Groupoids and Computational Topology

PhD thesis

by

Nisreen Alokbi

Supervisor: Professor Graham Ellis

School of Mathematics, Statistics and Applied Mathematics

National University of Ireland, Galway

May 2019

Summary

This thesis contributes to the computational theory of finitely presented groupoids. It develops, implements and illustrates data types and algorithms aimed at pure and applied topology. In particular, the thesis designs and implements data types for:

- free groupoids (Data type 2.7.1),
- elements in free groupoids (Data type 2.7.2),
- finitely presented (fp) groupoids (Data type 2.7.3),
- homomorphisms of fp groupoids (Data type 2.7.4).

The thesis designs and implements algorithms for:

- composition of elements in a free groupoid (Algorithm 2.7.1),
- path components of a fp groupoid (Algorithm 2.7.3),
- a finite presentation for the vertex group of a fp groupoid (Algorithm 2.7.2),
- a finite presentation for finite index subgroups of an fp group (Algorithm 2.8.1),
- pushouts of fp groupoids (Algorithm 3.7.1),
- a finite presentation for the fundamental groupoid of a finite, regular CW-complex (Algorithm 3.5.1),
- the homomorphism of fundamental fp groupoids induced by an inclusion of finite regular CW-complexes (Section 3.6),

- the low-dimensional cup product on the cohomology of a finite regular CW-complexes (Chapter 5),
- a re-implementation of the Mapper algorithm for obtaining examples of finite simplicial complexes derived from experimental data (Algorithm 4.4.3),
- a re-implementation of an approximation for the dominant eigenvectors of a floating point symmetric matrix (for use with the Mapper algorithm) (Algorithm 4.4.1).

The thesis contains illustrations of the above data types and algorithms such as:

- the computation of a finite presentation of the fundamental group of a finite regular CW-complex based on the groupoid version of the van-Kampen theorem. This allows for parallel computation of low-dimensional cup products (Section 3.6),
- the fundamental groupoid (and group) of simplicial complexes arising, via Mapper, from gait analysis data (Section 4.6),
- the fundamental groupoid (and group) of simplicial complexes arising from time-series data (Section 3.8).

Contents

Declaration

I, Nisreen Alokbi, certify that the 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 this work.

Nisreen Alokbi

Certification

This is to certify that Nisreen Alokbi has complied with all the requirements for the submission of this Doctor of Philosophy thesis to the National University of Ireland, Galway.

Prof Graham Ellis

Acknowledgement

Praise be to God, Lord of the worlds, the Almighty, with whose gracious help it was possible to be accomplish this task.

I would like to capture this opportunity to express my honest appreciation and deep gratitude to my supervisor, Prof Graham Ellis, for his guidance and encouragement, without whose helpful suggestions and advices I would never finish.

To all of my officemates, all of you have provided a friendly and fun environment to work in, and I have enjoyed a room with all of you.

In addition, I am grateful to staff of the Department of Mathematics / NUI Galway for their interesting conservations in the common room.

It is my great pleasure to thank my husband Faik Mayah for long-suffering, patience and understanding which made it possible. I owe a great deal to my childreen, Sarah, Ali and Hassan who give me much happiness at all times.

It is a great privilege and pleasure to thank my parents for their support, encouragements, sacrifices and prayers throughout the years.

Finally, I would like to thank my Government (MOHE), Iraq, for their financial support.

Galway, May, 2019

Nisreen Alokbi

List of symbols

$\mathbb{Z}$	integer numbers $\{\ldots,-2,-1,0,1,2,\ldots\}$
$\mathbb{R}$	real line
M^n	n-dimensional manifold
GAP	Groups, Algorithms, Programming
HAP	GAP package - "Homological Algebra Programming"
FpGd	GAP package - "Finitely presented Groupoid"
$\square$	end of proof
D^n	closed unit n-ball
$\mathring{D}^n$	open unit n-ball
S^n	n-sphere
$H_n(X)$	n-dimensional homology of CW-space X
$H^n(X)$	n-dimensional cohomology of CW-space X
X^n	n-skeleton of a CW-space X
$C_*(X)$	cellular chain complex
Ker	kernel
Im	image
$\mathcal{C}_{\mathsf{set}}$	category of sets
$\mathcal{C}_{\mathsf{grp}}$	category of gorups
$\mathcal{C}_{\mathsf{top}}$	category of topologiacl spaces
$\mathcal{C}_{\mathsf{R\text{-}mod}}$	category of R-modules
$\mathcal{C}_{\mathsf{Vect}}$	category of vector spaces
$\mathcal{G}raphs$	category of directed graphs
$\mathcal{G}roupoids$	category of groupoids

$\mathcal{F}$	functor
$Obj(\mathcal{G})$	objects of groupoid $\mathcal{G}$
$Arr(\mathcal{G})$	arrows of groupoid $\mathcal{G}$
Γ	directed graph
$\mathcal{F}(\Gamma)$	free groupoid on Γ

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Aims of the thesis

It is well recognised that the fundamental groupoid of a topological space has an important role to play in the basic theory of algebraic topology concerning fundamental groups and covering spaces [16]. It also plays a role in the area of geometric group theory [56, 15]. For examples of how the fundamental groupoid can simplify traditional proofs, see for instance [17]. The main aim of this thesis is to develop computational aspects of groupoids and their use in computational algebra.

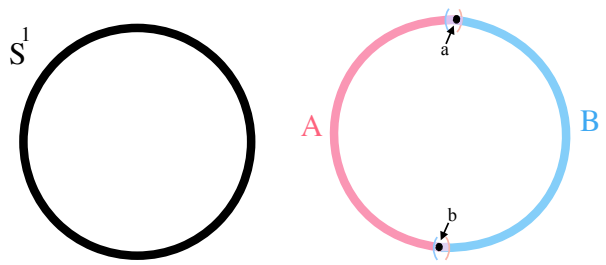
Computational algebra is an interdisciplinary area which helps with traditional research problems in mathematics as well as generating new research questions of its own. In the particular area of group theory, research in computational algebra has led to the creation of the two substantial computer algebra systems GAP [47] and Magma [8]. Some attempts have been focussed on applying computational algebra to questions in topology. The initial topological focus was on problems concerning calculations in cohomology groups [1, 7, 19, 45, 49, 60, 82], homotopy groups of spaces [10, 30] and geometric group theory (see [57] for an overview). More recently the focus has grown to include topics in the area of applied topology. In applied topology the basic idea is to investigate experimental data by associating cellular spaces to the data and then computing and studying and interpreting homotopical invariants of the associated spaces. To date most research in applied topology deals

with the most easily computed invariants of a space, namely its homology and cohomology. However, there has been some work on computing fundamental groups [31, 10]. Due to the large size of data sets in applied topology, efficiency of the algorithms is of paramount importance.

The main aim of this thesis is to develop the use of finitely presented groupoids in computational algebra. It is notable that none of the main computer algebra systems [GAP], [Magma], [Singular] and [Maple] has functionality for finitely presented groupoids and do not appear to use fp groupoids in their underlying algorithms. The Gpd package for GAP does provide basic computations in small finite groupoids. From the viewpoint of topology or geometric group theory one would like to compute with finitely presented infinite groupoids and to incorporate infinite groupoid techniques into computational algebra algorithms. There is also a case, which we make in this thesis, for using finitely presented infinite groupoids in applied topology.

A main contribution of this PhD work is a package for computing with finitely presented groupoids in the GAP system. We expect that the algorithms developed and implemented for the package should find use in theoretical topology, and geometric group theory. However, in this thesis we emphasise the role that the package can play in applied topology.

To make clear why groupoids can be useful, let us consider van Kampen's theorem - a standard tool for calculating fundamental groups. One of the most basic objects in topology is the circle S^1. Its fundamental group $\pi_1 S^1$ can not be computed using the standard version of van Kampen's theorem. If we consider S^1 as a union of two contractible connected components,



A and B, then $\pi_1 A = \pi_1 B = 1$ and so no algebraic construction produces $\pi_1 S^1 = \mathbb{Z}$

from two copies of the trivial group! However the fundamental groupoids $\pi_1(A, \{a, b\})$ and $\pi_1(B, \{a, b\})$ are non-trivial and are easily computed, and can be combined to calculate $\pi_1 S^1 \cong \mathbb{Z} \cong \pi_1(A, \{a, b\}) *_{\pi_1(A \bigcap B, \{a,b\})} \pi_1(B, \{a, b\})$

Applied topological methods have become important for developing algorithms for problems in data and shape analysis. For instance, topological methods are used for problems in topology inference [64, 86, 76, 91, 25], manifold reconstruction [26, 72, 48, 22, 9, 70, 58], Reeb graph construction [87, 85, 37, 78], homology cycle extractions [6, 24], and shape distance computations [83, 81, 79, 21].

In general, data in applied topology is large and often high-dimensional. Consequently, applied topology should benefit from work in the direction of a parallel computation to reduce the time and distribute the memory requirements of a computation.

The aim of this thesis is to develop and implement algorithms for finitely presented groupoids in order to calculate some of the topological invariants of spaces extracted from large data sets. The general goal is to design and implement algorithms that input large finite sets S of experimental data from an unknown manifold M and, using unsupervised learning, attempt to return homotopical invariants of M. One such algorithm involves Gunnar Carlsson's procedure. We use Mapper as a tool to produce a space from data. One goal is to describe how a computer implementation of the basic theory of finitely presented groupoids can be used to efficiently enhance the output of Mapper in a way that captures extra low-dimensional homotopy theoretic information. In particular, the enhanced output should be able to distinguish between data sets sampled from homotopy inequivalent spaces such as the cylinder, torus and Klein bottle. The enhanced output is not intended to distinguish between data from homotopy equivalent spaces such as the cylinder and Möbius strip [4].

1.2 Outline of the thesis

This thesis has six chapters.

Chapter 1 begins by outlining the aim of the thesis, namely methods for calculating the fundamental groupoids of spaces and using groupoid techniques in applied topology. It then reviews in Section 1.3 the standard material that will be used in the thesis.

Chapter 2 recalls the mathematical concepts of *groupoid, finitely presented groupoid, free groupoid* from Philip J. Higgins's paper [54]. It then goes on to describe our implementation of these concepts in GAP .

➢ Section 2.7 describes our implementations of free groupoids and finitely presented groupoids. Furthermore, it implements an algorithm for a finite presentation of the vertex group of a connected finitely presented groupoid.

➢ Section 2.8 implements an algorithm for determining the finite presentation for finite index subgroups of a fp group using groupoid techniques.

Chapter 3 recalls the mathematical concepts of *fundamental group, fundamental groupoid* and *van Kampen theorem* from [11] and [16].

➢ Section 3.5 describes our implementations of the fundamental groupoid for a connected regular CW-space.

➢ Section 3.6 describes our implementations of the van Kampen theorem.

➢ Section 3.8 builds a space from time-series data and then tests our implementations on this space.

Chapter 4 recalls the definition of the *Mapper cluster procedure* from [86]. The Mapper is used as a tool for constructing examples of simplicial complexes from real-life data.

➢ Section 4.4 describes our re-implementation of Mapper and re-implementation of an approximation for the dominant eigenvector of a floating point matrix that allows to use the *principal components analysis* PCA.

➢ Section 4.5 explains the limitations of Mapper by applying the procedure to different sets of data taken from homotopically equivalent spaces.

➢ Section 4.6 tests our implementation of Mapper on some experimental data taken from CMU Graphic Lab [66].

Chapter 5 is devoted to the distributed computation of low dimensional cup prod-

ucts. It builds on a practical algorithm for finding a finite presentation of the fundamental group $\pi_1(X, x_0)$ of an arbitrary finite regular CW-space X.

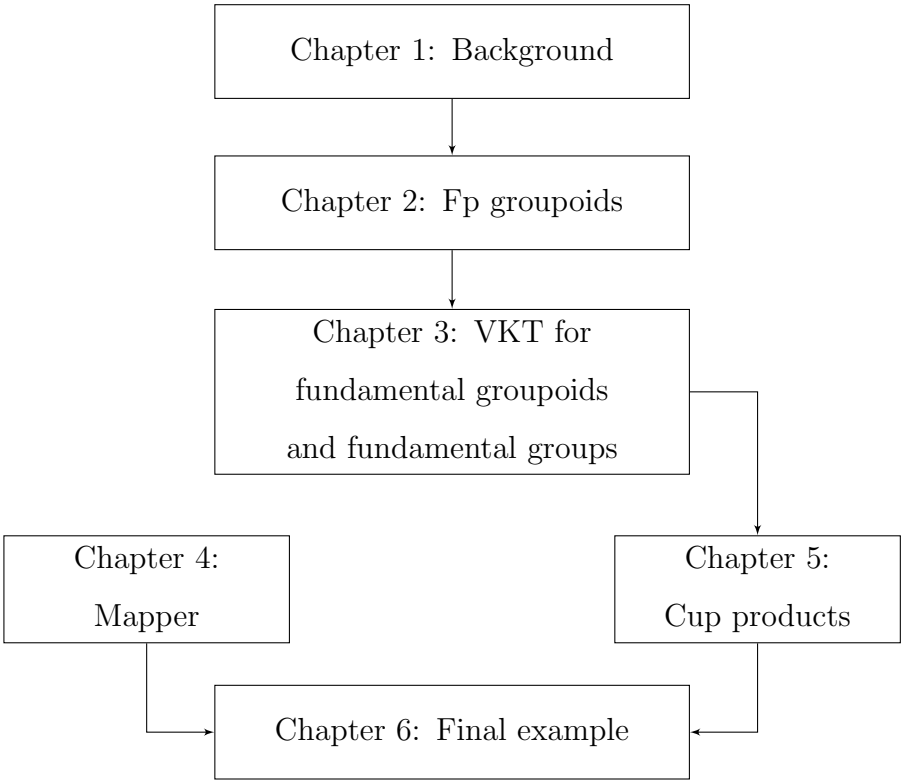
➢ Section 5.2 explains the calculation of the cup product from the presentation of the fundamental group without need for any further significant computations.

➢ Section 5.3 illustrates the method on the integral cohomology ring of a 3-dimensional digital image.

➢ Section 5.4 explains how the van Kampen theorem for fundamental groupoids [12, 18] yields a distributed version of the fundamental group algorithm of [11], and hence a distributed method for computing the cup product.

Chapter 6 is devoted to explaining how to distinguish between two data sets, one sampled from a torus T and the other sampled from a space X obtained from the sphere S^2 by attaching the two ends of two arcs to one point on the sphere.

Appendix A includes a full list of implemented functions in GAP programming language, these functions for the FpGd package and Appendix B include samples from the datasets that used in thesis. Finally, in Appendix C, we put all the codes which are written for the purpose of this thesis.



The above diagram aims to give a sketch of the chapters' dependency.

Most of the datasets that are generated randomly, we store them in text files available in the package FpGd .

1.3 Background

This section recalls some concepts and definitions needed in this thesis.

1.3.1 CW-Spaces

This section gives a brief review of CW-spaces. In the definition of a CW-space we use the following notation for the closed unit n-ball, the open unit n-ball and the unit n-sphere:

$$\begin{aligned} D^n &= \{x \in \mathbb{R}^n : \quad ||x|| \leqslant 1\} \\ \mathring{D}^n &= \{x \in \mathbb{R}^n : \quad ||x|| < 1\} \\ S^n &= \{x \in \mathbb{R}^{n+1} : ||x|| = 1\} \end{aligned}$$

where $||(x_1, x_2, ..., x_n)|| = \sqrt{x_1^2 + x_2^2 + ... + x_n^2}$.

Definition 1.3.1. [50] An *n-cell* is a topological space homeomorphic to $\mathring{D}^n$. A cell is a space which is an n-cell for some $n \geqslant 0$. An n-cell will be said to have *dimension* n.

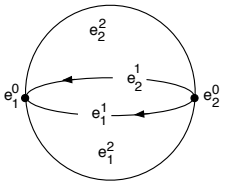
Definition 1.3.2. [50] A *cell-decomposition* (or cell-structure) of a topological space X is a family $\mathcal{E} = \{e_\alpha | \alpha \in A\}$ of subspaces of X such that each e_α is a cell and the space X is the disjoint union

$$X = \bigsqcup_{\alpha \in A} e_\alpha.$$

A cell-decomposition of a space X can have many different dimensions.
The *n-skeleton* of X is the subspace

$$X^n = \bigsqcup_{i \in A} e_i,$$

such that $\dim(e_\alpha) \leqslant n$.
Of course, $X^0 \subset X^1 \subset X^2 \subset ...$ and $X = \bigsqcup_{k \geqslant 0} X^k$. A finite n-dimensional CW complex structure for X expresses X as being built in stages as $X^0 \subset X^1 \subset X^2 \subset ... \subset X^n = X$.
The figure below shows a CW decomposition of the sphere, it includes two 0-cells e_1^0 and e_2^0, two 1-cells e_1^1 and e_2^1 and two 2-cells e_1^2 and e_2^2. There are no restrictions on the number of cells in a cell-decomposition. We can have uncountably many cells in the cell-decomposition. Any space X has a cell-decomposition where each point of X is a 0-cell. A finite cell-decomposition is a cell decomposition consisting of finitely many cells.



Definition 1.3.3. [50] A pair $(X, \mathcal{E})$ consisting of a Hausdorff space X and a cell-decomposition $\mathcal{E}$ of X is called a *CW-space* if the following axioms are satisfied

1. For each n-cell $e \in \mathcal{E}$ there is a map $\psi_e : D^n \to X$ restricting to a homeomorphism $\psi_e|_{\mathring{D}^n} : \mathring{D}^n \to e$ and taking S^{n-1} into X^{n-1} .

2. For any cell $e \in \mathcal{E}$ the closure $\bar{e}$ intersects only a finite number of other cells in $\mathcal{E}$.

3. A subset $A \subset X$ is closed iff $A \cap \bar{e}$ is closed in X for each $e \in \mathcal{E}$.

The axioms (2) and (3) are automatically satisfied if $\mathcal{E}$ is finite, so they are needed only when $\mathcal{E}$ is infinite. The restrictions $\psi_e|_{\partial D^n}$ are called the *attaching maps*.

The letters CW stand for the closure finiteness of axiom (2) and the weak topology of axiom (3).

Definition 1.3.4. [53] A CW-space is called *regular* if all its attaching maps are homeomorphisms.

Definition 1.3.5. [50] A subspace Y of a CW-space X is called a *CW-subspace* of X if it is a union of cells in X such that the closure of each cell is also contained in Y; therefore, it is also a CW-space.

The n-skeleton X^n is a CW-subspace of X, for every n.

Definition 1.3.6. [38] Let X and Y be CW-spaces, the map $f : X \to Y$ is called *cellular* if it satisfies $f(X^n) \subseteq Y^n$ for all n.

1.3.2 Simplicial complexes

The earliest formulation of homology theory is simplicial homology, based on triangulations of topological spaces called simplicial complexes. The theoretical properties of this homology have some drawbacks when dealing with general topological spaces and successive improvements over the past century or so have resulted in a more general version based on singular homology and general cell complexes.
We use simplicial homology here since it is concrete and easy to adapt for implementation on a computer.

Definition 1.3.7. [94] A *simplicial complex* Δ on a finite set X is a collection of subsets of X satisfying:

- if $\sigma \in \Delta$ and $\tau \subset \sigma$ then $\tau \in \Delta$;
- $\{x\} \in \Delta$ for each $x \in X$.

The sets $\sigma \in \Delta$ are called *simplices* and the elements $x \in X$ are called *vertices*. When $|\sigma| = n + 1$, we called σ an *n-simplex*. A 1-simplex is called an *edge*. The complex Δ is said to be *finite* if the vertex set X is finite, and the complex Δ is said to be of *dimension* k if some simplex contains $k + 1$ vertices and no simplex contains more.

One example of a simplicial complex arises from any collection $X = \{X_i\}_{i\in I}$ of sets X_i.

Definition 1.3.8. [95] The *nerve* of a collection $\mathcal{X} = \{X_i\}_{i\in I}$ is the simplicial complex with vertex set $V = \mathcal{X}$ and with simplex set

$$Nerve(\mathcal{X}) = \{\sigma \subseteq \mathcal{X} : \sigma \text{ finite and } \bigcap_{X\in\sigma} X \neq \varnothing\}.$$

1.3.3 (Co)Homology of a CW-complex

Let X be a finite regular CW-space. The *cellular chain complex*

$$C_*(X) = \cdots \longrightarrow C_n(X) \xrightarrow{\partial_n} C_{n-1}(X) \xrightarrow{\partial_{n-1}} \cdots \xrightarrow{\partial_1} C_0(X)$$

is constructed by taking $C_n(X)$ to be the free abelian group with free generators corresponding to the n-cells of X, with boundary map

$$\partial_n(e_i^n) = \sum_j [e^n, e_j^{n-1}]\ e_j^{n-1}$$

where $[e^n, e_j^{n-1}]$ denotes the degree of the composition map Γ_j,

$$\begin{array}{ccc} S^{n-1} & \longrightarrow X^{n-1} & \longrightarrow X^{n-1}/X^{n-2} \\ {\scriptstyle f}\uparrow & & \downarrow \\ S_j^{n-1} & \xrightarrow[\Gamma_j]{} & \bigvee S_j^{n-1} \end{array}$$

where $\bigvee S_j^{n-1}$ is the wedge sum of spheres. Each sphere corresponds to $n-1$ cells. For each n, $\partial_n\partial_{n+1} = 0$ or equivalently Im $\partial_{n+1} \subseteq$ Ker ∂_n.

Definition 1.3.9. [23] The n-dimensional *homology* of a chain complex $C_*(X)$ of a regular CW-space X is defined as

$$H_n(C_*(X)) = \frac{\text{Ker}\,\partial_n}{\text{Im}\ \partial_{n+1}}.$$

We abbreviate $H_n(C_*(X))$ to $H_n(X)$.

The elements of $H_n(X)$ are called *homology classes*. Each homology class is an equivalence class of *cycles* where a cycle is an element of $\operatorname{Ker} d_n$.

Fix an abelian group A, and replace each group C_n by its dual group $C_n^* = \operatorname{Hom}(C_n, A)$, and ∂_n by its dual homomorphism

$$d_{n-1} : C_{n-1}^* \to C_n^*,$$

to obtain the cochain complex

$$\cdots \longrightarrow C_{n-1}^*(X) \xrightarrow{d_{n-1}} C_n^*(X) \xrightarrow{d_n} C_{n+1}^*(X) \longrightarrow \cdots$$

The nth *cohomology group* of X with coefficients in A is

$$H^n(X) = \frac{\operatorname{Ker} d_n}{\operatorname{Im}\ d_{n-1}}$$

Let R be a commutative ring and X be a regular CW-space. Then there is a bilinear map, called the *cup product*:

$$H^n(X, R) \times H^m(X, R) \to H^{n+m}(X, R),$$

defined by an explicit formula on cochains. The product of cohomology classes u and v is written as $u \cup v$. This product makes the direct sum

$$H^*(X, R) = \oplus_i H^i(X, R)$$

into a graded ring, called the *cohomology ring* of X. It is graded-commutative in the sense that

$$uv = (-1)^{nm} vu$$

for u in $H^n(X, R)$ and v in $H^m(X, R)$.

Chapter 2

Groupoids and Fp Groupoids

2.1 Introduction

The mathematical structure that we are studying is a groupoid. This is a special case of the notion of a category, so we first recall the definition of a category.

Definition 2.1.1. [55, 69] A *category* $\mathcal{C}$ consists of

- a collection of *objects* denoted by $Obj(\mathcal{C})$,
- a collection $Arr(\mathcal{C})$ whose members are the *morphisms* of $\mathcal{C}$. Each morphism $f \in Arr(\mathcal{C})$ has an associated object $s(f) \in Obj(\mathcal{C})$ called the *source* of f, and an associated object $t(f) \in Obj(\mathcal{C})$ called the *target* of f. Morphisms are also known as arrows and denoted as $f : S \to T$ where $S = s(f), T = t(f)$. The *composite* $g \circ f$ of two morphisms $f, g \in Arr(\mathcal{C})$ exists if, and only if, $s(g) = t(f)$. This composite is itself a morphism with source $s(g \circ f) = s(f)$ and target $t(g \circ f) = t(g)$.
- for each $A \in Obj(\mathcal{C})$ a distinguished element $1_A : A \to A$ of $Arr(\mathcal{C})$, this special element is called the *identity* morphism on A.

These are required to satisfy the following two axioms:

- Associativity: if $f : A \to B$, $g : B \to C$, and $h : C \to D$ then $(h \circ g) \circ f = h \circ (g \circ f)$,
- Identity: if $k : A \to B$ then $k \circ 1_A = k = 1_B \circ k$

Example 2.1.1. Some examples of categories.

1. The category of sets $\mathcal{C}_{\mathsf{set}}$, whose objects are sets and whose arrows are functions between those sets.
2. The category of groups $\mathcal{C}_{\mathsf{grp}}$, whose objects are groups and whose arrows are group homomorphisms.
3. The category of topological spaces $\mathcal{C}_{\mathsf{top}}$, whose objects are topological spaces and whose arrows are continuous maps.
4. The category $\mathcal{C}_{\mathsf{R\text{-}mod}}$, whose objects are R-modules over a fixed ring R and whose arrows are all module homomorphisms between R-modules..
5. Any monoid is an example of a category with a single object, where all elements of the monoid are considered to be morphisms with source and target equal to this single object.

A *functor* is a type of mapping between categories analogous to homomorphisms of monoids.

Definition 2.1.2. [69] Let $\mathcal{C}$ and $\mathcal{D}$ be categories. A *functor* $\mathcal{F}$ from $\mathcal{C}$ to $\mathcal{D}$

- associates to each object $A \in \mathcal{C}$ an object $\mathcal{F}(A) \in \mathcal{D}$,
- associates to each morphism $f : A \to B$ in $\mathcal{C}$ a morphism $\mathcal{F}(f) : \mathcal{F}(A) \to \mathcal{F}(B)$ in $\mathcal{D}$ such that the following two conditions hold:

  i. $\mathcal{F}(1_A) = 1_{\mathcal{F}(A)}$ for every object A in $obj(\mathcal{C})$,

  ii. $\mathcal{F}(g \circ f) = \mathcal{F}(g) \circ \mathcal{F}(f)$ for all morphisms $f : A \to B$ and $g : B \to C$ in $Arr(\mathcal{C})$. In other words, the functor respects composition.

That is, functors must preserve identity morphisms and composition of morphisms. The *cartesian product* of two categories $\mathcal{C}, \mathcal{D}$, denoted by $\mathcal{C} \times \mathcal{D}$, is a category whose objects are ordered pairs (A, B) of objects $A \in Obj(\mathcal{C}), B \in Obj(\mathcal{D})$ and arrows $(A, B) \to (A', B')$ are pairs (f, g) where $f \in \mathcal{C}(A, A'), g \in \mathcal{D}(B, B')$ with composition defined by $(f, g) \circ (f', g') = (f \circ f', g \circ g')$.

2.2 Groupoids

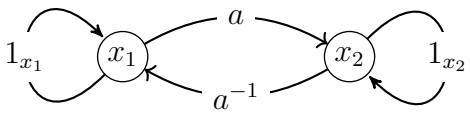
A *groupoid* is a special type of category which is a generalization of a group.

Definition 2.2.1. [55] A *groupoid* $\mathcal{G}$ is a category in which for each morphism $f : A \to B$ there is a morphism $f^{-1} : B \to A$ such that $f \circ f^{-1} = 1_B, f^{-1} \circ f = 1_A$. The morphism f^{-1} is called the *inverse* of f.

Definition 2.2.2. A groupoid $\mathcal{G}$ is called a *discrete* if each of its arrows has equal source and target.

Example 2.2.1. Any set S gives rise to a groupoid whose objects are the elements of S and whose only arrows are the identity arrows $1_x : x \to x, \ x \in S$. This is an example of a discrete groupoid on S.

Example 2.2.2. The *unit groupoid* consists of two objects with two identity arrows, and precisely two non-identity arrows which are inverse to each other. The unit groupoid plays the role of the "unit interval" in the theory of groupoids.



Example 2.2.3. (1) Any group can be considered to be a groupoid with one object. (2) The category $\mathcal{C}_{\mathsf{Vect}}$ with $Obj(\mathcal{C}_{\mathsf{Vect}})$ equal to the collection of all n-dimensional real vector spaces and $Arr(\mathcal{C}_{\mathsf{Vect}})$ equal to the collection of all isomomorphisms between such vector spaces is a groupoid.

Definition 2.2.3. Given a groupoid $\mathcal{G}$, a *subgroupoid* $\mathcal{H}$ of $\mathcal{G}$ consists of subsets $Obj(\mathcal{H}) \subset Obj(\mathcal{G})$ and $Arr(\mathcal{H}) \subset Arr(\mathcal{G})$ such that:

1. $s(f), t(f) \in Obj(\mathcal{H})$, for each $f \in Arr(\mathcal{H})$,
2. $1_A \in Arr(\mathcal{H})$, for each $A \in Obj(\mathcal{H})$,
3. if $f \in Arr(\mathcal{H})$ then $f^{-1} \in Arr(\mathcal{H})$
4. if $f, g \in Arr(\mathcal{H})$ with $t(f) = s(g)$ then $g \circ f \in Arr(\mathcal{H})$.

2.3 Groupoid Homomorphisms

A groupoid homomorphism is a mapping from one groupoid to another that respects multiplication, composition, inverses and identity morphisms. The way of creating groupoid homomorphisms is to give maps for a set of groupoid generators (which preserves relations).

A formal definition of groupoid homomorphism is given below.

Definition 2.3.1. If G_1 and G_2 are two groupoids, then a *homomorphism* between them, is a functor

$$\mathcal{F} : G_1 \to G_2$$

consisting of

- a map between the respective sets of objects

$$F_o : Obj(G_1) \to Obj(G_2)$$

- a map between the respective sets of arrows

$$\begin{aligned} F_a : Arr(G_1) &\to Arr(G_2) \\ f &\mapsto F_a(f) \end{aligned}$$

such that if $x = source(f)$ and $y = target(f)$ then $F_o(x) = source(F_a(f))$ and $F_o(y) = target(F_a(f))$.

A homomorphism is a an *isomorphism* if it is bijective on objects and bijective on arrows.

Example 2.3.1. A homomorphism between two groupoids with only one object is the same as a group homomorphism.

We can associate to each groupoid a *directed graph* (underlying graph) by omitting the compositions of the morphisms. Thus the language of graph theory is useful in describing groupoids. The next section starts by recalling the definition of a directed graph because it is used for defining the free groupoid.

2.4 Free groupoids

Definition 2.4.1. A *directed graph* $\Gamma = (V, E, s, t)$ consists of a set V called the set of *vertices*, a set E called the set of *edges* of Γ and two functions $s, t : E \to V$. The vertex $s(e)$ is the *source* of an edge $e \in E$. The vertex $t(e)$ is the *target* of an edge $e \in E$.

A map of directed graphs

$$(V, E, s, t) \;\; \mapsto \;\; (V', E', s', t')$$

consists of functions

$$\begin{aligned} f_1 : V &\;\; \to \;\; V' \\ f_2 : E &\;\; \to \;\; E' \end{aligned}$$

such that $s(f_2(e)) = f_1(s(e))$ and $t(f_2(e)) = f_1(t(e))$ for all $e \in E$.

Definition 2.4.2. The *disjoint union* $\Gamma = \Gamma_1 \sqcup \Gamma_2$ of directed graphs Γ_1 and Γ_2 with disjoint vertex sets $V(\Gamma_1)$ and $V(\Gamma_2)$ and edge sets $E(\Gamma_1)$ and $E(\Gamma_2)$ is the directed graph with $V(\Gamma) = V(\Gamma_1) \cup V(\Gamma_2)$ and $E(\Gamma) = E(\Gamma_1) \cup E(\Gamma_2)$.

Definition 2.4.3. A *maximal tree* T of a directed graph Γ is a subgraph which includes every vertex of Γ and contains no cycle.

Let $\mathcal{G}raphs$ denote the category whose objects are directed graphs and whose morphisms are maps of directed graphs. Let $\mathcal{G}roupoids$ denote the category whose objects are groupoids and whose morphisms are functors between groupoids. There is a functor

$$\mathcal{U} : \mathcal{G}roupoids \to \mathcal{G}raphs \tag{2.1}$$

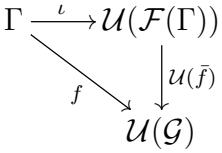
which simply forgets the partial composition on a groupoid. If $\mathcal{G}$ is a groupoid, then the vertices of $\mathcal{U}(\mathcal{G})$ are precisely the objects of $\mathcal{G}$. The directed edges of $\mathcal{U}(\mathcal{G})$ are the arrows of $\mathcal{G}$.

There is a functor

$$\mathcal{F} : \mathcal{G}raphs \to \mathcal{G}roupoids \tag{2.2}$$

where for a directed graph Γ, the groupoid $\mathcal{F}(\Gamma)$ is characterized, up to isomorphism, by the following universal property.

Universal property of a free groupoid on Γ. There is a map of directed graphs $\iota : \Gamma \to \mathcal{U}(\mathcal{F}(\Gamma))$. For any groupoid $\mathcal{G}$ and any map of directed graphs $f : \Gamma \to \mathcal{U}(\mathcal{G})$ there exists a unique groupoid morphism $\bar{f} : \mathcal{F}(\Gamma) \to \mathcal{G}$ for which the following diagram commutes in the category of directed graphs.

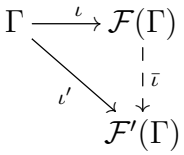


We call $\mathcal{F}(\Gamma)$ the *free groupoid* on Γ. The existence of $\mathcal{F}(\Gamma)$ is established by an explicit construction in terms of words $x_1^{\epsilon_1} x_2^{\epsilon_2} \ldots x_n^{\epsilon_n}$ where $\epsilon = \pm 1, x_i \in E(\Gamma)$, and $s(x_i^{\epsilon_i}) = t(x_{i+1}^{\epsilon_{i+1}})$. When the directed graph Γ has just a single vertex we say that

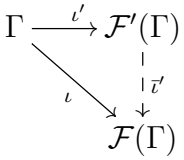
$\mathcal{F}(\Gamma)$ is the free group on the set $E(\Gamma)$.

Proposition 2.4.1. *$\mathcal{F}(\Gamma)$ is unique up to isomorphism of groupoids.*

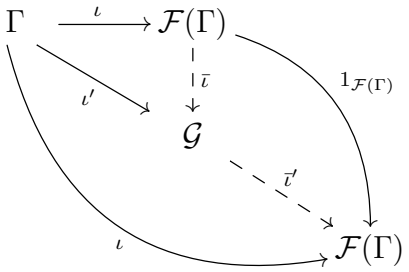
Proof. For simplicity we denote $\mathcal{U}(\mathcal{G})$ by $\mathcal{G}$ for any groupoid $\mathcal{G}$.
Let Γ be a directed graph, and let $\mathcal{F}(\Gamma)$ and $\mathcal{F}'(\Gamma)$ be free groupoids on Γ. Let $\iota : \Gamma \to \mathcal{F}(\Gamma)$ be a map, and another map $\iota' = \Gamma \to \mathcal{F}'(\Gamma)$. By the universal property of free groupoid there is a unique groupoid morphism $\bar{\iota} = \mathcal{F}(\Gamma) \to \mathcal{F}'(\Gamma)$ such that the following digram



commutes. Also, the following diagram



commutes. Now we obtain



By uniqueness, $\bar{\iota}' \circ \bar{\iota} = 1_{\mathcal{F}(\Gamma)}$.
Similarly, $\bar{\iota} \circ \bar{\iota}' = 1_{\mathcal{F}'(\Gamma)}$.
Therefore, $\mathcal{F}(\Gamma)$ is isomorphic to $\mathcal{F}'(\Gamma)$. □

Definition 2.4.4. A groupoid $\mathcal{G}$ is *connected* if for each pair of objects A and $B \in Obj(\mathcal{G})$ there is at least one arrow $w \in Arr(\mathcal{G})$ with the property

$$s(w) = A \text{ and } t(w) = B$$

Definition 2.4.5. A *component* of a groupoid $\mathcal{G}$ is a subgroupoid $\mathcal{S} \subseteq \mathcal{G}$ such that if $\mathcal{S}'$ is any connected subgroupoid of $\mathcal{G}$ with $Obj(\mathcal{S}) \subseteq Obj(\mathcal{S}')$ and $Arr(\mathcal{S}) \subseteq Arr(\mathcal{S}')$ then $\mathcal{S} = \mathcal{S}'$. The components of a groupoid are themselves groupoids.

Definition 2.4.6. The *disjoint union* $\mathcal{G} = \mathcal{G}_1 \bigsqcup \mathcal{G}_2$ of groupoids $\mathcal{G}_1$ and $\mathcal{G}_2$ with disjoint object sets $Obj(\mathcal{G}_1)$ and $Obj(\mathcal{G}_2)$ and arrow sets $Arr(\mathcal{G}_1)$ and $Arr(\mathcal{G}_2)$ is the groupoid with $Obj(\mathcal{G}) = Obj(\mathcal{G}_1) \cup Obj(\mathcal{G}_2)$ and $X = Arr(\mathcal{G}_1) \cup Arr(\mathcal{G}_2)$.

Every groupoid is uniquely expressible as the disjoint union of connected subgroupoids, namely, its components.

We sometimes refer to the edges in a directed graph Γ as a set of free generators of the groupoid $\mathcal{F}(\Gamma)$. The free groupoid $\mathcal{F}$ on a set of free generators $\underline{x}$ consists of all words

$$w = x_1^{\epsilon_1} x_2^{\epsilon_2} \dots x_n^{\epsilon_n}, \qquad x_i \in \underline{x} \text{ and } \epsilon_i = \pm 1,$$

such that $s(x_j^{\epsilon_j}) = t(x_{j+1}^{\epsilon_{j+1}})$, considering two words w_1 and w_2 different unless their equality follows from the axiom

$$w_1 h h^{-1} w_2 = w_1 w_2, \text{ for } h \in \underline{x}.$$

An arbitrary groupoid $\mathcal{D}$ is called *free* if it is isomorphic to $\mathcal{F}(\Gamma)$ for some Γ.
In the context of graph theory, the free groupoid on a directed graph is the groupoid whose objects are the vertices of the graph and whose morphisms are finite concatenations of the edges in the graph and formal inverses to them [14].

Let $\Gamma = \Gamma_1 \bigsqcup \Gamma_2$ be the disjoint union of connected graphs Γ_1 and Γ_2.

Lemma 2.4.2. $\mathcal{F}(\Gamma) = \mathcal{F}(\Gamma_1) \bigsqcup \mathcal{F}(\Gamma_2)$

Proof. This is clear from the construction of a free groupoid in terms of "words". □

Proposition 2.4.3. *A groupoid is free if and only if its components are free.*

Proof. Suppose $\mathcal{F}$ is the free groupoid on a directed graph Γ.
Suppose $\Gamma = \Gamma_1 \sqcup \Gamma_2 \sqcup \dots \sqcup \Gamma_n$ with Γ_i a connected graph.
Then

$$\mathcal{F}(\Gamma) = \mathcal{F}(\Gamma_1) \bigsqcup \mathcal{F}(\Gamma_2) \bigsqcup \dots \bigsqcup \mathcal{F}(\Gamma_n) \quad \text{by lemma 2.4.2.}$$

Each $\mathcal{F}(\Gamma_i)$ is a connected component of $\mathcal{F}(\Gamma)$. Hence each connected component is free.

Conversely, suppose each connected component $\mathcal{F}(\Gamma_i)$ is free.

Then the groupoid is the union of its components:

$$\mathcal{F}(\Gamma_1) \bigsqcup \mathcal{F}(\Gamma_2) \bigsqcup ... \bigsqcup \mathcal{F}(\Gamma_n)$$

By lemma 2.4.2, this union equals

$$\mathcal{F}(\Gamma_1 \sqcup \Gamma_2 \sqcup ... \sqcup \Gamma_n)$$

Hence, the groupoid is free on the graph

$$\Gamma_1 \sqcup \Gamma_2 \sqcup ... \sqcup \Gamma_n$$

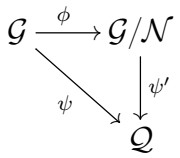
□

2.5 Finitely presented groupoids

Let $\mathcal{G}$ be a groupoid with object set $Obj(\mathcal{G}) = V$. Let $\mathcal{N}$ be a discrete subgroupoid of $\mathcal{G}$ with the same object set $Obj(\mathcal{N}) = V$. Thus every arrow of $\mathcal{N}$ is an arrow of $\mathcal{G}$ and $\mathcal{N}$ is closed under groupoid composition. The collection of groups $\{\mathcal{G}(v, v) \mid v \in V\}$ is an example of a discrete subgroupoid of $\mathcal{G}$. We say that a discrete subgroupoid $\mathcal{N}$ is *normal* in $\mathcal{G}$ if $\mathcal{N}(v, v)$ is a normal subgroup of $\mathcal{G}(v, v)$ for each $v \in V$. Given a discrete normal subgroupoid $\mathcal{N}$ in $\mathcal{G}$ we can form the quotient groupoid $\mathcal{G}/\mathcal{N}$ which is characterized up to groupoid isomorphism by the following universal property.

Universal property of a quotient groupoid. There is a morphism of groupoids $\phi : \mathcal{G} \to \mathcal{G}/\mathcal{N}$. For any groupoid $\mathcal{Q}$ with object set $Obj(\mathcal{Q}) = V$, and for any morphism $\psi : \mathcal{G} \to \mathcal{Q}$ that is the identity on V and that sends each element of $\mathcal{N}$ to an identity element, there exists a unique morphism of groupoids $\psi' : \mathcal{G}/\mathcal{N} \to \mathcal{Q}$

such that the following diagram in the category of groupoids commutes.



Proposition 2.5.1. *For discrete $\mathcal{N}$, $\mathcal{G}/\mathcal{N}$ is unique up to isomorphism of groupoids.*

Proof. Similar to the proof of the proposition 2.4.1. □

Definition 2.5.1. We say that a set $\underline{r}$ of arrows in a discrete subgroupoid $\mathcal{N}$ *normally generates* $\mathcal{N}$ if any normal discrete subgroupoid of $\mathcal{G}$ containing $\underline{r}$ also contains the subgroupoid $\mathcal{N}$.

Let $\mathcal{G}$ be a groupoid with vertex set $V = Obj(\mathcal{G})$, and let $\mathcal{F}(\Gamma)$ be a free groupoid on a directed graph $\Gamma = (V, \underline{x}, s, t)$, and suppose that there is a morphism of groupoids

$$\phi : \mathcal{F}(\Gamma) \twoheadrightarrow \mathcal{G} \tag{2.3}$$

that is the identity on objects and that is surjective on arrows. By $\ker \phi$ we mean the groupoid with vertex set V and with arrows those elements $\underline{r}$ in $\mathcal{F}(\Gamma)$ mapping to an identity arrow $1_{s(r)}$ in $\mathcal{G}$. The groupoid $\ker \phi$ is a discrete normal subgroupoid and $\mathcal{F}(\underline{x})/\ker \phi$ is isomorphic to $\mathcal{G}$. Let $\underline{r}$ be a set of elements in $\ker \phi$ that normally generates $\ker \phi$. The data $\langle \underline{x} \mid \underline{r} \rangle$ is called a *free presentation* of the groupoid $\mathcal{G}$.

2.6 Vertex group

Let $\mathcal{G}$ be a groupoid with object set $Obj(\mathcal{G}) = V$. For each object (vertex) $v \in V$ we let $\mathcal{G}(v, v)$ denote the group of arrows with source and target equal to v. We refer to $\mathcal{G}(v, v)$ as the *vertex group* or *isotropy group* or *object group* at v. The vertex group $\mathcal{G}(v, v)$ actually is a subgroupoid consisting of one object v and all arrows of the form $v \to v$.

Let $\mathcal{G}$ be a connected groupoid, we can define a homomorphism

$$\theta : \mathcal{G} \to \mathcal{G}(v, v) \tag{2.4}$$

in the following sense.

Let Γ be the generating graph of $\mathcal{G}$, (i.e. $\mathcal{F}(\Gamma) = \mathcal{G}$), and let T be a maximal tree in Γ. The tree T generates a subgroupoid $\mathcal{H}$ of $\mathcal{G}$, which called a *tree of groupoid*. The map θ is defined as

$$\begin{aligned} \theta(a) &= v \qquad a \in Obj(\mathcal{G}) \\ \theta(w) &= xwy, \quad w \in Arr(\mathcal{G}), x, y \in \mathcal{H} \end{aligned} \tag{2.5}$$

such that $t(y) = s(w), s(x) = t(w)$ and $s(y) = t(x) = v$.

For $c, d \in \mathcal{H}$ (such that $s(c) = t(d) = u$ and $t(c) = s(d) = v$), the product $dc = 1_u$. Its obvious that the map θ maps the whole $\mathcal{H}$ onto 1_v.

Proposition 2.6.1. *The vertex groups of a connected groupoid are all isomorphic.*

Proof. Let $\mathcal{G}$ be a groupoid with $Obj(\mathcal{G}) = V$. Let $v \in V$ and $\mathcal{G}(v, v)$ is the vertex group on v. To prove that all vertex groups are isomorphic to $\mathcal{G}(v, v)$, let us choose any object $w \in V$, and any arrow x such that $s(x) = v$ and $t(x) = w$. The map $h \mapsto xhx^{-1}$ is an isomorphism from the vertex group at $\mathcal{G}(v, v)$ to the vertex group at $\mathcal{G}(w, w)$. □

Theorem 2.6.2. *Let $\mathcal{G} = \langle \underline{x} \mid \underline{r} \rangle$ be a finitely presented connected groupoid, If $\mathcal{G}(v, v)$ is the vertex group at $v \in Obj(\mathcal{G})$, then $\mathcal{G}(v, v) = \langle \underline{x}' \mid \underline{r}' \cup \underline{t} \rangle$, where $\underline{x}' = \{\theta(x) : x \in \underline{x}\}$ and $\underline{r}' = \{\theta(r) : r \in \underline{r}$ with expressing $\theta(r)$ as a word $x_1^{\epsilon_1} x_2^{\epsilon_2} ... x_k^{\epsilon_k}$, $x_i \in \underline{x}', \epsilon_i \in \pm 1\}$ and $\underline{t} = \{t : t$ edge in a maximal tree of $\mathcal{G}\}$.*

Proof. Let $\underline{x} = (V, E, s, t)$ be a connected directed graph. Let $\mathcal{F}(\underline{x})$ denote the free groupoid on $\underline{x}$. An arrow $r \in Arr(\mathcal{F}(\underline{x}))$ is said to be a *loop* if $s(r) = t(r)$. Let $\underline{r}$ denote a set of loops in the groupoid $\mathcal{F}(\underline{x})$. Let R denote the normal subgroupoid of $\mathcal{F}(\underline{x})$ generated by $\underline{x}$.

The data $\langle \underline{x} \mid \underline{r} \rangle$ is a presentation for the quotient groupoid

$$\mathcal{G} = \mathcal{F}(\underline{x})/R.$$

Let $\underline{t}$ denote a maximal tree in the graph $\underline{x}$. Fix some vertex $v \in V$. Then each vertex $w \in V$ determines a unique simple path $p(w)$ in the tree $\underline{t}$ with $s(p(w)) = w$

and $t(p(w)) = v$. In other words, $p(w)$ is a path in $\underline{t}$ from w to v.
For each arrow a in the groupoid $\mathcal{F}(\underline{x})$ let us set

$$\theta(a) = p(s(a))^{-1} * a * p(t(a)).$$

Thus $\theta(a)$ is a loop in the groupoid $\mathcal{F}(\underline{x})$ with source and target equal to v.
Now define

$$\begin{aligned} \underline{x}' &= \{\theta(a) : a \text{ is a dircted edge in } \underline{x} \text{ and } a \notin \underline{t}\}, \\ \underline{r}' &= \{\theta(a) : a \text{ is an arrow in } \underline{r}\}. \end{aligned}$$

Note that $\underline{x}'$ is a free generating set for the free group $\mathcal{F}(v, v)$. here we are writing $\mathcal{F} = \mathcal{F}(\underline{x})$ and letting $\mathcal{F}(v, v)$ denote the vertex group at v.
Note that $\underline{r}'$ is a subset of $\mathcal{F}(v, v)$. Let $\mathcal{R}(v, v)$ denote the normal subgroup of $\mathcal{F}(v, v)$ normally generated by $\underline{r}'$.
We can now regard $\langle \underline{x}' \mid \underline{r}' \rangle$ as a free presentation for the finitely presented group

$$\mathcal{F}(v, v)/\mathcal{R}(v, v).$$

To prove the theorem we need to see that $\mathcal{F}(v, v)/\mathcal{R}(v, v)$ is isomorphic to the vertex group $\mathcal{G}(v, v)$ in $\mathcal{G}$.
There is a canonical set theoretic function $\lambda' : \underline{x} \to \mathcal{G}$. This function induces a group homomorphism

$$\lambda : \mathcal{F}(v, v) \to \mathcal{G}(v, v)$$

The kernel of λ, by definition, consists of all loops in $\mathcal{F}(\underline{x})$ at v that can be written as a product of conjugates of loops in $\underline{r}$. So clearly the kernel of λ is normally generated by $\underline{r}'$ and the proof is complete.

□

Example 2.6.1. Let $\mathcal{G}$ be a groupoid whose object set is $\{a, b\}$ with the following presentation

$$\langle x, y, z, w \mid y^{-1}x^{-1}z, x^{-1}y^{-1}wzw^{-1} \rangle \tag{2.6}$$

where

$$\begin{aligned} a &= s(x) = t(y) = t(w) \\ b &= t(x) = s(y) = s(w) = s(z) = t(z) \end{aligned}$$

Figure 2.1 shows the generating graph of the groupoid $\mathcal{G}$.

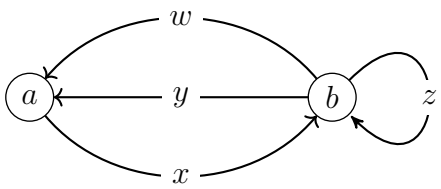


Figure 2.1: Generating graph of the groupoid of the presentation 2.6.

In order to find the presentation of the vertex group on the object, say a, we must determine a tree in the underlying graph $\mathcal{U}(\mathcal{G})$ which includes only one edge corresponding to one generator in $\{x, y, w, z\}$. Let us pick x and set $\bar{x}$ to be a generator of the vertex group $G(a, a)$. Now, we must write the other generators as loops starting and ending at the object a. We obtain

$$\begin{aligned} \bar{y} &= yx, \\ \bar{z} &= x^{-1}zx, \\ \bar{w} &= wx. \end{aligned}$$

Also, we could do the same for the relators to get

$$\begin{aligned} x^{-1}y^{-1}x^{-1}zx &= \bar{y}^{-1}\bar{z}, \\ x^{-1}y^{-1}wzw^{-1} &= \bar{y}^{-1}\bar{w}\bar{z}\bar{w}^{-1}. \end{aligned}$$

Now the presentation of the vertex group $G(a, a)$ is

$$\langle \bar{x}, \bar{y}, \bar{z}, \bar{w} \mid \bar{x}, \bar{y}^{-1}\bar{z}, \bar{y}^{-1}\bar{w}\bar{z}\bar{w}^{-1} \rangle,$$

which can be simplified to the form

$$\langle \bar{z}, \bar{w} \mid \bar{z}^{-1}\bar{w}\bar{z}\bar{w}^{-1} \rangle.$$

Corollary 2.6.3. *If $\mathcal{G}$ is a free groupoid, each of its vertex groups is a free group.*

Proof. Let $\mathcal{G}$ be a free groupoid on a directed graph Γ. By construction $\mathcal{G}$ has $Obj(\mathcal{G}) = V(\Gamma)$. If $\underline{x}$ is the set of directed edges of the graph Γ then $\mathcal{G}$ has the following presentation

$$\mathcal{G} = \langle \underline{x} \mid \varnothing \rangle$$

Let T be a maximal tree in the groupoid $\mathcal{G}$.
Let $v \in Obj(\mathcal{G})$ and G_v be a vertex group. Applying theorem 2.6.2 yields

$$G_v = \langle \underline{x} \backslash \underline{t} \mid \varnothing \rangle,$$

where $\underline{t}$ is the edges set of T, that means G_v is free group.

□

2.7 Implementations

We develop practical tools in the form of a GAP package, for computing finitely presented groupoids. The package is called FpGd , it depends on the HAP package for GAP [3].
In this section we describe the functionality of FpGd aimed at purely algebraic calculations relating to finitely presented groupoids. In the next Chapter we describe functionality aimed at the fundamental groupoid of a finite regular CW-space.
To display the generating graphs of a groupoid, say H, first construct a graph G using the FpGd function `GeneratingGraphOfGroupoid`(H) and then use the command `Display` to display the graph G.

2.7.1 Implementation of free groupoids

The FpGd package uses the following representation of free groupoid.

Data Type 2.7.1. *A Free Groupoid is represented by a component object* **F** *with the following components:*

- F!.objects is a list of integers, each corresponds to one object in F.
- F!.generators is a list of integers, each represents one generator in F.
- F!.sources is a list of integers belong to F!.objects, they represent the sources of the generators of F.
- F!.targets is a list of integers belong to F!.objects, they represent the targets of the generators of F.

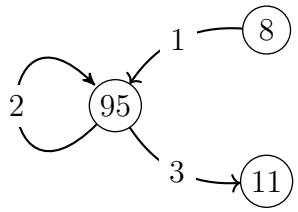
The FpGd command FpGdFreeGroupoid($\mathcal{O}, \mathcal{S}$)) can be used to construct a free groupoid on an object set $\mathcal{O}$ and a set of generators $\mathcal{S}$ as well, where

$$\mathcal{S} = \{[s_i, g_i, t_i], s_i, t_i \in \mathcal{O}, g_i \text{ is integer}\},$$

here $s_i = s(g_i)$ and $t_i = t(g_i)$.
The commands GeneratorsOfGroupoid(), Source(), and Target() can be used to display the generators of the free groupoid and their sources and targets.

The associated boolean valued function IsFpGdFreeGroupoid(F) returns true when F is of this data type.

Example 2.7.1. In the following gap session, we create a free groupoid F on three objects labeled by $\{8, 11, 95\}$ and generated by three free generators labeled by $1, 2$, and 3. The following figure shows the graph representing generators of the groupoid F in order source and target of the generators. Also, we show some FpGd functions like Source and Target that return the source and target of the generators.



GAP session 2.7.1

```
gap> O:=[8,11,95];;
gap> S:=[[8,1,95],[95,2,95],[95,3,11]];;
gap> F:=FreeGroupoid(O,S);
< free groupoid on the generators [ f1 , f2 , f3 ] >
gap> g:=GeneratorsOfGroupoid(F);
[ f1 , f2 , f3 ]
gap> List(g,x->[Source(x),Target(x)]);
[ [8,95], [95,95] ,[95,11] ]
gap> IsFpGdFreeGroupoid(F);
true
```

The following representation is the data type that used for arrows a in a free groupoid F.

Data Type 2.7.2. *An arrow in a free groupoid F is represented by a component object a with the following components:*

- a!.list is a list of integers representing the arrow (sequence of composible generators and their inverses) in F.
- a!.source is an integer representing the source of a which belong to F!.objects.
- a!.target is an integer representing the target of a which belong to F!.objects.
- a!.parent is a component object which is the free groupoid F.

The commands Source(), and Target() can be used also to display the sources and targets for arrows.

The composition of arrows a and b in a free groupoid F is defined as usual using the asterisk between them. It is based in Algorithm 2.7.1.

Algorithm 2.7.1 Composition of arrows in a free groupoids

Input: Two arrows a and b in a free groupoid G.
Output: The composition a∗b.

1: **procedure**
2: if a!.target = b!.source then return a∗b
3: else
4: return fail;
5: end if
6: **end procedure**

2.7.2 Implementation of fp groupoids

The FpGd package uses the following representation of a finitely presented groupoid.

Data Type 2.7.3. *A Fp Groupoid is represented by a component object G with the following components:*

- G!.objects is a list of integers, each corresponds to one object in G.
- G!.generators is a list of integers, each represents one generator in G.
- G!.sources is a list of integers belong to G!.objects, they represent the sources of the generators of G.
- G!.targets is a list of integers belong to G!.objects, they represent the targets of the generators of G.
- G!.relators is a list, each entry correspond to one relator which is also a list of integers.

The FpGd command FpGdFpGroupoid($\mathcal{O}, \mathcal{S}, \mathcal{R}$) can be used to construct a fp groupoid on an object set $\mathcal{O}$ which is generated by a set of free generators $\mathcal{S}$ with a set of relators $\mathcal{R}$. Also, we can construct a finitely presented groupoid by creating first a free groupoid F and then define a set of relators $rels$, so F/$rels$ returns a fp groupoid. The associated boolean valued function IsFpGdFpGroupoid(G) returns true when G is of this data type.
The following example shows a free groupoid of a set of objects $\{5, 9\}$ generated by three generators.

Example 2.7.2. Recall the groupoid in the example 2.6.1. Let us name the objects and generators as follows

$$a = 5,\ b = 9,$$

and

$$x = 1,\ y = 2,\ z = 3,\ w = 4.$$

In the following GAP session, this groupoid is created. It is subject to two relators to produce a fp groupoid.
The FpGd command RelatorsOfFpGroupoid used to display the relators of the given fp groupoid.

GAP session 2.7.2

```
gap> O:=[5, 9];;
gap> S:=[[5,1,9],[9,2,5],[9,3,9],[9,4,5]];;
gap> F:=FreeGroupoid(O,S);
< free groupoid on the generators [ f1 , f2 , f3 , f4 ] >
gap> g:=GeneratorsOfGroupoid(F);; x:=g[1]; y:=g[2]; z:=g[3];w:=g[4];
gap> rels:=[y^-1*x^-1*z, x^-1*y^-1*w*z*w^-1];;
gap> G:=F/rels;
< fp groupoid on the generators [ f1 , f2 , f3 , f4 ] >
gap> IsFpGdFpGroupoid(G);
true
gap> RelatorsOfFpGroupoid(G);
[ f2^-1*f1^-1*f3, f1^-1*f2^-1*f4*f3*f4^-1 ]
```

2.7.3 Implementation of vertex groups

Let $\mathcal{G}$ be a connected groupoid, and let $v \in Obj(\mathcal{G})$. The arrows a_i such that $s(a_i) = t(a_i) = v$ are the elements of the vertex group $\mathcal{G}(v, v)$.
Let T be a maximal tree of the groupoid $\mathcal{G}$. And define $\tau : Obj(\mathcal{G}) \to T$ as follows

$$\tau(x) = w,\ \text{ such that }\ s(w) = v\ \text{ and }\ t(w) = x.$$

There is a corresponding path

$$\bar{\tau}(x) = \bar{w},\ \text{ such that }\ s(\bar{w}) = x\ \text{ and }\ t(\bar{w}) = v.$$

in which the edges appear in reversed order.

Now, if $A = E(\mathcal{U}(\mathcal{G}))\backslash T$, then according to Theorem 2.6.2, the group $\mathcal{G}(v, v)$ is generated by

$$\{g = \bar{\tau}(t(a))\ a\ \tau(s(a)) \mid a \in A\}$$

and it is subject to the following relators

$$\{\bar{\tau}(t(r))\ r\ \tau(s(r)) \mid r \text{ is a relator of } \mathcal{G}\} \cup \{t \mid t \in T\}$$

We introduce the algorithm 2.7.2 based on the above instruction and we implement it in GAP .

Algorithm 2.7.2 Vertex Group

Input: A finitely presented groupoid $\mathcal{G}$ and one object (vertex) $v \in Obj(\mathcal{G})$.
Output: A finite presentation for the vertex group $\mathcal{G}_v$ on the vertex v.

1: **procedure**
2: The generators $\{x_1, ..., x_n\}$ of $\mathcal{G}$ can be viewed as the edges of a directed graph $X = (V, E, s, t)$, where s and t are the source and target functions.
3: Construct a rooted tree T_v emanating from v in X.
4: Such a tree determines a function,

$$\begin{aligned} \tau : V &\rightarrow \text{(paths in } T_v \text{ staring at } v \text{)} \\ v_i &\mapsto e_{i_1} e_{i_2} ... e_{i_k}, \end{aligned}$$

such that $s(e_{i_1}) = v$ and $t(e_{i_k}) = v_i$,
5: An element in $E \backslash T_v$ determines a loop at v, for example if $\tilde{e} \in E \backslash T_v$ such that $s(\tilde{e}), t(\tilde{e})$ and v are different, then $\tau(s(\tilde{e}))\tilde{e}\,\tau^{-1}(t(\tilde{e}))$ is the required loop. These loops correspond to free generators of a free group, say F.
6: Also, each relator of $\mathcal{G}$ determines a loop by the same sense as in step 5, and then rewrite each loop in terms of the elements obtained in step 5.
7: Return the finitely presented group $\mathcal{G}_v$ determined by F and the set of relators.
8: **end procedure**

The FpGd function VertexGroup(G, v) returns a finitely presented group where G is a fp groupoid and $v \in Obj(G)$.

Example 2.7.3. Recall the groupoid $\mathcal{G}$ in example 2.7.2 in order to test the function VertexGroup. In advance we know the result from example 2.6.1.

GAP session 2.7.3

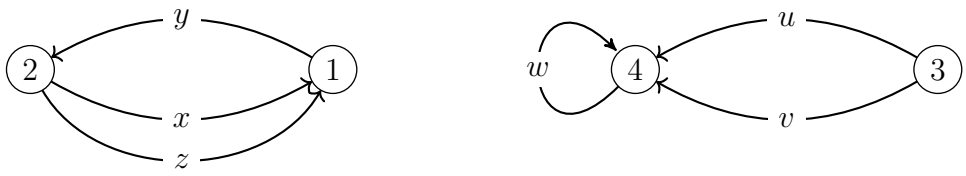
```
gap> V5:=VertexGroup(G,5);
< fp group on the generators [ f1, f2 ] >
gap> RelatorsOfFpGroup(V5);
[ f2^-1*f1*f2*f1^-1 ]
```

Algorithm 2.7.3 Path components of fp groupoid

Input: A finitely presented groupoid $\mathcal{G}$.
Output: A list of path components G_i of $\mathcal{G}$.

1: **procedure**
2: let obj be the object set of $\mathcal{G}$;
3: let $gens$ be the generator set of $\mathcal{G}$;
4: let $rels$ be the relator set of $\mathcal{G}$;
5: classify $[obj, gens, rels]$ into classes $\{[obj_\lambda, gens_\lambda, rels_\lambda]\}_{\lambda\in\Lambda}$ such that for each $\lambda \in \Lambda$, any $g \in gens_\lambda$, the source and target of g are in obj_λ and if $r \in rels_\lambda$ then $r = 1_x$ with $x \in obj_\lambda$.
6: return List$(\Lambda, \lambda \to$ FpGroupoid$([obj_\lambda, gens_\lambda, rels_\lambda])$);
7: **end procedure**

Example 2.7.4. Let $\mathcal{G} = \langle \underline{x} \mid \underline{r} \rangle$ be an fp groupoid with $\underline{x} = \{x, y, z, u, v, w\}$ and $\underline{r} = \{zyxz^{-1}xy, vu^{-1}w^{-1}uv^{-1}w\}$ such that $s(x) = s(z) = t(y) = 2, s(y) = t(x) = t(z) = 1$ and $s(u) = s(v) = 3, s(w) = t(u) = t(v) = t(w) = 4$. The following figure show the graph representing the generators of $\mathcal{G}$.



GAP session 2.7.4

```
gap> obj:=[1,2,3,4];;
gap> gens:=[[2,1,1],[1,2,2],[2,3,1],[3,4,4],[3,5,4],[4,6,4]];;
gap> F:=FreeGroupoid(obj,gens);;
gap> g:=GeneratorsOfGroupoid(F);;
gap> x:=g[1];;  y:=g[2];;  z:=g[3];;  u:=g[4];;  v:=g[5];;  w:=g[6];;
gap> G:=F/[z*y*x*z^-1*x*y, v*u^-1*w^-1*u*v^-1*w];
<fp groupoid on the generators [f1, f2, f3, f4, f5, f6]>
gap> C:=ComponentsOfFpGroupoid(G);;
gap> Length(C);
2
gap> RelatorsOfFpGroupoid(C[1]);
[ f3*f2*f1*f3^-1*f1*f2 ]
gap> RelatorsOfFpGroupoid(C[2]);
[ f5*f4^-1*f6^-1*f4*f5^-1*f6 ]
```

2.7.4 Implementation of groupoid homomorphisms

The FpGd package uses the following data type for the homomorphism of groupoids.

Data Type 2.7.4. *A Homomorphism of Groupoids is represented by a component object* H *with the following components:*

- H!.source is a component object (groupoid) which is the source of the homomorphism H.
- H!.target is a component object (groupoid) which is the target of the homomorphism H.
- H!.mappingObj is a function whose input is an object in H!.source.
- H!.mappingArr is a function whose input is an arrow in H!.source.

The associated function IsFpGdGroupoidHomomorphism(Y) returns the boolean true when Y is of this data type.

Example 2.7.5. Consider two fp groupoids G and K, each generated by three generators x, y, z and a, b, c, respectively. The following figure shows the graphs that generate G and K.

Now let us define the following groupoid homomorphism,

$$H : G \to K$$

which maps the generators of G as follows

$$\begin{aligned} x &\mapsto c, \\ y &\mapsto a, \\ z &\mapsto c * b, \end{aligned}$$

see the following GAP session.

GAP session 2.7.5

```
gap> F1:=FreeGroupoid([1,2],[[1,1,2],[1,2,1],[2,3,2]]);; G:=F1/[];;
gap> g:=GeneratorsOfGroupoid(G);;   x:=g[1];;  y:=g[2];;  z:=g[3];;
gap> List(g,x->[Source(x),Target(x)]);
[ [ 1, 2 ], [ 1, 1 ], [ 2, 2 ] ]
gap> F2:=FreeGroupoid([5,7],[[7,1,7],[5,2,7],[7,3,5]]);; K:=F2/[];;
gap> k:=GeneratorsOfGroupoid(K);;   a:=k[1];;  b:=k[2];;  c:=k[3];;
gap> List(k,x->[Source(x),Target(x)]);
[ [ 7, 7 ], [ 5, 7 ], [ 7, 5 ] ]
gap> H:=GroupoidMorphismByImages(G,K,[[ x, y, z ], [ c, a, c*b ]]);
Objects Mapping : [ 1, 2 ] -> [ 7, 5 ]
Arrows Mapping  : [ f1 , f2 , f3 ] -> [ f3 , f1 , f3*f2 ]
gap> w:=y*x^-1*z*x*y^-1*x^-1*z^-1*x;;
gap> ImageOfArrow(H,w);
f1*f2*f3*f1^-1*f3^-1*f2^-1
```

The FpGd command ImageOfArrow produce the image of and arrow w under the homomorphism H, as shown in the last line of the GAP session.

2.8 Presentation of subgroups

Definition 2.8.1. An action of a group G on a set S is a mapping $G \times S \to S, (g, s) \mapsto {}^{g}s$ satisfying ${}^{g}({}^{h}s) = {}^{gh}s$ for all $g, h \in G, s \in S$ and the identity element of the group acts as the identity permutation.

Proposition 2.8.1. *Given an action of a group G on a set S we can construct a groupoid with object set S and with one arrow for each pair (g, s) in $G \times S$. The source of (g, s) is s and the target of (g, s) is ${}^{g}s$. We denote this groupoid by $\mathfrak{Gpd}(G, S)$.*

Proof. If $(h, {}^{g}s)$ and (g, s) are two arrows in $\mathfrak{Gpd}(G, S)$, then their composition is defined as $(h, {}^{g}s)(g, s) = (hg, s)$. Let $a_1 = (k, {}^{hg}s), a_2 = (h, {}^{g}s)$ and $a_3 = (g, s)$ be three arrows in $\mathfrak{Gpd}(G, S)$; their composition is associative, i.e. $(a_1a_2)a_3 = a_1(a_2a_3)$ since

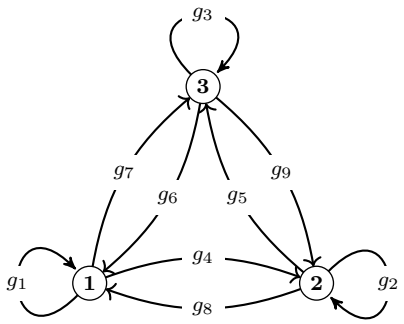
$$\begin{aligned}
(a_1a_2)a_3 &= ((k, {}^{hg}s)(h, {}^{g}s))(g, s) \\
&= (kh, {}^{g}s)(g, s) \\
&= ((kh)g, s) \\
&= (k(hg), s) \\
&= (k, {}^{hg}s)(hg, s)
\end{aligned}$$

$$\begin{aligned} &= (k, {}^{hg}s)((h, {}^{g}s)(g, s)) \\ &= a_1(a_2a_3). \end{aligned}$$

The identity element e of the group G defines the identity arrow (morphism) $(e, s) = 1_s$ at each object $s \in S$. The inverse of any arrow $a = (g, s)$ in the groupoid is the arrow $a^{-1} = (g^{-1}, {}^{g}s)$ where $a^{-1}a = 1_s$ and $aa^{-1} = 1_{{}^{g}s}$.

□

Example 2.8.1. Let $S = \{\mathbf{1}, \mathbf{2}, \mathbf{3}\}$ and G be the permutation group generated by (123). The groupoid $\mathfrak{Gpd}(G, S)$ has object set S and 9 generators $g_1 = ((), \mathbf{1}), g_2 = ((), \mathbf{2}), g_3 = ((), \mathbf{3}), g_4 = ((123), \mathbf{1}), g_5 = ((123), \mathbf{2}), g_6 = ((123), \mathbf{3}), g_7 = ((132), \mathbf{1}), g_8 = ((132), \mathbf{2}), g_9 = ((132), \mathbf{3})$, as shown in the following picture.



Proposition 2.8.2. *Suppose that a group G acts on a set S and that $\underline{x}$ is a set of generators for G. Then the groupoid $\mathfrak{Gpd}(G, S)$ is generated by the collection of arrows $\underline{x} \times S = \{(x, s) : x \in \underline{x}, s \in S\}$.*

Proof. An arbitrary arrow (g, s) in $\mathfrak{Gpd}(G, S)$ can be expressed as

$$(x_1^{\epsilon_1}, x_2^{\epsilon_2}...x_n^{\epsilon_n}s)(x_2^{\epsilon_2}, x_3^{\epsilon_3}...x_n^{\epsilon_n}s)...(x_{n-1}^{\epsilon_{n-1}}, x_n^{\epsilon_n}s)(x_n^{\epsilon_n}, s)$$

where

$$g = x_1^{\epsilon_1}x_2^{\epsilon_2}...x_n^{\epsilon_n}$$

and $x_i \in \underline{x}, \epsilon_i = \pm 1$. If $\epsilon_i = -1$ then

$$(x_i^{-1}, s) = (x_i, {}^{x_i^{-1}}s)^{-1}.$$

Each arrow in $\mathfrak{Gpd}(G, S)$ is a sequence of arrows in $\underline{x} \times S$. □

Let G be a group with subgroup U. Let $G/U = \{gU : g \in G\}$ denote the collection of left cosets $gU = \{{}^{g}u : u \in U\}$. There is an action of G on the set $X = G/U$ given by $(g, hU) \to ghU$ for $g, h \in G$. This action gives rise to a groupoid $\mathfrak{Gpd}(G, U)$.

Proposition 2.8.3. *For a group G and subgroup U the groupoid $\mathfrak{Gpd}(G, U)$ is connected and all vertex groups are isomorphic to U.*

Proof. The object set of the groupoid $\mathfrak{Gpd}(G,U)$ is

$$\{U, U_1, ..., U_n\}, \quad \text{where } n = \text{Index}(U) - 1.$$

Since any coset $U_i = y_i U$ for some $y_i \in G$, the groupoid is connected.
To prove that all vertex groups are isomorphic to U, let us choose any object U_i and any element $x = x_1^{\epsilon_1}...x_n^{\epsilon_n}$ such that $x_2^{\epsilon_2}...x_n^{\epsilon_n}U = U_i$. The map $h \mapsto x^{-1}hx$ is an isomorphism between the vertex group at U to the vertex group at U_i. □

Proposition 2.8.4. *Let $G = \langle \underline{x} \mid \underline{r} \rangle$ be a finitely presented group with finite index subgroup U. Then the groupoid $\mathfrak{G} = \mathfrak{Gpd}(G,U)$ is finitely presented as follows. The objects of $\mathfrak{G}$ are the left cosets gU. The generators of $\mathfrak{G}$ are the arrows (x, gU) for $x \in \underline{x}$. Each relator $r = x_1^{\epsilon_1}x_2^{\epsilon_2}...\ x_n^{\epsilon_n} \in \underline{r}$ and coset gU give rise to a word*

$$(r, gU) = (x_1^{\epsilon_1}, x_2^{\epsilon_2}...\ x_n^{\epsilon_n}gU)...(x_{n-1}^{\epsilon_{n-1}}, x_n^{\epsilon_n}gU)(x_n^{\epsilon_n}, gU) \tag{2.7}$$

in the groupoid generators. These words (r, gU) are the relators for the groupoid.

Proof. Let $F(\underline{x})$ be the free group on $\underline{x}$. Let R denote the normal subgroup of F normally generated by $\underline{r}$. It yields

$$F/R \cong G = \langle \underline{x} \mid \underline{r} \rangle$$

Let U be a subgroup of the group G and let G/U be the set of left cosets of U in G. Let $\mathfrak{G}$ denote the finitely presented groupoid $\mathfrak{Gpd}(G,U)$. By definition $\mathfrak{G}$ is generated by the set

$$\underline{x}' = \{(x, gU) \mid x \in \underline{x}, gU \in G/U\}.$$

Let $\mathfrak{F}$ be the free groupoid generated by $\underline{x}'$ (i.e. $\mathfrak{F} = \mathfrak{Gpd}(F,U)$). So each arrow $a \in \mathfrak{F}$ can be expressed as

$$a = (g_i, S_j),$$

where $S_j \in F/U$ and

$$g_i = x_{i_1}^{\epsilon_{i_1}} x_{i_2}^{\epsilon_{i_2}}...x_{i_k}^{\epsilon_{i_k}} \in F$$

There is a groupoid homomorphism $\phi : \mathfrak{F} \to \mathfrak{G}$ such that the kernel of ϕ consists of all arrows of the form (g_i, gU) for which the source and target is gU. That means $\phi(g_i) = 1_{gU}$ and that yields $g_i \in R$. It is readily seen that $\langle \underline{r} \rangle = R$.

□

Example 2.8.2. For $G = \langle x, y \mid x^2, y^2, (xy)^3 \rangle$ and $U = \langle xy \rangle \leqslant G$ the groupoid $\mathfrak{Gpd}(G,U)$ has presentation

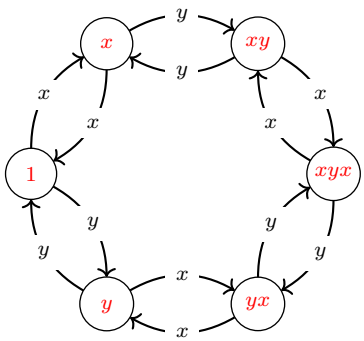
$$\langle \bar{x}, \bar{y}, \bar{z}, \bar{w} \mid \bar{x}\bar{y}, \bar{z}\bar{w}, (\bar{x}\bar{w})^3, \bar{y}\bar{x}, \bar{w}\bar{z}, (\bar{y}\bar{z})^3 \rangle \tag{2.8}$$

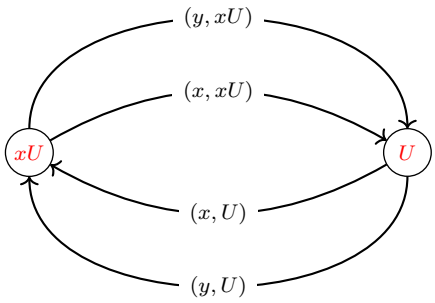
where $\bar{x} = (x, xU), \bar{y} = (x, U), \bar{z} = (y, xU), \bar{w} = (y, U)$.
Since the group G is generated by x and y and there are only two left cosets

$$G/U = \{ \ \ U := \{1, xy, yx\}, \quad xU := \{x, y, xyx\} \ \ \} \tag{2.9}$$

then the groupoid $\mathfrak{Gpd}(G, U)$ has four generators. The relators of this groupoid can be obtained by using the formula 2.7 and $x^2 = 1, y^2 = 1, (xy)^3 = 1$ as follows.

$$\begin{aligned}
(x^2, U) &= (x, xU)(x, U) = \bar{x}\bar{y} && (2.10)\\
(y^2, U) &= (y, yU)(y, U) = (y, xU)(y, U) = \bar{z}\bar{w} && (2.11)\\
((xy)^3, U) &= (x, y(xy)^2U)(y, (xy)^2U)(x, yxyU)(y, xyU)(x, yU)(y, U)\\
&= (x, xU)(y, U)(x, xU)(y, U)(x, xU)(y, U)\\
&= ((x, xU)(y, U))^3 = (\bar{x}\bar{w})^3 && (2.12)\\
(x^2, xU) &= (x, x^2U)(x, xU) = \bar{y}\bar{x} && (2.13)\\
(y^2, xU) &= (y, yxU)(y, xU) = (y, U)(y, xU) = \bar{w}\bar{z} && (2.14)\\
((xy)^3, xU) &= (x, (yx)^3U)(y, x(yx)^2U)(x, (yx)^2U)(y, xyxU)(x, yxU)(y, xU)\\
&= (x, U)(y, xU)(x, U)(y, xU)(x, U)(y, xU)\\
&= ((x, U)(y, xU))^3 = (\bar{y}\bar{z})^3 && (2.15)
\end{aligned}$$





The algorithm for finding a finite presentation of the vertex group of a finitely presented connected groupoid can be applied to the groupoid $\mathfrak{Gpd}(G, U)$ arising from a finitely presented group G and a finitely generated subgroup U in order to obtain a finite presentation for U.

Example 2.8.3. We will apply this idea to $G = \langle x, y \mid x^2 = 1, y^2 = 1, (xy)^3 = 1\rangle$ and $U = \langle xy\rangle$ to produce the finite presentation for U. From the presentation 2.8 the vertex group on U is generated by four generators corresponding to the generators of $\mathfrak{Gpd}(G, U)$. Now by looking on the underlying graph in the figure above (right), any tree in this graph includes only one edge, we can pick any one, say $\bar{x} = (x, xU) = X$.

Now, we should write other edges as loops at U,

$$\bar{y} = (x, U) \rightarrow (x, xU)(x, U) = Y \tag{2.16}$$
$$\bar{z} = (y, xU) \rightarrow (y, xU)(x, xU)^{-1} = Z \tag{2.17}$$
$$\bar{w} = (y, U) \rightarrow (x, xU)(y, U) = W \tag{2.18}$$

The relators of the groupoid each is equal to 1_U, we only need to rewrite these relators in terms of X, Y, Z and W.

$$(x, xU)(x, U) \rightarrow Y \tag{2.19}$$
$$(y, xU)(y, U) \rightarrow ZW \tag{2.20}$$
$$((x, xU)(y, U))^3 \rightarrow W^3 \tag{2.21}$$

Similarly, for the other 3 relators, we obtain $Y = 1, WZ = 1, Z^3 = 1$. From the last equations, we distinguish that $Y = 1$ and $Z = W^3$ which implies that $W^3 = 1$. The vertex group at U has the presentation

$$\langle X, Y, Z, W \mid X, Y, Z, W^3 \rangle$$

GAP session 2.8.3

```
gap> F:=FreeGroup(2);;
gap> G:=F/[F.1^2,F.2^2,(F.1*F.2)^3];;
gap> Size(G);
6
gap> x:=G.1;; y:=G.2;;
gap> S:=Subgroup(G,[x*y]);;
gap> U:=Image(IsomorphismFpGroup(S));
gap> RelatorsOfFpGroup(U);
[ F1^3 ]
gap> H:=FpGroupoid(G,[x*y]);;
gap> g:=GeneratorsOfGroupoid(H);;
gap> List(g,x->[Source(x),Target(x)]);
[ [ 2, 1 ], [ 1, 2 ], [ 2, 1 ], [ 1, 2 ] ]
gap> RelatorsOfFpGroupoid(H);
[ f2*f1, f4*f3, (f4*f1)^3, f1*f2, f3*f4, (f3*f2)^3 ]
gap> V:=VertexGroup(H,Source(g[1]));
<fp group on the generators [ f1, f2, f3, f4 ]>
gap> RelatorsOfFpGroup(V);
[ f2, f4*f3, f4^3, f2, f3*f4, (f3*f2)^3, f1 ]
```

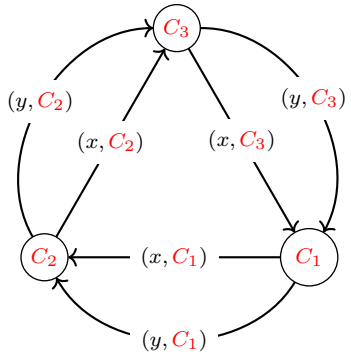
```
V:=SimplifiedFpGroup(V);
< fp group on the generators [ f3 ] >
gap> RelatorsOfFpGroup(V);
[ f3^3 ]
```

The following example involves a finite index subgroup of some infinite fp group.

Example 2.8.4. Let G be an infinite group with the following presentation

$$\langle x, y \mid xyx(yxy)^{-1} \rangle,$$

and let H be a subgroup of G generated by the three elements $yx^{-1}, y^{-1}x, x^3$. Let $\Omega = \{gH \mid g \in G\}$ be the set of the distinct left cosets which include exactly three left cosets, $C_1 = H, C_2 = xH$ and $C_3 = x^2H$. The fp groupoid induced by G and H is generated by six generators, see the following graph that generates $\mathfrak{G}(G, H)$.



The relators of the groupoid $\mathfrak{G}(G, H)$ can be obtained by using the formula 2.7 as follws.

$$\begin{aligned}
(xyx(yxy)^{-1}, H) &= (x, yx(yxy)^{-1}H)(y, x(yxy)^{-1}H)(x, (yxy)^{-1}H) \\
&\quad (y^{-1}, (yx)^{-1}H)(x^{-1}, y^{-1}H)(y^{-1}, H)
\end{aligned}$$

$$\begin{aligned}
&= (x, x^2H)(y, xH)(x, H)(y^{-1}, xH)(x^{-1}, x^2H)(y^{-1}, H) \\
&= (x, C_3)(y, C_2)(x, C_1)(y^{-1}, C_2)(x^{-1}, C_3)(y^{-1}, C_1) \\
(xyx(yxy)^{-1}, xH) &= (x, C_1)(y, C_3)(x, C_2)(y^{-1}, C_3)(x^{-1}, C_1)(y^{-1}, C_2) \\
(xyx(yxy)^{-1}, x^2H) &= (x, C_2)(y, C_1)(x, C_3)(y^{-1}, C_1)(x^{-1}, C_2)(y^{-1}, C_3)
\end{aligned}$$

The following GAP session shows the result using our implementation.

GAP session 2.8.5

```
gap> F:=FreeGroup(2);;
gap> x:=F.1;;  y:=F.2;;
gap> G:=F/[x*y*x*(y*x*y)^-1];;
gap> L:=LowIndexSubgroupsFpGroup(G,3);;
gap> H:=L[4];;
gap> GeneratorsOfGroup(H);
[ f2*f1^-1, f2^-1*f1, f1^3 ]
gap> K:=FpGroupoid(G,H);;
gap> g:=GeneratorsOfGroupoid(K);;
gap> V:=VertexGroup(K,Source(g[1]));;
gap> W:=SimplifiedFpGroup(V);
<fp group on the generators [ f3, f4, f6 ]>
gap> RelatorsOfFpGroup(W);
[ f6*f4^-1*f6^-1*f4^-1, f4*f3*f4*f6^-2*f3 ]
```

2.8.1 Implementation of fp groupoids induced by group actions

In order to get the presentation of a subgroup H of a finite fp group G, we need to create an fp groupoid induced by the group action of G on H. We then evaluate the vertex group on the subgroup under consideration. This is one of the applications of the groupoid techniques. We implement Propositions 2.8.2 and 2.8.4 This implementation follows the Algorithm 2.8.1.

Algorithm 2.8.1 Presentation of fp groupoid induced by group action.

Input: An fp group G and a list L of elements in G that generate a finite index subgroup H in G.
Output: A finite presentation for the groupoid $\mathfrak{G} = \mathfrak{Gpd}(G, H)$.

1: **procedure**
2: obj($\mathfrak{G}$) = G/H.
3: set gens($\mathfrak{G}$)= [ ].
4: **for** x in GeneratorsOfGroup(G) **do**
5: **for** c in obj($\mathfrak{G}$) **do**
6: add(gens($\mathfrak{G}$),${}^{x}c$)
7: **end do**
8: **end do**
9: set rels($\mathfrak{G}$) = [ ]
10: **for** r in RelatorsOfFpGroup(G) **do**
11: **for** c in obj($\mathfrak{G}$) **do**
12: add(rels($\mathfrak{G}$),${}^{r}c$)

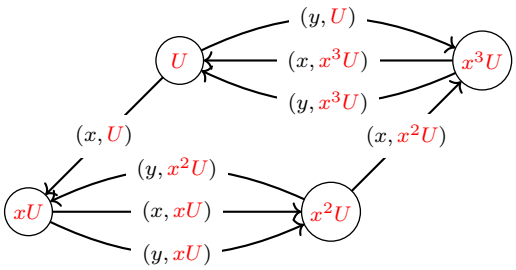
13: **end do**
14: **end do**
15: **return** FpGroupoid(obj($\mathfrak{G}$), gens($\mathfrak{G}$), rels($\mathfrak{G}$))
16: **end procedure**

The following example explains this method with dihedral group D_8.

Example 2.8.5. Consider the dihedral group of order 8,

$$D_8 = \langle x, y \mid x^4, y^2, (xy)^2 \rangle.$$

Let U be the subgroup of D_8 generated by xy. The left cosets are: $U = \{1, xy\}$, $xU = \{x, x^2y\}$, $x^2U = \{x^2, x^3y\}$ and $x^3U = \{x^3, y\}$.
The groupoid $\mathfrak{G} = \mathfrak{Gpd}(D_8, U)$ consists of four objects $\{U, xU, x^2U, x^3U\}$, and a set of generators $\{(x,U), (x,xU), (x,x^2U), (x,x^3U), (y,U), (y,xU), (y,x^2U), (y,x^3U)\}$.



The following GAP session shows the result using our implementation.

GAP session 2.8.5

```
gap> F:=FreeGroup(2);;
gap> G:=F/[F.1^4, F.2^2, (F.1*F.2)^2];; x:=G.1;; y:=G.2;;
gap> S:=Subgroup(G,[x*y]);;
gap> U:=Image(IsomorphismFpGroup(S));;RelatorsOfFpGroup(U);
[ F1^2 ]
gap> H:=FpGroupoid(G,[x*y]);; g:=GeneratorsOfGroupoid(H);;
gap> V:=VertexGroup(H,Source(g[1]));
gap> RelatorsOfFpGroup(V);
[ f5^2 ]
```

Chapter 3

Fundamental Groups, Groupoids and the van Kampen Theorem

3.1 Introduction

In the previous chapter, we described our GAP implementation of finitely presented groupoids and showed some applications in group theory. The fundamental groupoid of a topological space is an important example of a finitely presented groupoid. This chapter is devoted to introducing an implementation of the fundamental groupoid of a finite regular CW-space. We also implement the van Kampen theorem for fundamental groupoids due to R. Brown [16]. The van Kampen theorem yields an efficient algorithm for the distributed computation of finite presentations of fundamental groupoids and groups of large regular CW-complexs.

We first describe an algorithm for computing a finite presentation of the fundamental groupoid of a finite regular CW-space. The algorithm is based on simple homotopy collapses and uses the notion of discrete vector fields (due to [59] and [35]) to describe sequences of simple homotopy collapses. The algorithm is able to produce small presentations for some large CW-complexes arising in applied topology. In fact, the algorithm only requires the construction of a discrete vector field on the 3-skeleton of a space.

The chapter ends with a statement and computer implementation of the Seifert-van Kampen theorem for fundamental groupoids. There are certain advantages to working with groupoids in the computational setting of the Seifert-van Kampen theorem.

In the following section, we recall the definition of a discrete vector field and some important facts that will be used for the construction of the fundamental groupoid.

3.2 Discrete vector fields

Definition 3.2.1. [35, 59, 29] A *discrete vector field* V on a regular CW-space X is a collection of arrows $s \to t$ where

1. 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* of the arrow.

2. any cell is involved in at most one arrow.

A cell in X is said to be *critical* if it is not involved in an arrow.

Definition 3.2.2. [35, 59, 29] Let V be a discrete vector field on a regular CW-space X. Then a *chain* ρ of length n from a k-dimensional cell e_i^k to a k-dimensional cell e_j^k is a sequence of arrows $s_0 \to t_0, s_1 \to t_1, ..., s_{n-1} \to t_{n-1} \subset V$ satisfying:

1. $e_i^k = s_0$ and e_j^k is in the boundary of t_{n-1},

2. s_{i+1} lies in the boundary of t_i for all $0 \leqslant i < n - 1$.

A chain ρ is said to be a *circuit* if s_0 lies in the boundary of t_{n-1}.

Definition 3.2.3. [29] A discrete vector field is *admissible* if there is no circuit and for every source cell s, the length of any path starting from s is bounded by a fixed integer $\lambda(s)$, i.e. there is no chain of infinite length.

The following theorem can be viewed as a statement about simple homotopy equivalences phrased in the language of discrete vector fields. An explanation of this theorem is provided in [29].

Theorem 3.2.1. [92, 34, 29] *If X is a regular CW-space with admissible discrete vector field then there is a homotopy equivalence*

$$X \simeq Y$$

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

Example 3.2.1. We exhibit an easy example about building a discrete vector field on some given space. Consider the space of Figure 3.1 given by its skeleton (CW-complex) involving ten 0-cells, thirteen 1-cells and two 2-cells. Then a vector field (red arrows) is created. We distinguish some special cells (critical cells) indicated by a different color which are not involved in any arrow. These critical cells make $S^1 \vee S^1$ which is homotopy equivalent to the given space.

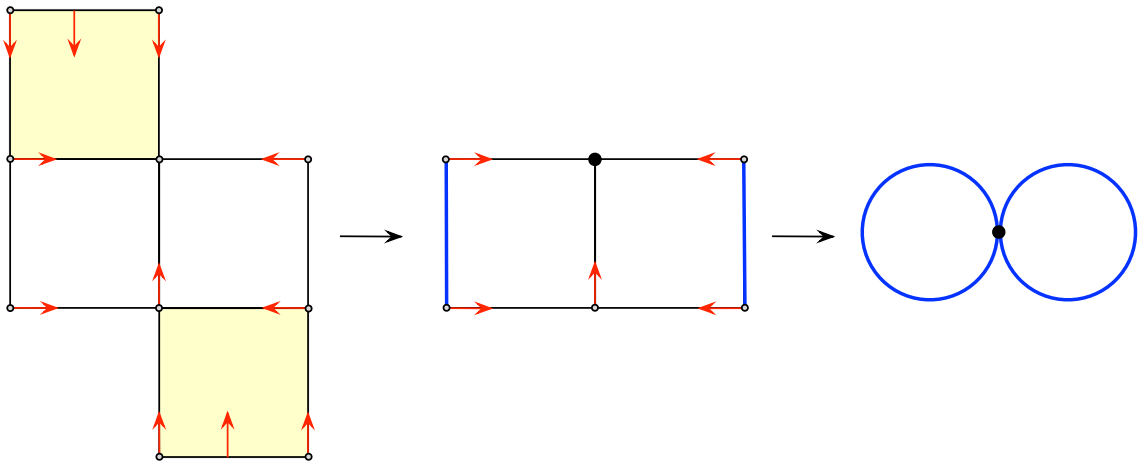


Figure 3.1 Stages of obtaining a small space (on the right) homotopy equivalent to a given space (on the left) by creating a discrete vector field on the regular CW-space.

Example 3.2.2. One method for representing a non-regular CW-complex X on a computer is via a simple homotopy equivalence $h : X \simeq Y$ where Y is regular, and where the simple homotopy equivalence is represented by a discrete vector field. An example of a discrete vector field on a regular CW-decomposition of a torus is illustrated in Figure 3.2, a non-regular CW-structure of a torus, and a regular CW-structure on a torus equipped with an admissible discrete vector field. It has one critical 0-cell, two critical 1-cells and one critical 2-cell. Let X be the torus and Y is a CW-space with one 0-cell, two 1-cells and one 2-cell as illustrated in Figure 3.2 (right). There is a homotopy equivalence $h : X \to Y$. This example is adapted from [29].

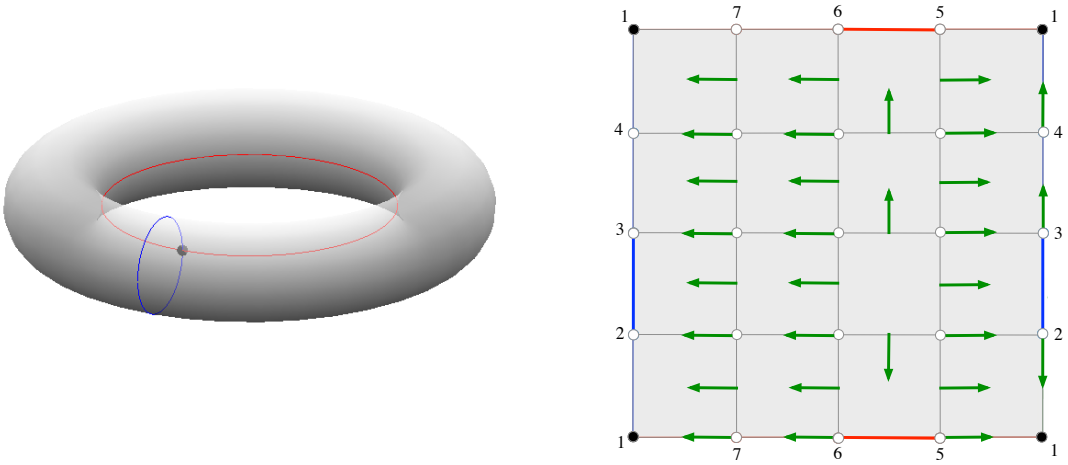


Figure 3.2 A non-regular CW-structure of a torus (left), and a regular CW-structure on a torus equipped with an acyclic discrete vector field (right).

Definition 3.2.4. An admissible discrete vector field is called *maximal* if it is not

possible to add an arrow while retaining admissibility.

There are many accounts in the literature of algorithms for constructing admissible discrete vector fields (see for instance[10, 31, 51, 29]). For connected X some of these algorithms ensure that precisely one of the 0-cells is critical. There are many approaches to constructing discrete vector fields, some of which are based on the following result.

Lemma 3.2.2. *Let X be a regular CW-space equipped with an admissible discrete vector field V. Suppose that there exist two critical cells $s, t \in X$ such that: $dim(t) = dim(s)+1$ and s lies in the boundary of t and any other cell of dimension $dim(s)+1$ containing s in its boundary is critical. Then the vector field V can be extended by adding the arrow $s \to t$ and the resulting discrete vector field is admissible.*

A GAP implementation of the construction of the discrete vector field on a given CW-space is available in HAP [31] which is based on Algorithm 1.5.1 [29].
The first step of the procedure is ordering the cells in some way that ensures any cell of dimension k is less than all cells of dimension $k+1$. 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.
The algorithm 1.5.1 in [29] is modified in this thesis by increasing the number of the critical 0-cells because the fundamental groupoid of a space is based on a set of base-points rather than one base-point.
Both algorithms produce an admissible discrete vector field V. The numbers of critical cells, when it is applied to some CW-complex X, satisfy the following formula

$$\chi(X) = \sum_{n=0}^{\dim(X)} (-1)^n \#\{\text{critical cells of } V \text{of dimension } n\},$$

where $\chi(X)$ is the Euler's characteristic of X.

We implement Algorithm 3.2.1 as a part of the function FundamentalGroupoid. Inputs to the algorithm are a regular CW-complex X and a set of 0-cells $X_0 \subset X^0$. It constructs a discrete vector field in which X_0 becomes the set of critical 0-cells.

Algorithm 3.2.1 Discrete vector field on a regular CW-complex.

Input: A finite regular CW-complex X and a set of 0-cells $X_0 \subset X^0$.
Output: A maximal admissible discrete vector field on X, with X_0 the set of critical 0-cells.

1: **procedure**
2: Partially order the cells of X in any fashion;
3: At any stage of the algorithm each cell will have precisely one of the following three states: (i) critical, (ii) potentially critical, (iii) non-critical.

Initially deem all cells of X to be potentially critical.
Deem all of X_0 to be critical;
while there exists a potentially critical cell **do**
    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**
        Choose such a pair (s, t) with s minimal in the given partial ordering.
        Add the arrow $s \to t$ and deem s and t to be non-critical.
    end while
      if there exists a potentially critical cell then
        Choose a minimal potentially critical cell and deem it to be critical.
      end if
end while
end procedure

3.3 Fundamental group of a topological space

A topological space X with some preferred point $x_0 \in X$ is called a *pointed space* and denoted by (X, x_0). The point x_0 is called a *base point*. For pointed spaces $(X, x_0), (Y, y_0)$, we say that a map $f : X \to Y$ is *pointed* if $f(x_0) = y_0$. Two pointed maps $f, g : X \to Y$ are *based homotopic* if there is a map $H : X \times [0, 1] \to Y$ with $H(x, 0) = f(x)$ and $H(x, 1) = g(x)$ for $x \in X$ and $H(x_0, t) = y_0$ for $t \in [0, 1]$. We denote by

$$[X, Y] = \{f : X \to Y : f(x_0) = y_0\}/ \simeq \tag{3.1}$$

the set of based homotopy classes of pointed maps $f : X \to Y$. We denote the based homotopy class of a pointed map f by $[f]$.
Let X be the unit circle $\mathbb{S}^1$. The classes defined in 3.1 admit a multiplication

$$* : [\mathbb{S}^1, Y] \times [\mathbb{S}^1, Y] \to [\mathbb{S}^1, Y], \tag{3.2}$$

defined below in Definition 3.3.1.
This defines a group which is called the *fundamental group* of Y denoted by $\pi_1(Y)$. The fundamental group $\pi_1(X, x_0)$ of a topological space X with base point x_0 describes when two paths starting and ending at x_0 can be continuously deformed into each other. It reflects geometric information about the basic shape or holes of the space. The fundamental group provides useful information about the space. For example, one can often show that two spaces are not homeomorphic, by establishing that their fundamental groups are not isomorphic.
A combinatorial description and an algorithm for finding a presentation for the fun-

damental group $\pi_1(X)$ of a finite CW-space X is introduced in [29]. The algorithm is illustrated in [29] on CW-spaces, involving large numbers of cells, that arise from the complements of protein backbones. The study of protein backbones leads on naturally to the general study of knot complements and an example is given in [29] that illustrates the computation of the peripheral system for a knot arising from a protein backbone.

Definition 3.3.1. Let X be a topological space. Let $x, y \in X$. A *path* from x to y is a map

$$\alpha : [0,1] \to X,$$

such that

$$\alpha(0) = x, \quad \text{and} \quad \alpha(1) = y.$$

If

$$\alpha : [0,1] \to X, \quad \text{and} \quad \beta : [0,1] \to X.$$

are paths from x and y, y to z, respectively, then $\beta \circ \alpha$ is a path from x to z defined by

$$(\beta \circ \alpha)(t) = \begin{cases} \alpha(2t) & 0 \leqslant t \leqslant \frac{1}{2}, \\ \beta(2t-1) & \frac{1}{2} \leqslant t \leqslant 1. \end{cases} \tag{3.3}$$

In particular, suppose we restrict attention to paths $f : [0,1] \to X$ with the same initial and ending point $f(0) = f(1) = x_0 \in X$. Such paths are called *loops*. The set of all based homotopy classes $[f]$ of loops $f : [0,1] \to X$ at the basepoint x_0 is denoted $\pi_1(X, x_0)$.

Proposition 3.3.1. [53] *$\pi_1(X, x_0)$ is a group with respect to the product $[\alpha][\beta] = [\alpha \circ \beta]$.*

The group $\pi_1(X, x_0)$ is called the *fundamental group* of the pointed topological space (X, x_0).

3.4 Fundamental groupoid of a space

Let X be a topological space. We can construct a category denoted by $\mathcal{P}(X)$, whose objects are the points of X and whose morphisms are all paths in X, with composition as defined in Eq. 3.3. $\mathcal{P}(X)$ is called the *path category* of X.

Let $\alpha : [0,1] \to X$ be a path in X. We call $\alpha(0)$ the *source* of α, denoted by $\mathbf{s}(\alpha)$ and $\alpha(1)$ is the target of α, denoted by $\mathbf{t}(\alpha)$. Given two paths α and β in X with $\mathbf{t}(\alpha) = \mathbf{s}(\beta)$, the composition $\beta \circ \alpha$ (or $\beta \cdot \alpha$) is defined as in Eq. 3.3. It is straightforward to check that $\mathcal{P}(X)$ is a category.

Let X and Y be topological spaces and $f : X \to Y$ a map. If α is a path in X, then $f \circ \alpha$ is a path in Y. And if α, β are two paths in X with $\mathbf{s}(\alpha) = \mathbf{t}(\beta)$, then

$$\begin{aligned}(f \circ (\alpha \cdot \beta))(t) &= \begin{cases}(f \circ \alpha)(2t) & 0 \leqslant t \leqslant \frac{1}{2}, \\ (f \circ \beta)(2t-1) & \frac{1}{2} \leqslant t \leqslant 1\end{cases} \\ &= ((f \circ \beta) \cdot (f \circ \alpha))(t).\end{aligned} \tag{3.4}$$

Thus the map f gives rise to a functor $\bar{f} : \mathcal{P}(X) \to \mathcal{P}(Y)$. The assignment $f \to \bar{f}$ is also functorial so we have a functor

$$\mathcal{P} : \mathsf{Top} \to \mathsf{Cat}$$

which assigns each space X in Top (category of topological spaces) its path category $\mathcal{P}(X)$ in Cat (the category of all path categories) and sends each continuous map f to $\bar{f}$.
For $x, y \in X$, let $\mathcal{P}(x, y)$ be the set of paths from x to y. We say that $\alpha, \beta \in \mathcal{P}(x, y)$ are path homotopic and write $\alpha \sim \beta$ if there is a map

$$\begin{aligned}H : [0,1] \times [0,1] &\to X \\ (t, s) &\mapsto H(t, s)\end{aligned}$$

called a *path-homotopy*, from α to β satisfying

$$\begin{aligned}H(t, 0) &= \alpha, \\ H(t, 1) &= \beta, \qquad t \in [0, 1],\end{aligned}$$

and

$$\begin{aligned}H(0, s) &= x, \\ H(1, s) &= y, \qquad s \in [0, 1].\end{aligned}$$

Proposition 3.4.1. *The relation $\sim$ is an equivalence relation on $\mathcal{P}(X)$.*

So we can form the quotient category $\mathcal{P}(X)/\sim$. It has the universal property that every functor from $\mathcal{P}(X)$ which sends homotopic paths to the same morphism uniquely factors through $\mathcal{P}(X)/\sim$.

Definition 3.4.1. Let X be a topological space and $V \subset X$. The *fundamental groupoid* classes of paths

$$\{p : [0,1] \to X \text{ with } p(0),\ p(1) \in V\}/\sim .$$

We denote this groupoid by $\pi_1(X, V)$.

Proposition 3.4.2. *Let X be a topological space with $V \subset X$. Every morphism in $\pi_1(X, V)$ is an isomorphism.*

Proof. Let $x, y \in V$, and $p \in \mathcal{P}(x, y)$. Let q be a path in $\mathcal{P}(y, x)$ given by $q : [0,1] \to X,\ t \mapsto p(1-t)$. It is obvious that $q \cdot p \in \mathcal{P}(x, x)$ and

$$\begin{array}{rcl} q \cdot p : [0,1] & \to & X \\ t & \mapsto & \begin{cases} p(2t) & 0 \leqslant t \leqslant \frac{1}{2}, \\ q(2t-1) = p(2-2t) & \frac{1}{2} \leqslant t \leqslant 1. \end{cases} \end{array}$$

Now define the following homotopy map

$$\begin{array}{rcl} H : [0,1] \times [0,1] & \to & X \\ (s,t) & \mapsto & \begin{cases} x & 0 \leqslant t \leqslant \frac{s}{2}, \\ p(t - s/2) & \frac{s}{2} \leqslant t \leqslant \frac{1}{2}, \\ q(t + \frac{s}{2}) & \frac{1}{2} \leqslant t \leqslant 1 - \frac{s}{2}, \\ x & 1 - \frac{s}{2} \leqslant t \leqslant 1. \end{cases} \end{array}$$

The mapping H is continuous and well-defined. Observe that $H(0, \cdot)$ is $q \cdot p$, $H(1, \cdot)$ is the identity map on $X, H(s, 0) = x$ and $H(s, 1) = x$. Thus $[q] \cdot [p] = \mathrm{id}_x$ in $\pi_1(X, V)$. Similarly, we can show that $[p] \cdot [q] = \mathrm{id}_y$. □

3.5 Implementation of presentations of fundamental groupoids

Let Y be a regular CW-complex with a set of base-points $Y_0 \subset Y^0$. Suppose the boundary vertices of a 1-cell e^1_i are labelled by e^0_{i-} and e^0_{i+}. We deem each edge to be directed from e^0_{i-} to e^0_{i+}.

Definition 3.5.1. A *combinatorial path* p in Y is a finite sequence

$$p = e^1_1, e^1_2, ..., e^1_k$$

of 1-cells such that

$$e^0_{i+} = e^0_{(i+1)-}$$

for $i = 1, 2, .., k-1$. The number of 1-cells is called the *length* of the path.
The 0-cells e^0_{1-} and e^0_{k+} are called the *initial vertex* $\mathbf{s}(p)$ and *final vertex* $\mathbf{t}(p)$ of the path p and referred to collectively as endpoints. If the endpoints of some path are the same then this path is said to be a *combinatorial loop*.

Notation. *We denote by* $\mathtt{P}(Y, Y_0)$ *the set of all combinatorial paths in* Y *that start and end at the 0-cells in* Y_0.

Let

$$\mathtt{P}(Y, Y_0) \times_{s,t} \mathtt{P}(Y, Y_0) = \{(\alpha, \beta) : \ \alpha, \beta \in \mathtt{P}(Y, Y_0), \ \mathbf{t}(\beta) = \mathbf{s}(\alpha)\}$$

The concatenation of combinatorial paths yields a function

$$\mathtt{P}(Y, Y_0) \times_{s,t} \mathtt{P}(Y, Y_0) \to \mathtt{P}(Y, Y_0)$$

Definition 3.5.2. Two combinatorial paths p and q in a regular CW-space Y are called *equivalent*, denoted by $p \simeq q$ if p and q have the following forms

$$\begin{aligned} p &= p_1^1, p_2^1, ..., p_l^1, p_{l+1}^1, ..., p_{n-k}^1, ..., p_n^1 \\ q &= q_1^1, q_2^1, ..., q_l^1, q_{l+1}^1, ..., q_{m-k}^1, ..., q_m^1 \end{aligned}$$

such that $p_i^1 = q_i^1$ for $1 \leqslant i \leqslant l$ and $p_{n-i}^1 = q_{m-i}^1$ for $0 \leqslant i \leqslant k$ and the rest of edges $p_{l+1}, ...p_{n-k+1}, q_{l+1}, ...q_{m-k+1}^1$ make the boundary of some 2-cell e_λ^2 in Y, see the Figure 3.3.

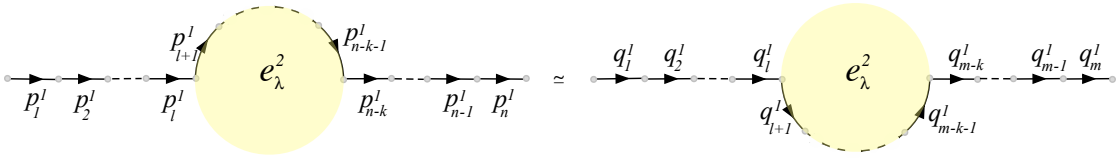


Figure 3.3 Equivalent paths in $\mathtt{P}(Y, Y_0)$.

Definition 3.5.3. The edge-path groupoid of Y is

$$\omega(Y, Y_0) \ = \ \mathtt{P}(Y, Y_0)/\simeq$$

with groupoid multiplication induced by the concatenation of paths in $\mathtt{P}(Y, Y_0)$.

Theorem 3.5.1. *If* Y *is a regular CW-complex with a set of base-points* Y_0 *then there is an isomorphism of groupoid*

$$\pi_1(Y, Y_0) \ \cong \ \omega(Y, Y_0)$$

Proof will be given after we have introduced the van Kampen Theorem.

Definition 3.5.4. Let Y be a path connected regular CW-space. The regular-subspace $T \subseteq Y^1$ is called a *spanning tree* of Y^1 if

1. $Y^0 \subset T$,

2. there is a unique path between any two 0-cells in T and

3. there is no loop in T.

Let Y be a path connected regular CW-space with a set of base points Y_0 and a spanning tree T. Any 0-cell $x \in Y^0$ determines a unique shortest path

$$P_T(x) \;=\; \{e_1^1, e_2^1, ..., e_k^1\}$$

in T with initial vertex equal to some base point $y \in Y_0$ and final vertex equal to x. There is a corresponding path

$$\bar{P_T}(x) \;=\; \{e_k^1, ..., e_2^1, e_1^1\}$$

in which the edges appear in reversed order, it starts at x and ends at y.
In a regular CW-space Y, any path $\gamma = \{e_1^1, e_2^1, ..., e_k^1\}$ of length $k \geqslant 1$ comes with an associated sequence of 0-cells $e_{1_-}^0, e_{1_+}^0, ..., e_{k_-}^0, e_{k_+}^0$. The path γ determines a new path

$$Q_T(\gamma) = P_T(e_{1_-}^0), e_1^1, \bar{P_T}(e_{1_+}^0), P_T(e_{2_-}^0), e_2^1, \bar{P_T}(e_{2_+}^0), ..., P_T(e_{k_-}^0), e_k^1, \bar{P_T}(e_{k_+}^0),$$

which is a concatenation of paths.
By an *orientation* on the 1-skeleton Y^1 we mean that for each 1-cell e^1 of Y some arbitrary but fixed ordering has been chosen on the two 0-cells in its boundary. We let $\partial^- e^1$ denote the first boundary vertex and $\partial^+ e^1$ denote the second. Given an orientation on Y^1 we define

$$Q_T(e^1) = P_T(\partial^- e^1), e^1, P_T(\partial^+ e^1) \tag{3.5}$$

to be the path obtained by concatenating the three paths $P_T(\partial^- e^1), e^1$ and $P_T(\partial^+ e^1)$. We denote by $[Q_T(e^1)]$ the equivalence class in the edge-path groupoid $\omega(Y)$ represented by $Q_T(e^1)$. We denote by $[[Q_T(e^1)]]$ the equivalence class in the edge-path groupoid $\omega(Y^1)$ represented by $Q_T(e^1)$. By an *orientation* on the 2-skeleton Y^2 we mean an orientation on Y^1 and that, additionally, for each 2-cell e^2 in Y an ordering has been placed on all the 1-cells $e_1^1, ..., e_k^1$ in its boundary so that $p(e^2) = \{e_1^1, ..., e_k^1\}$ is a path.
Let Y be a finite regular CW-space equipped with an admissible discrete vector field and an orientation on Y^1. We say that a cell in Y is *terminal* if it is either critical or is non-critical and the *target* of an arrow. We say that the cell is *initial* if it is the source of an arrow. Thus each cell is either terminal or initial.
Any 0-cell e^0 in Y can be associated with a unique terminal 0-cell $H(e^0)$ by recur-

sively defining

$$H(e^0) = \begin{cases} e^0 & \text{if } e^0 \text{ is terminal} \\ H(e'^0) & \text{if there exist an arrow } e^0 \to e^1 \text{ with } e'^0 \text{ and} \\ & \quad e^0 \text{ the boundary cells of the 1-cells } e^1. \end{cases} \tag{3.6}$$

This recursion simply identifies the unique chain of arrows in the vector field that starts at e^0 and ends at a 1-cell whose boundary contains $H(e^0)$.

Example 3.5.1. Consider the regular CW-space Y of Figure 3.4 (left) involving 16 0-cells, 24 1-cells and 8 2-cells. This space is homotopy equivalent to the circle S^1. A discrete vector field has been constructed on Y. The function $H(e^0)$ of 3.6 sends, for instance, the 0-cell e^0_7 to the 0-cell e^0_1. The recursion is

$$H(e^0_7) = H(e^0_6) = H(e^0_{10}) = H(e^0_{14}) = H(e^0_{13}) = H(e^0_9) = H(e^0_5) = e^0_1,$$

see Figure 3.4 (right).

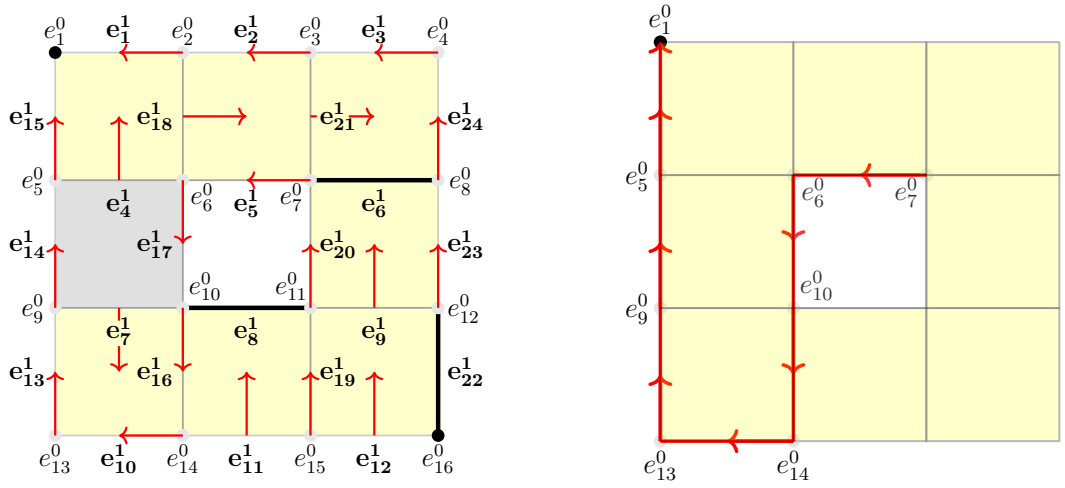


Figure 3.4 CW-complex endowed homotopy equivalent to the circle S^1 with a vector field (left), the recursion formula 3.6 applied on the 0-cell e^0_7 identifying a path from this cell to e^0_1 (right).

Any oriented 1-cell e^1 in Y can be associated with a unique path $H(e^1)$ of terminal 1-cells $H(e^1) = \{e^1_1, e^1_2, ..., e^1_n\}$ in Y which starts at the first boundary cell $\partial^- e^1$ of e^1 and ends at the second boundary cell $\partial^+ e^1$ of e^1. This association is explained below. We denote by $H(e^1)^{-1}$ the reversed path starting at $\partial^+ e^1$ and ending at $\partial^- e_1$. Once we have defined $H(e^1)$ we will be able to define , for any path of 1-cells $p = \{f^1_1, f^1_2, ..., f^1_m\}$ in Y, the path $H(p)$ to be the concatenation of the ordered sequence of paths $H(f^1_1)^{\epsilon_1}, H(f^1_2)^{\epsilon_2}, ..., H(f^1_m)^{\epsilon_m}$ with signs $\epsilon_i = \pm 1$ chosen to allow path concatenation.

For any initial 1-cell e^1 we have an associated 2-cell $e^1 \to e^2$. The boundary of e^2 specifies a path $f_1^1, f_2^1, ..., f_m^1$ from the vertex $\partial^- e^1$ to the vertex $\partial^+ e^1$, We denote this path by $\leftarrow e^1 \rightarrow$. The path involves all 1-cells of the boundary of e^2 except the 1-cell e^1.
We define $H(e^1)$ recursively by

$$H(e^1) = \begin{cases} e^1 & \text{if } e^1 \text{ is terminal} \\ H(\leftarrow e^1 \rightarrow) & \text{if } e^1 \text{ is initial.} \end{cases} \tag{3.7}$$

Example 3.5.1 (Continued)**.** The function $H(e^1)$ of 3.7 sends, for instance, the 1-cell e_4^1 to the 1-cell e_6^1. The recursion is

$$H(e_4^1) = H(e_{18}^1) = H(e_{21}^1) = e_6^1,$$

see Figure 3.5 (right).

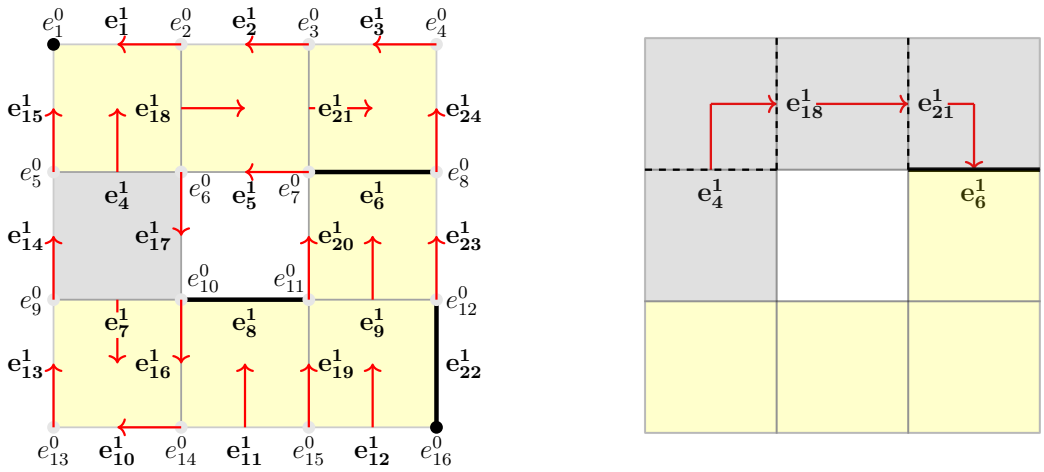


Figure 3.5 CW-complex endowed with a vector field (left), the recursion formula 3.7 applied on the 1-cell e_4^1 producing a sequence of 1-cells $\{e_4^1, e_{18}^1, e_{21}^1, e_6^1\}$ (right).

Note that in the recursive definitions of $H(e^0)$ and $H(e^1)$ the recursion will terminate because of the admissibility condition on the discrete vector field.

Proposition 3.5.2. *Let Y be a connected regular CW-space with Y_0 a set of base points. Let Y be equipped with an admissible discrete vector field. For any orientation on Y^2, the edge-path groupoid of Y is generated by the homotopy classes of paths corresponding to the critical 1-cells.*

Proof. Let Y be a regular CW-complex endowed with a discrete vector field V and let Y_0 be a set of base-points containing at least two 0-cells (otherwise the proof is similar to the proof of the edge-path group, see [29]). That makes the set of

all critical 1-cells Y_1 nonempty. Assume $Y_1 = \{e^1_1, ..., e^1_m\} \subset Y^1$. Use the recursion relation 3.6 to define a set of combinatorial paths $\mathcal{Y}_1 = \{p_1, ..., p_m\}$ and

$$p_k = P_T(e^0_{k_-})\ e^1_k\ P_T(e^0_{k_+}), \qquad 1 \leqslant k \leqslant m$$

where $e^0_{k_-}$ and $e^0_{k_+}$ are the boundary vertices of e^1_k. And $P_T(e^0_{k_+})$ is defined as in Eq. 3.5. This set of paths generates the groupoid. □

The algorithm 3.5.1 returns the presentation for the fundamental groupoid of a given CW-complex Y with a set of 0-cells $Y_0 \subset Y^0$.

Algorithm 3.5.1 Fundamental groupoid of a regular CW-complex

Input: A finite regular CW-complex Y and a set $Y_0 = \{y_0, ..., y_d\} \subset Y^0$ of basepoints which intersects non-trivially with each connected component of Y.
Output: A finitely presented groupoid G isomorphic to $\pi_1(Y, Y_0)$.

1: **procedure**
2: Create the 3-dimensional CW-complex Y^3 by forgetting all cells of Y in dimensions $\geqslant 4$.
3: Specify an orientation on 1-cells and 2-cells in Y^3.
4: Use Algorithm 3.2.1 to produce a maximal discrete vector field on Y^3 in which each base-point in Y_0 is critical and no other 0-cells are critical. (Use an ordering on cells such that each cell of dimension k is less than any cell of dimension $k+1$, and for each connected component of Y initially deem one 0-cell in Y_0 from this component to be critical. This will ensure that precisely one 0-cell per connected component will be critical. Having constructed a vector field, arrows can be omitted if necessary to ensure that every 0-cell in Y_0 is critical.)
5: Use the recursive definition in 3.7 to create the function that inputs a path p in Y^1 and returns the path $H(p)$ involving just terminal cells.
6: Create a free groupoid F with object set $Obj(F) = Y_0$ and with free generating set $y_1, ..., y_m$ corresponding to the critical 1-cells $e^1_1, ..., e^1_m$ of Y^3. The discrete vector field contracts the initial boundary vertex of e^1_i to a critical 0-cell in Y_0 and this critical 0-cell is the source of y_i. Similarly, the target boundary vertex of e^1_i is contracted to a critical 0-cell in Y_0 which is taken as the target of y_i.
7: Create a finite set R consisting of those words in F corresponding to the loops $H(Q_T(p(e^2)))$ for critical 2-cells e^2. In this context $Q_T(p(e^2))$ is a combinatorial path beginning at some vertex in Y_0 and ending at some vertex in Y_0, and need not be a loop.
8: Return the finitely presented groupoid G determined by F and R.
9: **end procedure**

Example 3.5.2 (The torus)**.** Figure 3.6 shows a regular CW-structure on the torus $Y = S^1 \times S^1$. Take Y_0 to be the three vertices numbered 1, 4 and 7. A discrete vector field has been constructed with these three vertices its only critical 0-cells. There are four critical 1-cells and one critical 2-cell. Following the procedure in Algorithm 3.5.1

we obtain a presentation for $\pi_1(Y, Y_0)$ with three objects 1, 4 and 7 and with four generators x, y, z and w corresponding to the critical edges with boundary vertices $\{1,6\}, \{2,3\}, \{4,5\}$ and $\{7,8\}$ respectively. Generator x has source 1 and target 7; generator y has source 1 and target 4; generator z has source 4 and target 1; the generator w has source 7 and target 1. The presentation has a single relator, namely $y^{-1}z^{-1}x^{-1}w^{-1}zywx$ representing the identity arrow at 1.
The vertex group of $\pi_1(Y, Y_0)$ at vertex $y_0 = 1$ is isomorphic to the fundamental group $\pi_1(Y, y_0)$. To extract a presentation for this group $\pi_1(Y, y_0)$ from the groupoid presentation we first regard the generating set $\{x, y, z, w\}$ as a directed graph on three vertices. We then construct a maximal tree in this graph. For instance, we can choose the tree consisting of the two edges y and w. We then 'contract' the maximal tree, which in practice means that we set $x' = wx, z' = zy$ and $r' = z'^{-1}x'^{-1}z'x'$ to obtain the group presentation $\pi_1(Y, y_0) = \langle x', z' \mid z'^{-1}x'^{-1}z'x' = 1 \rangle$.

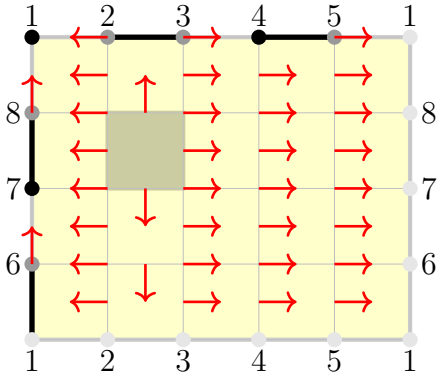


Figure 3.6 Admissible discrete vector field on the torus.

Definition 3.5.5. An *n-dimensional pure cubical complex* K is a subspace of n-dimensional Euclidian space arising as a union of finitely many n-dimensional unit cubes in $\mathbb{R}^n$ whose vertices have integral coordinates. A pure cubical complex can be created by specifying a n-dimensional array of 0s and 1s.

As an example consider the following 3-dimensional binary array

$$A = \left(\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \right) \tag{3.8}$$

This array corresponds to the following cubical complex.

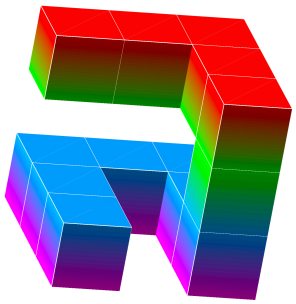


Figure 3.7 3-dimensional pure cubical complex corresponds to the array A given in Eq. 3.8.

Example 3.5.3. The following GAP session uses Algorithm 3.5.1 and Algorithm 2.7.2 to compute the fundamental groupoid of the pure cubical complex given in the Figure 3.8,

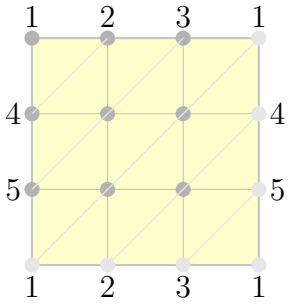


Figure 3.8 The pure cubical complex is homeomorphic to a torus.

and then computes the vertex group at some vertex which is the fundamental group of the torus.

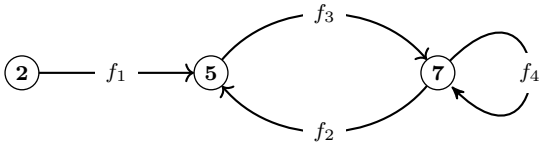


Figure 3.9 Generating graph of the fundamental groupoid of the torus $\pi_1(Y, Y_0)$.

GAP session 3.5.3

```
gap> L:= [ [ 1,2,4 ],[ 2,4,6 ],[ 2,3,6 ],[ 3,6,7 ],[ 1,3,7 ],
          [ 1,4,7 ],[ 4,5,6 ],[ 5,6,8 ],[ 6,7,8 ],[ 7,8,9 ],
          [ 4,7,9 ],[ 4,5,9 ],[ 1,5,8 ],[ 1,2,8 ],[ 2,8,9 ],
          [ 2,3,9 ],[ 3,5,9 ],[ 1,3,5 ] ];;
gap> Y:=RegularCWComplex(SimplicialComplex(L));;
gap> G:=FundamentalGroupoidOfRegularCWComplex(Y,[2,5,7]);
<fp groupoid on the generators [ f1 , f2 , f3 , f4 ]>
gap> g:=GeneratorsOfGroupoid(G);
[ f1, f2, f3, f4 ]
gap> List( g , x - > [ Source(x) , Target(x) ] );
[ [ 2 , 5 ] , [ 7 , 5 ] , [ 5 , 7 ] , [ 7 , 7 ] ]
gap> RelatorsOfFpGroupoid(G);
[ f1^-1*f3^-1*f4^-1*f3*f2*f4*f2^-1*f1 ]
gap> Gv:=Vertex_Group(G,7);
<fp group of size infinity on the generators [ f1, f2, f3, f4 ]>
gap> RelatorsOfFpGroup(Gv);
[ f2*f1*f2^-1*f1^-1, f3, f4 ]
gap> Sv:=SimplifiedFpGroupoid(Gv);
<fp group of size infinity on the generators [ f1, f2 ]>
gap> RelatorsOfFpGroup(Sv);
[ f2*f1*f2^-1*f1^-1 ]
```

Example 3.5.4. We can construct a 3-dimensional pure cubical space K involving three layers of cubes, as shown in the figure 3.10. This is homeomorphic to a double torus T^2. The construction of this pure cubical space and its fundamental groupoid $\pi_1(K, K_0)$, $K_0 = \{v_1, v_{10}\}$ is given in the following GAP session. The vertex group is a finitely presented group with four generators and one relator.

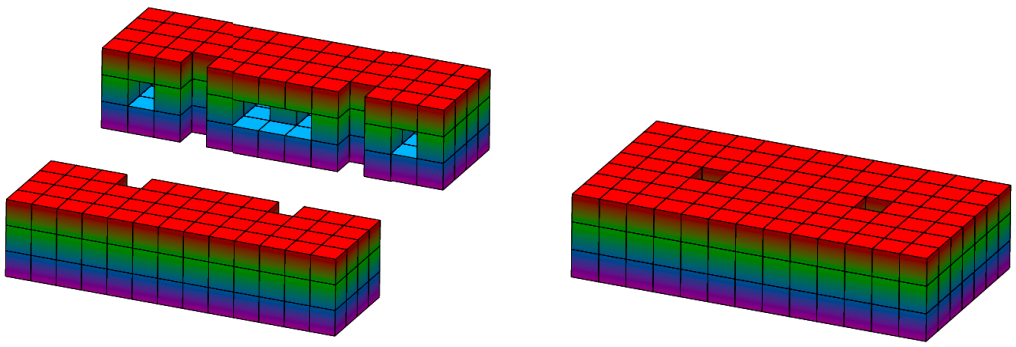


Figure 3.10 Pure cubical complex of the double torus T^2.

GAP session 3.5.4

```
gap> S:=List([1..3],x->List([1..13],y->List([1..7],z->1)));;
gap> for i in [1..3] do S[i][4][4]:=0;   S[i][10][4]:=0; od;;
gap> for i in [2..6] do S[2][2][i]:=0; S[2][6][i]:=0;
                        S[2][8][i]:=0; S[2][12][i]:=0;
     od;;
gap> for i in [3..5] do S[2][i][2]:=0; S[2][i+6][2]:=0;
                        S[2][i][6]:=0; S[2][i+6][6]:=0;
     od;;
gap> S[2][7][4]:=0;;
gap> K:=PureCubicalComplex(S);
Pure cubical complex of dimension 3.

gap> Y:=RegularCWComplex(K);;
gap> G:=FundamentalGroupoidOfRegularCWComplex(Y,[1,10]);
< fp groupoid on the generators [ f1, f2, f3, f4 ,f5, f6 ] >

gap> RelatorsOfFpGroupoid(G);
[ f2^-1*f6^-1*f4^-1*f2*f4*f6*f5^-1*f3^-1*f1*f3*f5*f1^-1 ]

gap> V:=VertexGroup(G,1);
#I  there are 4 generators and 1 relator of total length 8
<fp group of size infinity on the generators [ f1, f2, f3, f4 ]>

gap> RelatorsOfFpGroup(V);
[ f2^-1*f4^-1*f2*f4*f3^-1*f1*f3*f1^-1 ]
```

3.6 van Kampen's theorem

Let $\mathcal{G}, \mathcal{H}, \mathcal{K}$ be groupoids with object sets V_G, V_H, V_K respectively. Suppose that $V_K \subset V_G$ and $V_K \subset V_H$ and that $V_K = V_G \cap V_H$.
Suppose that

$$\phi : \mathcal{K} \to \mathcal{G}$$
$$\psi : \mathcal{K} \to \mathcal{H}$$

are morphisms of groupoids that are inclusions on object sets.
The *pushout groupoid* $\mathcal{G} *_{\mathcal{K}} \mathcal{H}$ is characterized up to isomorphism by the following universal property.

Universal property of a pushout. There are morphisms of groupoids

$$\mathcal{G} \to \mathcal{G} *_{\mathcal{K}} \mathcal{H}$$
$$\mathcal{H} \to \mathcal{G} *_{\mathcal{K}} \mathcal{H}.$$

For any groupoid morphisms

$$\theta : \mathcal{G} \to \mathcal{Q},$$
$$\theta' : \mathcal{H} \to \mathcal{Q}$$

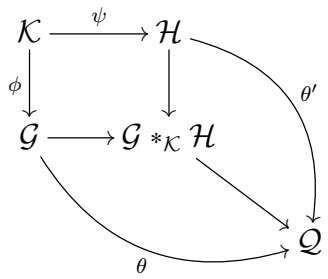
satisfying

$$\theta \circ \phi = \theta' \circ \psi$$

there is a unique groupoid morphism

$$\mathcal{G} *_{\mathcal{K}} \mathcal{H} \to \mathcal{Q}$$

such that the following diagram of groupoids commutes.



(3.9)

Given free presentations

$$\mathcal{K} \cong \langle \underline{x}_K \ : \ \underline{r}_K \rangle,$$
$$\mathcal{G} \cong \langle \underline{x}_G \ : \ \underline{r}_G \rangle,$$
$$\mathcal{H} \cong \langle \underline{x}_H \ : \ \underline{r}_H \rangle$$

it can be verified that the pushout in 3.9 has vertex set $V_G \cup V_H$ and presentation

$$\mathcal{G} *_{\mathcal{K}} \mathcal{H} \cong \langle \underline{x}_G \cup \underline{x}_H \mid \underline{r}_G \cup \underline{r}_H \cup \{\phi(x)\psi(x)^{-1} \ : \ x \in \underline{x}_K\} \rangle \tag{3.10}$$

As a special case, we have the following definition.

Definition 3.6.1. Let $\phi : K \to G$ and $\psi : K \to H$ be group homomorphisms. Suppose that we have free presentations $G \cong \langle \underline{x} \ : \ \underline{r} \rangle$ and $H \cong \langle \underline{y} \ : \ \underline{s} \rangle$ and a set $\underline{w} \subset K$ that generates K. For each $w \in \underline{w}$ let the image $\phi(w)$ be represented by some word w_ϕ in the free group on $\underline{x}$ and let the image $\psi(w)$ be represented by some word w_ψ in the free group on $\underline{y}$. Then the *amalgamated free product* $G *_K H$

is a group defined up to isomorphism by the presentation

$$G *_K H \cong \langle \underline{x} \cup \underline{y} : \underline{r} \cup \underline{s} \cup \{w_\phi w_\psi^{-1} : w \in \underline{w}\}\rangle. \tag{3.11}$$

The group $G *_K H$ is also referred to as the *pushout* of the homomorphisms ϕ and ψ. It is called the *free product* of G and H in the case when K is trivial.

Theorem 3.6.1 (Seifert and van Kampen [16])**.** *Let Y be a path-connected topological space with base-point $y_0 \in Y$. Let $A, B \subset Y$ be open and path-connected subsets such that: i) $Y = A \cup B$ and ii) the intersection $A \cap B$ is path-connected and contains y_0. Then there is an isomorphism*

$$\pi_1(Y) \cong \pi_1(A) *_{\pi_1(A\cap B)} \pi_1(B) \tag{3.12}$$

where the homomorphisms $\pi_1(A \cap B) \to \pi_1(A)$ and $\pi_1(A \cap B) \to \pi_1(B)$ are induced by the inclusion $A \cap B \hookrightarrow A$ and $A \cap B \hookrightarrow B$.

Theorem 3.6.2. [16, 13] *Let Y be a topological space with subspaces Y_1, Y_2. Let $\mathring{Y}_i$ denote the interior of Y_i. Let Y_0 be a set of points in Y that has non-empty intersection with each connected component of $Y_1 \cap Y_2$, with each connected component of Y_1, and with each connected component of Y_2. Then the induced square of fundamental groupoids*

$$\begin{array}{ccc} \pi_1(Y_1 \cap Y_2, Y_1 \cap Y_2 \cap Y_0) & \longrightarrow & \pi_1(Y_2, Y_2 \cap Y_0) \\ \downarrow & & \downarrow \\ \pi_1(Y_1, Y_1 \cap Y_0) & \longrightarrow & \pi_1(Y, Y_0) \end{array} \tag{3.13}$$

is a pushout in the category of groupoids.

Figure 3.11 shows a space Y with subspaces Y_1 and Y_2. The intersection of the subspaces consists of three disjoint pieces. In order to calculate the fundamental groupoid of Y, one must select at least one point in each piece, say $Y_0 = \{y_0, y_1, y_2\}$.

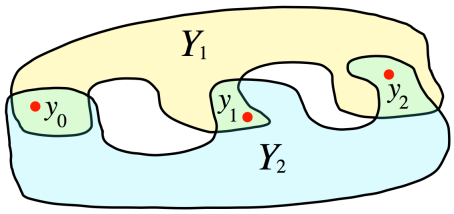
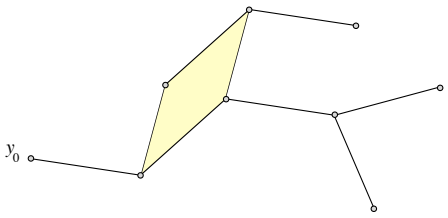


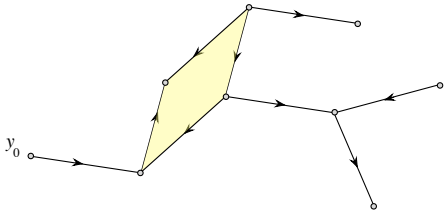
Figure 3.11 Two subspaces Y_1 and Y_2 of a space $Y = Y_1 \cup Y_2$ with $Y_0 = \{y_0, y_1, y_2\} \subset Y_1 \cap Y_2$.

Lemma 3.6.3. *A subgroupoid of a free groupoid is free.*

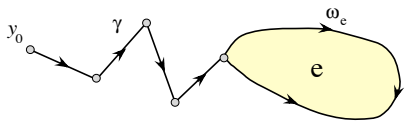
To prove Theorem 3.5.1, let us first give an alternative description of $\omega(Y, Y_0)$, where Y is a regular CW-space and Y_0 is a subset of its vertices.
The 1-skeleton Y^1 is a graph.



We can choose an arbitrary orientation on the edges of Y^1,



to make Y^1 a directed graph.
We let $\mathcal{P}(Y^1)$ denote the free groupoid on Y^1 (see Chapter 2). Let $\mathcal{P}(Y^1, Y_0)$ denote the subgroupoid of $\mathcal{P}(Y^1)$ consisting of those elements with source and target in Y_0.
Each 2-cell e in Y has a boundary that determines a circuit ω_e in $\mathcal{P}(Y^1)$.
Let γ be a simple path in Y^1 from some $y_0 \in Y_0$ to the start/end vertex of the circuit ω_e.

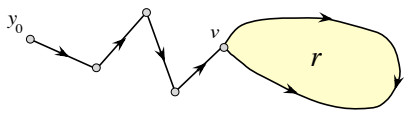


Then $r_e = \gamma^{-1}\omega_e\gamma$ is an element of $\mathcal{P}(Y^1, Y_0)$.
The edge path groupoid $\omega(Y, Y_0)$ can be described as the quotient of the groupoid $\mathcal{P}(Y^1, Y_0)$ by the relators r_e for e a 2-cell in Y.

If $\dim(Y) = 1$ then clearly $\omega(Y)$ is free. [The Lemma 3.6.3 holds essentially because a subgroup of free group is free.

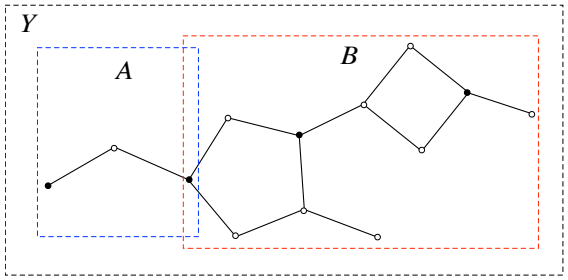
$$\omega(Y, Y_0) \subset \omega(Y).$$

If $\omega(Y, Y_0)$ not free then we have a relator



Then r is a relator for the vertex group at v. But a vertex group in $\omega(Y, Y_0)$ is a subgroup of a vertex group in $\omega(Y)$. So it is free.]

The fundamental groupoid of a space with only one 1-cell is clearly free when the set of base-points include its boundary vertices. $\pi_1(Y, Y_0)$ is free. Consider the following space Y with a set of base-point, Y_0 marked by solid black nodes and the space is a union of two pieces A and B.

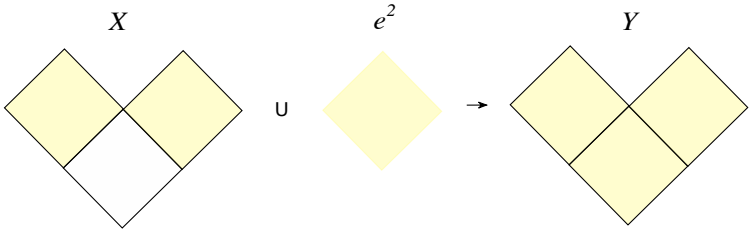


Then by using van-Kampen Theorem

$$\pi_1(Y^1, Y_0) = \pi_1(A, A \cap A_0) *_{\pi_1(A\cap B, A\cap B\cap Y_0)} \pi_1(B, B \cap Y_0)$$

Thus $\pi_1(Y^1, Y_0)$ is free by induction.

Now, let $Y = X \cup e^2$ is shown in the following



It is obvious to see the inclusion maps in the following diagram

$$\begin{array}{ccc} Y^1 & \hookrightarrow & Y^1 \cup e^2 \\ \downarrow & & \downarrow \\ X & \hookrightarrow & Y \end{array}$$

which lead to the following pushout

$$\begin{array}{ccc} \pi_1(Y^1, Y_0) & \longrightarrow & \pi_1(Y^1 \cup e^2, Y_0) \\ \downarrow & & \downarrow \\ \pi_1(X, Y_0) & \longrightarrow & \pi_1(Y, Y_0) \end{array} \tag{3.14}$$

3.7 Implementation of van Kampen's theorem

In order to implement the van Kampen theorem, we first define the data type 2.7.4 for a homomorphism of groupoids. Such a homomorphism, say H, consists of a source Source(H) and target Target(H). Also, it consists of two functions H!.mappingObj and H!.mappingArr, they map the objects and the arrows from the source into the corresponding objects and arrows in the target, respectively.

The package HAP includes a function for defining an inclusion mapping of two regular CW-complexes called a RegularCWMap. Based on Algorithm 3.7.1, the fundamental groupoid of this mapping is implemented.

Algorithm 3.7.1 Fundamental Groupoid of Regular CW-Map

Input: Regular CW-Map $f : X \to Y$, and a set of base-points $X_0 \subset X^0$.
Output: Groupoid morphism $\pi_1(f) : \pi_1(X, X_0) \to \pi_1(Y, Y_0)$ where $Y_0 = f(X_0)$.

1: **procedure**
2: Calculate the fundamental groupoids $G = \pi_1(X, X_0)$ and $H = \pi_1(Y, f(X_0))$, by which the maximal discrete vector fields are constructed on X and Y.
3: Let $\underline{g}$ and $\underline{h}$ be sets of generators of G and H, respectively.
4: The discrete vector field on any regular CW-complex determines a function $\gamma = \gamma(e^1)$ for any 1-cell e^1, such function returns a path joining two critical 0-cells.
5: For each critical 1-cell e_i^1 in X, there is a path $p_i = \gamma(e_i^1)$ joining two critical 0-cells in X_0. Each critical 1-cell e_i^1 corresponds to one generator $g_i \in \underline{g}$.
6: Find the images $q_i = f(p_i)$ for each critical 1-cell in X, which is also a path in Y joining two critical 0-cells.
7: The path q_i may pass through a sequence of critical 1-cells $[\tilde{e}_{i_1}^1, .., \tilde{e}_{i_j}^1]$ in Y. Then it determines a sequence of generators of H, say $[h_{i_1}^{\epsilon_1}, ..., h_{i_j}^{\epsilon_j}]$, where $\epsilon_r \in \{-1, +1\}$ depending on the orientation of $\tilde{e}^1$.
8: The product $a_i = h_{i_1}^{\epsilon_1} ... h_{i_j}^{\epsilon_j}$, where $\epsilon_r \in \{-1, +1\}$ is an arrow in H and it is the image of g_i. Let $\underline{a}$ denote to the set of such product for all q_i.
9: **return** Groupoid morphism given on generators by the function $\underline{g} \to \underline{a}$
10: **end procedure**

Let Y be a regular CW-space, and let A and B be subspaces of Y. Then we can construct the inclusion maps $i : A \hookrightarrow Y$ and $j : B \hookrightarrow Y$, with the help of the HAP function RegularCWMap. Let Y_0 be a set of base-points, $Y_0 \subset A \cap B$. The FpGd function FundamnetalGroupoidOfRegularCWMap can be used to produce a pair of groupoid homomorphisms $f : \pi_1(A, A \cap Y_0) \to \pi(Y, Y_0)$ and $g : \pi_1(B, B \cap Y_0) \to \pi(Y, Y_0)$.
We implement the pushout 3.13. The FpGd function PushoutOfFpGroupoid inputs a pair of groupoid homomorphisms and returns their pushout. So, PushoutOfFpGroupoid(f, g) will return the presentation of the fundamental groupoid $\pi_1(Y, Y_0)$.

Example 3.7.1. For testing our implementation of the van Kampen theorem, we let T be a 2-dimensional simplicial complex homeomorphic to the torus with 36 2-cells. We divide the torus T into two pieces A and B as shown in the figure 3.12 with $A \cap B \neq \emptyset$.

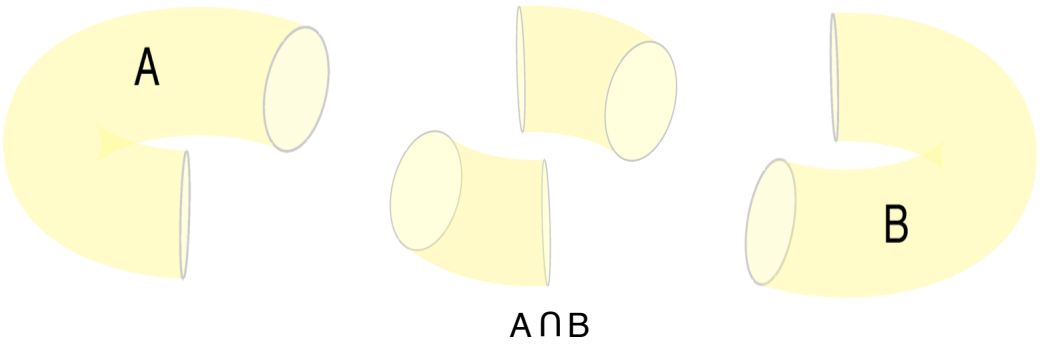


Figure 3.12 The torus T divided into two pieces A and B such that their intersection is nonempty.

In the first step of the following GAP session, we read the simplicial complex of the torus $T = A \cup B$. The simplicial complexes of the three pieces A, B and $A \cap B$ are stored in the file "data321.txt", see Appendix B.1. Figure 3.13 shows the simplicial complex T which is a union of A and B.

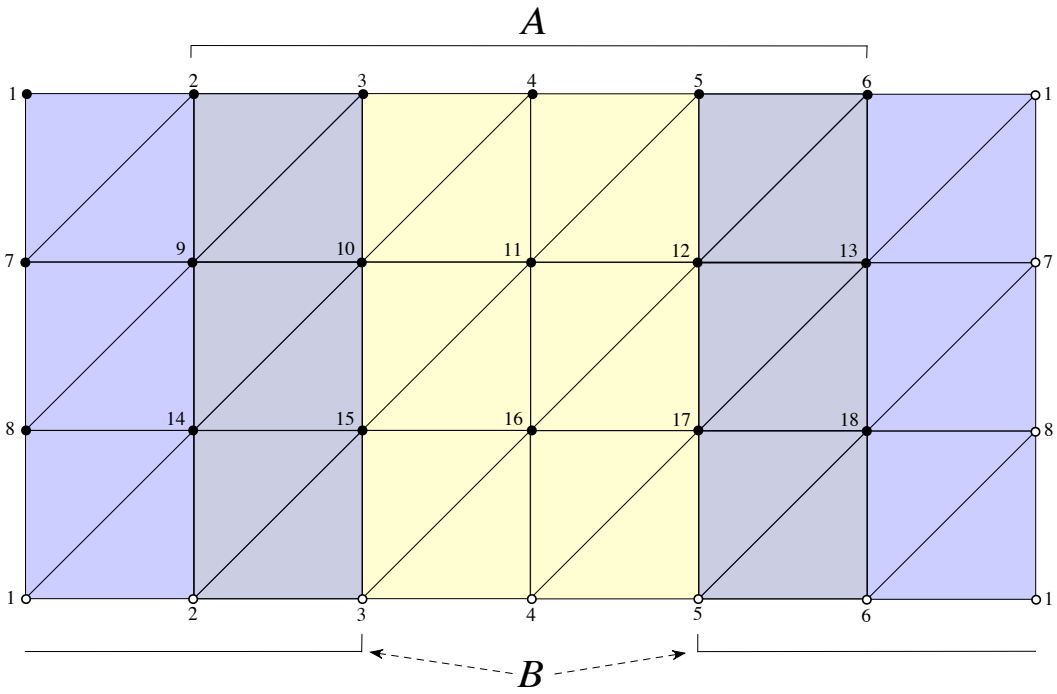


Figure 3.13 A 2-dimensional simplicial complex homeomorphic to the torus with 36 2-cells. Two sub-simplicial sets A and B are indicated by different colours yellow and blue, respectively. The intersection $A \cap B$ is non-empty.

In the second line, the mappings i and j are the inclusion maps from $A \cap B$ to A and from $A \cap B$ to B, respectively. The list $V = \{1, 3\}$ contains a single 0-cell in each component of $A \cap B$, the 0-cells 1 and 3 are corresponding to the vertices 7 and 10 in Figure 3.13.

GAP session 3.7.1

```
gap> Read("data321.gi");
gap> i:=RegularCWMap(A,AB);;
>    j:=RegularCWMap(B,AB);;
gap> pnt:=[1..Source(i)!.nrCells(0)];;
gap> p:=PiZero(Source(i))[2];;
gap> V:=SSortedList(List(pnt,p));
[ 1, 3 ]
```

Now, let f and g be two the induced groupoid homomorphisms

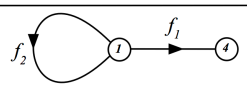
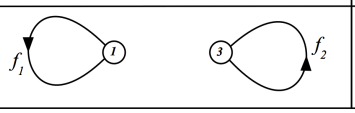
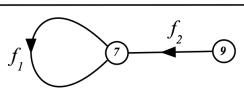
$$\begin{aligned} f &= \pi_1(A \cap B, V) \to \pi_1(A, f(V)) \\ g &= \pi_1(A \cap B, V) \to \pi_1(B, g(V)) \end{aligned}$$

The images of the generators of the groupoid $\text{Source}(f) = \pi_1(A \cap B, V)$ under f are words in the free groupoid of $\text{Target}(f) = \pi_1(A, f(V))$ and they are words in the free groupoid of $\text{Target}(g) = \pi_1(B, g(V))$ under g, as shown in the GAP session.

```
gap> f:=FundamentalGroupoidOfRegularCWMap(i,V);
Objects Mapping : [ 1, 3 ] -> [ 1, 4 ]
Arrows Mapping  : [ f1 , f2 ] -> [ f2 , f1*f2*f1^-1 ]

gap> g:=FundamentalGroupoidOfRegularCWMap(j,V);
Objects Mapping : [ 1, 3 ] -> [ 7, 9 ]
Arrows Mapping  : [ f1 , f2 ] -> [ f1 , f2^-1*f1*f2 ]
```

The following table shows the free generating graph of the groupoids $\pi_1(A, f(V)), \pi_1(B, g(V))$ and $\pi_1(AB, V)$

A	AB	B
		

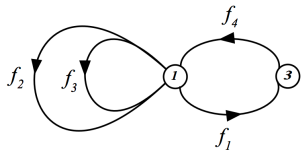
The source and target of each generator can be seen using the FpGd functions Source and Target. They are illustrated in the following GAP session and displayed in the above table as graphs.

```
gap> List(GeneratorsOfGroupoid(Source(f)),x->[Source(x),Target(x)]);
[ [ 1, 1 ], [ 3, 3 ] ]
gap> List(GeneratorsOfGroupoid(Target(f)),x->[Source(x),Target(x)]);
[ [ 1, 4 ], [ 1, 1 ] ]
gap> List(GeneratorsOfGroupoid(Target(g)),x->[Source(x),Target(x)]);
[ [ 7, 7 ], [ 9, 7 ] ]
```

Now, we are ready to calculate the pushout of the homomorphisms f and g. The function PushoutOfFpGroupoids(f,g) return the fundamental groupoid of the torus.

```
gap> C:=PushoutOfFpGroupoids(f,g);
<fp groupoid on the generators [ f1 , f2 , f3 , f4 ]>
gap> gensC:=GeneratorsOfGroupoid(C);;
gap> List(gensC,x->[Source(x),Target(x)]);
[ [ 1, 3 ], [ 1, 1 ], [ 1, 1 ], [ 3, 1 ] ]
gap> RelatorsOfFpGroupoid(C);
[ f2^-1*f3, f1*f2^-1*f1^-1*f4^-1*f3*f4 ]
```

The generating graph of the fundamental groupoid of the torus is



The final step is calculating the vertex group at some vertex. We choose the vertex 1 which is the source of the first generator f_1 of the groupoid C. The vertex group is generated by two generators f_2 and f_4. The set of relators in last line in the GAP session for the vertex group contains only one relators $f_2^{-1}f_4^{-1}f_2f_4$, which is exactly a presentation for the torus.

```
gap> V:=VertexGroup(C,Source(gensC[1]));
<fp group of size infinity on the generators [ f1, f2, f3, f4 ]>
gap> RelatorsOfFpGroup(V);
[ f2^-1*f3, f2^-1*f4^-1*f3*f4, f1 ]
gap> U:=SimplifiedFpGroup(V);
<fp group of size infinity on the generators [ f2, f4 ]>
gap> RelatorsOfFpGroup(U);
[ f2^-1*f4^-1*f2*f4 ]
```

3.8 Groupoid techniques for time series analysis

A time series is a sequence of records (data points) in successive order over a specified period of time. Identifying the patterns that form trends, cycles, and variances is important in order to understand such records which is usually involve large amounts of data. Time series analysis aims to understand patterns evolving over time and use these patterns to predict future behaviour, for instance monthly sales, heart arrhythmias, stock prices and so on.

Persistent homology can be a powerful topological approach for analysing large data sets. Topological tools, such as barcodes and Betti numbers of the spaces can be employed on cloud data to extract various features. Time-delay coordinate embedding has mostly been used in the analysis of time series. Time-delay embedding of a time series might recover the underlying dynamics of a system in most cases [33]. The delay coordinate embedding technique has many applications and it is useful for analysing the time series. The idea of using the time-delay embedding is approximating the data sets by simplicial complexes and by analysing their persistent homology. In most cases the size of data is big and then constructing such simplicial complexes is required large memory for computing. We think the groupoid technique is required for this purpose. In this section, we will show how the simplicial complexes can arise from time-series and then calculating the fundamental group(oid) of such complexes using the time-delay embedding method.

3.8.1 Time-delay embedding

The time-delay embedding is a method can be used for constructing a point cloud from time series and extracting their periodic behaviour. The mathematical foundation of the delay-coordinate embedding method which embeds a scalar time series into an d-dimensional space, see for instance [75, 88]. For a time series $\{x_i\}, i = 1, 2, ...,$ a representation of the delay coordinate embedding in $\mathbb{R}^d$ can be described as follows:

$$X_i = (x_i, x_{i+j}, x_{i+2j}, ..., x_{i+(d-1)j}),$$

where j is the index delay and d is the embedding dimension. If the sampling time is T_s, then the delay time τ is connected to the index delay j by the equality $\tau = j \cdot T_s$.
A good source of examples of time delay embeddings used in real-world data sets is available in [63].

Consider the following time series

$$y_t = A_t \sin(\varphi_t + t\ \omega_t) \tag{3.15}$$

where A_t is the amplitude, ω_t is the frequency, and φ_t the phase.

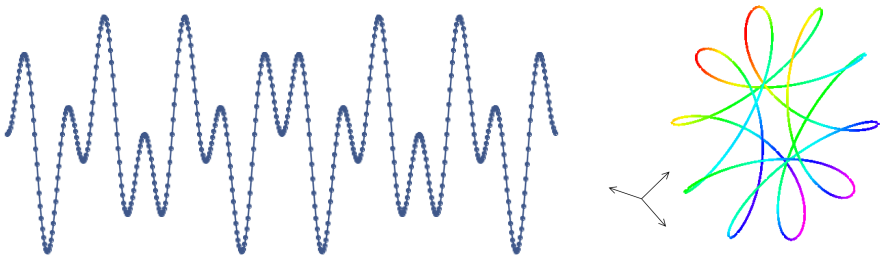


Figure 3.14 A time series plot of Eq. 3.15 (left), the corresponding time-delay embedding in 3-dimensional space (right).

A time-delay embedding of the time series $\{y_i\}$ is a lift to a time series ϕ_i defined as follows

$$\phi(t) = (y_t, y_{t+\tau}, ..., y_{t+(d-1)\tau})$$

Figure 3.14 (left) shows the corresponding delay embedding of the time series given by Eq. 3.15. And the corresponding time-coordinate embedding in 3-dimensional space on the right, it is homotopically equivalent to the circle because we designed the time-series using Equation 3.15 which is a sine function and its magnitude $A_t = -\exp(\cos(t))$.

Example 3.8.1. Consider the time-series given by the following function

$$f(t) = 3\cos(t) + 2\ \cos(\pi t), \tag{3.16}$$

using time-delay embedding in the 3-dimensional space ($d = 3$) and delay time $\tau = 3$ will produce a set of points

$$P = \{(f(t), f(t+3), f(t+6)) \mid t = 0, 1, 2, ...\} \tag{3.17}$$

The data are rotated about the line spanning by the dominant eigenvector $u = (0.7, 0.02, -0.7)$ of the covariance matrix of the data set P. The rotational matrix used here is

$$R = (\cos\theta(t))\, I + (\sin\theta(t))\, [\mathbf{u}]_\times + (1 - \cos\theta(t))\, (\mathbf{u} \otimes \mathbf{u}),$$

where $[u]_\times$ is the cross product matrix of u, $\otimes$ is the tensor product, and $\theta(t) = \cos^2(t)$.

The following GAP session shows how to produce a time-series $\{s_i\}$ using equation 3.16 and then form a set of points p using the time-delay embedding 3.17.

GAP session 3.8.1

```
gap> f:=function(t) return 3*Cos(t)+2*Cos(22/7*t); end;
gap> s:=List([0..800],x->f(x*22/35.0));;
gap> P:=List([1..264],i->[s[1+3*i],s[1+3*(i+1)],s[1+3*(i+2)]]);;
```

Now rotate the points P_i^T by an angle $\theta = \frac{\pi}{2}(1+\cos(t))$ about the vector

$$u = \begin{pmatrix} 0.7 \\ 0.02 \\ -0.7 \end{pmatrix} = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

using the following matrix

$$R = \begin{bmatrix} \cos\theta + a^2(1-\cos\theta) & ab(1-\cos\theta) - c\sin\theta & ac(1-\cos\theta) + b\sin\theta \\ ba(1-\cos\theta) + c\sin\theta & \cos\theta + b^2(1-\cos\theta) & bc(1-\cos\theta) - a\sin\theta \\ ca(1-\cos\theta) - b\sin\theta & cb(1-\cos\theta) + a\sin\theta & \cos\theta + c^2(1-\cos\theta) \end{bmatrix}.$$

to obtain a data set Q,

$$Q = \{R\, p^T \mid p \in P\}$$

```
gap> dx:=EuclideanApproximatedMetric;;
gap> M:=VectorsToSymmetricMatrix(Q,dx);;
gap> G:=SymmetricMatrixToGraph(M,19/10);;
gap> K:=CliqueComplex(G,2);
Simplicial complex of dimension 2.
gap> Y:=RegularCWComplex(K);
Regular CW-complex of dimension 2
gap> H:=FundamentalGroupoidOfRegularCWComplex(Y,[1,99]);
<fp groupoid on the generators [ f1, f2, f3 ]>
gap> V:=VertexGroup(H,1);
<fp group of size infinity on the generators [ f2, f3 ]>
gap> RelatorsOfFpGroup(V);
[ f2^-1*f3^-1*f2*f3 ]
```

The presentation of the vertex group suggests that the data set is sampled from a torus.

Chapter 4

Simplicial Complexes and Mapper

4.1 Introduction

Topological data analysis (TDA) is a growing area of research that attempts to use topological ideas to gain insight into large, and often high dimensional, data sets. Many mathematicians and computer scientists are contributing to TDA, such as Carlsson [20], Edelsbrunner and Harer [27], Ghrist [39], King, Knudson and Mramor [65] Oudot [77] and Zomorodian [94, 96].
In this chapter we discuss one technique from TDA, namely Mapper clustering [20]. This technique is a method for representing large data sets as simplicial complexes. In the main successful applications to date [93, 5, 86] data is represented by 1-dimensional simplicial complexes, i.e. graphs. However, in this chapter we are more interested in attempting to represent data by simplicial complexes of dimensions > 1, and then using the techniques of previous chapters to investigate the low-dimensional homotopy (fundamental group) of these simplicial complexes.

4.2 Simplicial complexes

Let X be a set consisting of k points $x_1, x_2, ..., x_k \in \mathbb{R}^n$. An *affine combination* is any summation of these points $\sum_{i=1}^{k} \mu_i x_i$ such that $\sum_{i=1}^{k} \mu_i = 1$. The affine combination $\sum_{i=1}^{k} \mu_i x_i$ is called a *convex combination* of x_i if the coefficients μ_i are all non-negative.
The *convex hull* is the set of all possible convex combinations of x_i, denoted by

$$Conv(X) = \left\{ \sum_{i=1}^{k} \mu_i x_i \in \mathbb{R}^n \mid \mu_i \geqslant 0, \sum_{i=1}^{k} \mu_i = 1 \right\}$$

Definition 4.2.1. [29] The *standard n-simplex*

$$\Delta^n = Conv\{e_1, ..., e_{n+1} \in \mathbb{R}^{n+1}\}$$

is the convex hull of the $n+1$ standard basis vectors of $\mathbb{R}^{n+1}$.

Definition 4.2.2. [29] A subspace X of a CW-space Y is said to be a CW-*subspace* if X is a CW-space whose cells are cells of Y. A CW-subspace of Δ^n is called a *finite simplicial space*.

A formal definition a the simplicial complex is already delivered in Section 1.3, see Definition 1.3.7.
We can construct a finite simplicial space $|K|$ from any finite simplicial complex K by first choosing a bijection $\phi : V \xrightarrow{\cong} \{e_1, ..., e_{|V|}\}$ between the vertex set of K and the standard basis vectors of $\mathbb{R}^{|V|}$, and then taking $|K|$ to be the union of the convex hulls $Conv(\phi(\sigma))$ for each $\sigma \in K$. The simplicial space $|K|$ is said to be a *geometric realization* of K.
The boundaries (faces) of an n-simplex are defined as the $n-1$ subsets of cardinality n. For example the 3 boundaries of the 2-simplex $[1, 2, 3]$ are $[1, 2], [2, 3], [1, 3]$. Each boundary of an n-simplex is an $(n-1)$-simplex. The union of the n-dimensional simplices of every simplex in a simplicial complex K gives the n-skeleton of K.

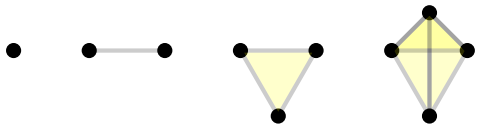


Figure 4.1 n-simplices for $n = 0, 1, 2, 3$.

A collection of standard simplices glued to each other produce a simplicial space. . The collection can easily grow large. The dimension of a simplicial complex K is the maximum dimension of its simplices. The underlying space, say Y, is the union of its standard simplices together with the topology inherited from $\mathbb{R}^n$. A *polyhedron* is the underlying space of a simplicial complex. A *triangulation* of a topological space X is a simplicial complex K together with a homeomorphism between X and $|K|$. Figure 4.2 shows a simplicial complex with 8 vertices, 12 triangles which represents a 2-sphere.

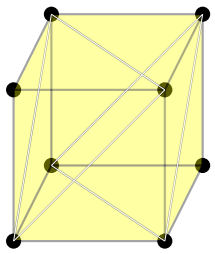


Figure 4.2 A simplicial complex of dimension 2 representing a triangulation of the 2-sphere.

Building such simplicial complexes is not so convenient for manual (hand) calculations but close to ideal for computer implementations. The package HAP provides an easy way to build simplicial complexes. This package uses the following data type for finite simplicial complexes.

Data Type 4.2.1. [28] *A finite simplicial complex is represented as a component object* K *with the following components:*

- K!.vertices is a list whose elements are distinct and represent the vertices of a simplicial complex. It must be possible to test $u < v$ for any two elements u, v in the list where $<$ is some fixed total order.
- K!.simplices(k,i) is a function which returns the list of vertices in the ith k-simplex. The order of the vertices in the returned list is important and is chosen to be in increasing order.
- K!.nrSimplices(k) is a function which returns the number of k-dimensional simplices.
- K!.enumeratedSimplex(x) is a function which inputs a list x of vertices, in increasing order, representing a k-simplex and returns the position of this simplex in the list of all k-simplices.
- K!.properties is a list of pairs such as ["dimension", 2].

4.3 Mapper

Given a data set (or cloud of data) sampled from some high dimensional space, topological data analysis focuses on recovering topological information of an unknown space from which the dataset is sampled. This information is obtained mainly using algorithms for computing some algebraic homotopy invariants.
Mapper requires a finite data set, say S sampled from an unknown metric space X and a function f from X onto some parameter space Z. For instance $Z = \mathbb{R}, \mathbb{R}^2$, or an interval $[a, b]$ in the real line or the unit circle S^1 in the plane or any subspace of the plane. The function f is called a *filter function* for the data set S. This function in most cases reflects geometric properties of S. The following are examples of filter functions that reflect interesting geometric properties of the cloud data S:

- **Density**: Any density estimator on the cloud data S produce a non-negative function such that

$$f_\epsilon(s) = C_\epsilon \sum_{s' \in S} \exp\left(-\frac{d(s,s')^2}{\epsilon}\right),$$

where C_ϵ is constant, d is Euclidean metric and $\epsilon > 0$. The function $f_{\epsilon(s)}$ gives useful information about the cloud data S.

- **Data depth**: This notion refers to any attempt to quantify the notion of nearness to the centre of the cloud data S. The eccentricity function is commonly used as a data depth function, which gives a filter function of the following form

$$E_p(s) = \frac{1}{|S|}\left(\sum_{s' \in S} d(s, s')^p\right)^{1/p},$$

where $|S|$ is the size of S. This filter function is a useful choice since it does not require any specific knowledge about the data.

- **Principal-component analysis** PCA: The idea of PCA is to treat a data set $\{v_1, v_2, ..., v_n\} \in \mathbb{R}^d$ with zero mean $\frac{1}{n}(v_1 + v_2 + ... + v_n)$ as a matrix $S \in \mathbb{R}^{d \times n}$, and find the eigenvectors for the *covariance matrix* $M = S^T S$. We choose the k largest eigenvalues of M with eigenvectors $e_1, e_2, ..., e_k \in \mathbb{R}^d$ and set $A \in \mathbb{R}^{k \times d}$ to be the matrix whose ith row is e_i for $1 \leqslant i \leqslant k$. Then we have a filter function

$$\begin{aligned} f \quad : \quad & \mathbb{R}^d \to \mathbb{R}^k \\ & v \mapsto Av \end{aligned}$$

If no prior knowledge of the space X is available then one can always choose a point $x_0 \in S$ that minimizes $b = \mathrm{Max}_{x \in S} d_X(x_0, x)$. More precisely, the function $f|_S : S \to Z$ is assumed to be the filter function $f(s) = d(x_0, s)$. We refer to such an f as an *intrinsic filter function* and note that, since x_0 is not necessarily unique, the data set S may give rise to more than one intrinsic filter function. The main purpose of this function is dividing the data into subsets, not necessary to be disjoint.

Mapper requires a user defined finite open cover $U = \{U_p\}_{p \in P}$ of Z. It also inputs a user defined procedure *cluster* for clustering any finite subsets $\tilde{U} \subset X$ into a number of distinct clusters V_i which yield a partition $\tilde{U} = V_1 \sqcup V_2 \sqcup ... \sqcup V_n$.
Any clustering algorithm, such as single-linkage, complete-linkage, hierarchical and k-means clustering, can be use to find the path connected components. These algorithms can be found in the textbooks of data classification, we recommend the reader to find them in [61, 2].
The Mapper procedure outputs a finite simplicial complex K which is intended to serve as a model for X. The distance $d_X(x, y)$ must be known for all $x, y \in S$ but no further details of the metric space X are required in the construction of K. For simplicity, we assume that the user has specified a finite subset $P \subset Z$ and a sufficiently large constant $r > 0$, and that the open cover U is defined by setting

$U_p = \{z \in Z : d_Z(z,p) < r\}$. As part of the Mapper procedure each finite subset $\tilde{U}_p = \{x \in S : f(x) \in U_p\} \subseteq S$ is determined and partitioned $\tilde{U}_p = V_{p,1} \sqcup V_{p,2} \sqcup ... \sqcup V_{p,n_p}$ using the *procedure cluster*.
The simplicial complex K is then defined to be the nerve of the cover $\mathcal{V}$ of S.

$$K = Nerve(\{V_{p,i}\}_{p \in P, 1 \leqslant i \leqslant n_p}).$$

We say this Mapper procedure in the ideal situation is **Mapper of S with filter f, cover $\mathcal{V}$ and clustering function** *cluster*. The output can be viewed as an approximate model of the homotopy type of the space X from which S is sampled. Theorem 3.2.1 provides an heuristic justification for viewing the output as such a model.

Note that K has one k-simplex for each subset $\sigma \subset V$ of size $k+1$ with $\bigcap_{V \in \sigma} V \neq \varnothing$. The above description of the Mapper clustering procedure can be encoded as a GAP function

$$Mapper(S, P, f, d_X, \epsilon, d_Z, r, cluster)$$

which returns the simplicial complex K. Here d_X and d_Z denote the metrics on X and Z.
In many applications one chooses P, r and f so that the resulting simplicial complex K is 1-dimensional. Graph visualization software such as [36] can then be used to investigate K. Each vertex V of K corresponds to a finite subset of S and the size of this subset can be represented by varying the size of nodes in the visualization of the graph. Information about f could be incorporated into the visualization using colours, see Figure 4.3.

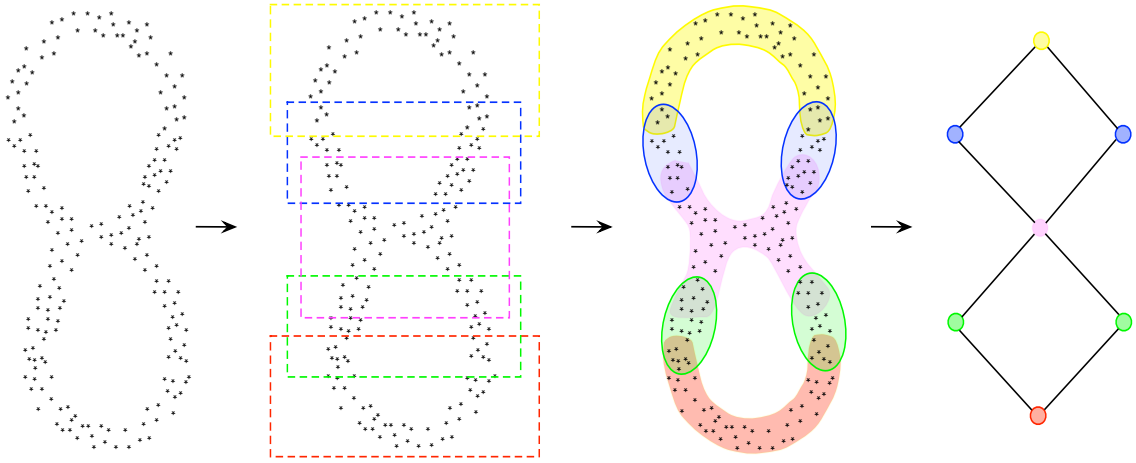


Figure 4.3 Mapper for random data around eight-figure produces a 1-dimensional simplicial complex capturing the shape of the data.

Our implementation omits this option of colouring the vertices. For instance see Figure 4.4, the data is a set of 1000 points sampled randomly from a picture of

Figure 4.4 Image $X \subset \mathbb{R}^2$ of spider (left), and the corresponding Mapper output (right).

spider. The Mapper returns a 1-dimensional simplicial complex (graph) of 79 nodes. The graph is a tree with nine long branches corresponding to the eight legs and the abdomen of the spider.
One method of clustering a set $\tilde{U}$ involves choosing a fixed parameter $\epsilon > 0$, forming the graph $G(\tilde{U}, \epsilon)$ with vertex set equal to $\tilde{U}$ and with vertices $x, x' \in \tilde{U}$ connected by an edge if $d_X(x, x') \leqslant \epsilon$, and then taking all vertices in a single connected component of $G(\tilde{U}, \epsilon)$ to constitute a cluster. We refer to this naive method as *graph clustering*. It can be encoded as a function

$$cluster(\tilde{U}, \epsilon, d_X).$$

There are various heuristic approaches for determining a suitable value for ϵ in terms of U and d_X, and so the parameter ϵ can be omitted from the input if desired. One heuristic is described in Chapter 5.

4.4 Implementations of Mapper

Consider a set $S = \{s_1, s_2, ..., s_n\}$ of points in $\mathbb{R}^d$. Mapper requires the data S to be entered in a matrix form, so the data S is a $n \times d$ matrix. And $f : S \to Z$ is a function that maps S into a parameter space Z. PCA is used widely as a filter function, so let us explain the method and how it is implemented. First treat the set S as a matrix of the following form

$$X = \begin{bmatrix} | & | & & | \\ s_1\text{-}\mu & s_2\text{-}\mu & \dots & s_n\text{-}\mu \\ | & | & & | \end{bmatrix}.$$

The covariance matrix is

$$M = \frac{1}{n} X^T X,$$

where μ is the mean of S (centre of gravity). The following GAP function `VectorsToCovarianceMatrix` inputs the data set S and returns the covariance matrix M.

```
gap> VectorsToCovarianceMatrix:=function( S )
>       local n, a, A, V;
>       n:=Length(TransposedMat(S)[1]);
>       a:=S-List([1..n],i->1)*S/n;
>       A:=TransposedMat(a)*a;
>       V:=A/n;
>       return V;
>     end;
```

The next step in PCA is calculating the eigenvalues and their corresponding eigenvectors of the covariance matrix M. The power method is used to find the first dominant eigenvalue and the corresponding eigenvector and then we use this method with deflation to find other eigenvalues and eigenvectors. We implement Algorithm 4.4.1 in GAP to find the set of eigenvalues and the corresponding eigenvectors of a symmetric matrix. The function `FloatSpectrum` inputs a symmetric matrix M and a natural number n and returns a list of eigenvalues and list of the corresponding eigenvectors of M. The eigenvectors are ordered according to the absolute value of their eigenvalues, from the largest to smallest. We have also implement the Jacobi algorithm [84] in GAP , but it is not used in this example.

Algorithm 4.4.1 Float Spectrum

Input: a symmetric matrix M of dimension $n \times n$ and real number ϵ.
Output: List $L = [e, V]$, where e is a list of eigenvalues and V is a list of eigenvectors.

1: **procedure**
2: Choose a random vector $v_0 \in \mathbb{R}^n$
3: $v_1 := M\ v_0$;
4: i:=2;
5: **while** $||v_{i-1} - v_{i-2}|| > \epsilon$ **do**
6: i:=i+1;
7: $w_i = Mv_{i-1}$
8: $v_i = \frac{w_i}{||w_i||}$
9: **end do**
10: $x = v_i$;
11: $e = [\ ]$, and $V = [\ ]$;
12: **for** $j = 1..n$ **do**

13: $\lambda = (x^T Mx)/||x||$;
14: add(e, λ);
15: add(V, x);
16: $M = M - \lambda v^T v$
17: **repeat** steps 3-9 to get new λ and x;
18: **end do**
19: **return** $[e, V]$;
20: **end procedure**

The k *principal components* are the eigenvectors corresponding to the k largest eigenvalues. Now, construct a new matrix A, whose rows are the k principal components. The filter function f is

$$\begin{aligned} f \quad : \quad & S \to Z = \mathbb{R}^k \\ & s \mapsto A(s - \mu) \end{aligned}$$

Let $P = \{p_1, p_2, ..., p_k\} \subset Z$ be a set of points with open cover $\mathcal{U} = \{U_i \mid U_i \text{ is open ball whose centre } p_i \text{ with radius } r\}$ of the space Z. So the function f, the cover $\mathcal{U}$ and some metric function d_Z will subdivide the data set S into parts by the sense $x \in \tilde{U}_j = f^{-1}(U_j)$ if $d_Z(f(x), p_j) \leqslant r$.
A clustering function $\mathcal{C}$ is required in the Mapper procedure to break each set $\tilde{U}_j$ into parts called clusters. The clustering function inputs a set of point $\tilde{U}_j$ and a real number ϵ to returns subsets of $\tilde{U}_j$ that never share common points.
The FpGd function `cluster` inputs a data set $\tilde{U}$, a metric function d and some real number (threshold value) ϵ and returns a family of subsets of S called clusters. The implementation is based on the following Algorithm 4.4.2.

Algorithm 4.4.2 cluster

Input: A data set $\tilde{U}$, a metric function d and a real number ϵ.
Output: A family of subsets of $\tilde{U}$.

1: **procedure**
2: Consider each $x \in \tilde{U}$ as a vertex for a graph, say G.
3: Two vertices x and y in the vertex set of G are connected by an edge when $d(x, y) \leqslant \epsilon$.
4: **if** G is connected graph
5: **then** return $\tilde{U}$
6: **else** return $[\tilde{U}_1, \tilde{U}_2, ..., \tilde{U}_n]$, where $\tilde{U}_i, 1 \leqslant i \leqslant n$ is a set of vertices of a connected component subgraph in G and $\tilde{U} = \bigcup_{1 \leqslant i \leqslant n} \tilde{U}_i$.
7: **end if**
8: **end procedure**

The following Algorithm 4.4.3 is reimplemented in GAP . The function `GeneralMapper` inputs data set S, a metric function d_S, a filter function f, a metric

function d_Z, a set of uniform points P, a real number r and a clustering function $\mathcal{C}$ which is also required to define a another real number ϵ.

Algorithm 4.4.3 Mapper algorithm, producing a simplicial complex from dataset.

Input:

- a dataset S,
- a filter function $f : X \to Z$, where X is a metric space (X, d_S) and Z is a parametric space defined with some metric function d_Z,
- a set of uniform points $P = \{p_1, ..., p_n\} \subset Z$, and a real value r,
- a clustering function $\mathcal{C} = \mathcal{C}(U, d_S, \epsilon)$.

Output: A simplicial complex K.

1: **procedure**
2: create an open cover $\mathcal{U} = \{U_i\}_{1 \leqslant i \leqslant n}$ for Z such that p_i is the centre of the open ball U_i.
3: map the set S into Z using the filter function f.
4: find the family of preimages $\mathcal{W} = \{W_i = f^{-1}(U_i)\}_{1 \leqslant i \leqslant n}$;
5: use $\mathcal{C}$ to define the family of subsets $\mathcal{V} = \{V_{i,\alpha} = \mathcal{C}(W_i, d_S, \epsilon_S)\}_{1 \leqslant i \leqslant n}$.
6: **return** the nerve of cover, $Nerve(\mathcal{V})$.
7: **end procedure**

In case of using a filter function that maps the data set to 1-dimensional space, the output of Mapper is a graph. To display the graph that corresponds to the output of Mapper, we use the HAP command `Display`. This command allows the user to visualise the Mapper of some data set as a graph containing vertices joined by edges. Mapper employs the program GraphViz [32] to create a geometric realization of the graph encoded in the adjacency matrix.
The number of vertices is the same number of clusters and the size of each vertex reflects the ratio of the size of the cluster to the size of the data set. Two vertices are joint by an edge when the corresponding clusters share any common data points.

In the following example, we will show that Mapper can produce different simplicial complexes or the same data set when the user use different choices for the filter functions. Using an appropriate filter function will lead to a simplicial complex capturing the shape of the data.

Example 4.4.1. First, let us generate 1000 3-dimensional points on a torus distributed randomly, the outer radius $a = 18$ and the inner radius $b = 6$, see the following GAP session . The GAP package "float" should be loaded first.

GAP session 4.4.1

```
gap> n:=1000;;   a:=18;;    b:=6;;
gap> rand:=function(i)
> return 2*3.14*Random([1..1000])/1000;
> end;;
gap> points:=[];;
gap> for i in [1..n] do
>  s:=rand(i);  t:=rand(i);
>  x:= a+b+(a+b*Cos(s))*Cos(t);
>  y:= a+b+(a+b*Cos(s))*Sin(t);
>  z:= b*(1+Sin(s));
>  points[i]:=[x,y,z];
>  od;
```

Now, create a set of uniform points P and give values for r the radius of the open balls that cover the parametric space and ϵ the threshold values for the clustering function. And define the filter function that project the points to z-axis.

```
gap> P:=[0..100];      P:=List(P, i->[i]);;
gap> r:=3/4;;
gap> epsilon:=12;;
gap> d:=EuclideanApproximatedMetric;;
gap> f1:=function(x) return x[3]; end;;
gap> M1:=GeneralMapper(points,P,f1,d,epsilon,d,r,cluster);;
gap> Display(GraphOfSimplicialComplex(M1));
```

The Mapper output is a simplicial complex shown in figure 4.5 (up) which doesn't match the torus. Now keep all values, only change the filter function to be the projection of the data to the x-axis. The Mapper output is shown in the same figure (down), in this case we obtained a circle which is the desired simplicial complex.

```
gap>  f2:=function(x) return x[1]; end;;
gap> M2:=GeneralMapper(points,P,f2,d,epsilon,d,r,cluster);;
gap> Display(GraphOfSimplicialComplex(M2));
```

4.5 Some illustrations of Mapper

To illustrate Mapper clustering we choose a set S of 1000 random points selected from a uniform distribution on the square $X = [0, 1000] \times [0, 1000]$. Four quotients of X, namely the cylinder, the Möbius strip, the torus and the Klein bottle are shown in Figures 4.6, 4.7, 4.8, 4.9 and are obtained from the square X by identifying

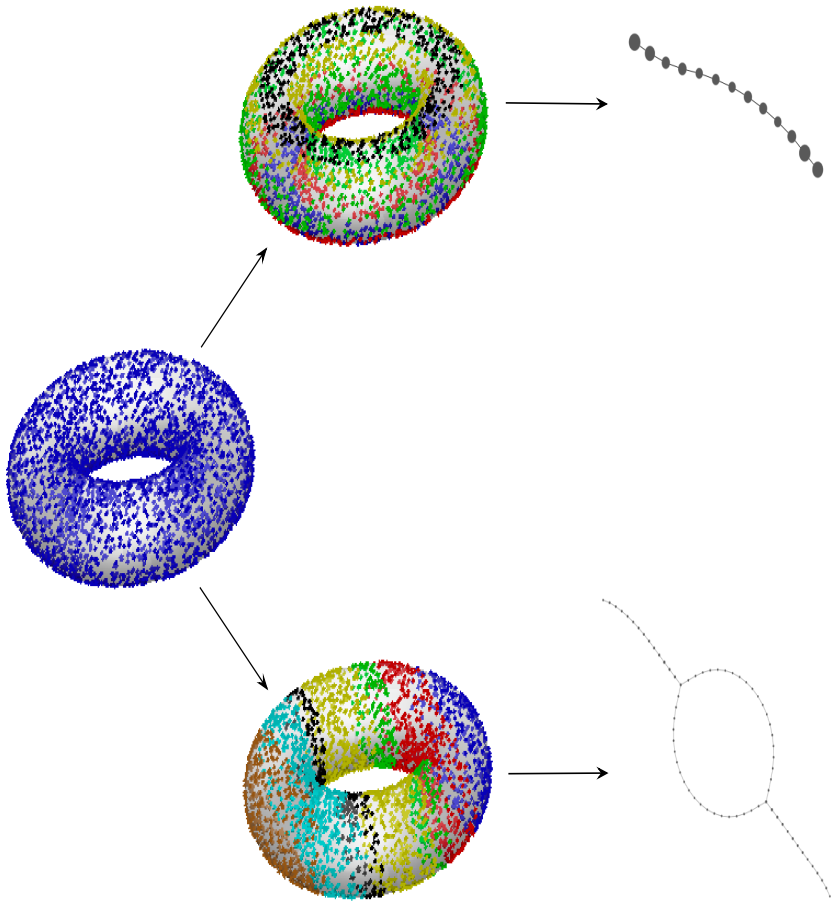


Figure 4.5 Using different filtere functions produce different Mapper for torus, the filter functions are shown by the colouring of the points for both cases (top: $f(x, y, z) = z$, bottom: $f(x, y, z) = x$).

opposite sides in the usual manner. The standard Euclicean metric on X induces distinct metrics on each of the four quotients. In the following example, we deal with this in detail.

Example 4.5.1. Let `points` be a set of 1000 points in the square $X = [0, 1000] \times [0, 1000]$, see the following GAP session.

GAP session 4.5.1

```
 gap> n:=1000;    a:=1;     b:=1;
gap> x:=List([1..n],i->Random([0..n*a]));;
gap> y:=List([1..n],i->Random([0..n*b]));;
gap> points:=List([1..n],i->[x[i],y[i]]);;
```

Now, let us define a metric function, called `cylinder_metric` to identify the point $(0, y)$ with $(1, y)$ in X, for all $y \in [0, 1]$.

```
gap> cylinder_metric:=function(x,y)
>      local L,d;
>      L:=[ [x,y],[x,[a*n+y[1],y[2]]],[y,[a*n+x[1],x[2]]] ];
>      d:=List(L,z-> EuclideanApproximatedMetric(z[1],z[2]));
>      return Minimum(d);
>      end;
```

The graph of Figure 4.6 (left) was obtained by applying our Mapper implementation to S with d_X equal to the induced metric on the cylinder, with an intrinsic filter function $f : X \to Z = [0, 5000]$ and with a set P of 50 points chosen (automatically) so that the $\tilde{U}_p$ have roughly equal size. The cycle in the graph of Figure 4.6 (right) represents an homology 1-cycle on the torus.

```
gap> P:=50*[0..100];;      P:=List(P, i->[i]);;
gap> r:=45;;
gap> epsilon:=150;;
gap> f:=function(x) return AbsoluteValue(x[1]-500); end;;
gap> dx:=cylinder_metric;;
gap> dz:=EuclideanApproximatedMetric;;
gap> M:=GeneralMapper(points,P,f,d,epsilon,d,r,cluster);;
gap> Display(GraphOfSimplicialComplex(M));
```

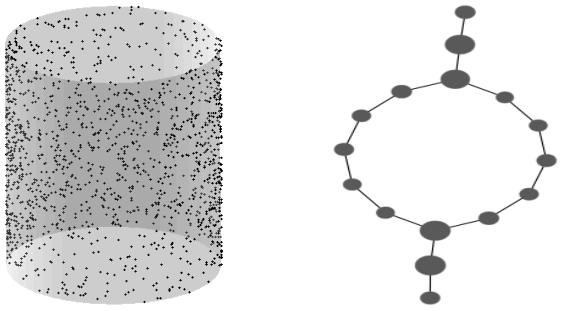


Figure 4.6 Data set of 1000 points around a cylinder (left), the Mapper returns 1-dimensional simplicial complex (graph of 16 vertices) (right).

Graphs almost identical to that of Figure 4.6 (right) are obtained when the Möbius metric d_X is replaced by the metric on the cylinder. The Möbius metric is defined to identify the points $(0, y)$ with $(1, 1 - y)$ in X, for all $y \in [0, 1]$.

```
gap> mobius_metric:=function(x,y)
>      local L,d;
>      L:=[ [x,y],[x,[a*n+y[1],b*n-y[2]]],[y,[a*n+x[1],b*n-x[2]]] ];
>      d:=List(L,z->EuclideanApproximatedMetric(z[1],z[2]));
>      return Minimum(d);
>      end;
gap> dx:=mobius_metric;;
gap> M:=GeneralMapper(points,P,f,d,epsilon,d,r,cluster);
gap> Display(GraphOfSimplicialComplex(M));
```

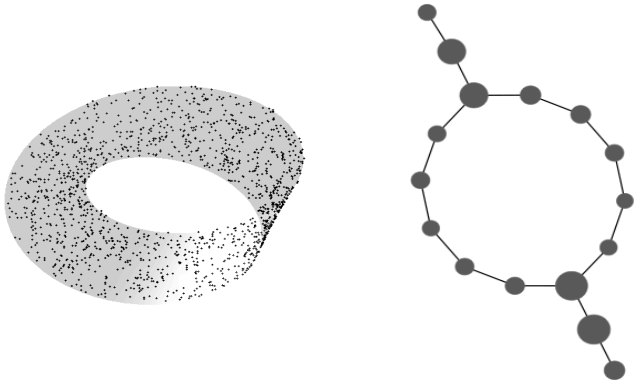


Figure 4.7 Data set of 1000 points around a Möbius strip (left), the Mapper returns 1-dimensional simplicial complex (graph of 16 vertices) (right).

Now, let us define the torus metric, by which the point $(0, y)$ are identified with $(1, y)$ and the point $(x, 0)$ identified with $(x, 1)$ for all $x, y \in [0, 1]$.

```
gap> torus_metric:=function(x,y)
> local X,Y,XY,d;
> X:=[ [a*n+x[1],x[2]],[x[1],b*n+x[2]] ];
> Y:=[ [a*n+y[1],y[2]],[y[1],b*n+y[2]] ];
> XY:=[ [x,y],[x,Y[1]],[x,Y[2]],[y,X[1]],[y,X[2]] ];
> d:=List(XY,z->EuclideanApproximatedMetric(z[1],z[2]));
> return Minimum(d);
> end;
gap> dx:=torus_metric;;
gap> M:=GeneralMapper(points,P,f,d,epsilon,d,r,cluster);;
gap> Display(GraphOfSimplicialComplex(M));
```

Finally, let us define a Klein metric to identify the point $(0, y)$ with $(1, 1 - y)$ and $(x, 0)$ with $(x, 1)$ for all $x, y \in [0, 1]$.

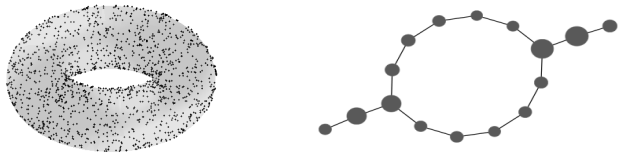


Figure 4.8 Data set of 1000 points around a torus (left), the Mapper returns 1-dimensional simplicial complex (graph of 16 vertices) (right).

```
gap> klein_metric:=function(x,y)
> local X,Y,XY,d;
> X:=[ [a*n+x[1],x[2]],[a*n-x[1],b*n+x[2]] ];
> Y:=[ [a*n+y[1],y[2]],[a*n-y[1],b*n+y[2]] ];
> XY:=[ [x,y],[x,Y[1]],[x,Y[2]],[y,X[1]],[y,X[2]] ];
> d:=List(XY,z->EuclideanApproximatedMetric(z[1],z[2]));
> return Minimum(d);
> end;

gap> epsilon:=75;;
gap> dx:=klein_metric;;
gap> M:=GeneralMapper(points,P,f1,d,epsilon,d,r,cluster);
gap> Display(GraphOfSimplicialComplex(M));
```

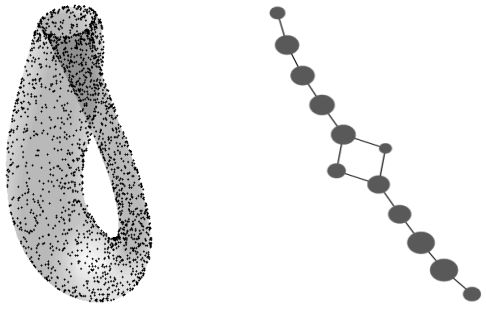


Figure 4.9 Data set of 1000 points around a Klein bottle (left), the Mapper returns 1-dimensional simplicial complex (graph of 12 vertices) (right).

Mapper remaining unchanged when the input any one of these data. Thus, while this application of Mapper manages to represent certain geometric features of the data, it fails to distinguish between data sampled from uniform distributions on four geometrically and topologically distinct spaces.

4.6 Gait analysis

We now illustrate our (unenhanced) implementation of Mapper on data from gait analysis. The example is based on [90]. The periodic motions are plentiful in human and animal behaviour. The human gait (or robot locomotion) is an example of periodic motion. Human motion, specifically walking, has been studied widely in recent decades. Motion capture databases that are available in [66] provide a rich source of motion patterns which can be used to animate characters in computer games and movies. This kind of data has been used to create a controller for a simulated human character and allowing it to walk based on the sequences of recorded motion captures [74].
There is a lot of mathematical work on gait analysis in the literature. Some of this work is about investigating the periodicity and detecting the kinds of motion. For instance, Mikael Vejdemo-Johansson and et al applied a persistent cohomology based method in a graphic application to recover circular coordinates of motions [90]. They proposed a framework for dealing with a motion and detecting the motion pattern by using persistent cohomology. The work is based initially on [67, 89] in which they applied a topological signature on a persistence diagram to the problem using gait data.

In this section, we use Mapper for real data of motion such as walking and gait or any motion. The periodicity can be detected without marking of the period start and end points. The method only required two parameters and two functions, one serve as a PCA and the other one determines the clusters according to the closeness of the points.
Motion databases such as [66] provide a rich source of motion patterns. Each motion can be consider as a trajectory in the object's configuration space. They model the configuration space by a *root joint* which is located at the hip in the skeleton, the translations along x, y, and z and *Euler angle* rotations around x, y and z, together with 56 additional joint angles of the various attached joints.
The data used in this section was obtained from mocap.cs.cmu.edu [66]. The data is stored in different format such as asf, amc and vsk files. We pick the data indexed by 01 in group 69, it is for forward walking. It contains 469 records. In each second they read 62 records. The motion 69:01 is used here as an experimental science data, three clips of human motion appear in Figure 4.11 (left). We use this data as an example to test our implementation of Mapper procedure. To read this kind of file, we wrote a GAP function called `ReadAMCfileAsPatternMatrex` to convert the text file into a data file. The function returns a matrix of dimension $n \times 63$, where n is the number of records. For the motion 69:01 under consideration, we have 469 records.
In the first step of the following GAP session, we read the file 69_01.amc and we call

the data set by S. Then we remove the first column that represents the time. Now S is a matrix of dimension 469×62. To apply PCA, we first calculate the covariance matrix M of the data S using the function `VectorsToCovarianceMatrix`. Then find the eigenvalues of M using the function `FloatSpectrum` which return a list of eigenvalues ordered from the large to the small.

GAP session 4.6

```
gap> S:=ReadAMCfileAsPatternMatrex("69_01.amc");;
gap> S:=S{[2..Length(S)]};;
gap> M:=VectorsToCovarianceMatrix(S);;
gap> L:=FloatSpectrum(M);
```

The first six eigenvalues are

$$616.060, 283.985, 217.461, 155.580, 76.547, 39.603, ...$$

other eigenvalues become small and are shown in the figure 4.10.

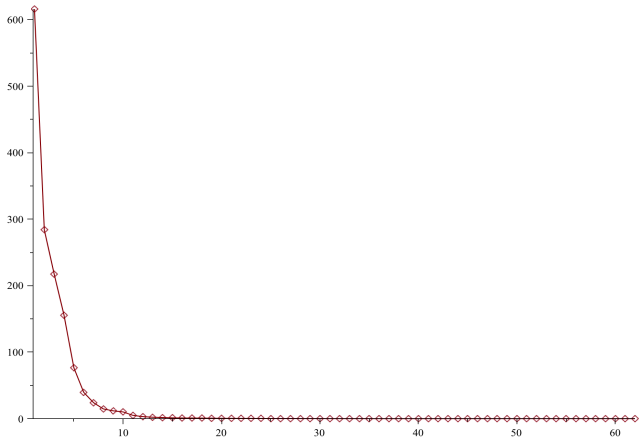


Figure 4.10 The eigenvalues of the covariance matrix for the data of the walk number 69:01.

Since the first eigenvalue is much greater than the others, that means its corresponding eigenvector is dominant and the direction along which the data set has the maximum variance. Now we will project our data to the line spanning by this vector.

First, create a set of uniform points P lie on the line spanned by v, and choosing suitable r and ϵ as shown in the GAP session and applying Mapper will return a simplicial complex with 41 clusters as shown in figure 4.11.

```
gap> v:=L[2][1];;
gap> dx:=EuclideanApproximatedMetric;;
gap> v:=v/dx(v,List(v,i->0));
gap> f:=function(x) return (x * v / v ^ 2) * v; end;;
gap> P:=List(3*[-20..20],x->x*v);; P:=List(P,i->[i]);;
gap> r:=29/10;
29/10
gap> epsilon:=50;;
gap> K:=GeneralMapper(SS,P,f,dx,epsilon,dx,r,cluster);;
Simplicial complex of dimension 1.

gap> GraphOfSimplicialComplex(K);
Graph on 41 vertices.
gap> Display(G);
```

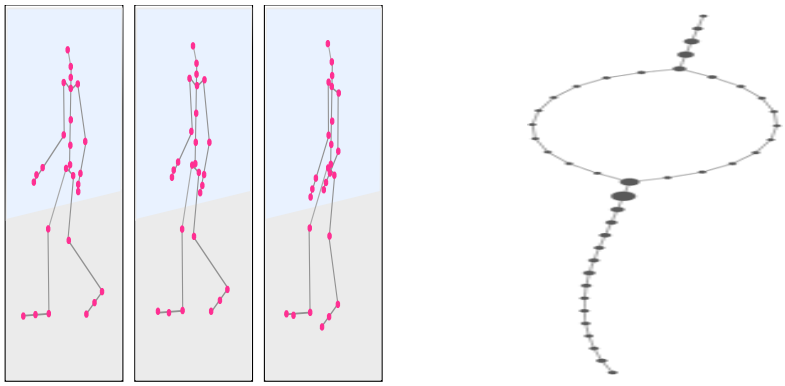


Figure 4.11 Clips of human motion presented as skeleton with root joint placed at the hip (left). Mapper of the motion 96:01 (walk forward) (right).

The periodicity in this motion has been shown in the [74], the output of the `Mapper` in Figure 4.11 (left) is also show that this motion is periodic.

Chapter 5

Distributed Computation of Cup Products

5.1 Introduction

The additive structure of the low-dimensional homology and cohomology of a space is the mainstay of applied computational topology due to its ease of computation and interpretation. The goal of this chapter is to advertize that the multiplicative structure on low-dimensional cohomology is also easy to compute and interpret. The chapter builds on a practical algorithm for finding a finite presentation of the fundamental group $\pi_1(X, x_0)$ of an arbitrary finite regular CW-space X which was illustrated in [31] and described in detail in [11]. In Section 5.2 we explain how, from such a presentation, one can directly read off the cup product

$$\cup\colon H^1(X,\mathbb{Z}) \times H^1(X,\mathbb{Z}) \to H^2(X,\mathbb{Z}) \tag{5.1}$$

without need for any further significant computations since this product is essentially an invariant of $\pi_1(X, x_0)$. In Section 5.3 we illustrate the method on the integral cohomology ring of a 3-dimensional digital image. Previous papers [42, 44, 40, 41] have described different approaches to computing the cohomology ring, over $\mathbb{Z}/2\mathbb{Z}$, of cubical and simplicial spaces arising from 3-dimensional digital images; these papers are based on techniques in [80, 43, 62]. In Section 5.4 we explain how the van Kampen's theorem for fundamental groupoids [12, 18] yields a distributed version of the fundamental group algorithm of [11], and hence a distributed method for computing (5.1). This chapter is published in Homology, Homotopy and Applications (HHA) [4].

5.2 The low dimensional cup product

Let C_*X denote the cellular chain complex of a finite regular CW-space X, and let $H^k(X,\mathbb{Z}) = H^k(Hom_{\mathbb{Z}}(C_*X,\mathbb{Z}))$ denote the cellular cohomology group (see [52] for details). Recall that the cohomology ring structure is derived from the *diagonal map* $\Delta\colon X \to X \times X, x \mapsto (x,x)$ and induced *diagonal homomorphism*

$$H^k(\Delta,\mathbb{Z})\colon H^k(X\times X,\mathbb{Z}) \to H^k(X,\mathbb{Z}) \tag{5.2}$$

together with a bilinear *cross product* function

$$H^m(X,\mathbb{Z}) \times H^n(X,\mathbb{Z}) \to H^{m+n}(X\times X,\mathbb{Z}) \tag{5.3}$$

for $k,m,n \geqslant 0$. The ring multiplication $H^*(X,\mathbb{Z}) \times H^*(X,\mathbb{Z}) \to H^*(X,\mathbb{Z})$ is obtained by composing (5.2) with (5.3) for $k = m+n$ and extending bilinearly over $H^*(X,\mathbb{Z})$. For an arbitrary regular CW-space the diagonal homomorphism (5.2) can be a challenge to implement efficiently on a computer. In contrast, an efficient implementation of (5.3) is straightforward. To implement (5.2) one can use the Alexander-Whitney diagonal approximation formula in the case of simplicial spaces, and Serre's analogue of this for cubical spaces. Details of the cubical analogue can be found in [71], and details on practical implementations of these two formulae can be found in [42, 44, 40, 41, 80, 43, 62]. In this section we assume that X is an arbitrary connected regular finite CW-space and observe that for $k = 2$ the homomorphism (5.2), and hence the cup product (5.1), can be read directly from a group presentation $\mathcal{P} = \langle \underline{x} \mid \underline{r} \rangle$ for the fundamental group $\pi_1 X$. One algorithm for computing such a presentation is described in [11].

The presentation $\mathcal{P} = \langle \underline{x} \mid \underline{r} \rangle$ for $\pi_1 X$ is produced by first constructing an admissible discrete vector on the 3-skeleton of X such that only one of the 0-cells is critical. By Theorem 3.2.1 there is then a homotopy equivalence $X \simeq Y$ where Y is a non-regular CW-space with a single 0-cell. The 2-skeleton of Y corresponds to the presentation $\mathcal{P} = \langle \underline{x} \mid \underline{r} \rangle$. The generators in $\underline{x}$ correspond to the critical 1-cells of X and to the 1-cells of Y. The relators in $\underline{r}$ correspond to the critical 2-cells in X and to the 2-cells in Y. We shall denote the 2-skeleton Y^2 by $K(\mathcal{P})$ since this is precisely the 2-complex associated to the presentation $\mathcal{P}$ in geometric group theory.

On the computer we store Y by storing the face lattice of the regular CW-space X together with a list of those pairs of non-critical cells (s,t) that constitute the discrete vector field on X. This data can be used to realize a chain homotopy equivalence $h_*\colon C_*X \xrightarrow{\simeq} C_*Y$ on the computer; this in turn could be used to compute an isomorphism $H^k(X,\mathbb{Z}) \cong H^k(Y,\mathbb{Z})$ if desired. So we focus on computing the cup product $\cup\colon H^1(Y,\mathbb{Z}) \times H^1(Y,\mathbb{Z}) \to H^2(Y,\mathbb{Z})$.

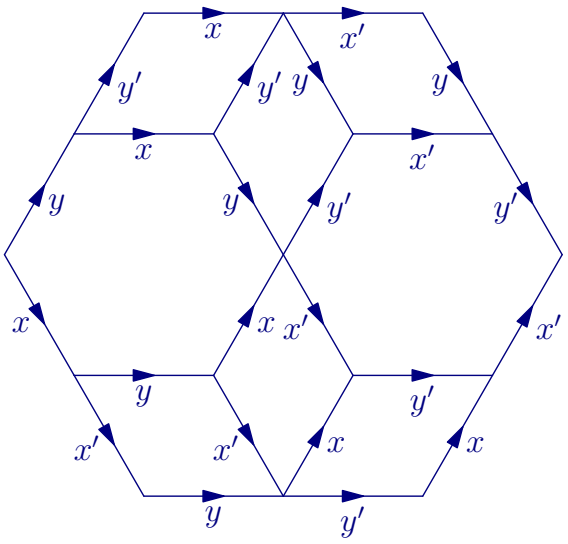


Figure 5.1 van Kampen diagram over the presentation $\langle x, y, x', y' \mid xyx = yxy, x'y'x' = y'x'y', xx' = x'x, yy' = y'y, xy' = y'x, yx' = x'y \rangle$.

The following notion, due to van Kampen and illustrated in Figure 5.1, is an aid to visualizing the diagonal homomorphism (5.2) for $k = 2$. Recall that a word in a free group is *cyclically reduced* if no conjugate of it is a shorter word, and that a presentation is *cyclically reduced* if all of its relators are. In this section we assume that presentations are cyclically reduced.

Definition 5.2.1. A *van Kampen diagram* over a cyclically reduced group presentation $\mathcal{P} = \langle \underline{x} \mid \underline{r} \rangle$ is a finite, planar, connected and simply connected CW-space $\mathcal{D} \subset \mathbb{R}^2$ with each 1-cell labelled by an arrow and a generator in $\underline{x}$ in such a way that: for each 2-cell e^2 in $\mathcal{D}$, the sequence of oriented and labelled 1-cells in the boundary ∂e^2, for some choice of initial vertex and some choice of orientation, spells a relator word in $\underline{r} \subset F(\underline{x})$.

It is not difficult to see that for any initial vertex in the boundary of a van Kampen diagram $\mathcal{D}$ over $\mathcal{P} = \langle \underline{x} \mid \underline{r} \rangle$, and for either orientation on the boundary of the diagram, the labelled 1-cells of the boundary spell a word in $R = \langle \underline{r} \rangle^{F(\underline{x})}$ where R is the normal closure in $F(\underline{x})$ of the subgroup generated by $\underline{r}$. The converse is also true: any non-trivial word in R is the boundary of a van Kampen diagram over $\mathcal{P}$. We are interested in a very easy and very particular case of this converse statement. Given a presentation $\mathcal{P} = \langle \underline{x} \mid \underline{r} \rangle$ of a group G, let $\underline{x}' = \{x' : x \in \underline{x}\}$ be an isomorphic copy of the set $\underline{x}$. Let $F(\underline{x} \cup \underline{x}')$ be the free group on the disjoint union $\underline{x} \cup \underline{x}'$ and let $\iota_1, \iota_2 \colon F(\underline{x}) \hookrightarrow F(\underline{x} \cup \underline{x}')$ be the inclusion homomorphisms defined by $\iota_1(x) = x$ and $\iota_2(x) = x'$ for $x \in \underline{x}$. We define the presentation

$$\mathcal{P} \times \mathcal{P} = \langle \underline{x} \cup \underline{x}' \mid \iota_1(r), \iota_2(r), \iota_1(x)\iota_2(y) = \iota_2(y)\iota_1(x) \quad (x, y \in \underline{x}, r \in \underline{r}) \rangle$$

for the direct product $G \times G$. In this context we define the *diagonal* group homomorphism $\Delta\colon F(\underline{x}) \to F(\underline{x} \cup \underline{x}')$ by $\Delta(x) = \iota_1(x)\iota_2(x)$ for $x \in \underline{x}$.
In the following lemma we use the notation $i <^{\epsilon} j$ to mean $i < j$ if $\epsilon = +1$ and $i \geqslant j$ if $\epsilon = -1$.

Lemma 5.2.1. *For any presentation $\mathcal{P} = \langle \underline{x} \mid \underline{r} \rangle$ and any reduced word $w = x_1^{\epsilon_1} x_2^{\epsilon_2} \dots x_n^{\epsilon_n} \in R = \langle\langle \underline{r} \rangle\rangle^{F(\underline{x})}$, where $\epsilon_i = \pm 1$ and $x_i \in \underline{x}$, there is a van Kampen diagram $\mathcal{D}$ over $\mathcal{P} \times \mathcal{P}$ whose boundary spells the word $\Delta(w)$ for some choice of initial vertex and orientation. The diagram $\mathcal{D}$ consists of precisely one 2-cell e^2_w with boundary word $\iota_1(w)$, one 2-cell $e^2_{w'}$ with boundary word $\iota_2(w)$ and, for each $1 \leqslant i <^{\epsilon_i} j \leqslant n$, one 2-cell e^2_{ij} whose boundary spells the relator $\iota_1(x_i)\iota_2(x_j)\iota_1(x_i)^{-1}\iota_2(x_j)^{-1}$.*

The van Kampen diagram $\mathcal{D}$ of Lemma 5.2.1 is typically *non-reduced* in the sense that there may be two CW-subspaces $\mathcal{D}_1, \mathcal{D}_2 \subset \mathcal{D}$ containing no common 2-cell whose union $\mathcal{D}_1 \cup \mathcal{D}_2$ is connected, simply-connected and has boundary that spells a word representing the trivial element in the free group of the presentation $\mathcal{P}$. A van Kampen diagram is said to be *reduced* if it contains no such subspaces $\mathcal{D}_1, \mathcal{D}_2$. It is possible to transform any van Kampen diagram $\mathcal{D}$ into a reduced diagram $\mathcal{D}'$ such that the boundary words of $\mathcal{D}$ and $\mathcal{D}'$ represent the same element in the free group of $\mathcal{P}$. The details of this transformation are not required for our cup product algorithm.

Example 5.2.1. Consider the presentation $\mathcal{P} = \langle x, y \mid xyxy^{-1}x^{-1}y^{-1} \rangle$ and word $w = xyxy^{-1}x^{-1}y^{-1} \in F(\{x, y\})$. A reduced van Kampen diagram over $\mathcal{P} \times \mathcal{P}$ with boundary word $\Delta(w)$ is shown in Figure 5.1.

Lemma 5.2.1 describes a cellular diagonal approximation $\Delta\colon K(\mathcal{P}) \to K(\mathcal{P}) \times K(\mathcal{P})$ which, in turn, induces a diagonal chain approximation

$$\Delta_*\colon C_*(K(\mathcal{P})) \to C_*(K(\mathcal{P} \times \mathcal{P})) \cong C_*(K(\mathcal{P})) \otimes C_*(K(\mathcal{P})). \tag{5.4}$$

Using e^0, e^1_x, e^2_r ($x \in \underline{x}, r \in \underline{r}$) to denote free abelian chain group generators, the chain map (5.4) is given in degrees $0, 1$ by

$$\Delta_0(e^0) = e^0,$$

$$\Delta_1(e^1_x) = e^1_x + e^1_{x'}.$$

In degree 2 it is defined for each $r = x_1^{\epsilon_1} x_2^{\epsilon_2} \dots x_n^{\epsilon_n} \in \underline{r}$ by

$$\Delta_2(e^2_r) = e^2_r + e^2_{r'} + \sum_{1 \leqslant i < j \leqslant n} e^2_{ij}. \tag{5.5}$$

The cup product on $H^*(K(\mathcal{P}), \mathbb{Z})$ is determined by the chain map (5.4) which, in turn, can be read off directly from the presentation $\mathcal{P}$ using Lemma 5.2.1. We

emphasize that one can apply this Lemma algebraically – there is no need to exhibit an actual van Kampen diagram in order to apply formula (5.5).
The inclusion $K(\mathcal{P}) \hookrightarrow Y$ and homotopy equivalence $Y \xrightarrow{\simeq} X$ induce a ring homomorphism $H^*(X,\mathbb{Z}) \to H^*(K(\mathcal{P}),\mathbb{Z})$ which restricts to an isomorphism $H^k(X,\mathbb{Z}) \xrightarrow{\cong} H^k(K(\mathcal{P}),\mathbb{Z})$ for $k = 0, 1$ and inclusion $H^2(X,\mathbb{Z}) \hookrightarrow H^2(K(\mathcal{P}),\mathbb{Z})$. We can thus compute the cup product (5.1) by using Lemma 5.2.1 to compute the cup product in the bottom row of the following commutative diagram.

$$\begin{array}{ccc}
H^1(\pi_1 X,\mathbb{Z}) \times H^1(\pi_1 X,\mathbb{Z}) & \xrightarrow{\cup} & H^2(\pi_1 X,\mathbb{Z}) \\
\downarrow\cong & & \downarrow \\
H^1(X,\mathbb{Z}) \times H^1(X,\mathbb{Z}) & \xrightarrow{\cup} & H^2(X,\mathbb{Z}) \\
\downarrow\cong & & \downarrow \\
H^1(K(\mathcal{P}),\mathbb{Z}) \times H^1(K(\mathcal{P}),\mathbb{Z}) & \xrightarrow{\cup} & H^2(K(\mathcal{P}),\mathbb{Z})
\end{array}$$

By attaching cells to X in dimensions $\geqslant 3$ we can construct a cellular inclusion $X \hookrightarrow B(\pi_1 X)$ into a space $B(\pi_1 X)$ with trivial homotopy groups $\pi_n(B(\pi_1 X)) = 0$ for $n \geqslant 2$ and with the inclusion inducing an isomorphism $\pi_1 X \cong \pi_1(B(\pi_1 X))$. The induced ring homomorphism $H^*(\pi_1 X,\mathbb{Z}) = H^*(B(\pi_1 X),\mathbb{Z}) \to H^*(X,\mathbb{Z})$ shows that the cup product (5.1) is induced from the cohomology of the group $\pi_1 X$ and hence can be calculated from *any* presentation $\mathcal{P}$ for $\pi_1 X$.

5.3 Illustration: digital images

Let T be the free abelian group of rank n with a specified free action on $\mathbb{R}^n$ as translations. Thus the elements $t \in T$ are distinct translations $\mathbb{R}^n \to \mathbb{R}^n, x \mapsto t(x)$. A fundamental domain for this action is

$$V = \{x \in \mathbb{R}^n \ : \ ||x|| \leqslant ||x - t(0)|| \text{ for all } t \in T\}$$

where $||\ ||$ is the Euclidean metric and 0 denotes the zero vector in $\mathbb{R}^n$. This domain V is an intersection of finitely many half-spaces or, equivalently, the convex hull of a finite collection of vertices. We define a *digital image* to consist of a subset $S \subset T$. For each $t \in S$ we say that $tV = \{t(x) \ : \ x \in V\}$ is a *voxel* of the digital image S. We define the *geometric realization* of S to be the union of the voxels of S.
If the elements of a free generating set for T act on $\mathbb{R}^n$ as the unit translations along the standard axes then V is an n-dimensional cube. The geometric realization of a digital image in this case is called a *pure cubical complex*. But there are other useful choices of action. If we identify $\mathbb{R}^n$ with the hyperplane $\mathbb{R}^n = \{(x_1,\dots,x_{n+1}) \in \mathbb{R}^{n+1} \ : \ x_1 + \cdots + x_{n+1} = 0\}$ and take the free generators of T to act as $x \mapsto x + v_i$ for $v_1 = (-n, 1, 1, \dots, 1, 1), v_2 = (1, -n, 1, \dots, 1, 1), \dots, v_n = (1, 1, 1, \dots, -n, 1)$ then V

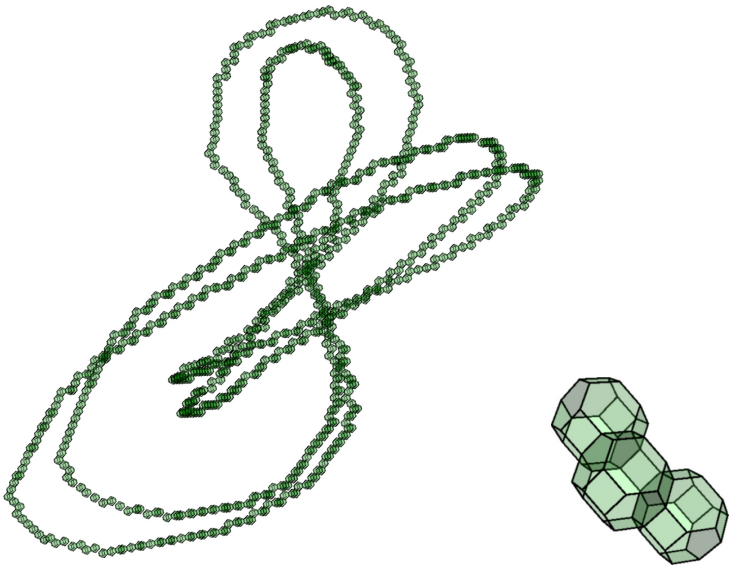


Figure 5.2 Pure permutahedral complex L representing a link with two components (left), an enlarged segment of which is also shown (right).

is the n-dimensional permutahedron. In this case the geometric realization of a digital image is called a *pure permutahedral complex*. An advantage to permutahedral complexes is that their boundary is a manifold; this is not necessarily so for pure cubical complexes. The term *lattice space* is used in [31] to refer to the geometric realizaton of an arbitrary digital image.

Example 5.3.1. The pure permutahedral complex L in Figure 5.2 represents a link with two components, one component winding around the other. To investigate the link we embed it into the interior of a contractible pure permutahedral complex R and form the complement $M = R \backslash \mathring{L}$ of the interior of L. The following GAP session loads such a complex M from the file `permutahedralcomplex.txt` available in Appendix B.2. The pure permutahedral complex M has 257851 voxels. As a CW-space it has 6982671 cells. The session first constructs a smaller homotopy equivalent pure permutahedral complex $XM \simeq M$ with just 3728 voxels and, as a CW-space, 156127 cells. The space XM is constructed using a *zig-zag deformation retract* technique based on simple homotopy collapses which is described explicitly in [31]. The session then uses XM to compute

$$H^n(M, \mathbb{Z}) = \begin{cases} \mathbb{Z}, & n = 0, \\ \mathbb{Z} \oplus \mathbb{Z}, & n = 1, 2, \\ 0, & n \geqslant 3, \end{cases}$$

and a presentation for $G = \pi_1 M$ using the algorithm of [11].

GAP session 5.3.1

```
gap> Read("permutahedralcomplex.txt");
gap> Size(M); # number of 3-d voxels
257851
gap> XM:=RegularCWComplex(ZigZagContractedComplex(M));
Regular CW-complex of dimension 3
gap> Size(XM); # number of CW cells
156127
gap> List([0..3],n->Cohomology(XM,n));
[ [ 0 ], [ 0, 0 ], [ 0, 0 ], [  ] ]
gap> G:=FundamentalGroup(XM);
<fp group of size infinity on the generators [ f1, f2, f3 ]>
```

The following continuation of the GAP session uses the method described in Section 5.2 to compute the cup product $\alpha \cup \beta = 6(-\gamma + \delta)$ where α, β are free generators of $H^1(M, \mathbb{Z})$ and γ, δ are free generators of $H^2(M, \mathbb{Z})$.

```
gap> cup:=CupProduct(G);
function( a, b ) ... end
gap> cup([1,0],[0,1]);
[ -6, 6 ]
```

It is well known that the cup product $\alpha \cup \beta$ can be interpreted in terms of the linking number $Lk(K_1, K_2)$ where K_i are the two components in the link of Figure 5.2 (see for instance [73]).

We mention that to any lattice space M one can associate a homotopy equivalent simplicial complex SM simply by taking the nerve of its collection of voxels; we let XSM denote SM regarded as a CW-space. The following continuation of the GAP session shows that for our example above this construction leads to a smaller CW-space than the one above involving permutahedral cells; the CW-space XSM has 31013 cells compared to the 156127 cells of XM. So it would have been slightly advantageous to work with XSM rather than XM.

```
gap> Size(XM);
156127
gap> XSM:=RegularCWComplex(Nerve(ZigZagContractedComplex(M)));
Regular CW-complex of dimension 3
gap> Size(XSM);
31013
```

Indeed, one could have reduced the size of the problem further by applying a sequence of simple homotopy collapses to obtain deformation retracts $XM1 \subseteq XM$ and $XSM1 \subseteq XSM$. The following commands

```
gap> XM1:=ContractedComplex(XM);;
gap> Size(XM1);
78341
gap> XSM1:=ContractedComplex(XSM);;
gap> Size(XSM1);
25185
```

produce CW-spaces $XM1$, $XSM1$ with 78341 cells and 25185 cells respectively. The simplicial complex SM provides a method for computing cup products in all dimensions when dealing with higher-dimensional M: one can use the Alexander-Whitney diagonal approximation on the cellular chain complex $C_*(SM)$. For practical computations one would construct an admissible discrete vector field on SM to produce a homotopy equivalence to a smaller non-regular CW-space Y and incorporate the induced chain homotopy equivalences $C_*(SM) \overset{\simeq}{\rightleftarrows} C_*Y$ into the computation. For the low-dimensional cup product (5.1) this approach tends to be slower than that described above. The following continuation of the GAP session computes, for our 3-dimensional example, a chain complex C_*Y with $C_0Y = \mathbb{Z}$, $C_1Y = \mathbb{Z}^3$, $C_2Y = \mathbb{Z}^3$, $C_nY = 0$ for $n \geqslant 3$ and so it is certainly quite practical to work with the chain equivalences $C_*(SM) \rightleftarrows C_*Y$ in this example.

```
gap> CY:=ChainComplex(XSM);
Chain complex of length 3 in characteristic 0 .
gap> List([0..3],CY!.dimension);
[ 1, 3, 3, 0 ]
```

5.4 Fundamental groupoids and a distributed algorithm

Let us analyse the computation of the previous section to see how it might be distributed over several computers. The computation begins with the representation of the link L as a pure permutahedral complex consisting of 864 voxels. The complement $M = R\backslash\mathring{L}$ is then constructed as a pure permutahedral complex with 257851 voxels. The voxels of M are stored as a list of points in $\mathbb{R}^3$. There are then three steps to computing the cohomology cup product on M, the first and second of which consume the most time and memory.

1. A zig-zag homotopy contraction (details of which are given in [31]) is performed to compute a pure permutahedral complex M' which is homotopy equivalent to M and which involves only 3728 voxels.

2. The complex M' is realized as a regular CW-space $X = XM$ involving 156127 cells in dimensions $\leqslant 3$. A maximal admissible discrete vector field is computed on X and the critical 1-cells and 2-cells are used to construct a presentation $\mathcal{P}$ for $\pi_1 X \cong \pi_1 M$.

3. The cup product $\alpha \cup \beta$ is read off directly from $\mathcal{P}$.

It is difficult to make meaningfully precise statements about the complexity of Steps 1–3. Step 2 involves the computation of a discrete vector field on a regular CW-space X. One could always deem every cell of X to be critical, in which case there is nothing to compute. At the other extreme one could aim for a vector field involving a minimal number of critical cells – but the construction of such a vector field is known to be an NP hard problem (see for instance [68]). In Step 1 the zig-zag homotopy reduction procedure is based on repeated applications of procedures which can be regarded as producing discrete vector fields on a lattice complex and so the comments for Step 2 also apply to Step 1. In Step 3 the 1- and 2- dimensional cohomology groups of a cochain complex need to be computed using some version of a Smith Normal Form algorithm and here again it is possible to produce examples where small input data can yield lengthy computations. Rather than attempt to make any precise complexity statement we simply provide the running times (on a Linux laptop with Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz) for each of the three steps applied to the example of Section 5.3: Step 1 took 5.7 seconds; Step 2 took 5.4 seconds; Step 3 took 0.004 seconds.

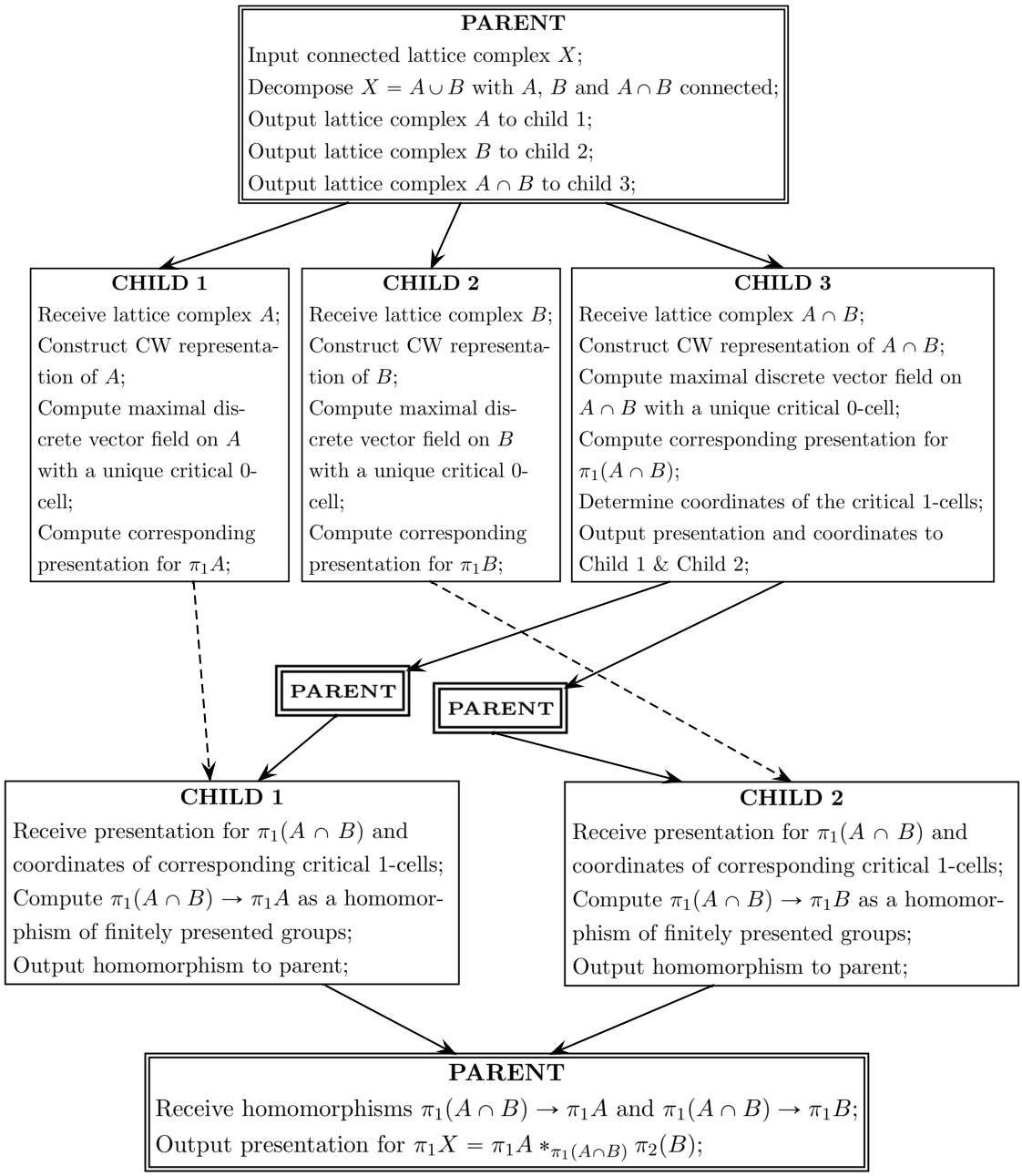
Step 1 involves repeatedly applying a basic homotopy deformation procedure which inputs a pure lattice complex M and outputs a homotopy equivalent pure lattice complex M' with potentially fewer voxels. The basic deformation procedure is applied until no size reduction occurs. The initial application of the procedure takes 1.1 seconds to reduce the number of voxels from 257851 to 25719 – a 90 percent reduction in size. The basic procedure has to be applied a total of six times in order to achieve a complex with 3728 voxels – a 98 percent reduction in size. So in this example Step 1 is best speeded up simply by applying the basic procedure only once rather than six times.

We thus focus on Step 2, with the aim of performing a distributed computation of a presentation $\mathcal{P}$ for $\pi_1 X$.

We start by supposing that in Step 2 we have some expression for $M' = A \cup B$ as a union of two pure permutahedral complexes A, B whose intersection $A \cap B$ is also a pure permutahedral complex. (For instance, we could choose two suitable integers $l < r \in \mathbb{Z}$ and let A consist of those voxels of M' whose centre has x-coordinate $< r$,

and B consist of those voxels whose centre has x-coordinate $> l$.) For the moment let us make an assumption which is mathematically convenient but which is not so realistic for many applied settings: assume that the spaces A, B and $A \cap B$ are all connected.

Given three computers PC_1, PC_2, PC_3 each directly connected to a parent computer we can send, from the parent computer, the data describing the pure permutahedral complex A to PC_1, the data describing B to PC_2, and the data describing $A \cap B$ to PC_3. We can then simultaneously construct the face lattices for A, B, $A \cap B$ and represent these spaces as regular CW-spaces $X_A, X_B, X_{A\cap B}$ on PC_1, PC_2, PC_3 respectively. Then we can simultaneously compute maximal discrete vector fields on the 3-skeleta of X_A and X_B, and on the 2-skeleton of $X_{A\cap B}$. Because of the connectivity assumptions the computation can be done so that in each case only one 0-cell is critical. We can then read off on PC_1 the presentation $\mathcal{P}_A = \langle \underline{x}_A \mid \underline{r}_A \rangle$ for $\pi_1(X_A)$, and read off on PC_2 the presentation $\mathcal{P}_B = \langle \underline{x}_B \mid \underline{r}_B \rangle$ for $\pi_1(X_B)$. On PC_3 we have that the critical 1-cells of $X_{A\cap B}$ correspond to a generating set $\underline{x}_{A\cap B}$ for $\pi_1(X_{A\cap B})$. The identities of these critical 1-cells on $X_{A\cap B}$ can be sent from PC_3 to both PC_1 and PC_2. Then, on PC_1 we can use the discrete vector field on X_A to compute the homomorphism of free groups $f_A \colon F(\underline{x}_{A\cap B}) \to F(\underline{x}_A)$ induced by the inclusion $X_{A\cap B} \hookrightarrow X_A$ while, simultaneously on PC_2, we can use the discrete vector field on X_B to compute the corresponding homomorphism $f_B \colon F(\underline{x}_{A\cap B}) \to F(\underline{x}_B)$. Here $F(\underline{x}_A)$ denotes the free group on $\underline{x}_A$. The presentations for $\pi_1 A$, $\pi_1 B$, $\pi_1(A \cap B)$ and the homomorphisms f_A, f_B can now all be sent to the parent. The presentation $\mathcal{P}_{A\cup B} = \langle \underline{x}_A, \underline{x}_B \mid \underline{r}_A, \underline{r}_B, f_A(x) = f_B(x) \ (x \in \underline{x}_{A\cap B}) \rangle$ for the amalgamated product $\pi_1 A *_{\pi_1(A\cap B)} \pi_1 B$ can now be computed on the parent. By van Kampen's theorem $\mathcal{P}_{A\cup B}$ is a presentation for the fundamental group $\pi_1 X = \pi_1(A \cup B)$. The following is a flow chart of the parallel computation.



This distributed computation of a presentation for $\pi_1 X$ is correct because the classical theorem of van Kampen can be invoked thanks to the connectivity assumptions on $A, B, A \cap B$. The distributed computation is practical because: (i) the digital images $A, B, A \cup B$ can be efficiently transmitted between computers simply by transmitting the lists of their voxel centres (there is no need to transmit the CW face lattice); (ii) the homotopy-theoretic information described by the critical cells of the CW-spaces $X_A, X_B, X_{A \cap B}$ in dimensions $\leqslant 2$ can be encoded as group presentations and then efficiently transmitted between computers. The method, under the connectivity assumptions, is implemented in the current version of HAP [28].

But one only has to think of a digital image representing a torus in $\mathbb{R}^3$ to realize that many of the most natural decompositions $M = A \cup B$ of a digital image M will

have disconnected intersection $A \cap B$. To overcome this difficulty we pay attention to Alexander Grothendieck [46]:

> Ceci est lié notamment au fait que les gens s'obstinent encore, en calculant avec des groupes fondamentaux, à fixer un seul point base, plutôt que d'en choisir astucieusement tout un paquet qui soit invariant par les symétries de la situation, lesquelles sont donc perdues en route. Dans certaines situations (comme des théorèmes de descente à la Van Kampen pour groupes fondamentaux) il est bien plus élégant, voire indispensable pour y comprendre quelque chose, de travailler avec des groupoïdes fondamentaux par rapport à un paquet de points base convenable ...

The above description of a distributed computation of a presentation for the fundamental group $\pi_1(X, x_0)$ of a regular CW-space $X = X_A \cup X_B$, under severe connectivity assumptions, extends to a computation of a presentation for the fundamental groupoid $\pi_1(X, X_0)$ where the connectivity assumptions are relaxed to the requirement that the set X_0 has non-empty intersection with each connected component of $X_{A\cap B}$, each connected component of X_A, and each connected component of X_B.
If a regular CW-space X is connected then there is a simple procedure, given in [54], for extracting a presentation $\mathcal{P}$ for $\pi_1(X, x_0)$ from a presentation for $\pi_1(X, X_0)$. This provides a practical method for distributing the computation of $\mathcal{P}$ over several computers.

The above distributed computation extends to regular CW-spaces $X = X_1 \cup X_2 \cup \cdots \cup X_n$ arising as the union of CW-subspaces X_i for $1 \leqslant i \leqslant n$. Let X_0 denote a subset of the 0-skeleton X^0 that has non-empty intersection with each connected component of $X_i \cap X_j$ and non-empty intersection with each connected component of $X_i \cap X_j \cap X_k$ for all $1 \leqslant i, j, k \leqslant n$. There is a diagram of groupoid morphisms

$$\bigsqcup_{1\leqslant i<j\leqslant n} \pi_1(X_i \cap X_j, X_0 \cap X_i \cap X_j) \overset{\alpha}{\underset{\beta}{\rightrightarrows}} \bigsqcup_{1\leqslant i\leqslant n} \pi_1(X_i, X_0 \cap X_i) \xrightarrow{\gamma} \pi_1(X, X_0) \qquad (5.6)$$

where $\bigsqcup$ denotes disjoint union in the category of groupoids; α is induced by the inclusion $X_i \cap X_j \hookrightarrow X_i$; β is induced by the inclusion $X_i \cap X_j \hookrightarrow X_j$; γ is induced by the inclusion $X_i \hookrightarrow X$. The generalization of van Kampen's theorem given in [18] implies that in (5.6) the morphism γ is the coequalizer of α and β in the category of groupoids. Thus to compute a presentation for $\pi_1(X, X_0)$ one needs only compute presentations for each $\pi_1(X_i, X_0 \cap X_i)$ and the images under α and β of generators for each $\pi_1(X_i \cap X_j, X_0 \cap X_i \cap X_j)$. One can achieve this by first choosing a set X_0 of vertices that has non-empty intersection with each connected component of all double and triple intersections. Then compute admissible discrete vectors fields on the 2-skeleta of $X_i \cap X_j$ for all $1 \leqslant i < j \leqslant n$ such that all vertices in $X_0 \cap X_i \cap X_j$

are critical. Also compute admissible discrete vector fields on the 3-skeleta of the X_i such that all vertices in $X_0 \cap X_i$ are critical. From the vector fields on the $X_i \cap X_j$ and X_i one can construct the coequalizer (5.6) as a diagram of finitely presented groupoids and thus obtain a presentation for $\pi_1(X, X_0)$.

It is instructive to give an example showing the necessity for X_0 to have non-empty intersection with each connected component of three-fold intersections. Let X be the simplicial graph with vertex set $X^0 = \{1, 2, 3, 4, 5\}$ and edges $\{1, 4\}$, $\{2, 4\}$, $\{3, 4\}$, $\{1, 5\}$, $\{2, 5\}$, $\{3, 5\}$. For $1 \leqslant i \leqslant 3$ let X_i be the subgraph consisting of all vertices except i and those edges not incident with i. Let $X_0 = \{1, 2, 3\}$. In this case, for each $1 \leqslant i < j \leqslant 3$ the groupoid $\pi_1(X_i \cap X_j, X_0 \cap X_i \cap X_j)$ consists of a single object and a single identity arrow. It follows that for $X = X_1 \cup X_2 \cup X_3$ the diagram (5.6) is not a coequalizer diagram.

Chapter 6

Final Example

In this final chapter we give an example that summarizes and illustrates the aims of the thesis. We start with a data set S of $n = 1700$ points in $\mathbb{R}^3$ which is stored in the text file "data61.txt", see Appendix B.3. We construct a filter function

$$f : \mathbb{R}^3 \to \mathbb{R}^2$$

by using our GAP implementation of principal component analysis and project onto the hyperplane spanned by the two principal components.

GAP session 6

```
gap> Read("data61.txt");
gap> Length(S);
1700
gap> M:=VectorsToCovarianceMatrix(S)*1.0;
[[8.70847, -0.283718e-3, -0.354466e-1],
 [-0.283718e-3, 8.71985, -0.527455e-1],
 [-0.354466e-1, -0.527455e-1, 1.62771]]
gap> eig:=FloatSpectrum(M,100);;
gap> eig[1];
[ 8.72024229079632, 8.70864740941752, 1.62714029978615 ]
gap> A:=[eig[2][1],eig[2][2]];
[ [ 0.999986, 0.00169704, -0.00501849 ],
  [ -0.00173429, 0.999971, -0.00742788 ] ]
gap> f:=function(x) return A*x; end;
function( x ) ... end
```

Next we use our GAP implementation of the Mapper clustering algorithm to produce a simplicial complex K representing the data S and f.
The simplicial complex K requires the choice of an open cover of $\mathbb{R}^2$. For this we took of square grid of $9^2 = 81$ points lying close to the image $f(S)$.

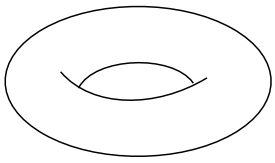
```
gap> P:=Cartesian(List([0..8],x->1/2+x),List([0..8],y->1/2+y));;
gap> r:=95/100;;
gap> epsilon:=2;;
gap> dx:=EuclideanApproximatedMetric;;
gap> dz:=EuclideanApproximatedMetric;;
gap> K:=GeneralMapper(S,f,P,dx,epsilon,dz,r,cluster);
Simplicial complex of dimension 3.
```

The constructed simplicial complex K has dimension 3. To investigate the topology of K we computed homology groups .

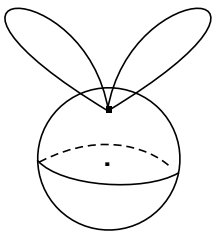
$$\begin{aligned} H_0(K) &= \mathbb{Z} \\ H_1(K) &= \mathbb{Z} \oplus \mathbb{Z} \\ H_2(K) &= \mathbb{Z} \end{aligned}$$

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

There are two likely candidates for the homology type of K. We might have $K \simeq \mathbb{S}^1 \times \mathbb{S}^1$.



or we might have $K \simeq \mathbb{S}^2 \vee \mathbb{S}^1 \vee \mathbb{S}^1$.



To investigate these two possibilities we construct a presentation for the fundamental group $\pi_1(K)$. For the given example it is, in fact, possible to compute a presentation

for $\pi_1(K)$ directly from a discrete vector field on K. However, in general a computation of $\pi_1(K)$ can require lots of CPU time and memory and so there is merit in computing a presentation $\pi_1(K)$ in a distributed fashion using the van Kampen's theorem for fundamental groupoids . To do this for $a > b$ we could consider

$$\begin{aligned} R^{\leqslant a} &= \{(x, y) \in \mathbb{R}^2 : x \leqslant a\}, \\ R^{\geqslant b} &= \{(x, y) \in \mathbb{R}^2 : x \geqslant b\}. \end{aligned}$$

Now we can form

$$\begin{aligned} S^a &= \{s \in S : f(s) \in R^{\leqslant a}\}, \\ S^b &= \{s \in S : f(s) \in R^{\geqslant b}\}. \end{aligned}$$

We now could use our GAP implementation of Mapper to construct simplicial complexes K^a, K^b representing the data (S^a, f) and (S^b, f).
Let V, V^a, V^b, denote the vertex sets for K, K^a, K^b. Note that, by construction, $V^a \subset V, V^b \subset V$ and $V = V^a \cup V^b$. In fact by choosing a, b appropriately, we get

$$K = K^a \cup K^b$$

So now could apply the van-Kampen theorem and our GAP implementation of fp groupoids to $K^a, K^b, K^a \cap K^b$ in order to obtain a presentation for $\pi_1(K)$.

To illustrate the required GAP commands in a succinct fashion, we now show how to apply our fp groupoid implementations directly on K (rather than on K^a, K^b).

The following commands compute a finite presentation for the fundamental groupoid

$$\pi_1(K, \{x, y, z\})$$

for (an arbitrary choice of) vertices $x, y, z \in K$.

```
gap> Y:=SimplicialComplexToRegularCWComplex(K);
Regular CW Complex of dimension 3;
gap> V:=[1, 27, 43];;
gap> G:=FundamentalGroupoidOfRegularCWComplex(Y,V);
< fp groupoid on the generators [ f1 , f2 , f3 , f4 ] >

gap> gens:=GeneratorsOfGroupoid(G);;
gap> List(gens, x-> Source(x), Target(x));
[ [1, 43], [27, 1], [43, 1], [1, 27] ]
```

Now we use our GAP implementation of the vertex group to find a presentation for the group $\pi_1(K)$. Since $H_0(K) = \mathbb{Z}$, we know that K is connected and the choice of base point is not important.

```
gap> v:=Source(gens[1]);
27
gap> Gv:=VertexGroup(G,v);
<fp group of size infinity on the generators [ f3, f4 ]>

gap> RelatorsOfFpGroup(Gv);
[ f3*f4*f3^-1*f4^-1]
```

We obtain the presentation

$$\pi_1(K) = \langle x, y \mid xyx^{-1}y^{-1} \rangle$$

In this example we are able to see, directly from the presentation, that

$$\pi_1(K) \cong \mathbb{Z} \times \mathbb{Z}$$

However, one can not so easily identify the homotopy type from the group.
But we can use our GAP implementation of cup products to identify the cup product

$$\cup : H^1(K, \mathbb{Z}) \times H^1(K, \mathbb{Z}) \to H^2(K, \mathbb{Z}) \tag{6.1}$$

```
gap> cup:=CupProduct(Gv);
function( a, b ) ... end
gap> cup([1,0], [0,1]);
[ -1 ]
```

This shows that the cup product 6.1 is non-zero.
We know that the cup product for $\mathbb{S}^2 \vee \mathbb{S}^1 \vee \mathbb{S}^1$ is trivial so this suggests that K may have the homotopy type of a torus.

Appendix A

FpGd functions

A.1 Fp Groupoids

FreeGroupoid

`FpGdFreeGroupoid(objs,gens)::  List, List  → FreeGroupoid`
Input a list `objs` of integers represent the objects of the groupoid and a list `gens`= $[[s_1, g_1, t_1], ..., [s_n, g_n, t_n]]$ of triples of integers for the generators with their sources s_i and targets t_i, and returns a `free groupoid` on `objs` and generated by `gens`, where $s_i, t_i \in$ `obj`.

FpGroupoid

`FpGdFpGroupoid(objs,gens,rels)::  List, List, List  → FpGroupoid`
`FpGdFpGroupoid(G,S)::  Group, Group  → FpGroupoid`
Input a list `objs` of integers represent the objects of the groupoid, a list `gens`= $[[s_1, g_1, t_1], ..., [s_n, g_n, t_n]]$ of triples of integers for the generators with their sources s_i and targets t_i and a list `rels`= $[r_1, ..., r_m]$, where $r_i = [g_{i_1}, ..., g_{i_n}]$, $g_{i_k} \in \{\pm g_1, ..., \pm g_n\}$ and returns a `fp groupoid` on the objects `objs` and generated by `gens` subject to a finite set of relations `rels` that these generators satisfy.
Finitely presented groupoids can also obtained by factoring a free groupoid `F` by a set of relators `rels`.
Input a Group G and a subgroup S of G, returns a finitely presented groupoid $\mathfrak{Gpd}(G, S)$ induced by the group action of G on the left cosets G/S.

FpGdProduct

`FpGdProduct(L):: List → FreeGroupoidElm`

`FpGdProduct(L):: List → FpGroupoidElm`

Input a list `L`= $[g_1, ..., g_n]$ of composed generators of some `free groupoid` and return their product.

Input a list `L`= $[g_1, ..., g_n]$ of composed generators of some `fp groupoid` and return their product.

GeneratorsOfGroupoid

`GeneratorsOfGroupoid(F):: FreeGroupoid → List`

`GeneratorsOfGroupoid(G):: FpGroupoid → List`

Input a `free groupoid F` and returns a list of the generators of `F`.

Input a `fp groupoid G` and returns a list of the generators of `G`.

RelatorsOfFpGroupoid

`RelatorsOfFpGroupoid(G):: FpGroupoid → List`

Input a `fp groupoid G` and returns a list of arrows represents the relators of `G`.

FreeGroupoidOfFpGroupoid

`FreeGroupoidOfFpGroupoid(G):: FpGroupoid → FreeGroupoid`

Input a `fp groupoid G` and returns the underlying free groupoid `G`. This is the groupoid generated by the free generators of `G`.

IsConnectedFpGroupoid

`IsConnectedFpGroupoid(G):: FpGroupoid → Boolean`

Input `fp groupoid G` and returns `true` if `G` is connected, otherwise returns `false`.

ComponentsOfFpGroupoid

`ComponentsOfFpGroupoid(G):: FpGroupoid → List`

Input a `fp groupoid G` and returns a list of connected groupoids, such that the disjoint union of these connected groupoids is `G`.

FundamentalGroupoid

`FundamentalGroupoid(Y,V):: RegCWComplex, List → FpGroupoid`

`FundamentalGroupoid(f,V):: RegCWMap, List → GroupoidMorphism`

Input a regular CW-complex `Y` and a list `V` $\subset$ `Y`^{0} and returns an fp groupoid of `Y`.

Input a regular CW-map `f` : $X \to Y$ and a list `V` $\subset$ `X`^{0} and returns the fundamental groupoid $\pi_1(f) : \pi_1(X, V) \to \pi_1(Y, f(V))$.

VertexGroup

`VertexGroup(G,v)::  FpGroupoid, Int → FpGroup`
Input `fp groupoid G` and an integer `v` and returns an fp group $G(v, v)$ represents the vertex group of `G` at `v`.

FpGroupToFpGroupoid

`FpGroupToFpGroupoid(G)::  FpGroup → FpGroupoid`
Input an fp group `G` and returns a presentation of groupoid on one object.

UnionOfGroupoids

`UnionOfGroupoids(G,H)::  FreeGroupoid, FreeGroupoid→FreeGroupoid`
`UnionOfGroupoids(G,H)::  FpGroupoid, FpGroupoid → FpGroupoid`
Input two `free groupoids G` and `H` and returns a free groupoid `F` on the objects `objs(G)` ∪ `objs(H)` and generated by `gens(G)` ∪ `gens(H)`.
Input two `fp groupoids G` and `H` and returns an fp groupoid `F` on the objects `objs(G)` ∪ `objs(H)`, generated by `gens(G)` ∪ `gens(H)` and it subjects to the relators `rels(G)` ∪ `rels(H)`.

GroupoidMorphismByImages

`GroupoidMorphismByImages(G,H,gens,imgs)::  Groupoid, Groupoid, List, List → GroupoidMorphism`
Input two groupoids `G` and `H` and two lists `gens` and `imgs` of the generators of `G` and arrows in `H`, respectively. It returns the groupoid morphism for which the source is `G` and the range is `H`. The groupoid morphism maps the members of `gens` to their images in `H`.

ImageOfArrow

`ImageOfArrow(f,a)::  GroupoidMomorphism, GroupoidElm→GroupoidElm`
Input groupoid morphism `f`: $G \to H$ and an arrow `a` in the gorupoid G returns the image `f`(a) in the groupoid H.

PushoutOfFpGroupoids

`PushoutOfFpGroupoids(f,g)::  GroupoidMorphism, GroupoidMorphism → FpGroupoid`
Input two groupoid morphisms `f` : $\pi_1(X, V) \to \pi_1(Y, f(V))$ and `g` : $\pi_1(X, V) \to \pi_1(Z,$`g`$(V))$ and returns the pushout $\pi_1(Y,$`f`$(V)) *_{\pi_1(X,V)} \pi_1(Z,$`g`$(V))$.

Source

`Source(w)::` `GroupoidElm` → `Int`

`Source(F)::` `GroupoidMorphism` → `Groupoid`

Input a groupoid arrow `w` and returns an integer represents the source of `w`.

Input a groupoid morphism `F` : $G \to H$ returns the groupoid G.

Target

`Traget(w)::` `GroupoidElm` → `Int`

`Target(F)::` `GroupoidMorphism` → `Groupoid`

Input a groupoid arrow `w` and returns an integer represents the taget of `w`.

Input a groupoid morphism `F` : $G \to H$ returns the groupoid H.

Display

`Display(M)::` `List` → Void

Display a graph G for which, M is the adjacency matrix.

A.2 Mapper

VectorsToCovarianceMatrix

`VectorsToCovarianceMatrix(L)::` `List` → `Matrix`

Input a list `L` = $[p_1, p_2, ..., p_n]$, where $p_k = [p_{k_1}, p_{k_2}, ..., p_{k_d}]$ is a list of real numbers, and returns a symmetric $d \times d$ matrix M called the covariance matrix of `L`.

FloatSpectrum

`FloatSpectrum(M,n)::` `Matrix`, Int → `List`

`FloatSpectrum(M,n,S)::` `Matrix`, Int, String → `List`

Input a symmetric matrix `M` and an integer n and returns a list $[L_1, L_2]$, where L_1 is a list of the eigenvalues of `M` in descending order and L_2 is a list of the corresponding eigenvectors.

Input a symmetric matrix `M`, an integer n and a string "Jacobi" and returns a list $[L_1, L_2]$, where L_1 is a list of the eigenvalues of `M` in descending order and L_2 is a list of the corresponding eigenvectors based on Jacobi eigenvalue algorithm.

cluster

`cluster(S, e, d)::` `List, float, function` → `List`

Input a list `S` = $[p_1, p_2, ..., p_n]$, where $p_k = [p_{k_1}, p_{k_2}, ..., p_{k_d}]$ is a list of real numbers, real number `e` and a metric function $d : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ and returns a list $[S_1, S_2, ..., S_m]$ such that for any $i, j \in \{1, 2, ..., m\}$, $S_i \cap S_j = \varnothing$, i.e. if $x \in S_i$ and $y \in S_j$ then $d(x, y) > e$.

GeneralMapper

`GeneralMapper(S, P, f, dx, a, dz, b, c):: List, List, function, function, Rat, function, Rat, function → Simplicial Complex`

Input a list S, a list P, a function f, a function dx, a real number a, a function dz, a real number b and a function c and returns a simplicial complex K.

ReadAMCfileAsPatternMatrix

`ReadAMCfileAsPatternMatrix(str):: String → List`

Reads a AMC file identified by a string `str` such as "file.amc" or "path/file.amc" and returns a list L. The list L represents a matrix of 62 columns, the entries are rational numbers.

Appendix B

Data sets

B.1 dataset321.txt

```
A:=[
[2,3,9],[3,9,10],[9,10,14],[10,14,15],[2,14,15],[2,3,15],
[3,4,10],[4,10,11],[10,11,15],[11,15,16],[3,15,16],[3,4,16],
[4,5,11],[5,11,12],[11,12,16],[12,16,17],[4,16,17],[4,5,17],
[5,6,12],[6,12,13],[12,13,17],[13,17,18],[5,17,18],[5,6,18],
];

B:=[
[1,2,7],[2,7,9],[7,8,9],[8,9,14],[1,8,14],[1,2,14],
[2,3,9],[3,9,10],[9,10,14],[10,14,15],[2,14,15],[2,3,15],
[5,6,12],[6,12,13],[12,13,17],[13,17,18],[5,17,18],[5,6,18],
[1,6,13],[1,7,13],[7,13,18],[7,8,18],[6,8,18],[1,6,8]
];

AB:=[
[2,3,9],[3,9,10],[9,10,14],[10,14,15],[2,14,15],[2,3,15],
[5,6,12],[6,12,13],[12,13,17],[13,17,18],[5,17,18],[5,6,18]
];
```

B.2 permutahedralcomplex.txt

The text file `permutahedralcomplex.txt` include a binary array A of size $84 \times 77 \times 40$. As a sample, $A[44, :, :]$ is below.

```
[
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
```

```
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1,1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
```

```
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
```

```
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
```

```
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
];
```

B.3 dataset61.txt

The text file `dataset61.txt` is stored in our package `FpGd` [3], it is a set of 1700 3-dimensional points. The list S below include the first 100 points in `dataset61.txt`.

```
S:=[[237/200,189/40,121/40],[1749/200,179/20,3/2],[33/4,9/4,59/20],
[1111/200,363/40,41/25],[-3/40,267/200,17/10],[121/20,807/200,469/
```

200],[249/200,114/25,1/20],[1737/200,357/40,163/200],[166/25,241/
40,3/40],[27/100,267/40,-3/40],[357/40,137/50,251/100],[869/100,
319/40,-1/40],[1483/200,1227/200,3],[41/8,7/8,3],[61/20,149/25,141
/50],[19/4,277/40,3],[767/200,1341/200,59/20],[-1/20,1029/200,211/
200],[1793/200,161/20,-1/40],[179/20,34/5,209/200],[39/20,39/10,3/
40],[118/25,363/40,6/25],[1753/200,433/50,121/40],[299/50,357/40,
9/4],[241/40,467/100,369/200],[-13/200,1307/200,59/20],[781/200,
1669/200,3],[151/40,329/40,119/40],[313/100,349/40,-3/40],[179/20
,69/40,99/200],[127/40,359/40,83/40],[109/20,243/40,51/100],[-1/40
,267/200,137/50],[1111/200,71/50,-1/40],[3/40,209/25,3/40],[157/50,
61/50,1/40],[571/100,237/40,1],[9,563/200,33/40],[1/20,37/10,101/
40],[62/25,767/200,-1/40],[57/40,-1/40,13/100],[97/40,71/20,-1/20],
[239/40,541/100,123/40],[289/40,41/5,119/40],[0,183/50,141/50],
[216/25,1193/200,-1/40],[1637/200,1101/200,123/40],[179/20,57/
40,21/20],[189/50,-1/20,311/200],[759/200,-1/20,561/200],[-3/40,
27/10,19/40],[3/40,1233/200,129/200],[79/40,57/20,59/20],[1541/
200,1/40,213/200],[-1/40,771/200,223/100],[359/200,3/40,289/200]
,[41/20,249/40,3/40],[193/40,353/40,119/40],[309/40,203/40,3/40],
[151/40,329/40,1/40],[191/40,41/5,1/20],[191/40,9/40,1/40],[481/200
,381/100,1/20],[237/40,591/200,19/200],[121/20,1069/200,43/25],
[207/100,683/100,3],[133/20,359/40,73/50],[121/40,0,46/25],[1411/
200,1731/200,119/40],[33/40,9/4,1/20],[1461/200,-9/200,59/20],[29/
5,117/40,219/200],[1793/200,99/40,-3/40],[7/20,219/25,3/40],[3/8,
103/40,-1/40],[1793/200,247/40,121/40],[619/200,119/40,181/100],
[803/100,216/25,61/20],[1143/200,237/40,99/200],[353/40,289/40,
123/40],[101/40,541/100,119/40],[2/25,9,81/200],[123/200,1147/200,
-1/20],[29/40,361/40,361/200],[263/40,157/200,1/20],[411/200,363/40
,221/100],[1/40,347/100,1/20],[319/40,1/4,59/20],[179/20,117/20,41/
20],[89/40,27/40,3],[1607/200,67/50,119/40],[1783/200,363/40,447/
200],[771/100,1129/200,-1/20],[-3/40,603/200,393/200],[1727/200,
21/50,61/20],[166/25,1731/200,-3/40],[83/25,1231/200,117/40],[15/
4,663/100,3],[13/8,-3/40,199/200],[623/200,-1/40,13/20]];

Appendix C

Codes of FpGd

C.1 Groupoid

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

#                    Functions & Methods

#############################################################################
#0
#F  IdentityGroupoidElm
##
##
##
InstallMethod(IdentityGroupoidElm,
               "Method for Identity Element of Free Groupoid",
         [IsFpGdFreeGroupoid,IsInt],
function(G,obj)
local a, rec_a, A;
a:=[];
rec_a:=rec(list:=[], source:=obj, target:=obj, parent:=G);
A:=Objectify(FpGdFreeGroupoidElm,rec_a);
return A;
end);
#
#############################################################################
#0
#F  =
##
##
```

```
##
InstallMethod(\=,"for two words in Free Groupoid", IsIdenticalObj,
 [IsFpGdFreeGroupoidElm, IsFpGdFreeGroupoidElm], 0,
function(w1,w2)
return
      w1!.source = w2!.source and
      w1!.target = w2!.target and
      w1!.list = w2!.list;
end);
#
############################################################

InstallTrueMethod(IsMultiplicativeElement,IsFpGdFreeGroupoidElm);

#
############################################################################
#0
#F  SimplifyList
##
##
##
SimplifyList:=function(w)
local l, n, v,i,w_new,x, j;
 l:=Length(w);
 if l=0 then
     return [];
 else
     w_new:=ShallowCopy(w);
     n:=Int(l/2);
    for j in [1..n] do
          v:=ShallowCopy(w_new);
          i:=0;
          for x in List([1..Length(v)-1],i->v[i]) do
                i:=i+1;
               if x=-v[i+1] then
                     Unbind(v[i]);
                     Unbind(v[i+1]);
               fi;
         od;
         w_new:=Filtered(v,a->IsBound(a));
```

```
    od;
  return w_new;
  fi;
end;
#
#############################################################################
#0
#F  *
##
##
##
InstallMethod
(  \*,
   "for two free groupoid elements in 'IsFpGdFreeGroupoidElm' ",
   [IsFpGdFreeGroupoidElm, IsFpGdFreeGroupoidElm],

function( w1 , w2 )
local w, F ;
if not (w1!.target = w2!.source) then
    Print("Words are not composable. \n");
    return fail;
else
    F:=w1!.parent;
    w:=SimplifyList(Concatenation(w1!.list,w2!.list));
    return Objectify(FpGdFreeGroupoidElm,
    rec(list:=w, source:=w1!.source, target:=w2!.target, parent:=F));
fi;
end
);
#
#############################################################################
#0
#F  FpGdProduct
##
##
##
InstallGlobalFunction(FpGdProduct, [IsFpGdFreeGroupoidElm],
function(w)
local k,L,i;
k:=Length(w);
```

```
L:=w[1];
for i in [2..k] do
    L:=w[i]*L;
od;
return L;
end);
#
#############################################################################
#0
#F  ^
##
##
##
InstallMethod
(  \^,
   "for one free groupoid element in 'IsFpGdFreeGroupoidElm' ",
    [IsFpGdFreeGroupoidElm, IsInt],
function( g,n )
local r,L, ginvList;
L:=g!.list;
r:=Length(L);
if (n=-1) then
    if (r=1) then
       return Objectify(FpGdFreeGroupoidElm,
       rec(list:=-g!.list, source:=g!.target, target:=g!.source,
       parent:=g!.parent));
    else
      ginvList:=List([1..r],i->-L[r-i+1]);
      return Objectify(FpGdFreeGroupoidElm,
      rec(list:=ginvList, source:=g!.target, target:=g!.source,
      parent:=g!.parent));
    fi;
elif (n=1) then
if (r=1) then
    return Objectify(FpGdFreeGroupoidElm,
    rec(list:=g!.list, source:=g!.source, target:=g!.target,
    parent:=g!.parent));
else
    ginvList:=List([1..r],i->L[r-i+1]);
    return Objectify(FpGdFreeGroupoidElm,
```

```
    rec(list:=ginvList, source:=g!.source, target:=g!.target,
    parent:=g!.parent));
fi;
else return fail;
fi;
end );
#
#############################################################################
#0
#F  GeneratorsOfGroupoid
##
##
##
InstallOtherMethod(GeneratorsOfGroupoid, [IsFpGdFreeGroupoid],
                      function(G)
local g, gens, k, i, s, t, a, A;
g:=G!.generators;
s:=G!.sources;
t:=G!.targets;
k:=Length(g);
a:=[];
A:=[];
gens:=[];
for i in [1..k] do
    a[i]:=rec(list:=[g[i]], source:=s[i], target:=t[i], parent:=G);
    A[i]:=Objectify(FpGdFreeGroupoidElm,a[i]);
    Add(gens,A[i]);
od;
return gens;
end);
#
#############################################################################
#0
#F  FreeGroupoid
##
##
##
InstallGlobalFunction(FreeGroupoid,
function(arg)
local l,k,L,s,t,g,o,Rec,Name;
```

```
  o:=arg[1];
  L:=arg[2];
  k:=Length(L);
  l:=Length(arg);
  if l > 2 then
       Name:=arg[3];
  else
      Name:="G";
  fi;
  if Length(SSortedList(L)) <> k then
     Print ("error"," ","some generators are duplicated", "\n");
     return fail;
   else
      s:=List([1..k],i->L[i][1]);
      g:=List([1..k],i->L[i][2]);
      t:=List([1..k],i->L[i][3]);
      Rec:=rec(objects:=o, generators:=g, sources:=s, targets:=t,
      name:=Name);
     return Objectify(FpGdFreeGroupoid,Rec);
  fi;
end);
#
#############################################################################
#0
#F  GeneratorsOfGroupoid
##
##
##
InstallOtherMethod(GeneratorsOfGroupoid, [IsFpGdFpGroupoid],
                      function(G)
local g, gens, k, i, s, t, a, A;
g:=G!.generators;
s:=G!.sources;
t:=G!.targets;
k:=Length(g);
a:=[];
A:=[];
gens:=[];
for i in [1..k] do
      a[i]:=rec(list:=[g[i]], source:=s[i], target:=t[i], parent:=G);
```

```
      A[i]:=Objectify(FpGdFpGroupoidElm,a[i]);
      Add(gens,A[i]);
od;
return gens;
end);
#
#############################################################################
#0
#F  FpGroupoid
##
##
##
InstallMethod(FpGroupoid,
function(arg)
local l,k,L,s,t,g,o,r,Rec,Name;
  o:=arg[1];
  L:=arg[2];
  k:=Length(L);
  r:=arg[3];
  l:=Length(arg);
if l > 3 then
  Name:=arg[4];
else
  Name:="G";
fi;
if Length(SSortedList(L)) <> k then
     Print ("error,"," ","some generators are duplicated", "\n");
     return fail;
else
      s:=List([1..k],i->L[i][1]);
      g:=List([1..k],i->L[i][2]);
      t:=List([1..k],i->L[i][3]);
      Rec:=rec(objects:=o, generators:=g, sources:=s, targets:=t,
      relators:=r, name:=Name);
     return Objectify(FpGdFpGroupoid,Rec);
fi;
end);
#
#############################################################################
#0
```

```
#F  FpGroupoid
##
##
##
InstallOtherMethod(FpGroupoid,
function(G,L)
local U,RC,LC,gens,Gens,Gens_inv,W,sep,rels,R,x,C,Rels,r,Y,ftr,K,i,
j,b,ff,y,p,WW,find1,find2,find3,find4,RELS;
if IsList(L) then
   U:=Subgroup(G,L);
else
   U:=L;
fi;
RC:=RightCosets(G,U);
LC:=List(RC,x->x!.Representative);
gens:=GeneratorsOfGroup(G);
W:=[];
for i in [1..Length(gens)] do
 for j in [1..Length(LC)] do
    Add(W,[LC[j],gens[i],LC[j]*gens[i]]);
 od;
od;
###################### begin 1 ######################
find1:=function(x) local k,i;
k:=0;
for i in [1..Length(RC)] do
      if x in RC[i] then
            k:=i; break;
       fi;
od;
return k;
end;
###################### end 1 ########################

###################### begin 1 ######################
find2:=function(x)
local i,j,k,z;
j:=1;
for i in [1..Length(gens)] do
   j:=j+1;
```

```
   if PrintString(gens[i]*x^-1)=PrintString(Identity(G)) then
      k:=i;  break; fi;
od;
if j>Length(gens) then
i:=0;
for z in List(gens,y->y^-1) do
   i:=i+1;
   if PrintString(z*x^-1)=PrintString(Identity(G)) then
      k:=-i; break; fi;
od;
fi;
return k;
end;
###################### end 1 ########################
Gens:=List([1..Length(W)],x->[find1(W[x][1]),find2(W[x][2]),
       find1(W[x][3])]);
Gens_inv:=List(Gens,m->[m[3],-m[2],m[1]]);
###################### begin 1 ######################
sep:=function(G,w)
local g,ww,W,x,U,j,i,F,gens,f,N;
F:=FreeGroupOfFpGroup(G);
gens:=GeneratorsOfGroup(F);
g:=Concatenation(gens,List(gens,x->x^-1));
ww:=StructuralCopy(w);
W:=[];
while not Length(ww)=0 do
    for x in g do
        if Length(x^-1*ww)+1=Length(ww) then
           Add(W,x);
           ww:=x^-1*ww;
        fi;
    od;
od;
U:=List([1..Length(W)],i->[]);
Add(U[1],W[1]);
j:=1;
for i in [2..Length(w)] do
   if W[i]=W[i-1] then
         Add(U[j],W[i]);
   else
```

```
         Add(U[j+1],W[i]);
         j:=j+1;
    fi;
od;
N:=Flat(Filtered(U,x->not x=[]));
f:=GroupHomomorphismByImages(F,G,gens,GeneratorsOfGroup(G));
return List(N,x->Image(f,x));
end;
######################## end 1 ########################
rels:=RelatorsOfFpGroup(G);
R:=List(rels,r->sep(G,r));
R:=List(R,L->Reversed(L));
R:=List(R,r->List(r,x->find2(x)));

RELS:=List([1..Length(LC)*Length(R)],i->[]);

######################## begin 1 ########################
find3:=function(x)
local p;
if x in List(Gens,y->y{[1,2]}) then
   p:=Position(List(Gens,y->y{[1,2]}),x);
   return Gens[p][3];
else
   p:=Position(List(Gens_inv,y->y{[1,2]}),x);
   return Gens_inv[p][3];
fi;
end;
######################## end 1 ########################
i:=0;
for y in [1..Length(LC)] do
   for r in R do
      i:=i+1;
      for x in r do
         Add(RELS[i],[y,x,find3([y,x])]);
         y:=find3([y,x]);
      od;
   od;
od;
######################## end 1 ########################
find4:=function(x)
```

```
if x in Gens then
   return Position(Gens,x);
else
   return -Position(Gens_inv,x);
fi;
end;
###################### end 1 ######################
RELS:=List(RELS,L->List(L,x->find4(x)));

return FpGroupoid[[1..Length(LC)],Gens,RELS];;
end);
#
############################################################################
#0
#F  IdentityGroupoidElm
##
##
##
InstallOtherMethod(IdentityGroupoidElm,
        "Method for Identity Element of Fp Groupoid",
         [IsFpGdFpGroupoid,IsInt],
function(G,obj)
local a, rec_a, A;
a:=[];
rec_a:=rec(list:=[], source:=obj, target:=obj, parent:=G);
A:=Objectify(FpGdFpGroupoidElm,rec_a);
return A;
end);
#
############################################################

InstallTrueMethod(IsMultiplicativeElement,IsFpGdFpGroupoidElm);

############################################################################
#0
#F  *
##
##
##
InstallOtherMethod
```

```
(  \*,
   "for two Fp groupoid elements in 'IsFpGdFpGroupoidElm' ",
   [IsFpGdFpGroupoidElm, IsFpGdFpGroupoidElm],
function( w1 , w2 )
local w, F ;
if not (w1!.target = w2!.source) then
     Print("Words are not composable. \n");
     return fail;
else
    F:=w1!.parent;
    w:=SimplifyList(Concatenation(w1!.list,w2!.list));
    return Objectify(FpGdFpGroupoidElm,
    rec(list:=w, source:=w1!.source, target:=w2!.target, parent:=F));
fi;
end
);
#
#############################################################################
#0
#F  ^
##
##
##
 InstallOtherMethod
(  \^,
   "for one Fp groupoid element in 'IsFpGdFpGroupoidElm' ",
   [IsFpGdFpGroupoidElm, IsInt],
function( g,n )
local r,L, ginvList;
L:=g!.list;
r:=Length(L);
if (n=-1) then
     if (r=1) then
          return Objectify(FpGdFpGroupoidElm,
          rec(list:=-g!.list, source:=g!.target, target:=g!.source,
           parent:=g!.parent));
     else
          ginvList:=List([1..r],i->-L[r-i+1]);
          return Objectify(FpGdFpGroupoidElm,
         rec(list:=ginvList, source:=g!.target, target:=g!.source,
```

```
         parent:=g!.parent));
fi;
elif (n=1) then
     if (r=1) then
           return Objectify(FpGdFpGroupoidElm,
           rec(list:=g!.list, source:=g!.source, target:=g!.target,
          parent:=g!.parent));
     else
          ginvList:=List([1..r],i->L[r-i+1]);
          return Objectify(FpGdFpGroupoidElm,
          rec(list:=ginvList, source:=g!.source, target:=g!.target,
         parent:=g!.parent));
fi;
else
       return fail;
fi;
end );
#
#############################################################################
#O
#F  RelatorsOfFpGroupoid
##
##
##
InstallGlobalFunction(RelatorsOfFpGroupoid,
                      function(G)
local rels,t,gens, ListToWord,Rels, r;
rels:=G!.relators;
t:=Length(rels);
gens:=GeneratorsOfGroupoid(G);
##########################################
ListToWord:=function(L,Gens)
local Lnew, LL, K, i, j, g, k;
g:=Concatenation(Gens,List(Gens,x->x^-1));
Lnew:=Filtered(g,x->x!.list[1] in L);
LL:=[];
for k in L do
     for j in [1..Length(Lnew)] do
            if Lnew[j]!.list[1]=k then
                  Add(LL,Lnew[j]);
```

```
            fi;
     od;
od;
K:=LL[1];
for i in [2..Length(LL)] do
     K:=K*LL[i];
od;
return K;
end;
##########################################
Rels:=List([1..t],i->ListToWord(rels[i],gens));
if "parent" in NamesOfComponents(G) then
    for r in Rels do
       r!.parent:=G!.parent;
   od;
fi;
return Rels;
end);
#
#############################################################################
#0
#F  FreeGroupoidOfFpGroupoid
##
##
##
InstallGlobalFunction(FreeGroupoidOfFpGroupoid, function(G)
local objs, gens, s, t, N, R;

objs:= G!.objects;
gens:=G!.generators;
s:=G!.sources;
t:=G!.targets;
N:=G!.name;
R:=rec(objects:=objs,generators:=gens,sources:=s,targets:=t,name:=N);
return Objectify(FpGdFreeGroupoid,R);
end);
#
#############################################################################
#0
#F  TreeOfFpGroupoid
```

```
##
##
##
InstallMethod(TreeOfFpGroupoid,"Method for Tree of Fp Groupoid",
              [IsFpGdFpGroupoid, IsInt],
function(G,v)
local g,T, V, objs, gens, ftr;
T:=[];     V:=[v];
gens:=GeneratorsOfGroupoid(G);
ftr:=Filtered(gens,x->not Source(x)=Target(x));
objs:=G!.objects;

if not v in objs then
   Print("The vertex should be an object of the Groupoid. \n");
   return fail;
else
while Length(T)<Length(objs)-1 do
   for g in ftr do

      if g!.source in V then
         if not g!.target in V then
            Add(V,g!.target);
             Add(T,g);
          fi;
      fi;
      if g!.target in V then
          if not g!.source in V then
               Add(V,g!.source);
               Add(T,g^-1);
          fi;
      fi;
   od;
od;
return T;
fi;
end);
#
#############################################################################
#0
#F  IdentityFpGroupoidElm
```

```
##
##
##
InstallOtherMethod(IdentityFpGroupoidElm,
               "Method for Identity Element of Fp Groupoid",
               [IsFpGdFpGroupoid,IsInt],
function(G,obj)
local a, rec_a, A;
a:=[];
rec_a:=rec(list:=[], source:=obj, target:=obj, parent:=G);
A:=Objectify(FpGdFpGroupoidElm,rec_a);
return A;
end);
#
#############################################################################
#0
#F  =
##
##
##
InstallOtherMethod(\=,"for two words in Fp Groupoid", IsIdenticalObj,
 [IsFpGdFpGroupoidElm, IsFpGdFpGroupoidElm], 0, function(w1,w2)
return
      w1!.source = w2!.source and
      w1!.target = w2!.target and
      w1!.list = w2!.list;
end);
#
#############################################################################
#0
#F  ComponentsOfFpGroupoid
##
##
##
InstallMethod( ComponentsOfFpGroupoid,
"function for components of Fp Groupoid", [IsFpGdFpGroupoid],
function(G)
local gens,L,C,i,j,g,M,k,N,NN, W, r, Q,SubGroupoid,NeighbouredGenerators;
NeighbouredGenerators:=function(g1,g2)
if g1!.source=g2!.target or g2!.source=g1!.target or
```

```
     g1!.source=g2!.source or g2!.target=g1!.target
     then return true; else return false;
fi;
end;
gens:=GeneratorsOfGroupoid(G);
L:=Length(gens);;
C:=List([1..L],i->[gens[i]]);;
for i in [1..L-1] do
   for j in [i+1..L] do
         if NeighbouredGenerators(gens[i],gens[j]) then
              Add(C[i],gens[j]);
         fi;
   od;
od;
g:=function(R,r) if Position(R,r)=fail then return 0; else return
Position(R,r); fi; end;
M:=List([1..L],i->[]);;
for k in [1..L] do
    M[k]:=List([1..L],i->g(C[k],gens[i]));
od;
N:=List([1..L],i->[]);;
for i in [1..L] do
     for j in [1..L] do
           if not M[i][j]=0 then
               N[i][j]:=j;
          else
              N[i][j]:=[];
           fi;
     od;
od;
for i in [1..L-1] do
for j in [i+1..L] do
if not Filtered(Intersection(N[i],N[j]), k -> not k=[])=[] then
N[i]:=Union(N[i],N[j]); N[j]:=[];
fi;
od;
od;
NN:=SSortedList(N);;
W:=[];;
r:=0;;
```

```
for i in [1..Length(NN)] do
if not NN[i]=[] then r:=r+1; W[r]:=NN[i]; fi;
od;
W:=List([1..Length(W)], i-> Filtered(W[i], k -> not k=[]));
Q:=List([1..Length(W)],i->List([1..Length(W[i])],j->gens[W[i][j]]));
#####################################################
SubGroupoid:=function(G,Gens)
local Rec, Rels, objs, gens, sours, targs, rels,  r, g, N,k;
Rels:=RelatorsOfFpGroupoid(G);
objs:= SSortedList( Flat (List(Gens, g -> [g!.source,g!.target]) ));
gens:=List(Gens, g -> g!.list[1] );
sours:=List(Gens, g -> g!.source );
targs:=List(Gens, g -> g!.target );
rels:=[];
for r in Rels do
    if not Intersection(r!.list,gens)=[] then Add(rels,r!.list); fi;
od;
Rec:=rec(objects:=objs, generators:=gens, sources:=sours,
 targets:=targs, relators:=rels, name:="G" );
return
Objectify(FpGdFpGroupoid,StructuralCopy(Rec));
end;
#####################################################
return List([1..Length(Q)],i->SubGroupoid(G,Q[i]));
end);
#
#############################################################################
#0
#F  IsConnectedFpGroupoid
##
##
##
InstallMethod( IsConnectedFpGroupoid,
               "method for connectivity of Fp Groupoid",
               [IsFpGdFpGroupoid],
function(G)
if Length(ComponentsOfFpGroupoid(G))=1 then return true; else
 return false; fi;
end);
#
```

```
#############################################################################
#0
#F  RelatorToGroupWord
##
##
##
InstallMethod( RelatorToGroupWord,
               "method for connectivity of Groupoid",
               [IsFpGdFpGroupoid, IsFpGdFpGroupoidElm, IsInt],
function(G,w,v)
local T,L,s,e;
s:=w!.source;
T:=TreeOfFpGroupoid(G,v);
L:=Length(T);
if s=v then return w;
else
while not w!.source=v do
     for e in T do
          s:=w!.source;
          if e!.target=s then
              w:=e*w*e^-1;
          fi;
     od;
od;
return w;
fi;
end);
#
#############################################################################
#0
#F  VertexGroup
##
##
##InstallMethod( VertexGroup, "method for finding vertex group",
               [IsFpGdFpGroupoid, IsInt],
         function(G,v)
local T,Tinv,gens,ftr,fun,rels,W1,W2,x,RM,L1,LL1,prod,R,F,M,f,K,
H,LL,L2,h,ff,s1,s2,s12,i,j;

T:=TreeOfFpGroupoid(G,v);
```

```
Tinv:=List(T,x->x^-1);
gens:=GeneratorsOfGroupoid(G);
ftr:=Filtered(gens,x->not x in Concatenation(T,Tinv));
#####################################################
fun:=function(p)     # return path from v to p.
local A,xx,TT,y;
 A:=[];
 xx:=Filtered(T,x->Target(x)=p)[1];
 if Source(xx)=v then return xx; fi;
 TT:=StructuralCopy(T);
 while not Source(xx)=v do

    for y in TT do
        if Target(y)=p then
            Add(A,y);
            p:=Source(y);
            Unbind(TT[ Position(TT,y) ]);
            xx:=y;
        fi;
     od;
 od;
A:=List([0..Length(A)-1],i->A[Length(A)-i]);
return FpGdProduct(A);
end;
#####################################################
W1:=[];
for x in ftr do
    if Source(x)=v and Target(x)=v then Add(W1,x); fi;
    if Source(x)=v and not Target(x)=v then Add(W1,
    x*fun(Target(x))^-1); fi;
    if Target(x)=v and not Source(x)=v then Add(W1,
    fun(Source(x))*x); fi;
    if not Source(x)=v and not Target(x)=v then
       Add(W1,fun(Source(x))*x*fun(Target(x))^-1);
    fi;
od;
W2:=[];
R:=RelatorsOfFpGroupoid(G);
for x in R do
     if Source(x)=v and Target(x)=v then Add(W2,x); fi;
```

```
     if Source(x)=v and not Target(x)=v then
     Add(W2,x*fun(Target(x))^-1); fi;
     if Target(x)=v and not Source(x)=v then
     Add(W2,fun(Source(x))*x); fi;
     if not Source(x)=v and not Target(x)=v then
       Add(W2,fun(Source(x))*x*fun(Target(x))^-1);
     fi;
od;
L1:=List(W1,x->x!.list);
L2:=List(W2,x->x!.list);

F:=FreeGroup(Length(gens));
H:=FreeGroup(Length(gens));

f:=GeneratorsOfGroup(F);
h:=GeneratorsOfGroup(H);

LL:=List([1..Length(gens)],x->[x]);
LL1:=[];
for x in LL do
   if not x[1] in List(Flat(L1),y->AbsoluteValue(y)) then
   Add(LL1,x); fi;
od;

if  Length(LL1)<Length(L1)+Length(L1) then
  for x in LL do
    if not x in Concatenation(L1,LL1)  then
       Add(LL1,x);
          if Length(LL1)=Length(gens)-Length(L1) then break; fi;
    fi;
  od;
fi;

prod:=function(L) if Length(L)>1 then return FpGdProduct(L); else
return L[1]; fi; end;

#L1:=Concatenation(L1,LL1);

ff:=function(M,x) if x in M then return Position(M,x); else
return Position(List(M,x->x^-1),x); fi; end;
```

```
s1:=List(T,x->ff(gens,x));
s2:=Difference([1..Length(gens)],s1);
s12:=[];
for i in s1 do
    s12[i]:=h[i];
od;
j:=0;
for i in s2 do
    j:=j+1;
    s12[i]:=prod(List(L1[j],x->h[AbsoluteValue(x)]^SignInt(x)));
od;
K:=GroupHomomorphismByImages(F,H,f,s12);
M:= List ( L2, L-> prod( List( L , x -> h[AbsoluteValue(x)]^
SignInt(x) ) )  );
rels := Concatenation( List ( M , x-> PreImage(K,x) ) ,
List(s1,x-> f[x]) );
return F/rels;
end);
#
#############################################################################
#0
#F  Source
##
##
##
InstallOtherMethod(Source,
"source of of an arrow of a Fp Groupoid",
[IsFpGdFreeGroupoidElm],
function(f) return f!.source;
end);
#
#############################################################################
#0
#F  Target
##
##
##
InstallOtherMethod(Target,
"target of of an arrow of a Fp Groupoid",
[IsFpGdFreeGroupoidElm],
```

```
function(f) return f!.target;
end);
#
############################################################################
#0
#F  Source
##
##
##
InstallOtherMethod(Source,
"source of of a generator of a Free Groupoid",
[IsFpGdFpGroupoidElm],
function(f) return f!.source;
end);
#
############################################################################
#0
#F  Target
##
##
##
InstallOtherMethod(Target,
"target of of a generator of a Free Groupoid",
[IsFpGdFpGroupoidElm],
function(f) return f!.target;
end);
#
############################################################################
#0
#F  Length
##
##
##
InstallOtherMethod(Length,
"length of of an alement of a Groupoid",
[IsFpGdFpGroupoidElm],
function(f) return Length(f!.list);
end);
#
############################################################################
```

```
#0
#F  Length
##
##
##
InstallOtherMethod(Length,
"length of of an element of a Groupoid",
[IsFpGdFreeGroupoidElm],
function(f) return Length(f!.list);
end);
#
#############################################################################
#0
#F
##
##
##
InstallOtherMethod
(  \/, "  ",
   [IsFpGdFreeGroupoid, IsList],
   function( F , L )
     local Rels, N, G;
     Rels:=List([1..Length(L)],i->L[i]!.list);
    if L=[] then G:=Objectify(FpGdFpGroupoid,
                rec(objects:=F!.objects,
                 generators:=F!.generators,
                 sources:=F!.sources,
                 targets:=F!.targets,
                 relators:=Rels,
                 name:="G"));
    else
     N:=L[1]!.parent;
     Rels:=List([1..Length(L)],i->L[i]!.list);
     G:= Objectify(FpGdFpGroupoid,
            rec(objects:=F!.objects,
                  generators:=F!.generators,
                  sources:=F!.sources,
                  targets:=F!.targets,
                  relators:=Rels,
                  name:=N!.name));
```

```
     G!.parent:=N;
     fi;
     return G;
   end);
#
#############################################################################
#O
#F  in
##
##
##
InstallOtherMethod( \in , "for groupoid elements" , [IsObject, IsObject],
function (g,G)
local N;
if IsFpGdFreeGroupoidElm(g) or IsFpGdFpGroupoidElm(g)
   then
   N:=g!.parent;
   if N=G then
       return true;
   else
       return false;
    fi;
else
   return false;
fi;
end);
#
#############################################################################
#O
#F  in
##
##
##
InstallOtherMethod( \in , "for free groupoid elements" ,
                          [IsFpGdFpGroupoidElm , IsFpGdFpGroupoid] ,
function (g,G)
local N;
if IsFpGdFpGroupoidElm(g)  then
N:=g!.parent;
if N=G then
```

```
return true;
else
 return false;
fi;
else
return false;
fi;
end);
#
#############################################################################
#0
#F  =
##
##
##
InstallOtherMethod( \= , "for two free groupoids" ,
                           [IsFpGdFreeGroupoid, IsFpGdFreeGroupoid],
function (F1,F2)
if F1!.objects = F2!.objects and
F1!.generators = F2!.generators and
F1!.sources = F2!.sources and
F1!.targets = F2!.targets then
return true;
else
return false;
fi;
end);
#
#############################################################################
#0
#F  =
##
##
##
InstallOtherMethod( \= , "for two fp groupoids" ,
                           [IsFpGdFpGroupoid, IsFpGdFpGroupoid],
function (G1,G2)
if G1!.objects = G2!.objects and
G1!.generators = G2!.generators and
G1!.sources = G2!.sources and
```

```
G1!.targets = G2!.targets and
G1!.relators = G2!.relators then
return true;
else
return false;
fi;
end);
#
#############################################################################
#0
#F  FundamentalGroupoidOfRegularCWComplex
##
##
##
InstallMethod( FundamentalGroupoidOfRegularCWComplex,
               "method for finding fundamental groupoid",
               [IsHapRegularCWComplex, IsList],
function(Y,V)
local  st,F,critical_cells,vf,kappa1,u,VF,2bounds,Deform,gens,RELS,
Rels,x,y,P,G,c_cells,vfield,verCrt,edgCrt,facCrt,edgTar,facTar,verSor,
edgSor,test,z;
####################################
#
critical_cells:=function(Y,V)
local cc,vf,vf1,vf2,0c,0c0,1c,1c1,x,y,CC,cc0,cc1,cc2;
Y!.criticalCells:=fail;
Y!.vectorField:=fail;
cc:=CriticalCells(Y);
vf:=Y!.vectorField;
vf2:=SortedList(Filtered(vf[2],x->IsInt(x)));
0c:=Union(List(Y!.boundaries[2],x->x{[2,3]}));
0c0:=Difference(0c,V);
1c:=Union(List(Y!.boundaries[3],x->x{[2,3,4]}));
1c1:=Difference(1c,vf2);
vf1:=[];
for x in 0c0 do
    for y in 1c1 do
         if x in Y!.boundaries[2][y]{[2,3]} then
         vf1[y]:=x;
         break;
```

```
         fi;
    od;
    1c1:=1c1{Difference([1..Length(1c1)],[Position(1c1,y)])};
od;
Y!.vectorField:=[vf1,vf[2]];
cc0:=List(V,x->[0,x]);
cc1:=List(Difference(1c,
                  Union(List(Filtered(vf1,x->IsInt(x)),y->
                  Position(vf1,y)) ,Filtered(vf2,x->IsInt(x))))
     ,z->[1,z]);
cc2:=Filtered(cc,x->x[1]=2);
CC:=Union(cc0,cc1,cc2);
Y!.criticalCells:=Filtered(CC,x->not x=[]);
return true;
end;
#
#####################################

critical_cells(Y,V);;
vfield:=Y!.vectorField;
c_cells:=Y!.criticalCells;

verCrt:=List(Filtered(c_cells,x->x[1]=0) , y->y[2]);;
edgCrt:=List(Filtered(c_cells,x->x[1]=1) , y->y[2]);;
facCrt:=List(Filtered(c_cells,x->x[1]=2) , y->y[2]);;
edgTar:=List(Filtered(vfield[1],x->IsInt(x)),y->Position(vfield[1],y));;
facTar:=List(Filtered(vfield[2],x->IsInt(x)),y->Position(vfield[2],y));;
verSor:=List(edgTar,x->[vfield[1][x],x]);;
edgSor:=List(facTar,x->[vfield[2][x],x]);;
vf:=[[verCrt,edgCrt,facCrt],[edgTar,facTar],[verSor,edgSor]];;

#####################################
#
kappa1:=function( 1c )
local 1c_targ,1c_sour,1c_Targ,1c_Sour,bool,ee,path;
1c_targ:=Y!.boundaries[2][1c][2];;
1c_sour:=Y!.boundaries[2][1c][3];;
path:=[[],[]];;
if not 1c_targ in vf[1][1] then
   bool:=true;
```

```
while bool=true do
    ee:=Filtered(vf[3][1],x->x[1]=1c_targ)[1][2];
    1c_Targ:=Y!.boundaries[2][ee][2];
     if 1c_Targ=1c_targ then
            1c_targ:=Y!.boundaries[2][ee][3];
             Add(path[1],ee);
    else
            1c_targ:=1c_Targ;
              Add(path[1],ee);
    fi;
    if 1c_targ in vf[1][1] then
          bool:=false;
     fi;
od;
fi;
if not 1c_sour in vf[1][1] then
bool:=true;
while bool=true do
   ee:=Filtered(vf[3][1],x->x[1]=1c_sour)[1][2];
   1c_Sour:=Y!.boundaries[2][ee][3];
   if 1c_Sour=1c_sour then
         1c_sour:=Y!.boundaries[2][ee][2];
          Add(path[2],ee);
   else
         1c_sour:=1c_Sour;
         Add(path[2],ee);
   fi;
   if 1c_sour in vf[1][1] then bool:=false; fi;
od;
fi;
if not path[2]=[] then
    path[2]:=List([0..Length(path[2])-1],x->
                 path[2][Length(path[2])-x]);
fi;
if not 1c in path[2] and not 1c in path[1] then
    Add(path[2],1c);
fi;
    path:=Concatenation( path[2] , path[1] );
return [1c_sour,1c_targ,path];
end;
```

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

Y!.path:=function(1cell) return kappa1(1cell)[3]; end;

st:=List(vf[1][2],x->kappa1(x));;
F:=FreeGroupoid(V,List([1..Length(vf[1][2])],x->[st[x][1],x,st[x][2]]));
gens:=GeneratorsOfGroupoid(F);

VF:=Y!.vectorField;;
u:=[[],[]];
for x in SortedList(VF[1]) do
      u[1][x]:=Position(VF[1],x);
od;
for x in SortedList(VF[2]) do
      u[2][x]:=Position(VF[2],x);
od;
Y!.inverseVectorField:=u;;
#
#####################################
#
P:=SSortedList(List(Y!.orientation[1],Sum));;
if P=[0] then Y!.homotopyOrientation:=Y!.orientation{[1,2,3]};
else
P:=TruncatedRegularCWComplex(Y,2);;
P!.orientation:=fail;
OrientRegularCWComplex(P);
Y!.homotopyOrientation:=P!.orientation;
fi;
Unbind(P);
#
#####################################
2bounds:=List(vf[1][3],x->[Y!.boundaries[3][x],
Y!.homotopyOrientation[3][x]]);;
Apply(2bounds,x->[x[1]{[2..Length(x[1])]},x[2]]);
Apply(2bounds,x->List([1..Length(x[1])],i->x[1][i]*x[2][i]));
#####################################
#

Deform:=function(n,kk)
```

```
local sgnn,x,f,k,sgnk,cnt,bnd,def,sn,tog,def1,def2,DCSrec,dim;
dim:=Dimension(Y);
DCSrec:=List([1..dim+1],i->[]);;
k:=AbsInt(kk);
sgnk:=SignInt(kk);
if [n,k] in Y!.criticalCells then
   return [kk];
fi;
if n>0 then
    if IsBound(Y!.vectorField[n][k]) then
        return [];
    fi;
fi;
if IsBound(DCSrec[n+1][k]) then
    if sgnk=1 then
         return DCSrec[n+1][k];
    else
         return -DCSrec[n+1][k];
     fi;
fi;
f:=Y!.inverseVectorField[n+1][k];
bnd:=Y!.boundaries[n+2][f];
sn:=Y!.orientation[n+2][f];
def:=[]; def1:=[];def2:=[];
for x in [2..Length(bnd)] do
       if not bnd[x]=k then
                Add(def1,sn[x-1]*bnd[x]);
       else sgnn:=sn[x-1];
                break;
        fi;
od;
cnt:=x+1;
for x in [cnt..Length(bnd)] do
Add(def2,sn[x-1]*bnd[x]);
od;
if sgnn=1 then def:=-Concatenation(def1,def2);
else
    def:=Concatenation(def2,def1);
fi;
Apply(def,x->Deform(n,x));
```

```
def:=Flat(def);
Apply(def,x->[x,0]);
def:=AlgebraicReduction(def);
Apply(def,x->x[1]);
DCSrec[n+1][k]:=def;
if sgnk=1 then return
    def;
else return
  -def;
fi;
end;
#
#####################################

Apply(2bounds,x->List(x,a->Deform(1,a)));
2bounds:=Filtered(2bounds,x->not x=[]);
2bounds:=List(2bounds,L->Filtered(L,x->not x=[]));
#2bounds:=Filtered(List(2bounds,L->Flat(L)),x->not x=[]);

RELS:=List(2bounds,L->List(L,x->[]));
for x in 2bounds do
    for z in x do
        for y in z do

            Add(RELS[ Position(2bounds,x) ][ Position(x,z) ],
            gens[ Position(vf[1][2],AbsInt(y)) ]^(SignInt(y)));

         od;
    od;
od;

RELS:=List(RELS,L->List([0..Length(L)-1],i->L[Length(L)-i]));
#
#####################################

test:=function(L)
local k,v,BOOL,i;
k:=Length(L);
v:=Target(L[k]);
BOOL:=true;
```

```
for i in Reversed([1..k-1]) do
     if Source(L[i])=v then
         v:=Target(L[i]);
     else
       BOOL:=false;
       break;
      fi;
od;
return BOOL;
end;

RELS:=List(RELS,L->Flat(List(L,N->Filtered(List(
PermutationsList([1..Length(N)]),M->List(M,i->N[i])),x->test(x))[1])));

if RELS=List(RELS,x->[]) then
   Rels:=[];
else
   Rels:=Filtered(RELS,L->not L=[]);
   Rels:=List(Rels,L->FpGdProduct(L));
fi;

G:=F/Rels;
return G;
end);
#
############################################################################
#0
#F  UnionOfGroupoids
##
##
##
InstallGlobalFunction( UnionOfGroupoids,
"method for defining union of two groupoids",
function(G1,G2)
local f,o,g1,g2,g12,r1,r2,R1,R2,rels;
################
f:=function(L,LL,n,m)
local K,J,x;
J:=[n..n+m-1];
K:=[];
```

```
for x in L do
   if x>0 then Add(K,J[Position(LL,x)]); else Add(K,(-1)*
       J[Position(LL,AbsoluteValue(x))]);
    fi;
od;
return K;
end;
################
o:=Union(G1!.objects,G2!.objects);
g1:=GeneratorsOfGroupoid(G1);
g2:=GeneratorsOfGroupoid(G2);
g12:=Concatenation(g1,g2);
if IsFpGdFpGroupoid(G1) and IsFpGdFpGroupoid(G2) then
    r1:=G1!.relators;
    R1:=List(r1, x-> f(x,Concatenation(List(g1,x->x!.list)),1,Length(g1)));
    r2:=G2!.relators;
    R2:=List(r2, x-> f(x,Concatenation(List(g2,x->x!.list)),
    Length(g1)+1,Length(g2)));
    rels:=Concatenation(R1,R2);
    return FpGroupoid(o,List([1..Length(g12)],x->
          [Source(g12[x]),x,Target(g12[x])]),rels);
else
     return FreeGroupoid(o,List([1..Length(g12)],x->
     [Source(g12[x]),x,Target(g12[x])]));
fi;
end);
#
############################################################################
#0
#F  FpGroupToFpGroupoid
##
##
##
InstallGlobalFunction( FpGroupToFpGroupoid,
               "method for changing Fp Group to be Fp Groupoid",
function(arg)
local G,v,F, gensF, rels, L, Rels, Objs, Gens, O, Rel2List;
if Length(arg)>1 then G:=arg[1]; v:=arg[2]; else G:=arg[1]; fi;
F:=FreeGroupOfFpGroup(G);
gensF:=GeneratorsOfGroup(F);
```

```
rels:=RelatorsOfFpGroup(G);
L:=Concatenation(List([1..Length(gensF)],i->[gensF[i],gensF[i]^-1]));
########################
Rel2List:=function(R,L)
local l1,l2,K,C,i,j,k,A;
l1:=Length(R);
l2:=Length(L);
K:=Concatenation(List([1..l2/2],i->[-i,i]));
C:=[];
k:=1;
A:=[];
A[1]:=StructuralCopy(R);
  for i in [1..l1] do
     for j in[1..l2] do
         if Length(L[j]*A[k])=Length(A[k])-1
              then  k:=k+1;
                A[k]:=L[j]*A[k-1];
                Add(C,K[j]);
         fi;
     od;
  od;
return C;
end;
######################
Rels:=List([1..Length(rels)],i->Rel2List(rels[i],L));
Gens:=[1..Length(gensF)];
if Length(arg)>1 then
    return FpGroupoid([v],List(Gens,x->[v,x,v]),Rels);
else
   return FpGroupoid([1],List(Gens,x->[1,x,1]),Rels);
fi;
end);
#
#############################################################################
#0
#F  GroupoidMorphismByImages
##
##
##
InstallGlobalFunction( GroupoidMorphismByImages,
```

```
 "method for defining groupoid morphism",
function(arg)
local G,H,Smo,Tmo,Sma,Tma,gens,MapObj,MapArr,L,K,Word2List;

if Length(arg)>3 then
  G:=arg[1];
  H:=arg[2];
  Smo:=arg[3][1];
  Tmo:=arg[3][2];
  Sma:=arg[4][1];
  Tma:=arg[4][2];
else
  G:=arg[1];
  H:=arg[2];
  Sma:=arg[3][1];
  Tma:=arg[3][2];
fi;
gens:=GeneratorsOfGroupoid(G);
####################
MapObj:=function(x)
return Tma[Position(Sma,x)];
end;
####################
L:=Sma;
Append(L, List (L,x->x^-1) );
K:=Tma;
Append(K, List (K,x->x^-1) );
####################
Word2List:=function(w)
 local i,j, LfromW;
LfromW:=[];
j:=0;
for i in w!.list do
   j:=j+1;
   if i>0 then
       LfromW[j]:=gens[i];
   else
       LfromW[j]:=gens[-i]^-1;
   fi;
od;
```

```
return LfromW;
end;
###################
MapArr:=function(w)
local Lw,k,x,i;
Lw:=Word2List(w);
k:=Length(Lw);
if k=1 then
    x:=Lw[1];
    return K[Position(L,x)];
else
    x:=K[Position(L,Lw[1])];
for i in [2..k] do
     x:=x*K[ Position(L,Lw[i]) ];
od;
return x;
fi;
end;
if Length(arg)>3 then
    return Objectify(GroupoidMorphism,rec(source:=G,target:=H,
    mappingObj:=MapObj,mappingArr:=MapArr,name:="N"));
else
    return Objectify(GroupoidMorphism,rec(source:=G,target:=H,
    mappingArr:=MapArr,name:="N"));
fi;
end);
#
#############################################################################
#0
#F  Source
##
##
##
InstallOtherMethod(Source,
"source of of a groupoid morphim",
[IsGroupoidMorphism],
function(f) return f!.source;
end);
#
#############################################################################
```

```
#0
#F  Target
##
##
##
InstallOtherMethod(Target,
"target of of a groupoid morphim",
[IsGroupoidMorphism],
function(f) return f!.target;
end);
#
#############################################################################
#0
#F   ImageOfArrow
##
##
##
InstallGlobalFunction(ImageOfArrow,
"image of of an arrow under a groupoid morphism",
function(arg)
local a, f, x;
f:=arg[1];
x:=arg[2];
a:=f!.mappingArr;
return a(x);
end);
#
#############################################################################
#0
#F  FundamentalGroupoidOfRegularCWMap
##
##
##
InstallGlobalFunction(FundamentalGroupoidOfRegularCWMap,
              "method for defining the groupoid functor",
function(map,V)
local S,T,f,GS,gensS,GT,gensT,VV,L,cc,1c,path,Deform,PATHS,x,i,
prod,CC,1C,v,f1c,B,y,f_path,dom,codom,g,rev;

S:=Source(map);
```

```
T:=Target(map);
f:=map!.mapping;
VV:=List(V , y -> f(0,y) );
GS:=FundamentalGroupoidOfRegularCWComplex(S,V);
GT:=FundamentalGroupoidOfRegularCWComplex(T,VV);

gensS:=GeneratorsOfGroupoid(GS);
gensT:=GeneratorsOfGroupoid(GT);

cc:=S!.criticalCells;
1c:=List(Filtered(cc,x->x[1]=1),y->y[2]);

path:=List( 1c , x-> S!.path(x) );

f_path:=List(path , x-> List(x,y-> f(1,y)) );


#####################################
#
Deform:=function(Y,n,kk)
local sgnn,x,f,k,sgnk,cnt,bnd,def,sn,tog,def1,def2,DCSrec,dim,rev;
dim:=Dimension(Y);
DCSrec:=List([1..dim+1],i->[]);;
k:=AbsInt(kk);
sgnk:=SignInt(kk);
if [n,k] in Y!.criticalCells then
   return [kk];
fi;
if n>0 then
     if IsBound(Y!.vectorField[n][k]) then
         return [];
     fi;
fi;
if IsBound(DCSrec[n+1][k]) then
     if sgnk=1 then
         return DCSrec[n+1][k];
     else return -DCSrec[n+1][k];
     fi;
fi;
f:=Y!.inverseVectorField[n+1][k];
```

```
bnd:=Y!.boundaries[n+2][f];
sn:=Y!.orientation[n+2][f];
def:=[]; def1:=[];def2:=[];
for x in [2..Length(bnd)] do
    if not bnd[x]=k then
       Add(def1,sn[x-1]*bnd[x]);
    else
      sgnn:=sn[x-1]; break;
    fi;
od;
cnt:=x+1;
for x in [cnt..Length(bnd)] do
   Add(def2,sn[x-1]*bnd[x]);
od;
if sgnn=1 then
   def:=-Concatenation(def1,def2);
else
   def:=Concatenation(def2,def1);
fi;
Apply(def,x->Deform(Y,n,x));
def:=Flat(def);
Apply(def,x->[x,0]);
def:=AlgebraicReduction(def);
Apply(def,x->x[1]);
DCSrec[n+1][k]:=def;
if sgnk=1 then
   return def;
else
   return -def;
fi;
end;
#
#####################################

Apply(f_path,x-> List(x, a->Deform(T,1,a)));

rev:=function(L) return List([0..Length(L)-1],i->L[Length(L)-i]); end;

Apply(f_path,rev);
Apply(f_path,Flat);
```

```
f1c:=List(Filtered(T!.criticalCells,x->x[1]=1),y->y[2]);
B:= List(f_path, x-> List(x, y-> gensT[ Position(f1c,AbsInt(y)) ]^
(SignInt(y)) ) );

dom:=List(gensS,x->Source(x));
codom:=List(dom,x->f(0,x));

g:=function(L,0c)
local k,i,LL;
k:=Length(L);
LL:=[];
for i in [0..k-1] do
    if Source(L[k-i])=0c
       then
       LL[k-i]:=L[k-i];
       0c:=Target(L[k-i]);
    else
       LL[k-i]:=L[k-i]^-1;
       0c:=Target(L[k-i]);
    fi;
od;
return LL;
end;

B:=List([1..Length(B)],i->g(B[i],codom[i]));

prod:=function(L) if Length(L)>1 then return FpGdProduct(L);
else return L[1]; fi; end;
PATHS:=List(B,x->prod(x));

return GroupoidMorphismByImages(GS,GT,[gensS,PATHS]);
end);
#
#############################################################################
#0
#F  PushoutOfFpGroupoids
##
##
##
```

```
InstallMethod(PushoutOfFpGroupoids, [IsGroupoidMorphism,IsGroupoidMorphism],
function(f,g)
local P, rels, FhomP, FFhomP, GhomP, GGhomP, FGhomP, F, FF, G, GG,
      FG, U, W, gensF, gensFF, gensG, gensGG, gensP, x, gg,tot,
      UF,UG,mappingOBJ;
F:=Target(f);
FF:=FreeGroupoidOfFpGroupoid(F);
G:=Target(g);
GG:=FreeGroupoidOfFpGroupoid(G);
FG:=Source(f);
gensFF:=GeneratorsOfGroupoid(FF);
gensF:=GeneratorsOfGroupoid(F);
gensGG:=GeneratorsOfGroupoid(GG);
gensG:=GeneratorsOfGroupoid(G);
U:=FG!.objects;
#####################
mappingOBJ:=function(f,x)
local S,T,gensS,gensT,gS;
S:=Source(f);
T:=Target(f);
gensS:=GeneratorsOfGroupoid(S);
gensT:=GeneratorsOfGroupoid(T);
gS:=Filtered(gensS,y->Source(y)=x)[1];
return Source(f!.mappingArr(gS));
end;
#####################
UF:=List(U,x->mappingOBJ(f,x));
UG:=List(U,x->mappingOBJ(g,x));
W:=[];
tot:=[1..Length(gensF)+Length(gensG)];
Append(W,List([1..Length(gensF)], x -> [U[Position(UF,gensF[x]!.source)],
x,U[Position(UF,gensF[x]!.target)]]));
Append(W,List([1..Length(gensG)],x->[U[Position(UG,gensG[x]!.source)],
x+Length(gensF),U[Position(UG,gensG[x]!.target)]]));
P:=FreeGroupoid( U , W );
gensP:=GeneratorsOfGroupoid(P);
FFhomP:=GroupoidMorphismByImages(FF,P,[gensFF,
gensP{[1..Length(FF!.generators)]}]);
FhomP:=GroupoidMorphismByImages(F,P,[gensF,
gensP{[1..Length(FF!.generators)]}]);
```

```
GGhomP:=GroupoidMorphismByImages(GG,P,[gensGG,
gensP{[1+Length(FF!.generators)..Length(gensP)]}]);
GhomP:=GroupoidMorphismByImages(G,P,[gensG,
gensP{[1+Length(FF!.generators)..Length(gensP)]}]);
gg:=GroupoidMorphismByImages(FG,G,
[GeneratorsOfGroupoid(FG),List(GeneratorsOfGroupoid(Source(g)),
x->ImageOfArrow(g,x)) ]);
rels:=[];
Append(rels, List(RelatorsOfFpGroupoid(F),x->ImageOfArrow(FFhomP,x)) );
Append(rels, List(RelatorsOfFpGroupoid(G),x->ImageOfArrow(GGhomP,x)) );
for x in GeneratorsOfGroupoid(FG) do
     Add(rels,(ImageOfArrow(GhomP, ImageOfArrow(gg,x)))*
    (ImageOfArrow(FhomP, ImageOfArrow(f,x)))^-1);
od;
return P/rels;
end);
#
############################################################################
#0
#F  TestFpGd
##
##
##
InstallGlobalFunction(TestFpGd,
function()
local 2simplices,K,Y,G,V,V1,rels,Bool,Rels,L,g,resl,gens;

2simplices:=[[1,2,4],[2,4,8],[2,3,8],[3,8,9],[1,3,9],[1,4,9],[4,5,8],
[5,6,8],[6,8,9],[6,7,9],[4,7,9],[4,5,7],[1,5,6],[1,2,6],[2,6,7],[2,3,7],
[3,5,7],[1,3,5]];;

K:=SimplicialComplex(2simplices);
Y:=RegularCWComplex(K);

G:=FundamentalGroupoidOfRegularCWComplex(Y,[1,8]);

Rels:=RelatorsOfFpGroupoid(G);

gens:=GeneratorsOfGroupoid(G);
```

```
L:=List(gens,x->[Source(x),Target(x)]);

V1:=VertexGroup(G,1);;

V:=SimplifiedFpGroup(V1);;

g:=GeneratorsOfGroup(FreeGroupOfFpGroup(V));

rels:=RelatorsOfFpGroup(V);

if
L = [ [ 1, 1 ], [ 1, 8 ], [ 1, 1 ] ]
and
Rels = [gens[2]^-1*gens[1]*gens[3]*gens[1]^-1*gens[3]^-1*gens[2]]
  and
  rels = [g[2]^-1*g[1]^-1*g[2]*g[1]]
then
  Print("\n\n FpGd seems to be working fine. \n");
else
  Print("\n\n There are some problems with FpGd. \n");
fi;
end);
```

C.2 Mapper

```
#############################################################################
#0
#F  cluster
##
##
##
InstallGlobalFunction( cluster, function(S,f,r)
local clusters, Y, P, C, b, i, j;
if Length(S)=0 then
  return S;
fi;
b:=[];
b[1]:=List(S,x->[1,0]);
b[2]:=[];
```

```
for i in [1..Length(S)] do
    for j in [i+1..Length(S)] do
            if f(S[i],S[j])<=r then
                 Add(b[2],[2,i,j]);
            fi;
   od;
od;
if Length(b[2])>0 then b[3]:=[]; fi;
Y:=RegularCWComplex(b);;
P:=PiZero(Y);
C:=Classify([1..Length(S)],P[2]);
clusters:=List(C,x->S{x});
return clusters;
end);
#
#############################################################################
#0
#F  cluster_alte
##
##
##
InstallGlobalFunction( cluster_alte, function(S,dS,epsilon)
local SS,i,kk,l,t,K,k,ll,G,HausdorffDistance;
l:=Length(S);
if IsList(S[1]) then
  ll:=Length(S[1]);
fi;
##################
G:=function(arg,F,d,epsilon)
local I,n,j,i,l;
l:=Length(arg);
if IsList(arg[1])=false then
  I:=List([1..l],i->[arg[i]]);
else
  I:=arg;
fi;
n:=0;
for j in [1..l-1] do
        for i in [j+1..l] do
           if F(I[j],I[i],d)<>fail and
```

```
          F(I[j],I[i],d)<=epsilon then
              I[j]:=Concatenation(I[j],I[i]);
              I[i]:=[];
           fi;
       od;
od;
return Filtered(I,x->not x=[]);
end;
#################
HausdorffDistance:=function(N,M,d)
local n,m,x,y,N1,M1;
if (N=[] or M=[]) then return fail;
else
if Length(N)<Length(M) then
  N1:=N;
  M1:=M;
else
  N1:=M;
  M1:=N;
fi;
n:=Length(N1);  m:=Length(M1);
if m=1 then
    return AbsoluteValue(N1[1]-M1[1]);
elif m=1 and Length(N)>1 then
    return d(N1,M1);
else
     return Minimum(Flat(List([1..n],j->
           List([1..m],i->d(N1[j],M1[i])))));
  fi;
fi;
end;
#################
SS:=[];
SS[1]:=S;
SS[2]:=G(SS[1],HausdorffDistance,dS,epsilon);
i:=2;
while Length(SS[i-1])>Length(SS[i]) do
  i:=i+1;
  SS[i]:=G(SS[i-1],HausdorffDistance,dS,epsilon);
od;
```

```
if IsList(S[1])=false then
  return SS[i];
else
  k:=SS[i];
  kk:=Length(SS[i]);
  t:=List([1..kk],j->Length(k[j]));
  K:=List([1..kk],r->List([0..t[r]/ll-1],j->List([1+j*ll..(1+j)*ll],
  i->k[r][i])));
  return K;
fi;
end);
#
############################################################
#
FloatSpectorm_1:=function(arg)
local M,n,k,fun,K,a,b,N,A;
M:=arg[1];
n:=arg[2];
k:=Length(M);
####################
  fun:=function(N)
    local v,i,L,S,M,w;
      v:=List([1..k],x->1);
      for i in [1..n] do
        S:=List(N*v,x->x^2);
        M:=1/Sum(S)^0.5;
        v:=M*N*v;
      od;
      w:=Sum(List(v,x->x^2))^0.5;
      L:=[(v*N*v)/w,v/(v[Length(v)])];
  return L;
end;
####################
N:=M;
a:=fun(N);
K:=[[a[1]],[a[2]]];
for i in [2..k] do
  a:=[K[1][Length(K[1])],K[2][Length(K[2])]];
  A:=Sum(List(a[2],x->x^2))^0.5;
  b:=List(a[2]/A,x->[x]);
```

```
  N:=N-(a[1]*( b*TransposedMat(b) ));
  a:=fun(N);
  Add(K[1],a[1]);
  Add(K[2],a[2]);
od;

return K;
end;
#
############################################################
#
FloatSpectorm_2:=function(arg)
Jacobi_Eigen:=function(a,it_max)
local M,m,n,v,d,bw,zw,it_num,rot_num_U,i,j,thresh,p,q,
       gapq,termp,termq,theta,t,c,s,tau,h,g;
M:=arg[1];
m:=arg[2];
n:=Length(a);
v:=(List([1..n],i->List([1..n],j->1)))^0;
d:=List([1..n],i->a[i][i]);
bw:=List(d,x->x);
zw:=List(d,x->0);
it_num:=0;
rot_num:=0;
U:=List([1..n],i->List([1..n],j->0.0));
for i in [1..n] do
    for j in [i..n] do
        U[i][j]:=1.0*a[i][j]^2;
    od;
od;
while it_num < it_max do
     it_num := it_num + 1;
     thresh := Sqrt(Sum(List(U,x->Sum(x))))/(4.0 * n);
     if thresh = 0.0 then
      break;
     fi;

     for p in [1 .. n] do
        for q in [p + 1 .. n] do
            gapq := 10.0 * AbsoluteValue ( a[p][q] );
```

```
termp := gapq + AbsoluteValue ( d[p] );
termq := gapq + AbsoluteValue ( d[q] );
if (4 < it_num and
   termp = AbsoluteValue( d[p] ) and
   termq = AbsoluteValue( d[q] ) )
   then
   a[p][q] := 0.0;
elif thresh <= AbsoluteValue( 1.0*a[p][q] ) then
   h := d[q] - d[p];
   term := AbsoluteValue( h ) + gapq;
   if term = AbsoluteValue( 1.0*h ) then
        t := a[p][q] / h;
   else
        theta := 0.5 * h / a[p][q];
        t := 1.0 / ( AbsoluteValue( theta ) +
                        ( 1.0 + theta * theta )^0.5 );
        if theta < 0.0 then
             t := - t;
        fi;
   fi;

   c := 1.0 / ( 1.0 + t * t )^0.5;
   s := t * c;
   tau := s / ( 1.0 + c );
   h := t * a[p][q];

   zw[p] := zw[p] - h;
   zw[q] := zw[q] + h;
   d[p] := d[p] - h;
   d[q] := d[q] + h;
   a[p][q] := 0.0;

   for j in [1..p - 1] do
       g := a[j][p];
       h := a[j][q];
       a[j][p] := g - s * ( h + g * tau );
       a[j][q] := h + s * ( g - h * tau );
   od;

   for j in [p + 1..q - 1] do
```

```
                    g := a[p][j];
                    h := a[j][q];
                    a[p][j] := g - s * ( h + g * tau );
                    a[j][q] := h + s * ( g - h * tau );
                od;

                for j in [q + 1..n]  do
                    g := a[p][j];
                    h := a[q][j];
                    a[p][j] := g - s * ( h + g * tau );
                    a[q][j] := h + s * ( g - h * tau );
                od;

                for j in [1..n] do
                    g := v[j][p];
                    h := v[j][q];
                    v[j][p] := g - s * ( h + g * tau );
                    v[j][q] := h + s * ( g - h * tau );
                od;

                rot_num := rot_num + 1;
            fi;
        od;
    od;
    bw := bw + zw;
    d := bw;
    zw := 0.0*zw;
od;
return [d,v];
end;
end;
#
#############################################################################
#0
#F  FloatSpectrum
##
##
##
InstallGlobalFunction( FloatSpectrum,function(arg)
local k;
```

```
k:=Length(arg);
if k=2 then return FloatSpectrum_1(arg); fi;
if k=3 then return FloatSpectrum_2(arg); fi;
end);
#
#############################################################################
#0
#F  Mapper
##
##
##
InstallGlobalFunction( Mapper,
function(S,f,n,dS,dZ,epsilon_S,epsilon_Z,Clusters)
local k,l,c,S0,L,m,M,h, P, s, i, Preimagesf, clusters, W, Mapper,
IdentifyingSetOfUniformPoints, R, e;
k:=Length(S);
l:= Length(TransposedMat(S));
c:=CenterOfGravity(S);
S0:=List(S,x->x-c);
##################
IdentifyingSetOfUniformPoints:=function(S0,n)
local k,CM,v,w,PS0,Lps0,M,a,h,A;
k:=Length(S0);
CM:=VectorsToCovarianceMatrix(S0);
v:=SpectrumFloat(CM)[1];
w:=v/EuclideanMetric(v,List(v,i->0));
PS0:=List(S0,x->OrthogonalProjection(x,w));
Lps0:=List(PS0,x->EuclideanMetric(x,List(PS0[1],i->0)));
M:=Maximum(Lps0);
a:=Int(Round(M));
h:=2*a/n;
A:=List([0..n],i->-a+i*h);
return [PS0,List(A,x->x*w)];
end;
##################
R:=IdentifyingSetOfUniformPoints(S,n);
if f=OrthogonalProjection  then
  P:=R[2];
  h:=dZ(P[1],P[2]);
else
```

```
  L:=List(S0,x->f(x));
  m:=Minimum(L);
  M:=Maximum(L);
  h:=(M-m)/n;
  P:=List([0..n],i->m+i*h);
fi;
#################
Preimagesf:=List(P,x->[]);
e:=0;
for s in S do
   e:=e+1;
   for i in [1..Length(P)] do
      if f=OrthogonalProjection  then
        if dZ(R[1][e],P[i]) < (h/2)*(1+epsilon_Z) then
           Add(Preimagesf[i],s);
        fi;
      else
        if dZ(f(s),P[i]) < (h/2)*(1+epsilon_Z) then
           Add(Preimagesf[i],s);
        fi;
      fi;
   od;
od;
Preimagesf:=Filtered(Preimagesf,x->not x=[]);
clusters:=[];
for i in [1..Length(Preimagesf)] do
     clusters[i]:=Clusters(Preimagesf[i],dS,epsilon_S);
od;
W := Concatenation( clusters );
    Mapper := NerveOfCover( W, 20 );
    Mapper!.clustersizes := List( W, Size );
    return Mapper;
end);
#
#########################################################################
#0
#F  ReadAMCfileAsPatternMatrex
##
##
##
```

```
InstallGlobalFunction( ReadAMCfileAsPatternMatrex,
function(dir)
local instr,f,A,J,i,j,K,T,s,d,k,TexToList,S;
instr:=InputTextFile(dir);
f:=true;
A:=[]; J:=[]; i:=0; j:=0;
while not f=fail do
    j:=j+1;
    f:=ReadLine(instr);
    Add(A,f);
    if f=Concatenation(String(1+i),"\n") then
      i:=i+1; Add(J,j);
    fi;
od;
d:=function(f) return f{[1..Length(f)-1]}; end;
A:=List([4..Length(A)-1],i->d(A[i]));;
k:=Int(Length(A)/30);
A:=List([0..k-1],i-> List([1+i*30..(1+i)*30],j->A[j]));;
############################################################
TexToList:=function(x)
local l,i, j, I, Y;
l:=Length(x);
I:=[];
for i in [1..l] do
if x{[i]}=" " or x{[i]}="\r"  then Add(I,i); fi;
od;
Y:=[];
for j in [1..Length(I)-1] do
    Add(Y,Float(x{[I[j]+1..I[j+1]-1]}));
od;
return Y;
end;
############################################################
S:= List([1..Length(A)],j-> Flat(List([1..Length(A[j])] ,
i->TexToList(A[j][i]))));;
for i in [1..Length(S)] do
    S[i]:=Concatenation([i*1.0],S[i]);
od;
return S;
end);
```

```
######################      THE END     ##########################
```

Bibliography

[1] A. Adem. Recent devolopments in the cohomology of finite groups. *Notices AMS*, 44:806–812, 1997. Available online at http://www.ams.org/notices/199707/199707-toc.html. (Cited on page 1.)

[2] C. Charn Aggarwal. *A general theory of polyhedral sets*. CRC Press, Taylor and Francis Group, 2015. (Cited on page 71.)

[3] N. Alokbi. *FpGd – Finitely Presented Groupoid (*GAP *package)*, 2019. https://github.com/nalokbi/FpGd. (Cited on pages 24 and 111.)

[4] N. Alokbi and G. Ellis. Distributed computation of low-dimensional cup products. *Homology, Homotopy and Applications*, 20(2):41–59, 2018. (Cited on pages 3 and 85.)

[5] J. Arnold, L. M. Nicolau, and G. Carlsson. Topology based data analysis identifies a subgroup of breast cancers with a unique mutational profile and excellent survival. *Proceedings of the National Academy of Sciences of the United States of America PNAS*, 108(17):7265–7270, 2011. (Cited on page 68.)

[6] W. F. Basener. *"Module Presentations for use with Topology and Its Applications." Topology and Its Applications. Ed. William F. Basener.* National Science Foundation, 17 June . Web. 17 Aug. 2004. (Cited on page 3.)

[7] M. Bishop. The GAP package CRIME, available from http://www.gap-system.org/Packages/crime.html). (Cited on page 1.)

[8] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993),https://www.sciencedirect.com/science/article/pii/S074771719690125X. (Cited on page 1.)

[9] C. Bregler and S. M. Omohundro. Nonlinear manifold learning for visual speech recognition. *Proc. 5th Internat. Conf. Comput. Vision*, 4(3):494–499, 1995. (Cited on page 3.)

[10] P. Brendel, P. Dłotko, G. Ellis, M. Juda, and M. Mrozek. Computing fundamental groups from point clouds. *Appl. Algebra Eng. Commun. Comput.*, 26(1-2):27–48, 2015. (Cited on pages 1, 2, and 43.)

[11] P. Brendel, P. Dłotko, G. Ellis, M. Juda, and M. Mrozek. Computing fundamental groups from point clouds. *Appl. Algebra Engrg. Comm. Comput.*, 26(1-2):27–48, 2015. (Cited on pages 4, 5, 85, 86, and 90.)

[12] R. Brown. Groupoids and van Kampen's theorem. *Proc. London Math. Soc. (3)*, 17:385–401, 1967. (Cited on pages 5 and 85.)

[13] R. Brown. *Elements of modern topology*. McGraw-Hill Book Co., New York, 1968. (Cited on page 58.)

[14] R. Brown. From groups to groupoids: A brief survey. *Bull. London Math. Soc.*, 19:113–134, 1987. (Cited on page 18.)

[15] R. Brown. *Topology: a geometric account of general topology, homotopy types, and the fundamental groupoid*. Ellis Horwood, Chichester, 1988. (Cited on page 1.)

[16] R. Brown. *Topology and groupoids*. 2006. Third edition of *Elements of modern topology* [McGraw-Hill, New York, 1968], http://groupoids.org.uk. (Cited on pages 1, 4, 40, and 58.)

[17] R. Brown, P.J. Higgins, and R. Sivera. *Nonabelian algebraic topology: filtered spaces, crossed complexes, cubical homotopy groupoids*, volume 15. EMS Tracts in Mathematics, 2011. (Cited on page 1.)

[18] R. Brown and A.R. Salleh. A van Kampen theorem for unions of nonconnected spaces. *Arch. Math. (Basel)*, 42(1):85–88, 1984. (Cited on pages 5, 85, and 96.)

[19] J. Carlson. http://www.math.uga.edu/lvalero/cohointro.html. (Cited on page 1.)

[20] G. Carlsson. Topology and data. *Bulletin of the American Mathematical Society*, 46(2):255–308, 2009. (Cited on page 68.)

[21] Fr d ric Chazal, D. Cohen-Steiner, L. J. Guibas, F. M moli, and S. Y. Oudot. Gromov-Hausdorff Stable Signatures for Shapes using Persistence. *Computer Graphics Forum*, 28(5):1393–1403, July 2009. (Cited on page 3.)

[22] S. W. Cheng, T. K. Dey, and E. A. Ramos. Manifold reconstruction from point samples. *Proc. of ACM SIAM Symposium on Discrete Algorithms*, 2005. (Cited on page 3.)

[23] James Frederic Davis and Paul Kirk. *Lecture Notes in Algebraic Topology*. American Mathematical Soc., 2001. (Cited on page 9.)

[24] S. Derdar, M. Allili, and D. Ziou. Topological feature extraction using algebraic topology. *Vision Geometry XV. Edited by Latecki*, 6499:64990G, January 2007. (Cited on page 3.)

[25] T. K. Dey, F. Fan, and Y. Wang. Graph induced complex on point data. *Computational Geometry*, 48(8):575–588, 2015. (Cited on page 3.)

[26] T. K. Dey and S S. Goswami. Provable surface recon- struction from noisy samples. *Computational Geometry: Theory and Application*, 35(1-2):124–141, 2006. (Cited on page 3.)

[27] H. Edelsbrunner and J.L. Harer. Computational topology. *Computational Topology, AMS*, 2010. (Cited on page 68.)

[28] G. Ellis. *HAP – Homological Algebra Programming, Version 1.10.13*, 2013. http://www.gap-system.org/Packages/hap.html. (Cited on pages 70 and 95.)

[29] G. Ellis. *An invitation to Computational Homotopy*. 2018. (Cited on pages 41, 42, 43, 45, 51, 68, and 69.)

[30] G. Ellis and Fintan Hegarty. Computational homotopy of finite regular cw-spaces. *Journal of Homotopy and Related Structures*, 9(1):25–54, Apr 2014. (Cited on page 1.)

[31] G. Ellis and Fintan Hegarty. Computational homotopy of finite regular CW-spaces. *J. Homotopy Relat. Struct.*, 9(1):25–54, 2014. (Cited on pages 2, 43, 85, 90, and 93.)

[32] Gansner E. Koutsofios L. North S.C. Ellson, J. and G. Woodhull. Graphviz—open source graph drawing tools. graph drawing. pages 483–484, 2002. (Cited on page 76.)

[33] S. Emrani, T. Gentimis, H. Krim IEEE Signal Processing Letters, and 2014. Persistent homology of delay embeddings and its application to wheeze detection. *ieeexplore.ieee.org*. (Cited on page 65.)

[34] Robin Forman. Morse theory for cell complexes. *Advances in Mathematics*, 134 (1), 1998. (Cited on page 41.)

[35] Robin Forman. A user's guide to discrete morse theory. *Sém. Lothar. Combin*, 48, 2002. (Cited on pages 40 and 41.)

[36] E.R. Gansner and S.C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000. (Cited on page 72.)

[37] X. Ge, I.I. Safa, M. Belkin, and Y. Wang. Data skeletonization via reeb graphs. *NIPS'11 Proceedings of the 24th International Conference on Neural Information Processing Systems*, (24):837–845, 2011. (Cited on page 3.)

[38] Ross Geoghegan. *Topological Methods in Group Theory*. Springer Science+Business Media, LLC, 2008. (Cited on page 8.)

[39] R. Ghrist. *Elementary Applied Topology*. 2014. ISBN: 978-1502880857. (Cited on page 68.)

[40] R. Gonzalez-Diaz, M.J. Jimenez, and B. Medrano. Cohomology ring of 3D cubical complexes. *Springer Lecture Notes in Computer Science*, 5852:139–150, 2009. (Cited on pages 85 and 86.)

[41] R. Gonzalez-Diaz, J. Lamar, and R. Umble. Cup products on polyhedral approximations of 3D digital images. In *Combinatorial image analysis*, volume 6636 of *Lecture Notes in Comput. Sci.*, pages 107–119. Springer, Heidelberg, 2011. (Cited on pages 85 and 86.)

[42] R. Gonzalez-Diaz, J. Lamar, and R. Umble. Computing cup products in $\mathbb{Z}_2$-cohomology of 3D polyhedral complexes. *Found. Comput. Math.*, 14(4):721–744, 2014. (Cited on pages 85 and 86.)

[43] R. González-Díaz and P. Real. Computation of cohomology operations of finite simplicial complexes. *Homology Homotopy Appl.*, 5(2):83–93, 2003. Algebraic topological methods in computer science (Stanford, CA, 2001). (Cited on pages 85 and 86.)

[44] R. González-Díaz and P. Real. On the cohomology of 3D digital images. *Discrete Appl. Math.*, 147(2-3):245–263, 2005. (Cited on pages 85 and 86.)

[45] D. Green. http://www.math.uni-wuppertal.de/green/Cohov2/. (Cited on page 1.)

[46] A. Grothendieck. Esquisse d'un programme. In *Geometric Galois actions, 1*, volume 242 of *London Math. Soc. Lecture Note Ser.*, pages 5–48. Cambridge Univ. Press, Cambridge, 1997. With an English translation on pp. 243–283. (Cited on page 96.)

[47] The GAP Group. *GAP : Groups, Algorithms, and Programming, Version 4.5.6*, 2013. http://www.gap-system.org. (Cited on page 1.)

[48] Leonidas J Guibas and Steve Y Oudot. Reconstruction Using Witness Complexes. *Discrete and Computational Geometry*, 40(3):325–356, October 2008. (Cited on page 3.)

[49] P. Guillot. http://irma.math.unistra.fr/~guillot/. (Cited on page 1.)

[50] Soren Hansen. Lecture notes on algebraic topology, 2005. http://www.math.ksu.edu/~hansen/CWcomplexes.pdf). (Cited on pages 6, 7, and 8.)

[51] S. Harker, K. Mischaikow, M. Mrozek, and V. Nanda. Discrete morse theoretic algorithms for computing homology of complexes and maps. *Found. Comput. Math.*, 14(1):151–184, 2014. (Cited on page 43.)

[52] A. Hatcher. *Algebraic topology*. Cambridge University Press, Cambridge, New York, 2002. Autre(s) tirage(s) : 2003,2004,2005,2006. (Cited on page 86.)

[53] Allen Hatcher. *Algebraic topology*. Cambridge University Press, New York, NY, USA, 2010. (Cited on pages 8 and 45.)

[54] P. J. Higgins. Presentations of groupoids, with applications to groups. *Proc. Cambridge Philos. Soc.*, 60:7–20, 1964. (Cited on pages 4 and 96.)

[55] P. J. Higgins. Categories and groupoids, 1971. (Cited on pages 11 and 13.)

[56] P.J. Higgins. On the cohomology of 3d digital images. *Mathematical Studies, Volume 32. Van Nostrand Reinhold Co. London (1971); Reprints in Theory and Applications of Categories, No. 7 (2005) pp 1-195.* (Cited on page 1.)

[57] D. F. Holt, B. Eick, and E. A. O'Brien. Handbook of computational group theory. Chapman and Hall/CRC, 2005. (Cited on page 1.)

[58] J. B. J. B. Tenenbaum, V. de Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 5500(3), 2000. (Cited on page 3.)

[59] D. Jones. A general theory of polyhedral sets, 1988. (Cited on pages 40 and 41.)

[60] D. Joyner. A primer on computational group homology and cohomology in GAP and SAGE. *Algebra and Discrete Mathematics, Aspects of Infinite Groups: A Festschrift in Honor of of Anthony Gaglione*, Volume 1, 2008. (Cited on page 1.)

[61] Anil Jain K. and C. Richarad Dubes. Algorithms for clustering data, 1988. (Cited on page 71.)

[62] T. Kaczynski and M. Mrozek. The cubical cohomology ring: an algorithmic approach. *Found. Comput. Math.*, 13(5):789–818, 2013. (Cited on pages 85 and 86.)

[63] H. Kantz and T. Schreiber. *Nonlinear Time Series Analysis*. Cambridge University Press, 2003. (Cited on page 65.)

[64] J. Kilner and K. J. Friston. Topological inference for EEG and MEG data. *Annals of Applied Statistics*, 4(3):1272–1290, 2010. (Cited on page 3.)

[65] Henry King, Kevin Knudson, and Neža Mramor. Generating discrete morse functions from point data. *Experimental Mathematics*, 14(4):435–444, 2005. (Cited on page 68.)

[66] CMU Graphics Lab. *"CMU graphics lab motion capture database."*, 2012. http://mocap.cs.cmu.edu/. (Cited on pages 4 and 82.)

[67] J. Lamar-Len, E. Garca-Reyes, , and R. Gonzalez-Diaz. Human gait identification using persistent homology, in progress in pattern recognition. *Image Analysis, Computer Vision, and Applications*, 7441:244–251, 2012. (Cited on page 82.)

[68] T. Lewiner, H. Lopes, and G. Tavares. Optimal discrete Morse functions for 2-manifolds. *Comput. Geom.*, 26(3):221–233, 2003. (Cited on page 93.)

[69] Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, New York, 1971. Graduate Texts in Mathematics, Vol. 5. (Cited on pages 11 and 12.)

[70] T. Martinetz and K. Schulten. Topology preserving networks. *Neural Networks*, 7:507–522, 1994. (Cited on page 3.)

[71] W.S. Massey. *A basic course in algebraic topology*, volume 127 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1991. (Cited on page 86.)

[72] B. Mederos, N. Amenta, L. Velho, and L. H. de Figueiredo. Surface reconstruction for noisy point clouds. *In Proc. 3rd Sympos. on Geometry Processing*, pages 53—62, 2005. (Cited on page 3.)

[73] M. Morishita. *Knots and primes*. Universitext. Springer, London, 2012. An introduction to arithmetic topology. (Cited on page 91.)

[74] U. Muico, Y. Lee, J. Popovi'c, and Z. Popovi'c. Contact-aware nonlinear control of dynamic characters. *Applicable Algebra in Engineering, Communication and Computing*, 81:1–9, 2009. (Cited on pages 82 and 84.)

[75] J. D. Farmer N. H. Packard, J. P. Crutchfield and R. S. Shaw. Geometry from a time series. *Physical Review Letters*, 45(9):712, 1980. (Cited on page 65.)

[76] M. Nicolau, A. Levine, and G. G. Carlsson. Topology based data analysis identifies a subgroup of breast cancers with a unique mutational profile and excellent survival. *Proc. Natl. Acod. Sci. USA*, 108(17):7265—7270, 2011. (Cited on page 3.)

[77] S.Y. Oudot. Persistence theory: From quiver representations to data analysis. *AMS Mathematical Surveys and Monographs*, 2015. (Cited on page 68.)

[78] I.H. Park and C. Li. Dynamic ligand-induced-fit simulation via enhanced conformational samplings and ensemble dockings: a survivin example. *J. Phys. Chem.*, B(114):5144—5153, 2010. (Cited on page 3.)

[79] N. Peinecke, F-E. Wolter, and M. Reuter. Laplace spectra as fingerprints for image recognition. *Computer-Aided Design*, 39(6):460–476, June 2007. (Cited on page 3.)

[80] P. Pilarczyk and P. Real. Computation of cubical homology, cohomology, and (co)homological operations via chain contraction. *Adv. Comput. Math.*, 41(1):253–275, 2015. (Cited on pages 85 and 86.)

[81] M. Reuter, F-E. Wolter, and N. Peinecke. *Laplace-spectra as fingerprints for shape matching*. ACM, New York, New York, USA, June 2005. (Cited on page 3.)

[82] M. Roder. The GAP package HAPcryst, available from http://www.gap-system.org/Packages/undep.html. (Cited on page 1.)

[83] R. M. Rustamov. Laplace-beltrami eigenfunctions for deformation invariant shape representation. *In Symposium on Ge- ometry Processing*, pages 225–233, 2007. (Cited on page 3.)

[84] H. Rutishauser. The jacobi method for real symmetric matrices. *Numerische Mathematik*, 9(1):1–10, Nov 1966. (Cited on page 74.)

[85] M. Spagnuolo B. Falcidieno S. Biasotti, D. Giorgi. Reeb graphs for shape analysis and applications. *Theoret. Comput. Sci.*, (392):5–22, 2008. (Cited on page 3.)

[86] G. Singh, F. Memoli, and G. Carlsson. Topological Methods for the Analysis of High Dimensional Data Sets and 3D Object Recognition. In M. Botsch, R. Pajarola, B. Chen, and M. Zwicker, editors, *Eurographics Symposium on Point-Based Graphics*. The Eurographics Association, 2007. (Cited on pages 3, 4, and 68.)

[87] Václav Snášel, Jana Nowaková, Fatos Xhafa, and Leonard Barolli. Geometrical and topological approaches to Big Data. *Future Generation Computer Systems*, 67:286–296, February 2017. (Cited on page 3.)

[88] F. Takens. Detecting strange attractors in turbulence. *Dynamical Systems and Turbulence, Lecture Notes in Mathematics*, 898:366–381, 1981. (Cited on page 65.)

[89] R. Vasudevan, A. Ames, and R. Bajcsy. Persistent homology for automatic determination of human- data based cost of bipedal walking. *Nonlinear Analysis: Hybrid Systems, IFACWorld Congress*, 7(1):101 – 115, 2013. (Cited on page 82.)

[90] Mikael Vejdemo-Johansson, Florian T. Pokorny, Primoz Skraba, and Danica Kragic. Cohomological learning of periodic motion. *Applicable Algebra in Engineering, Communication and Computing*, 26(1-2):5–26, 2015. (Cited on page 82.)

[91] S. Weinberger. The Complexity of Some Topological Inference Problems. *Foundations of Computational Mathematics*, 14(6):1277–1285, January 2014. (Cited on page 3.)

[92] J. H. C. Whitehead. Simple homotopy types. *Amer. J. Math.*, (72):1–57, 1950. (Cited on page 41.)

[93] Yao Yuan, Jian Sun, Huang Xuhui, B. Gregory, G. Singh, L. Michael, G. Leonidas, P. Vijay, and G. Gunnar. Topological methods for exploring low-density states in biomolecular folding pathways. *The Journal of Chemical Physics*, 130(144115):1–10, 2009. (Cited on page 68.)

[94] A. Zomorodian. *Topology for Computing*. Cambridge University Press, New York, NY, 2005. (Cited on pages 8 and 68.)

[95] A. Zomorodian. Topological data analysis. *Proceedings of Symposia in Applied Mathematics, AMS*, 2011. (Cited on page 9.)

[96] A. Zomorodian. *Advances in Applied and Computational Topology*. 2012. AMS. (Cited on page 68.)