]=m);
Lvertices:=Filtered(vertices,x->x[1]=0);
Rvertices:=Filtered(vertices,x->x[1]=m);

for x in Rvertices do
    #AddSet(vertices, x+[p,0]);
    AddSet(edges, SortedList(1*[x,x+[p,0]]));
    AddSet(edges, SortedList(1*[x,x+[0,q]]));
    AddSet(edges, SortedList(1*[x+[p,0],x+[p,q]]));
od;
```

```
for x in Tvertices do
    #AddSet(vertices, 1*[x,x+[0,q]]);
    AddSet(edges, SortedList(1*[x,x+[0,q]]));
    AddSet(edges, SortedList(1*[x,x+[p,0]]));
    AddSet(edges, SortedList(1*[x+[0,q],x+[p,q]]));
od;

for x in Lvertices do
    AddSet(edges, SortedList(1*[x,x+[0,q]]));
od;

for x in Bvertices do
    AddSet(edges, SortedList(1*[x,x+[p,0]]));
od;

#AddSet(vertices,[m+p,m+q]);
#AddSet(edges,SortedList([[m,m],[m+p,m]]));
AddSet(edges,SortedList([[m+p,m],[m+p,m+q]]));

vertices:=[];
for e in edges do
    Append(vertices,1*e);
od;
vertices:=SSortedList(vertices);

Y:=RegularCWDiscreteSpace(Length(vertices));
for e in edges do
    RegularCWComplex_AttachCellDestructive(
        Y,
        1,
        [Position(vertices,e[1]),Position(vertices,e[2])]
    );
od;

for i in [0..q-1] do
    P:=[];
    Add(P,Position(edges,[[i*p,m],[(i+1)*p,m]]));
    Add(P,Position(edges,[[i*p,m+q],[(i+1)*p,m+q]]));
    Add(P,Position(edges,[[i*p,m],[i*p,m+q]]));
```

```
    Add(P,Position(edges,[[(i+1)*p,m],[(i+1)*p,m+q]]));
    Add(faces,P);
    RegularCWComplex_AttachCellDestructive(Y,2,SortedList(P));
od;

for i in [0..p] do
    P:=[];
    Add(P,Position(edges,[[m,i*q],[m,(i+1)*q]]));
    Add(P,Position(edges,[[m+p,i*q],[m+p,(i+1)*q]]));
    Add(P,Position(edges,[[m,i*q],[m+p,i*q]]));
    Add(P,Position(edges,[[m,(i+1)*q],[m+p,(i+1)*q]]));
    Add(faces,P);
    RegularCWComplex_AttachCellDestructive(Y,2,SortedList(P));
od;


LL:=[];
for v in Lvertices do
    L:=Filtered(edges, x->v in x);
    for x in L do
        if (not x[1][1]=x[2][1]) and (not x[1][2]=x[2][2]) then
            Add(LL,x);
        fi;
    od;
od;

LB:=[];
for v in Bvertices do
    L:=Filtered(edges, x->v in x);
    for x in L do
        if (not x[1][1]=x[2][1]) and (not x[1][2]=x[2][2]) then
            Add(LB,x);
        fi;
    od;
od;

for i in [0..p-2] do
    P:=[];
    x:=[[0,i*q],[0,(i+1)*q]];
```

```
        Add(P,x);
        pos:=Position(Lvertices,x[1]);
        if pos=fail then Print("OOPS\,"); fi;
        Add(P,LL[pos]);
        pos:=Position(Lvertices,x[2]);
        if pos=fail then Print("OOPS\,"); fi;
        Add(P,LL[pos]);
        x:=SortedList([P[2][2], P[3][2]]);
        Add(P,x);
        P:=List(P,a->Position(edges,a));
        Add(faces,P);
        RegularCWComplex_AttachCellDestructive(Y,2,P);
    od;
    i:=p-1;
    P:=[];
    x:=[[0,i*q],[0,(i+1)*q]];
    Add(P,x);
    pos:=Position(Lvertices,x[1]);
    if pos=fail then Print("OOPS\,"); fi;
    Add(P,LL[pos]);
    x:=[[0,m],[p,m]];
    Add(P,x);
    P:=List(P,a->Position(edges,a));
    Add(faces,P);
    RegularCWComplex_AttachCellDestructive(Y,2,P);

    for i in [0..q-2] do
        P:=[];
        x:=[[i*p,0],[(i+1)*p,0]];
        Add(P,x);
        pos:=Position(Bvertices,x[1]);
        if pos=fail then Print("OOPS\,"); fi;
        Add(P,LB[pos]);
        pos:=Position(Bvertices,x[2]);
        if pos=fail then Print("OOPS\,"); fi;
        Add(P,LB[pos]);
        x:=SortedList([P[2][2],P[3][2]]);
        Add(P,x);
        P:=List(P,a->Position(edges,a));
```

```
    Add(faces,P);
    RegularCWComplex_AttachCellDestructive(Y,2,P);
od;

P:=[];
x:=[[m-p,0],[m,0]];
Add(P,x);
pos:=Position(Bvertices,x[1]); if pos=fail then Print("OOPS\,"); fi;
Add(P,LB[pos]);
x:=[[m,0],[m,q]];
Add(P,x);
P:=List(P,a->Position(edges,a));
Add(faces,P);
RegularCWComplex_AttachCellDestructive(Y,2,P);

#####################################
vertices:=10*vertices;
edges:=10*edges;
edges:=List(edges,x->SortedList(x));
# faces stays unchanged
#####################################

loop:=[];
for x in edges do
    if x[1][1]<>x[2][1] and x[1][2]<>x[2][2] then
        Add(loop,x);
    fi;
od;

loopvertices:=List(loop,x->1*x[2]);
for v in loopvertices do
    if v[1]=10*m then
        Add(loop,[v,v+[10*p,0]]);
    fi;
    if v[2]=10*m then
        Add(loop,[v,v+[0,10*q]]);
    fi;
od;
pos:=Position(loop,10*[[m,m],[m+p,m]]);
```

```
Remove(loop,pos);
Add(loop,[[m,0],[m+p,0]]*10);

loop2:=[];
if p<q then
    Add(loop2,[10*[m,m]-[0,q],10*[m,m]+[p,0]]);
fi;
for x in loop do
    if x[1][1]=0 and x[1][2]>0 and x[2][1]<10*m then
        Add(loop2,[x[1]-[0,q],x[2]+[p,0]]);
    fi;
    if x[1][1]=0 and x[1][2]>0 and x[2][1]=10*m then
        Add(loop2,[x[1]-[0,q],x[2]+[0,-q]]);
        if x[2][2]=10*m then
            Add(loop2,[x[2]+[0,-q],x[2]+[p,0]]);
        fi;
    fi;

    if x[1][2]=0 and x[1][1]<10*m and x[2][1]=10*m then
        Add(loop2,[x[1]+[p,0],x[2]-[0,q]]);
    fi;

    if x[1][2]=0 and x[1][1]<10*m and x[2][1]<10*m then
        Add(loop2,[x[1]+[p,0],x[2]+[p,0]]); # NEW
    fi;

    if x[1][2]=10*m and x[1][1]<10*m then
        Add(loop2,[x[1]+[p,0],x[2]+[p,0]]);
    fi;
    if x[1][2]=10*m and x[1][1]=10*m then
        Add(loop2,[x[1]+[p,0],x[2]+[p,-q]]);
    fi;

    if x[1][1]=10*m and x[1][2]>0 and x[1][2]<10*m then
        Add(loop2,[x[1]+[0,-q],x[2]+[0,-q]]);
    fi;
od;

Add(loop2,[[m,m+q],[m+p,m+q]]*10);
```

```
Add(loop2,[10*[m,m+q]+[p,-q],10*[m+p,m+q]-[0,q]]);
Add(loop2,[[0,m+q]*10-[0,q],[0,m+q]*10+[p,0]]);

for i in Concatenation([0..p-2],[p]) do
    Add(loop2,[10*[0,i*q],10*[0,(i+1)*q]-[0,q]]);
od;
Add(loop2,[10*[0,(p-1)*q],10*[0,p*q]]);
Add(loop2,[10*[0,(p+1)*q]-[0,q],10*[0,p*q+q]]);

for i in [0..p-1] do
    Add(loop2,[10*[m,i*q],10*[m,(i+1)*q]-[0,q]]);
od;

for i in Concatenation([0..p-2],[p]) do
    Add(loop2,[10*[m+p,i*q],10*[m+p,(i+1)*q]-[0,q]]);
od;
Add(loop2,[10*[m+p,(p-1)*q],10*[m+p,p*q]]);
Add(loop2,[10*[m+p,(p+1)*q]-[0,q],10*[m+p,p*q+q]]);

for i in [0..q-1] do
    Add(loop2,[10*[i*p,0]+[p,0],10*[(i+1)*p,0]]);
od;

for i in [0..q-1] do
    Add(loop2,[10*[i*p,m+q]+[p,0],10*[(i+1)*p,m+q]]);
od;
Add(loop2,[10*[0,m+q],10*[0,m+q]+[p,0]]);

for i in [1..q] do
    Add(loop2,[10*[i*p,m]+[p,0],10*[(i+1)*p,m]]);
od;
Add(loop2,[10*[0,m],10*[p,m]]);

for i in [1..p-1] do
    Add(loop2,[10*[0,i*q]+[0,-q],10*[0,i*q]]);
    Add(loop2,[10*[m,i*q]+[0,-q],10*[m,i*q]));
    Add(loop2,[10*[m+p,i*q]+[0,-q],10*[m+p,i*q]]);
od;
i:=p;
```

```
Add(loop2,[10*[m,i*q]+[0,-q],10*[m,i*q]]);


for i in [0..q-1] do
    Add(loop2,[10*[i*p,0],10*[i*p,0]+[p,0]]);
od;
for i in [1..q] do
    Add(loop2,[10*[i*p,m],10*[i*p,m]+[p,0]]);
od;
for i in [1..q-1] do
    Add(loop2,[10*[i*p,m+q],10*[i*p,m+q]+[p,0]]);
od;

loop:=Concatenation(loop,loop2);
loop:=List(loop,x->SortedList(x));

#########################################################################
#########################################################################
## VISUAL CHECK THAT THE COMPLEX IS CORRECT
S:=[];

#for e in edges do
for e in loop do
    if (not e[1][1]=e[2][1] ) and (not e[1][2]=e[2][2]) then
        x:=10*e[1];
        #while x[1]<= 10*m*10 and x[2]<=10*m*10 do
        while x[1]<= 10*e[2][1] and x[2]<=10*e[2][2] do
            Add(S,x);
            x:=x+[p,q];
        od;
    fi;
    if e[1][1]=e[2][1] then
        for i in [e[1][2]..e[2][2]] do
            Add(S,[10*e[2][1],10*i]);
        od;
    fi;
    if e[1][2]=e[2][2] then
        for i in [e[1][1]..e[2][1]] do
            Add(S,[10*i,10*e[2][2]]);
```

```
        od;
    fi;
od;
return Scatterplot(S);
#########################################################################
#########################################################################

vertices:=[];
for e in loop do
    Append(vertices,1*e);
od;
vertices:=SSortedList(vertices);

Y:=RegularCWDiscreteSpace(Length(vertices));
for e in loop do
    RegularCWComplex_AttachCellDestructive(
        Y,
        1,
        [Position(vertices,e[1]), Position(vertices,e[2])]
    );
od;

TL0:=[];
TL1:=[];
TR0:=[];
TR1:=[];
BL0:=[];
BL1:=[];
BR0:=[];
BR1:=[];

P:=[];;
#Add(P,[10*[0,m+q]-[0,q],10*[0,m+q]]);
#Add(P,[10*[0,m+q],10*[0,m+q]+[p,0]]);
#Add(P,[10*[0,m+q]-[0,q],10*[0,m+q]+[p,0]]);
#Add(TL0,P);

P:=[];;
Add(P,[10*[0,m],10*[0,m+q]-[0,q]]);
```

```
Add(P,[10*[0,m+q]-[0,q],10*[0,m+q]+[p,0]]);
Add(P,[10*[0,m+q]+[p,0],10*[p,m+q]]);
Add(P,[10*[0,m],10*[p,m]]);
Add(P,[10*[p,m],10*[p,m+q]]);
Add(TL0,P);

for i in [1..q-1] do
    P:=[];;
    Add(P,[10*[i*p,m]+[p,0],10*[i*p,m+q]+[p,0]]);
    Add(P,[10*[(i+1)*p,m],10*[(i+1)*p,m+q]]);
    Add(P,[10*[i*p,m]+[p,0],10*[(i+1)*p,m]]);
    Add(P,[10*[i*p,m+q]+[p,0],10*[(i+1)*p,m+q]]);
    Add(TL0,P);
od;

for i in [1..q-1] do
    P:=[];;
    Add(P,[10*[i*p,m],10*[i*p,m+q]]);
    Add(P,[10*[i*p,m]+[p,0],10*[i*p,m+q]+[p,0]]);
    Add(P,[10*[i*p,m],10*[i*p,m]+[p,0]]);
    Add(P,[10*[i*p,m+q],10*[i*p,m+q]+[p,0]]);
    Add(TL1,P);
od;

P:=[];
Add(P,[10*[0,m+q]-[0,q],10*[0,m+q]]);
Add(P,[10*[0,m+q],10*[0,m+q]+[p,0]]);
Add(P,[10*[0,m+q]-[0,q],10*[0,m+q]+[p,0]]);
Add(TL1,P);

P:=[];;
Add(P,[10*[m,m]+[p,0],10*[m,m+q]+[p,-q]]);
Add(P,[10*[m,m]+[p,0],10*[m+p,m]]);
Add(P,[10*[m,m+q]+[p,-q],10*[m+p,m+q]+[0,-q]]);
Add(P,[10*[m+p,m],10*[m+p,m+q]+[0,-q]]);
Add(TR0,P);

P:=[];;
Add(P,[10*[m,m],10*[m,m+q]]);
```

```
Add(P,[10*[m,m],10*[m,m]+[p,0]]);
Add(P,[10*[m,m+q]+[p,-q],10*[m+p,m+q]+[0,-q]]);
Add(P,[10*[m,m+q],10*[m+p,m+q]]);
Add(P,[10*[m,m]+[p,0],10*[m,m+q]+[p,-q]]);
Add(P,[10*[m+p,m+q]+[0,-q],10*[m+p,m+q]]);
Add(TR1,P);

for i in [0..p-2] do
    P:=[];
    Add(P,[10*[m,i*q],10*[m+p,i*q]]);
    Add(P,[10*[m,(i+1)*q]+[0,-q],10*[m+p,(i+1)*q]+[0,-q]]);
    Add(P,[10*[m,i*q],10*[m,(i+1)*q]-[0,q]]);
    Add(P,[10*[m+p,i*q],10*[m+p,(i+1)*q]-[0,q]]);
    Add(BR0,P);
od;

P:=[];
i:=p-1;
Add(P,[10*[m,i*q],10*[m,(i+1)*q]-[0,q]]);
Add(P,[10*[m,(i+1)*q]-[0,q],10*[m,(i+1)*q]+[p,0]]);
Add(P,[10*[m,(i+1)*q]+[p,0],10*[m+p,(i+1)*q]]);
Add(P,[10*[m,i*q],10*[m+p,i*q]]);
Add(P,[10*[m+p,i*q],10*[m+p,(i+1)*q]]);
Add(BR0,P);

for i in [1..p-1] do
    P:=[];
    Add(P,[10*[m,i*q]-[0,q],10*[m+p,i*q]-[0,q]]);
    Add(P,[10*[m,i*q],10*[m+p,i*q]]);
    Add(P,[10*[m,i*q]-[0,q],10*[m,i*q]]);
    Add(P,[10*[m+p,i*q]-[0,q],10*[m+p,i*q]]);
    Add(BR1,P);
od;

P:=[];
Add(P,[10*[m,m],10*[m,m]+[p,0]]);
Add(P,[10*[m,m]-[0,q],10*[m,m]]);
Add(P,[10*[m,m]-[0,q],10*[m,m]+[p,0]]);
Add(BR1,P);
```

```
F:=Filtered(loop,e->e[1][1]=0);
F:=Filtered(F,e->e[2][1]>0);
F1:=List(F,e->e[1]);

for i in [0..p-2] do
    P:=[];
    A:=10*[0,i*q];
    B:=10*[0,(i+1)*q]-[0,q];
    Add(P,[A,B]);
    pos:=Position(F1,A);
    Add(P,F[pos]);
    pos:=Position(F1,B);
    Add(P,F[pos]);
    Add(P,SortedList([P[2][2],P[3][2]]));
    Add(BL0,P);
od;

P:=[];
A:=10*[0,m-q];
B:=10*[0,m];
pos:=Position(F1,A);
Add(P,F[pos]);
Add(P,[A,B]);
Add(P,[B,10*[p,m]]);
Add(BL0,P);

G:=Filtered(loop,e->e[1][2]=0);
G:=Filtered(G,e->e[2][2]>0);
G1:=List(G,e->e[1]);

for i in [0..q-2] do
    P:=[];
    A:=10*[i*p,0]+[p,0];
    B:=10*[(i+1)*p,0];
    Add(P,[A,B]);
    pos:=Position(G1,A);
    Add(P,G[pos]);
    pos:=Position(G1,B);
```

```
    Add(P,G[pos]);
    Add(P,SortedList([P[2][2],P[3][2]]));
    Add(BL0,P);
od;

P:=[];
A:=10*[m-p,0]+[p,0];
B:=10*[m,0];
Add(P,[A,B]);
pos:=Position(G1,A);
Add(P,G[pos]);
Add(P,[P[1][2],P[2][2]]);
Add(BL0,P);

for i in [1..p-1] do
    P:=[];
    A:=10*[0,i*q]-[0,q];
    B:=10*[0,i*q];
    Add(P,[A,B]);
    pos:=Position(F1,A);
    Add(P,F[pos]);
    pos:=Position(F1,B);
    Add(P,F[pos]);
    Add(P, SortedList([P[2][2],P[3][2]]));
    Add(BL1,P);
od;

for i in [0..q-1] do
    P:=[];
    A:=10*[i*p,0];
    B:=10*[i*p,0]+[p,0];
    Add(P,[A,B]);
    pos:=Position(G1,A);
    Add(P,G[pos]);
    pos:=Position(G1,B);
    Add(P,G[pos]);
    Add(P, SortedList([P[2][2],P[3][2]]));
    Add(BL1,P);
od;
```

```
faces:=Concatenation(TL0,TL1,TR0,TR1,BL0,BL1,BR0,BR1);

##########################
mymod:=function(x,m);
    if x=infinity then return infinity; fi;
    if x=-infinity then return -infinity; fi;
return x mod m;
end;
##########################

##########################
fmod:=function(x);
    return [mymod(x[1],10*(m+p) ),mymod(x[2],10*(m+q))];
end;
##########################
ffmod:=function(e);
    return SortedList([fmod(e[1]),fmod(e[2])]);
end;
##########################
fffmod:=function(P);
    return SortedList(List(P,ffmod));
end;
##########################

vertices:=List(vertices,fmod);
vertices:=SSortedList(vertices);

lloop:=[];
for e in loop do
    Add(lloop,ffmod(e));
od;
loop:=lloop;
loop:=SSortedList(loop);

faces:=List(faces,fffmod);

Faces:=List(1*faces,P->List(P,e->Position(loop,1*e)));
```

```
Y:=RegularCWDiscreteSpace(Length(vertices));
for e in loop do
    RegularCWComplex_AttachCellDestructive(
        Y,
        1,
        SortedList([Position(vertices,e[1]),Position(vertices,e[2])])
    );
od;

for P in Faces do
    RegularCWComplex_AttachCellDestructive(Y,2,P);
od;

# Now add the thickened relator disk to Y=TORUS
LeftVert:=fmod([-infinity,-infinity]);
RightVert:=fmod([infinity,infinity]);
MiddleEdge:=ffmod([LeftVert,RightVert]);
xtrvertices:=[LeftVert,RightVert];
Append(vertices,xtrvertices);
xtrloop:=[MiddleEdge];
xtrfaces:=[];
xtrballs:=[];

for P in TL1 do
    BALL:=[fffmod(P)];
    if Size(P)=4 then
        for e in P do
            A:=e[1]; B:=e[2];
            if A[2]<B[2] and (A[1] mod (10*p)=0) then
                Add(xtrloop,ffmod([A,LeftVert]));
                Add(xtrloop,ffmod([B,LeftVert]));
                Q:=[];
                Add(Q,e);
                Add(Q,ffmod([A,LeftVert]));
                Add(Q,ffmod([B,LeftVert]));
                Add(xtrfaces,fffmod(Q));
                Add(BALL,fffmod(Q));
            fi;
            if A[2]<B[2] and not (A[1] mod (10*p)=0) then
```

```
                Add(xtrloop,ffmod([A,RightVert]));
                Add(xtrloop,ffmod([B,RightVert]));
                Q:=[];
                Add(Q,e);
                Add(Q,ffmod([A, RightVert]));
                Add(Q,ffmod([B,RightVert]));
                Add(xtrfaces,fffmod(Q));
                Add(BALL,fffmod(Q));
            fi;
            if A[2]=B[2]  then
                ee:=SortedList(e);
                Q:=[e,MiddleEdge,[ee[1],LeftVert],[ee[2],RightVert]];
                Q:=fffmod(Q);
                Add(xtrfaces,Q);
                Add(BALL,fffmod(Q));
            fi;
        od;
    fi;
    if Size(P)=3 then
        for e in P do
            A:=e[1]; B:=e[2];
            if A[1]=0 and B[1]=0 then
                Add(xtrloop,ffmod([B,LeftVert]));
                Add(xtrloop,ffmod([A,RightVert]));
                Q:=[e,[B,LeftVert],[A,RightVert],MiddleEdge];
                Q:=fffmod(Q);
                Add(xtrfaces,Q);
                Add(BALL,fffmod(Q));
            fi;
            if A[2]=10*(m+q) and B[2]=10*(m+q) then
                Add(xtrloop,ffmod([A,LeftVert]));
                Add(xtrloop,ffmod([B,RightVert]));
                Q:=[e,[A,LeftVert],[B,RightVert],MiddleEdge];
                Q:=fffmod(Q);
                Add(xtrfaces,Q);
                Add(BALL,fffmod(Q));
            fi;
            if (not A[1]=B[1]) and (not A[2]=B[2])  then
                Q:=[e,[A,RightVert],[B,RightVert]];
```

```
                Q:=fffmod(Q);
                Add(xtrfaces,Q);   ##########??
                Add(BALL,fffmod(Q));
            fi;
        od;
    fi;
    Add(xtrballs,BALL);
od;

for P in TR1 do
    BALL:=[fffmod(P)];
    for e in P do
        A:=e[1]; B:=e[2];
        bool:=true;
        if (A[1]=10*m and B[1]=10*m) or
            (A[2]=10*(m+q) and B[2]=10*(m+q)) then
            bool:=false;
            Add(xtrloop,ffmod([A,LeftVert]));
            Add(xtrloop,ffmod([B,LeftVert]));
            Q:=[e,ffmod([A,LeftVert]),ffmod([B,LeftVert])];
            Q:=fffmod(Q);
            Add(xtrfaces,Q);
            Add(BALL,fffmod(Q));
        fi;
        if (A[1]=(10*m+p) and B[1]=(10*m+p)) or
            (A[2]=(10*m+9*q) and B[2]=(10*m+9*q)) then
            bool:=false;
            Add(xtrloop,ffmod([A,RightVert]));
            Add(xtrloop,ffmod([B,RightVert]));
            Q:=[e,ffmod([A,RightVert]),ffmod([B,RightVert])];
            Q:=fffmod(Q);
            Add(xtrfaces,Q);
            Add(BALL,fffmod(Q));
        fi;
        if bool then
            if A[1]=(10*m)  then
                Q:=[
                    e,
                    MiddleEdge,
```

```
                    ffmod([A,LeftVert]),
                    ffmod([B,RightVert])
                ];
                Q:=fffmod(Q);
                Add(xtrfaces,Q);   ############??
                Add(BALL,fffmod(Q));
            else
                Q:=[
                    e,
                    MiddleEdge,
                    ffmod([B,LeftVert]),
                    ffmod([A,RightVert])
                ];
                Q:=fffmod(Q);
                Add(xtrfaces,Q);  #################??
                Add(BALL,fffmod(Q));
            fi;
        fi;
    od;
    Add(xtrballs,BALL);
od;

for P in BR1 do
    BALL:=[fffmod(P)];
    if Size(P)=4 then
        for e in P do
            A:=e[1]; B:=e[2];
            if A[2]=B[2] and A[2] mod (10*q) =0 then
                Add(xtrloop,ffmod([A,LeftVert]));
                Add(xtrloop,ffmod([B,LeftVert]));
                Q:=[];
                Add(Q,e);
                Add(Q,ffmod([A, LeftVert]));
                Add(Q,ffmod([B,LeftVert]));
                Add(xtrfaces,fffmod(Q));
                Add(BALL,fffmod(Q));
            fi;
            if A[2]=B[2] and not (A[2]) mod (10*q) =0 then
                Add(xtrloop,ffmod([A,RightVert]));
```

```
                Add(xtrloop,ffmod([B,RightVert]));
                Q:=[];
                Add(Q,e);
                Add(Q,ffmod([A,RightVert]));
                Add(Q,ffmod([B,RightVert]));
                Add(xtrfaces,fffmod(Q));
                Add(BALL,fffmod(Q));
            fi;
            if A[1]=B[1] then
                Q:=[e,MiddleEdge,[A,RightVert],[B,LeftVert]];
                Add(xtrfaces,fffmod(Q));
                Add(BALL,fffmod(Q));
            fi;
        od;
    fi;
    if Size(P)=3 then
        for e in P do
            A:=e[1]; B:=e[2];
            if A[1]=10*m and B[1]=10*m then
                Add(xtrloop,[A,RightVert]);
                Add(xtrloop,[B,LeftVert]);
                Q:=[e,[A,RightVert],[B,LeftVert],MiddleEdge];
                Add(xtrfaces,fffmod(Q));
                Add(BALL,fffmod(Q));
            fi;
            if A[2]=10*m and B[2]=10*m then
                Add(xtrloop,[B,RightVert]);
                Add(xtrloop,[A,LeftVert]);
                Q:=[e,[B,RightVert],[A,LeftVert],MiddleEdge];
                Add(xtrfaces,fffmod(Q)); ########################??
                Add(BALL,fffmod(Q));
            fi;
            if (not A[1]=B[1]) and (not A[2]=B[2]) then
                Q:=[e,[A,RightVert],[B,RightVert]];
                Add(xtrfaces,fffmod(Q));  ########################??
                Add(BALL,fffmod(Q));
            fi;
        od;
    fi;
```

```
        Add(xtrballs,BALL);
    od;

    for P in BL1 do
        BALL:=[fffmod(P)];
        for e in P do
            A:=e[1]; B:=e[2];
            if (not A[1]=B[1]) and (not A[2]=B[2]) and
                (A[2] mod (10*q)=0) and (B[2] mod (10*q)=0) and
                    A[1] mod (10*p)=0 then # NEW#########################
                Add(xtrloop,ffmod([A,LeftVert]));
                Add(xtrloop,ffmod([B,LeftVert]));
                Q:=[e,[A,LeftVert],[B,LeftVert]];
                Add(xtrfaces,fffmod(Q));
                Add(BALL,fffmod(Q));
            fi;
            if (not A[1]=B[1]) and (not A[2]=B[2]) and
                (A[2] mod (10*q)=0) and (B[2] mod (10*q)=0) and
                    A[1] mod (10*p)=p then # NEW#########################
                Add(xtrloop,ffmod([A,RightVert]));
                Add(xtrloop,ffmod([B,RightVert]));
                Q:=[e,[A,RightVert],[B,RightVert]];
                Add(xtrfaces,fffmod(Q));
                Add(BALL,fffmod(Q));
            fi;
            if (not A[1]=B[1]) and (not A[2]=B[2]) and
                ((not A[2] mod (10*q)=0) or  (not B[2] mod (10*q)=0)) then
                Add(xtrloop,ffmod([A,RightVert]));
                Add(xtrloop,ffmod([B,RightVert]));
                Q:=[e,[A,RightVert],[B,RightVert]];
                Add(xtrfaces,fffmod(Q)); ###############??
                Add(BALL,fffmod(Q));
            fi;
            if A[1]=B[1] and A[1]=0 then
                Q:=[e,[A,RightVert],[B,LeftVert],MiddleEdge];
                Add(xtrfaces,fffmod(Q)); ##################??
                Add(BALL,fffmod(Q));
            fi;
            if A[1]=B[1] and A[1]=10*m then
```

```
            Q:=[e,[A,RightVert],[B,LeftVert],MiddleEdge];
            Add(xtrfaces,fffmod(Q)); ############################??
            Add(BALL,fffmod(Q));
        fi;
        if A[2]=B[2] and A[2]=0 then
            Q:=[e,[A,LeftVert],[B,RightVert],MiddleEdge];
            Add(xtrfaces,fffmod(Q)); ######################??
            Add(BALL,fffmod(Q));
        fi;
        if A[2]=B[2] and A[2]=10*m then
            Q:=[e,[A,LeftVert],[B,RightVert],MiddleEdge];
            Add(xtrfaces,fffmod(Q)); ##########################??
            Add(BALL,fffmod(Q));
        fi;
    od;
    Add(xtrballs,BALL);
od;

vertices:=List(vertices,fmod);
xtrloop:=List(xtrloop,e->ffmod(e));
xtrloop:=SSortedList(xtrloop);
Append(loop,xtrloop);
xtrfaces:=List(xtrfaces,f->fffmod(f));
xtrfaces:=SSortedList(xtrfaces);
xtrFaces:=List(xtrfaces,P->List(P,e->Position(loop,e)));
Append(Faces,xtrFaces);
Append(faces,xtrfaces);
xtrballs:=List(xtrballs,b->SortedList(b));
xtrballs:=SSortedList(xtrballs);
xtrballs:=List(xtrballs,P->SortedList(List(P,e->Position(faces,e))));

for v in xtrvertices do
    RegularCWComplex_AttachCellDestructive(Y,0);
od;

for e in xtrloop do
    RegularCWComplex_AttachCellDestructive(
        Y,
        1,
```

```
        SortedList([Position(vertices,e[1]),Position(vertices,e[2])])
    );
od;

for P in xtrFaces do
    RegularCWComplex_AttachCellDestructive(Y,2,P);
od;

for P in xtrballs do
    RegularCWComplex_AttachCellDestructive(Y,3,P);
od;

TORUS:=Y;
#return TORUS;

##   A----------B----------A
##   |          |          |
##   |          |          |
##   C----------D----------C
##   |          |          |
##   |          |          |
##   A----------B----------A

VA:=10*[0,m+q];;
VB:=10*[m,m+q];;
VC:=10*[0,m];;
VD:=10*[m,m];;
AB:=[];
for i in [0..q-1] do
    e:= [10*[i*p,m+q],10*[i*p,m+q]+[p,0]];
    Add(AB,e);
    f:=[10*[i*p,m+q]+[p,0],10*[(i+1)*p,m+q]];
    Add(AB,f);
od;
CD:=[ [10*[0,m],10*[p,m]]];
for i in [1..q-1] do
    e:= [10*[i*p,m],10*[i*p,m]+[p,0]];
    Add(CD,e);
    f:=[10*[i*p,m]+[p,0],10*[(i+1)*p,m]];
```

```
    Add(CD,f);
od;
BA:=[[10*[m,m+q],10*[m+p,m+q]]];
DC:=[[10*[m,m],10*[m,m]+[p,0]],[10*[m,m]+[p,0],10*[m+p,m]]];
CA:=[];
for i in [0..p-2] do
    e:=[10*[0,i*q],10*[0,(i+1)*q]+[0,-q]];
    Add(CA,e);
    f:=[10*[0,(i+1)*q]+[0,-q],10*[0,(i+1)*q]];
    Add(CA,f);
od;
Add(CA,[10*[0,(p-1)*q],10*[0,m]]);

DB:=[];
for i in [0..p-1] do
    e:=[10*[m,i*q],10*[m,(i+1)*q]+[0,-q]];
    Add(DB,e);
    f:=[10*[m,(i+1)*q]+[0,-q],10*[m,(i+1)*q]];
    Add(DB,f);
od;

AC:=[
    [10*[0,m],10*[0,m+q]-[0,q]],
    [10*[0,m+q]-[0,q],10*[0,m+q]]
];
BD:=[[10*[m,m],10*[m,m+q]]];

ABDC:=Concatenation(TL0,TL1);
BACD:=Concatenation(TR0,TR1);
CDBA:=Concatenation(BL0,BL1);
DCAB:=Concatenation(BR0,BR1);

Y:=SphericalKnotComplement(K);;
ss:=100000;;
s:=Size(BoundaryOfPureRegularCWComplex(Y));
while s<ss do
    ss:=s;
    Y:=RegularCWComplex(BarycentricSubdivision(Y));
    Y:=SimplifiedComplex(Y);
```

```
    s:=Size(BoundaryOfPureRegularCWComplex(Y));
od;

if s>16 then
    Print("PROBLEM: Torus has ",s," cells\n");
fi;
f:=BoundaryMap(Y);
T:=Source(f);
Y:=Target(f);

B3:=List(T!.boundaries[3],x->1*x{[2..5]});

abdc:=B3[1];   #I think this choice needs to be refined.
for P in B3{[2,3,4]} do
    if Intersection(abdc,P)=[] then
        dcab:=P;
        break;
    fi;
od;

diff:=Difference(B3,[abdc,dcab]);
vabdc:=List(abdc,i->T!.boundaries[2][i]{[2,3]});
sqboundary:=1*vabdc;
Vabdc:=[vabdc[1]]; vabdc:=Difference(vabdc,Vabdc);
for P in vabdc do
    if Size(Intersection(P,Vabdc[1]))>0 then
        Add(Vabdc,P);
        vabdc:=Difference(vabdc,Vabdc);
        break;
    fi;
od;
for P in vabdc do
    if Size(Intersection(P,Vabdc[2]))>0 then
        Add(Vabdc,P);
        vabdc:=Difference(vabdc,Vabdc);
        break;
    fi;
od;
Add(Vabdc,vabdc[1]);
```

```
 for i in [2,3,4] do
     if Vabdc[i-1][2]<>Vabdc[i][1] then
         Vabdc[i]:=Reversed(Vabdc[i]);
     fi;
 od;
 if Vabdc[1][2]<>Vabdc[2][1] then
     Vabdc[1]:=Reversed(Vabdc[1]);
 fi;

 Va:=Vabdc[1][1];
 Vb:=Vabdc[2][1];
 Vd:=Vabdc[3][1];
 Vc:=Vabdc[4][1];

 ab:=abdc[Position(sqboundary,SortedList(Vabdc[1]))];
L:=Filtered([1..8],i->T!.boundaries[2][i]{[2,3]}=SortedList(Vabdc[1]));
 ba:=Difference(L,[ab])[1];

 bd:=abdc[Position(sqboundary,SortedList(Vabdc[2]))];
L:=Filtered([1..8],i->T!.boundaries[2][i]{[2,3]}=SortedList(Vabdc[2]));
 db:=Difference(L,[bd])[1];

 cd:=abdc[Position(sqboundary,SortedList(Vabdc[3]))];
L:=Filtered([1..8],i->T!.boundaries[2][i]{[2,3]}=SortedList(Vabdc[3]));
 dc:=Difference(L,[cd])[1];

 ac:=abdc[Position(sqboundary,SortedList(Vabdc[4]))];
L:=Filtered([1..8],i->T!.boundaries[2][i]{[2,3]}=SortedList(Vabdc[4]));
 ca:=Difference(L,[ac])[1];

 B3:=List(B3,x->SSortedList(x));
 abdc:=Position(B3, SSortedList([ab,bd,cd,ac]));
 bacd:=Position(B3, SSortedList([ba,ac,dc,bd]));
 cdba:=Position(B3, SSortedList([cd,db,ab,ca]));
 dcab:=Position(B3, SSortedList([dc,ca,ba,db]));

 Va:=f!.mapping(0,Va);
 Vb:=f!.mapping(0,Vb);
 Vc:=f!.mapping(0,Vc);
```

```
Vd:=f!.mapping(0,Vd);
ab:=f!.mapping(1,ab);
ba:=f!.mapping(1,ba);
cd:=f!.mapping(1,cd);
dc:=f!.mapping(1,dc);
ac:=f!.mapping(1,ac);
ca:=f!.mapping(1,ca);
bd:=f!.mapping(1,bd);
db:=f!.mapping(1,db);
abdc:=f!.mapping(2,abdc);
bacd:=f!.mapping(2,bacd);
cdba:=f!.mapping(2,cdba);
dcab:=f!.mapping(2,dcab);

if SGN then
###########################################
PVa:=Vd;
PVb:=Vc;
PVc:=Vb;
PVd:=Va;
Pab:=cd;
Pba:=dc;
Pcd:=ab;
Pdc:=ba;
Pac:=bd;
Pca:=db;
Pbd:=ac;
Pdb:=ca;
Pabdc:=abdc;
Pbacd:=bacd;
Pcdba:=cdba;
Pdcab:=dcab;

Va:=PVa;
Vb:=PVb;
Vc:=PVc;
Vd:=PVd;
ab:=Pab;
ba:=Pba;
```

```
cd:=Pcd;
dc:=Pdc;
ac:=Pac;
ca:=Pca;
bd:=Pbd;
db:=Pdb;
abdc:=Pabdc;
bacd:=Pbacd;
cdba:=Pcdba;
dcab:=Pdcab;
##########################################
fi;

M:=RegularCWComplex_DisjointUnion(Y,TORUS);
MB:=1*M!.boundaries;

VA:=fmod(VA);
VB:=fmod(VB);
VC:=fmod(VC);
VD:=fmod(VD);
AB:=List(AB,ffmod);
AB:=List(AB,x->SortedList(x));
AB:=List(AB,e->Position(loop,e))+Y!.nrCells(1);
BA:=List(BA,ffmod);
BA:=List(BA,x->SortedList(x));
BA:=List(BA,e->Position(loop,e))+Y!.nrCells(1);
CD:=List(CD,ffmod);
CD:=List(CD,x->SortedList(x));
CD:=List(CD,e->Position(loop,e))+Y!.nrCells(1);
DC:=List(DC,ffmod);
DC:=List(DC,x->SortedList(x));
DC:=List(DC,e->Position(loop,e))+Y!.nrCells(1);
AC:=List(AC,ffmod);
AC:=List(AC,x->SortedList(x));
AC:=List(AC,e->Position(loop,e))+Y!.nrCells(1);
CA:=List(CA,ffmod);
CA:=List(CA,x->SortedList(x));
CA:=List(CA,e->Position(loop,e))+Y!.nrCells(1);
BD:=List(BD,ffmod);
```

```
BD:=List(BD,x->SortedList(x));
BD:=List(BD,e->Position(loop,e))+Y!.nrCells(1);
DB:=List(DB,ffmod);
DB:=List(DB,x->SortedList(x));
DB:=List(DB,e->Position(loop,e))+Y!.nrCells(1);
ABDC:=List(ABDC,fffmod);
BACD:=List(BACD,fffmod);
CDBA:=List(CDBA,fffmod);
DCAB:=List(DCAB,fffmod);

ABDC:=List(1*ABDC,P->SortedList(List(P,e->Position(loop,1*e))));
BACD:=List(1*BACD,P->SortedList(List(P,e->Position(loop,1*e))));
CDBA:=List(1*CDBA,P->SortedList(List(P,e->Position(loop,1*e))));
DCAB:=List(1*DCAB,P->SortedList(List(P,e->Position(loop,1*e))));
T3:=List(TORUS!.boundaries[3],x->x{[2..Length(x)]});;
ABDC:=List(ABDC,P->Position(T3,P));
BACD:=List(BACD,P->Position(T3,P));
CDBA:=List(CDBA,P->Position(T3,P));
DCAB:=List(DCAB,P->Position(T3,P));

#Reduce the number of 0-cells
MB[1]:=MB[1]{[1..Length(MB[1])-4]};
mins:=[Y!.nrCells(0)+Position(vertices,VA),
Y!.nrCells(0)+Position(vertices,VB),
Y!.nrCells(0)+Position(vertices,VC),
Y!.nrCells(0)+Position(vertices,VD)];
mins:=SSortedList(mins);
############################
fn:=function(k);
if k=Y!.nrCells(0)+Position(vertices,VA) then return Va; fi;
if k=Y!.nrCells(0)+Position(vertices,VB) then return Vb; fi;
if k=Y!.nrCells(0)+Position(vertices,VC) then return Vc; fi;
if k=Y!.nrCells(0)+Position(vertices,VD) then return Vd; fi;
if k<mins[1] then return k; fi;
if k<mins[2] then return k-1; fi;
if k<mins[3] then return k-2; fi;
if k<mins[4] then return k-3; fi;
return k-4;
end;
```

```
############################

MB[2]:=List(MB[2],x->Concatenation(
    [[x[1]],List(x{[2..x[1]+1]},k->fn(k))])
);

#Reduce the number of 1-cells
mins:=SSortedList([ab, ba, cd, dc, ac, ca, bd, db]);
############################
fn:=function(k);
if  k=ab then return AB; fi;
if  k=ba then return BA; fi;
if  k=cd then return CD; fi;
if  k=dc then return DC; fi;
if  k=ac then return AC; fi;
if  k=ca then return CA; fi;
if  k=bd then return BD; fi;
if  k=db then return DB; fi;
return k;
end;
############################
MB[3]:=List(MB[3],x->Concatenation(
    [[x[1]],List(x{[2..x[1]+1]},k->fn(k))])
);
MB[3]:=List(MB[3],x->Flat(x));
MB[3]:=List(MB[3],x->Concatenation(
    [[Length(x)-1],x{[2..Length(x)]}])
);
############################
fn:=function(k);
if k<mins[1] then return k; fi;
if k<mins[2] then return k-1; fi;
if k<mins[3] then return k-2; fi;
if k<mins[4] then return k-3; fi;
if k<mins[5] then return k-4; fi;
if k<mins[6] then return k-5; fi;
if k<mins[7] then return k-6; fi;
if k<mins[8] then return k-7; fi;
return k-8;
```

```
 end;
 ############################
MB[3]:=List(MB[3],x->Concatenation([x[1]],List(x{[2..Length(x)]},fn)));
 L:=[1..Length(MB[2])];
 L:=Difference(L,mins);
 MB[2]:=MB[2]{L};

 #Reduce the number of 2-cells;
 mins:=SSortedList([abdc, bacd, cdba, dcab]);
 ############################
 fn:=function(k);
 #return k;
 if k=abdc then return ABDC+Y!.nrCells(2); fi;
 if k=bacd then return BACD+Y!.nrCells(2); fi;
 if k=cdba then return CDBA+Y!.nrCells(2); fi;
 if k=dcab then return DCAB+Y!.nrCells(2); fi;
 return k;
 end;
 ############################
MB[4]:=List(MB[4],x->Concatenation([x[1]],List(x{[2..Length(x)]},fn)));
 MB[4]:=List(MB[4],x->Flat(x));
 MB[4]:=List(MB[4],x->Concatenation([Length(x)-1],x{[2..Length(x)]}));

 ############################
 fn:=function(k);
 if k<mins[1] then return k; fi;
 if k<mins[2] then return k-1; fi;
 if k<mins[3] then return k-2; fi;
 if k<mins[4] then return k-3; fi;
 return k-4;
 end;
 ############################
MB[4]:=List(MB[4],x->Concatenation([x[1]],List(x{[2..Length(x)]},fn)));
 L:=[1..Length(MB[3])];
 L:=Difference(L,mins);
 MB[3]:=MB[3]{L};


 W:=RegularCWComplex(MB);
```

```
    P:=Filtered([1..W!.nrCells(2)],k->W!.coboundaries[3][k][1]=1);;
    RegularCWComplex_AttachCellDestructive(W,3,P);

    return SimplifiedComplex(W);
end);
```