CHALMERS
UNIVERSITY OF TECHNOLOGY

Computing persistent homology in parallel with a functional language

Master's thesis in Engineering Mathematics and Computational Science

ERIK VON BRÖMSSEN

DEPARTMENT OF MATHEMATICAL SCIENCES

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021
www.chalmers.se

Master's thesis 2021

Computing persistent homology in parallel with a functional language

Erik von Brömssen

Department of Mathematical Sciences
Chalmers University of Technology
Gothenburg, Sweden 2021

Computing persistent homology in parallel with a functional language
Erik von Brömssen

Supervisor: John Hughes, Mary Sheeran, Department of Computer Science and Engineering
Examiner: Martin Raum, Department of Mathematical Sciences

Master's Thesis 2021
Department of Mathematical Sciences
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone: +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

Computing persistent homology in parallel with a functional language
Erik von Brömssen
Department of Mathematical Sciences
Chalmers University of Technology

Abstract

Persistent homology, first developed at the beginning of the millennium, is a tool within the field of topological data analysis. It is an extension of simplicial homology to filtrations of simplicial complexes, which allows one to, in a sense, compute topological features of finite sets of points in a metric space at a variety of scales. More precisely, persistent homology is concerned with sequences of homomorphisms between homology groups induced by inclusions on the underlying groups of cycles. Persistent homology is well suited for data analysis since it can be efficiently computed via a certain matrix reduction.

Graphics processing units (GPUs), originally designed for tasks such as image rendering, have recently become an integral part of high performance computing due to their massively parallel design. Futhark is a statically typed purely functional language that compiles to efficient code for GPUs. The goal of Futhark is to simplify the implementation of algorithms for GPUs, by giving a high-level functional perspective and hiding low-level concepts from the programmer.

In this thesis we present a massively parallel algorithm for computing persistent homology on GPUs, and we describe an implementation in Futhark. Our algorithm is conceptually simple, and its main parts are all entirely massively parallel. Our implementation utilises a sparse matrix data structure and exemplifies that non-trivial sparse matrix computations can be efficiently implemented in Futhark. We compare the performance of our algorithm to that of OpenPH, an existing GPU-based persistent homology algorithm, and achieve speedups of 2.3 to 5. Lastly, we briefly investigate the potential to use our algorithm for approximating persistent homology via early stopping.

Keywords: Persistent homology, functional programming, GPU, Futhark, sparse matrix.

Acknowledgements

I would like to thank my supervisors Mary Sheeran and John Hughes, as well as my examiner Martin Raum, for their collective help and guidance in the writing of this thesis. I especially extend my thanks to Mary for her help with writing the paper submitted to FHPNC that thesis is largely an extension of.

I would further like to thank Troels Henriksen for his comments and suggestions regarding the writing of the Futhark code presented in this thesis.

Lastly, I would like to thank all the friends who have made life during the current pandemic easier to endure. Of these I particularly thank David Elinder with whom I have had many discussions about the experience of writing a thesis in this period of remote work and social distancing.

Erik von Brömssen, Gothenburg, June 2021

Table of Contents

1. Introduction

1.1 Background

A topological perspective on data can expose interesting qualitative features that can be used in analysis. Many topological spaces can be distinguished by counting the number of holes they have. For instance, a circle has one hole while a disk has none, and a torus has (with a suitable definition of hole) more holes than a sphere.

There's more than one way to define holes, but one which lends itself to efficient computations is the theory of *homology* in algebraic topology. Homology can be defined on simplicial complexes — roughly speaking, a space built from n-dimensional triangles — where the boundary maps of the complex give rise to its homology groups. The generators of these groups then correspond to holes in the complex. In the last two decades, the field of topological data analysis has seen the rise of the theory of *persistent homology* [1] — a theory of holes in pointclouds. In this field, one constructs a filtered simplicial complex from a pointcloud, and then tracks how the homology groups change under the inclusion maps of the filtration. With a suitable filtration, this gives a tool to study holes at many *scales* simultaneously, and importantly, how these holes are born and killed as the scale changes. This analysis results in a set of persistence intervals, called a persistence diagram or a barcode, which can then be used as a statistic of the pointcloud to be used in further analysis, depending on the problem at hand.

Persistent homology has been applied in a wide range of sciences. Li et al. used it to identify morphological variation in plants, by applying it to processed greyscale images of leaves and roots [2]. Musa et al. used the persistent homology of the phase space of a water level time series to develop a flood early warning system [3]. In varying degrees of complexity, it has also been used in fields like quantum physics [4], brain morphometry [5], and tourism [6].

1.2 Purpose

Graphics processing units, with typically thousands of cores on a single card, can be used for massively parallel computations. Due to the hardware architecture of GPUs, they are poorly suited for task parallelism, i.e. performing different kinds of tasks in parallel, but are well suited for data parallelism, i.e. doing the same computation in parallel on different data. GPUs are typically programmed using the C-like languages in frameworks such as CUDA and OpenCL. These languages are low-level and require the programmer to reason about things such as individual threads or groups of threads (*work-items* and *work-groups* in OpenCL), synchronisation, and memory layout. Algorithms also have to be designed in a low-level fashion, similarly to writing code in C. In contrast, Futhark is a high-level purely functional language that can be compiled to CUDA or OpenCL. This allows a programmer to design algorithms with a high-level perspective, without having to consider threads or memory layout of data.

Persistent homology is computed by reducing a certain matrix representing the boundary maps of a filtered complex, and speeding up this reduction is currently the focus of much research [7]. These matrices are sparse and can quickly become large, so more sophisticated methods than using a dense matrix data structure is needed in practically useful algorithms. Many authors have parallelised the reduction algorithm in various ways, and most algorithms have been designed and implemented for CPUs and distributed systems. A notable exception is the algorithm constructed by Mendoza-Smith and Tanner [8], which has been successfully implemented for GPUs in CUDA by the same authors.

The purpose of this thesis is twofold: firstly, we further investigate whether it is possible to design a massively data-parallel algorithm for computing persistent homology that can be implemented for GPU hardware; and secondly, we investigate whether Futhark is a suitable language to use for sparse matrix algorithms of this nature.

1.3 Outline

In chapter 2 we give a self-contained description of Futhark. We do not describe the entire language, but enough for any reader to understand the code presented in this thesis. For a full description of Futhark, see e.g. the book Parallel Programming in Futhark [9]. Chapter 3 presents the necessary mathematical background of homology, chapter 4 describes how one can compute persistent homology, while our main contribution, a new parallel algorithm, is presented in 5. We present benchmarks and compare with OpenPH in chapter 6, and finally a discussion about our work can be found in chapter 7.

2. Futhark

Graphics processing units (GPUs), originally developed for computations in computer graphics, have during the last two decades been used for other applications. Computer graphics involves processing a large number of pixels and other data independently, and so GPUs generally have a large number of cores, typically thousands, in contrast to CPUs that typically have a number of cores on the order of 1 or 10. GPUs are poorly suited for task-parallel algorithms, where many different tasks have to be performed in parallel, but are well suited for data-parallel algorithms, where the same task has to be performed on many points of data in parallel.

GPUs are often programmed using the frameworks OpenCL [10] or CUDA [11]. While CUDA is developed by NVidia and only available on their hardware, OpenCL is a specification implemented on both NVidia and AMD graphics cards. These frameworks both utilise low-level C-like languages to write the programs that run on GPUs, generally requiring the programmer to consider the memory layout of data, access patterns and cache use, synchronisation, and dividing work across threads. In contrast, Futhark is a hardware-agnostic "statically typed, data-parallel and purely functional array language" with a heavily optimising compiler that can currently compile to OpenCL, CUDA, or multithreaded C code [12]. Being purely functional and high-level, Futhark does not expose such low-level concepts to the programmer; instead, programs are written in terms of operations on entire arrays. The syntax of Futhark is similar to languages like ML and Haskell, although it is far less general. Notably, it does not support recursion, and its type system is less expressive [9]. In the remainder of this chapter, we give an overview of Futhark, covering enough to enable the reader to understand all code presented in this thesis.

2.1 The basics

2.1.1 Primitives and function definitions

The type system in Futhark has a number of primitive types: signed integers `i8`, `i16`, `i32`, `i64`, unsigned integers `u8`, `u16`, `u32`, `u64`, floating-point numbers `f32`, `f64`, and booleans `bool`. The boolean values are `true` and `false`. The number suffixed to each numerical type denotes the number of bits in its representation. A number of functions exist for converting between primitive types, such as the function `i64.i32` that converts a 32-bit integer to a 64-bit integer. Other conversion functions follow the same naming pattern.

Types can be combined to make *tuple* types, e.g. (`i32`, `i32`) and (`bool`, (`u8`, `f32`)); values of these types include $(0, -5)$ and (`false`, $(1, 2.56)$). We extract elements from tuples using dot notation: The expression $(1.3, 5.6)$.0 will have the value 1.3 and $(1.3, 5.6)$.1 the value 5.6. As an example, we may represent complex numbers using 2-tuples where the first component is the real part and the second component is the imaginary part. Then we can define complex multiplication with the following function definition:

```
let cmul (z: (f32,f32)) (w: (f32,f32)): (f32,f32) =
  let re = z.0 * w.0 - z.1 * w.1
  let im = z.0 * w.1 + z.1 * w.0
  in (re,im)
```

The first line defines the function `cmul` and gives its type signature: it takes two arguments, `z` and `w`, both of type (`f32`, `f32`), and it evaluates to a value also of type (`f32`, `f32`). After the equals sign follows the function's `body`, that defines its value. Here we bind two local variables with the `let` keyword, `re` and `im` corresponding to the real and imaginary parts of the product of `z` and `w`. The body ends with the `in` keyword, defining the final value of the function application.

In general, a function f of type $t_1 \to t_2 \to \cdots \to t_n \to t$ taking n arguments can be defined with syntax of the following form:

let f $(x_1\colon\ t_1)$ $(x_2\colon\ t_2)$ ... $(x_n\colon\ t_n)\colon\ t$ = body

In the above, `body` is an expression of type t defining the return value of the function.

To make the definition of `cmul` slightly more readable, we may define a *type abbreviation* for complex numbers:

```
type complex = (f32,f32)
```

Then the type signature of `cmul` can instead be written as:

```
let cmul (z: complex) (w: complex): complex =
  ...
```

The body of `cmul` will be the same as before.

Functions can be *partially applied* to only some of their arguments. If f is a function of type $a \to b \to c$, and x a value of type a, the expression $f\ x$ has type $b \to c$. For instance, partially applying `cmul` to a single complex number as `cmul` $(1, 2)$, we get a function of type `complex` $\to$ `complex` that multiplies any complex number with $1 + 2i$. Specifically, we can define such a function as follows:

```
let cmul_with_1_2: complex → complex = cmul (1,2)
```

We can also partially apply infix operators; for instance, a function `half` that divides a float by two can be defined as such:

```
let half: f32 → f32 = (/2.0)
```

When partially applying infix operators, we need parentheses as above.

2.1.2 Records

Instead of representing complex numbers as 2-tuples, we can use a *record* type:

```
type complex' = {re: f32, im: f32}
```

Expressions of type `complex'` then contain two *fields*, called `re` and `im`, which are values of type `f32`. For instance, $1 + 2i$ can be written as $\{\texttt{re} = 1, \texttt{im} = 2\}$. The fields of a record are accessed using dot notation:

```
let cmul' (z: complex') (w: complex'): complex' =
  let real_part = z.re * w.re − z.im * w.im
  let imag_part = z.re * w.im + z.im * w.re
  in {re = real_part, im = imag_part}
```

The field names of a record are part of the type, so $\{\texttt{a} : \texttt{i32}\}$ and $\{\texttt{b} : \texttt{i32}\}$ are considered distinct types.

2.1.3 Lambda expressions

Like many other languages, Futhark supports lambda expressions. If e is an expression of type t possibly containing the variables $x_1, \dots x_n$ of types $t_1, \dots, t_n$ respectively, then $\lambda x_1 \dots x_n \to e$ is an expression of type $t_1 \to \cdots \to t_n \to t$. For instance, the code below shows three equivalent ways of defining $f(x) = 2x$:

```
let f x = 2 * x
let f = (2 *)
let f = λx → 2 * x
```

Note that we have omitted the type signatures in the definitions above. In this case, the types will be inferred by the compiler, and since integer literals default to the type `i32`, the type of both `f` and `f'` will be `i32` → `i32`. Lambda expressions are especially useful in higher-order functions, see section 2.5.

2.2 Polymorphism

A type signature does not need to mention a specific type, but can also contain *type variables*. For instance, we can define a function `first` of type $(a, b) \to a$ where a and b are type variables as follows:

```
let first 'a 'b (pair: (a,b)): a = pair.0
```

In the type signature of `first`, the single quotes indicate that a and b are type variables rather than specific types. This makes `first` a *polymorphic* function; it can be applied to *any* 2-tuple, regardless of the types of its elements, and the return type will depend on the type of the tuple. So, for instance, `first` $(1, 2)$ equals 1 and has type `i32`, while `first` $(\texttt{true}, 2)$ equals `true` and has type `bool`. While this example is not very interesting, polymorphism plays a crucial role in section 2.5.

2.3 Pattern matching

Consider a simple definition:

```
let x = foo
```

On the left-hand side of this definition, x is a variable name that will take the value of foo. This definition is always possible, regardless of the type of foo. We may consider the name x to be a *pattern* that matches any value regardless of its type. In Futhark definitions, we can also use more refined patterns, such as the pair pattern (x, y) where x and y are two different variable names. This pattern can only match values whose type is (t_1, t_2) for some types t_1 and t_2. When a pattern is matched, the variables x and y will be bound to the corresponding values that were matched. For instance, if the expression foo evaluates to the value (v_1, v_2) of type (t_1, t_2), we can define

```
let (x,y) = foo
```

after which x will equal v_1 with type t_1 and similarly for y. If the type of foo is not a pair type, the above definition is a type error.

Pattern matching can also occur in type signatures, so for instance, the first function from above can be defined as follows:

```
let first 'a 'b ((x,y): (a,b)): a = x
```

We can also pattern match in lambda expressions, so we can also define first as such:

```
let first 'a 'b: ((a,b) → a) = λ(x,y) → x
```

Patterns can also be records or nested combinations of records and tuples, although we will only use tuple pattern matching in the code in this thesis.

2.4 Loops and branching

Futhark does not support recursion, but does have a number of loop constructs, that are all executed entirely sequentially. One type of loop expression is

```
loop pattern = initial for i < bound do body
```

where pattern = initial is a pattern matching expression, binding the values in initial to the names in pattern. This construct first binds the value of initial to pattern, and then as long as i is less than bound, it evaluates body and binds the resulting value to pattern, and increments i by one. As an example, we can sum the ten first even non-negative numbers as follows:

```
let sum_of_evens = loop x = 0 for i < 10 do x + 2 * i
```

Another type of loop expression is the while loop construct:

```
loop pattern = initial while b do body
```

where b is an expression of type bool that depends on the names in pattern. As an example, consider function search below, that performs a sequential binary search in a sorted array of 64-bit integers (note that the loop only halts if the wanted integer is actually in the array):

```
let search [n] (xs: [n]i64) (x: i64): i64 =
  (loop (l,r,m) = (0,n,0) while xs[m] != x do
    let m = (l + r) / 2
    in if xs[m] < x then (m+1,r,m) else (l,m−1,m)
  ).2
```

Branching in Futhark is done with the `if-then-else` construct: if e_1 and e_2 are expressions of type t and b is an expression of type `bool`, then

```
if b then e1 else e2
```

is an expression of type t.

2.4.1 Uniqueness types

To allow for in-place updates while keeping the language pure and free from side effects, Futhark uses the concept of *uniqueness types*. A uniqueness type is a modification of a normal type, indicated with an asterisk before the normal type name; if t is a type, then $*t$ is a uniqueness variant of the same type. Whenever a value is given as a uniqueness typed argument to another function, neither that value nor any other value aliasing it can be used later in the same scope. For instance, if `foo` has type $*t \to t$, `bar` has type $t \to t$, and x has type t, then

```
let y = x
let z = foo x
in bar y
```

is illegal, since `y` aliases `x` and cannot be used after the call to `foo`. Further, if the returned value of a function has a uniqueness type, then it cannot share memory space with any of the non-unique arguments passed to the same function; the returned value cannot alias anything non-unique. Uniqueness types are described more in detail in the book Parallel Programming in Futhark [9].

The polymorphic function `copy` of type $a \to *a$ makes a copy of its argument in memory, and returns the new copy, with a uniqueness type. Thus, while the above code example is not valid, the following is:

```
let y = x
let z = foo (copy x)
in bar y
```

2.5 Arrays and second-order array combinators

An essential type in Futhark is the *array* type `[]`t denoting arrays whose elements have type t. Array types can also have a size, so `[`n`]`t is the type of arrays of length n whose elements have type t. Arrays can be multidimensional but must be *regular*, i.e. all inner arrays must have the same size: $[[1, 2], [3, 4], [5, 6]]$ is a valid array and has type `[3][2]i32` while $[[1, 2], [3, 4, 5]]$ is not a valid array. Futhark uses zero-based indexing, which is done into arrays using square braces; if the array `xs` $= [4, 5, 6]$, the expression `xs`$[0]$ equals 4. We can concatenate arrays with the `++` operator, so $[1]$ `++` $[4, 5] = [1, 4, 5]$. Two common functions on arrays are `head`, that returns the first element in an array, and `init`, that returns all but the last element.

The most useful functions in Futhark are the so-called *array combinators*. These are builtin functions that act on arrays that can be efficiently executed in parallel; they are the basic building blocks with which we build parallel programs. An often used array combinator is the polymorphic function `zip`, of type

$$\texttt{[}n\texttt{]}\,a \to \texttt{[}n\texttt{]}\,b \to \texttt{[}n\texttt{]}\,(a, b).$$

As the type suggests, `zip` combines two arrays of equal length into a single array of pairs. For instance,

$$\texttt{zip}\ [1, 2, 3]\ [\texttt{true}, \texttt{false}, \texttt{true}]$$

evaluates to

$$[(1, \texttt{true}), (2, \texttt{false}), (3, \texttt{true})].$$

Related to `zip` is `unzip`, of type

$$\texttt{[}n\texttt{]}\,(a, b) \to (\texttt{[}n\texttt{]}\,a,\ \texttt{[}n\texttt{]}\,b),$$

turning an array of pairs into a pair of arrays. Continuing the previous example,

$$\texttt{unzip}\ [(1, \texttt{true}), (2, \texttt{false}), (3, \texttt{true})]$$

evaluates to

$$([1, 2, 3], [\texttt{true}, \texttt{false}, \texttt{true}]).$$

Other common combinators are the *second order array combinators*, or *SOACs*. Among these are *map*, *reduce* and *filter*, which should be recognisable to any reader familiar with functional languages. These are polymorphic functions that take a *function as an argument* and operate on arrays. We describe those that we use in our work below.

2.5.1 Map and tabulate

The SOAC `map` has type

$$(a \to b) \to \texttt{[}n\texttt{]}\,a \to \texttt{[}n\texttt{]}\,b.$$

It applies its first argument to each element in the second argument.

Recall the simple function $f(x) = 2x$ from before. The application `map` f $[1, 2, 3]$ will then evaluate to $[2, 4, 6]$. We can also avoid having to name the function f by using a lambda, and instead write `map` $(\lambda x \to 2x)$ $[1, 2, 3]$.

A special case of `map` is the `tabulate` function, of type

$$(n : \texttt{i64}) \to (\texttt{i64} \to a) \to \texttt{[}n\texttt{]}\,a.$$

The expression `tabulate` n f is defined as equal to `map` f (`iota` n), where `iota` $n = [0, 1, \ldots, n-1]$. As an example, we can get all even numbers between 0 and 10 (exclusive) with the expression `tabulate` 5 $(*2)$.

2.5.2 Reduce and scan

While `map` operates on elements individually, `reduce` and `scan` combine the elements of an array. The type of `reduce` is

$$(a \to a \to a) \to a \to \texttt{[}n\texttt{]}\,a \to a.$$

If f has type $a \to a \to a$, is associative, and has neutral element e, then the expression `reduce` $f\ e\ [x_1, x_2, \ldots, x_n]$ equals $f\ x_1\ (f\ x_2\ (\ldots(f\ x_n\ e)))$. As an example, `reduce` $(+)\ 0$ is a function that sums each element in a list.

We can, for instance, combine `zip`, `map`, and `reduce` to compute the dot product of two arrays:

```
let dotprod (xs: []i32) (ys: []i32): i32 =
  let products = map (λ(x,y) → x * y) (zip xs ys)
  in reduce (+) 0 products
```

In our work we use a variant of `reduce` called `reduce_by_index`, which has the type

$$*\texttt{[}m\texttt{]}\,a \to (a \to a \to a) \to a \to \texttt{[}n\texttt{]i64} \to \texttt{[}n\texttt{]}\,a \to *\texttt{[}m\texttt{]}\,a.$$

Conceptually, the application `reduce_by_index dest` $f\ e$ `is as` creates index-value-pairs $(i, v) \in$ `zip is as`, and for each i, reduces the corresponding values into `dest` at index i — reduction is performed using f, which must be associative and have neutral element e. Conceptually, `reduce_by_index` is described by the following imperative pseudocode:

Algorithm 1: Reduce by index

```
1 for $(i, v) \in$ zip is as do
2     dest$[i] \leftarrow f$ dest$[i]$ $v$
3 end for
```

For instance, `reduce_by_index` $[0, 0]\ (+)\ 0\ [0, 1, 0, 0, 1]\ [1, 2, 3, 4, 5]$ evaluates to $[8, 7]$, since the values $[1, 3, 4]$ are added into index 0 of `dest` $:= [0, 0]$ while the values $[2, 5]$ are added into index 1.

We also use the `scan` function, also known as a prefix sum. This function is similar to `reduce`, but results in an array instead of a single value. It has the type $(a \to a \to a) \to a \to \texttt{[}n\texttt{]}\,a \to \texttt{[}n\texttt{]}\,a$, and it computes the reduction of every prefix of the given array. In other words, `scan` $f\ e\ [x_1, x_2, \ldots, x_n]$ is the same as the array [`reduce` $f\ e\ [x_1]$, `reduce` $f\ e\ [x_1, x_2], \ldots,$ `reduce` $f\ e\ [x_1, x_2, \ldots, x_n]$]. For instance, `scan` $(+)\ 0\ [1, 2, 3]$ evaluates to $[1, 3, 6]$.

2.5.3 Filter and partition

The `filter` SOAC has type

$$(a \to \texttt{bool}) \to \texttt{[]}\,a \to \texttt{[]}\,a.$$

`filter` p `xs` returns an array containing only those elements $x \in$ `xs` for which $p\ x$ is true. The type of `partition` is similar:

$$(a \to \texttt{bool}) \to \texttt{[]}a \to (\texttt{[]}a, \texttt{[]}a).$$

It returns both an array containing all elements for which the predicate holds, and an array with all elements for which it does not hold. As an example, if $p = \lambda x \to x\%2 == 0$, then `filter` p (`iota` 5) equals $[0, 2, 4]$ while `partition` p (`iota` 5) equals $([0, 2, 4], [1, 3])$.

2.5.4 Expand

The `expand` function is not part of the base Futhark language, but was defined by Elsman et al. [13] and is available via the external `segmented` library. It has type

$$(a \to \texttt{i64}) \to (a \to \texttt{i64} \to b) \to \texttt{[]}a \to \texttt{[]}b.$$

The function application `expand sz get xs` expands each $x \in$ `xs` into `sz` x elements of type b, such that the ith element in the expansion of x equals `get` $x\ i$. In other words, it is the concatenation of `tabulate` (`sz` x) (`get` x) for all $x \in$ `xs`. For instance, `expand` $(\lambda x \to x)$ $(\lambda x\ i \to x + 2 * i)$ $[2, 3, 4]$ evaluates to $[2, 4, 3, 5, 7, 4, 6, 8, 10]$.

2.5.5 Scatter

`scatter` is a first order array combinator that performs in-place updates in an array using uniqueness types. The type of `scatter` is

$$*\texttt{[}m\texttt{]}a \to \texttt{[}n\texttt{]i64} \to \texttt{[}n\texttt{]}a \to *\texttt{[}m\texttt{]}a.$$

The application `scatter dest is as` returns `dest` but with the element at index i equal to v for all $(i, v) \in$ `zip is as`. We say that the last two arguments are zipped to form *index-value-pairs*, a concept we will often refer to. For instance, `scatter` $[1, 2, 3, 4]$ $[0, 3]$ $[5, 1]$ evaluates to $[5, 2, 3, 1]$. This function ignores any index-value-pairs whose index is out-of-bounds.

2.6 Application operators

The infix operators |> and <| are two useful higher-order functions that are common in many languages in various forms. <| has type $(a \to b) \to a \to b$ and is the *application operator* that simply applies a function to an argument; it holds that f <| $x = f\ x$. Its use comes from the fact that it has low precedence and is right-associative, so expressions such as $f\ (g\ (h\ x))$ can be written as f <| g <| $h\ x$, allowing us to omit parentheses. |> is the same function but with its arguments flipped, so it has type $a \to (a \to b) \to b$. That is, $f\ x = x$ |> f. This function is useful for expressing computation "pipelines"; the expression $f\ (g\ (h\ x))$ can be written as x |> h |> g |> f. This pattern is common throughout chapter 5.

2.7 Size parameters

A function definition may place constraints on the size of its inputs and output. Consider the `dotprod` function from before. We can further refine its type signature by requiring that the two input arrays have equal length:

```
let dotprod [n] (xs: [n]i32) (ys: [n]i32): i32 =
  ...
```

Here, `[n]` denotes a *size parameter*. We do not explicitly give this argument when calling the function — we still apply it to just two arrays — but wherever we use `dotprod`, the Futhark compiler must be able to guarantee that the two arrays are of equal length.

Type abbreviations can also be given size parameters. We can, for instance, define a record type that stores two arrays of equal length as follows:

```
type two_arrays [n] = {array_a: [n]i32, array_b: [n]i32}
```

3. Theory

3.1 Topology

We start this chapter by defining some basic topological concepts. Topology is a field of its own, but here we only aim to give the definitions required for the rest of this chapter.

Definition 3.1.1 (Topological space). A *topological space* is a pair (X, τ) where X is a set and $\tau \subseteq \mathcal{P}(X)$ and $\mathcal{P}(X)$ is the powerset of X, such that

1. $\emptyset \in \tau$ and $X \in \tau$,
2. τ is closed under arbitrary unions, and
3. τ is closed under finite intersections.

τ is called a *topology* on X. The elements of τ are called *open sets*. If the specific topology is not important, or understood from context, we will call X a topological space.

On $\mathbb{R}^n$, the *standard topology* can be constructed by starting with the set of all open balls, and then including all unions and all finite intersections as open sets. Every subset of a topological space is also a topological space, inheriting the *subspace topology*. If (X, τ) is a topological space and $Y \subseteq X$, the subspace topology on Y is $\tau_Y = \{Y \cap U \mid U \in \tau\}$.

Maps between topological spaces admit a notion of continuity, which we now define.

Definition 3.1.2 (Continuous function). A function $f : X \to Y$ between two topological spaces X and Y is *continuous* iff the preimage of every open set is an open set.

A specific type of continuous function is *homeomorphisms*.

Definition 3.1.3 (Homeomorphism). A *homeomorphim* is a continuous function between topological spaces with a continuous inverse. Two spaces with a homeomorphism between them are said to be *homeomorphic*.

As a concrete example, a square without its interior is homeomorphic with a circle, while a circle and an annulus are not.

Definition 3.1.4 (Homotopy)**.** Let X and Y be topological spaces, and let $f, g : X \to Y$ be continuous functions. Then f and g are *homotopic* iff there exists a continuous map

$$H : \; X \times [0,1] \to Y$$

such that $H(x,0) = f(x)$ and $H(x,1) = g(x)$ for all $x \in X$, and H is a *homotopy* between f and g.

We can consider $H(\cdot, t)$ as a function of t that "continuously deforms" the function f into g.

Definition 3.1.5 (Homotopy equivalence)**.** Let X and Y be topological spaces, and let $f : X \to Y$ and $g : Y \to X$ be continuous functions. Then X and Y are *homotopy equivalent* iff $g \circ f$ is homotopic to id_X and $f \circ g$ is homotopic to id_Y.

Homotopy equivalence is a weaker condition than being homeomorphic; all homeomorphic spaces are homotopy equivalent. To exemplify the difference, consider the map $f : \mathbb{R}^n \to \{0\}$ for $n > 0$ projecting a Euclidean space onto a single point. Clearly, if $n > 0$, there is no bijection between $\mathbb{R}^n$ and $\{0\}$, so $\mathbb{R}^n$ is not homeomorphic with a single point. The spaces are however homotopy equivalent. Take any map $g : \{0\} \to \mathbb{R}^n$. Then clearly $f \circ g = \mathrm{id}_{\{0\}}$, and further, $g \circ f$ is homotopic to $\mathrm{id}_{\mathbb{R}^n}$ via the homotopy $H : \mathbb{R}^n \times [0,1] \to \mathbb{R}^n$ such that $H(x,t) = (1-t)g(f(x)) + tx$. Likewise, while a circle and an annulus are not homeomorphic, they are homotopy equivalent, and they are also homotopy equivalent with $\mathbb{R}^2 \setminus \{0\}$.

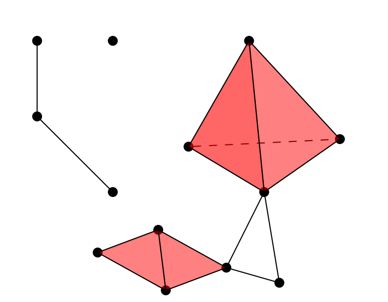
A basic intuitive understanding of these concepts is helpful but not required to understand homology as presented in the remainder of this chapter.

3.2 Simplicial complexes

There are many different homology theories, but the one used to define persistent homology is the theory of *simplicial homology.* To this end, we need to define simplices and simplicial complexes.

Definition 3.2.1 (Simplex)**.** An *n-simplex* σ, where $n \geq 0$, is the convex hull of $n+1$ affinely independent points in Euclidean space, and these points are called the *vertices* of the simplex. We call n the *dimension* of σ and write $\dim(\sigma) = n$. An m-simplex, with $m < n$, whose vertices are all vertices of σ is called a *face* of σ. If $v_0, \ldots, v_n$ are the vertices of σ, we write $\sigma = [v_0, \ldots, v_n]$. For the purposes of homology, we also equip simplices with an ordering of their vertices, so that $[v_0, \ldots, v_n]$ denotes an ordered set.

We often call 0-simplices *vertices*, 1-simplices *edges*, 2-simplices *triangles* and 3-simplices *tetrahedra.* Thus, simplices are a sort of generalisation of triangles. We can use these to construct simplicial complexes, a certain kind of set with an underlying topological space.


(a) *A set of simplices that is a simplicial complex.*

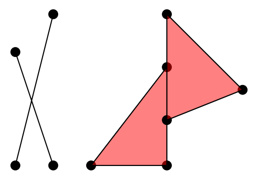

(b) *A set of simplices that is not a simplicial complex. The intersection of the lines to the left is a point which is not a face (i.e. endpoint) of any of the lines. The intersection of the triangles to the right is a line segment which is not a face (i.e. edge) of any of the triangles.*

Figure 3.1: Examples of valid and invalid simplicial complexes.

Definition 3.2.2 (Simplicial complex)**.** A *simplicial complex* is a set K of simplices such that

1. if $\sigma \in K$, then every face of sigma is also in K, and
2. for any $\sigma_1, \sigma_2 \in K$, $\sigma_1 \cap \sigma_2$ is either empty or is a simplex in K.

We define the *dimension* of K to be $\dim(K) := \max_{\sigma \in K} \dim(\sigma)$. Further, we write $|K| := \bigcup_{\sigma \in K} \sigma$, which we call the *underlying space* of K.

Simplicial complexes are particularly well-suited for practical computations, since they can easily be represented e.g. by listing their vertices, edges, triangles, etc. Note that the underlying space of a simplicial complex is a subset of $\mathbb{R}^n$, and so is a topological space. Many topological spaces are homeomorphic to the underlying space of a simplicial complex, so we may be able to compute properties of a space by computing it on such a complex. Such a homeomorphism is called a *triangulation*.

Definition 3.2.3 (Triangulation)**.** A *triangulation* of a topological space X is a simplicial complex K, with a homeomorphism $|K| \to X$. If such a triangulation exists, X is said to be *triangulable*, and we say that K *triangulates* X.

The concept of triangulations will help motivate the use of simplicial homology, and indeed of persistent homology.

While we will define homology using simplicial complexes, in order to exemplify how data analysis is often done we need the concept of an *abstract* simplicial complex.

Definition 3.2.4 (Abstract simplicial complex)**.** An *abstract simplicial complex* is a family of finite sets Δ such that if $\sigma \in \Delta$ and $\tau \subseteq \sigma$, then $\tau \in \Delta$. An element $\sigma \in \Delta$ with cardinality n is an *abstract* $(n-1)$*-simplex*, and its elements are the *vertices* of σ. The *dimension* of σ is $\dim(\sigma) = n - 1$. Further, the *dimension* of Δ is $\dim(\Delta) := \max_{\sigma \in \Delta} \dim(\sigma)$.

Every abstract simplicial complex can *realised* as a simplicial complex, by suitably mapping every abstract simplex to a simplex in $\mathbb{R}^n$ for n sufficiently large.

Definition 3.2.5 (Geometric realisation)**.** Let Δ be an abstract simplicial complex. A *geometric realisation* of Δ is a simplicial complex K such that there exists a bijection $f : \Delta \to K \cup \{\emptyset\}$ that

1. preserves intersections, i.e. $f(\tau \cap \sigma) = f(\tau) \cap f(\sigma)$, and
2. preserves dimensions of simplices, i.e. $\dim(\sigma) = \dim(f(\sigma))$ for all $\sigma \in \Delta \setminus \{\emptyset\}$.

In particular, every abstract simplicial complex of dimension d can be realised as a complex with simplices in $\mathbb{R}^{2d+1}$, see e.g. the Geometric Realisation Theorem in Computational Topology: An Introduction [1].

For the purpose of relating to data analysis, we use terminology from persistent homology literature and talk of *pointclouds*.

Definition 3.2.6 (Pointcloud)**.** A *pointcloud* is a finite subset of a metric space.

We now give two examples of abstract simplicial complexes that can be generated from pointclouds that are especially useful in data analysis.

Definition 3.2.7 (Čech complex)**.** Let X be a pointcloud in a metric space with metric d. The *Čech complex* on X with radius ε is the abstract simplicial complex

$$\check{C}_\varepsilon(X) \coloneqq \left\{ \sigma \subseteq X \mid \bigcap_{x \in \sigma} \overline{B}_\varepsilon(x) \neq \emptyset \right\}, \tag{3.1}$$

where $\overline{B}_\varepsilon(x)$ is the closed ball of radius ε centered at x.

A similar complex that is more often used in practice [7] is the *Vietoris-Rips complex*.

Definition 3.2.8 (Vietoris-Rips complex)**.** Let X be a pointcloud. The *Vietoris-Rips complex* on X with radius ε is the abstract simplicial complex

$$V_\varepsilon(X) \coloneqq \{\sigma \subseteq X \mid d(x,y) \leq 2\varepsilon \ \forall x, y \in \sigma\}. \tag{3.2}$$

Note that for $\varepsilon \leq \varepsilon'$ we have $\check{C}_\varepsilon(X) \subseteq \check{C}_{\varepsilon'}(X)$ and $V_\varepsilon(X) \subseteq V_{\varepsilon'}(X)$, a property we will use further in section 3.4. For now, we go on to define homology using simplicial complexes.

3.3 Simplicial homology

Definition 3.3.1 (Chains)**.** Let K be a simplicial complex. A p-chain is a formal linear combination

$$\sum_i a_i \sigma_i$$

of p-simplices $\sigma_i \in K$ with coefficients a_i in some ring. We define addition of p-chains component-wise, i.e.

$$\sum_i a_i \sigma_i + \sum_i b_i \sigma_i = \sum_i (a_i + b_i) \sigma_i.$$

Since the coefficients form an additive group, this gives us the *group of p-chains* $\mathrm{C}_p = \mathrm{C}_p(K)$.

The specific ring of coefficients should be clear from context, and in persistent homology, it is often the two-element field $\mathbb{F}_2$.

Next, we define the notion of the boundary of a simplex, and, by extension, the boundary of a chain.

Definition 3.3.2 (Boundary map)**.** Given a p-simplex $\sigma = [v_0, \dots, v_p]$, we define its *boundary* to be the $(p-1)$-chain

$$\partial_p \sigma := \sum_{i=0}^{p} (-1)^i [v_0, \dots, v_{i-1}, v_{i+1}, \dots, v_p]$$

where the ith vertex has been removed from the ith term. We extend this linearly and define the boundary of a p-chain $c = \sum_i a_i \sigma_i$ to be

$$\partial_p c := \sum_i a_i \partial_p \sigma_i.$$

This gives us the group homomorphisms $\partial_p \colon \mathrm{C}_p \to \mathrm{C}_{p-1}$, which we call *boundary maps.*

An important property of boundary maps is that the boundary of a boundary is always zero:

Theorem 3.3.1. *Let σ be a $(p+1)$-simplex. Then $\partial_p \partial_{p+1} \sigma = 0$.*

Proof. Note that every term in the $p-1$-chain $\partial_p \partial_{p+1} \sigma$ is of the form $(-1)^k \hat{\sigma}$, where $\hat{\sigma}$ is σ but with exactly two vertices removed, since first ∂_{p+1} removes one vertex, after which ∂_p removes a second. For such a $\hat{\sigma}$, let the indices of the two removed vertices be α and β, and assume $\alpha < \beta$. Then, in $\partial_p \partial_{p+1} \sigma$, there will be exactly two terms containing $\hat{\sigma}$: one where v_α is removed before v_β, and one where v_β is removed before v_α. In the first case, the term must be $(-1)^{\beta-1}(-1)^{\alpha}\hat{\sigma}$, while in the other case it must be $(-1)^{\alpha}(-1)^{\beta}\hat{\sigma}$. These cancel out, and this holds for all such $\hat{\sigma}$. Thus $\partial_p \partial_{p+1} \sigma = 0$. □

We give an example of boundaries below.

Example 3.3.1. Take the ring of coefficients to be $\mathbb{Z}$ and consider the simplicial complex with three vertices v_0, v_1, v_2 and two edges $e_0 = [v_0, v_1]$, $e_1 = [v_1, v_2]$. Then we have $\partial_1 e_0 = [v_1] - [v_0]$ and $\partial_1 e_1 = [v_2] - [v_1]$. Thus, the boundary of the chain $e_0 + e_1$ is $[v_2] - [v_0]$. Similarly, if we have a complex with two triangles $t_0 = [v_0, v_1, v_2]$ and $t_1 = [v_1, v_3, v_2]$, we see that the boundary $\partial_2(t_0 + t_1)$ is the 1-chain $[v_0, v_1] + [v_1, v_3] + [v_3, v_2] - [v_0, v_2]$, as in figure 3.2. Note that the contribution of the edge $[v_1, v_2]$ in the boundaries of t_0 and t_1 cancels in $\partial_2(t_0 + t_1)$ due to their opposite sign. This reflects the intuitive notion that $[v_1, v_2]$ is not part of the boundary of the triangulation that t_0 and t_1 form together. The orientation, i.e., the ordering of vertices in t_0 and t_1, is crucial for this to work out correctly.

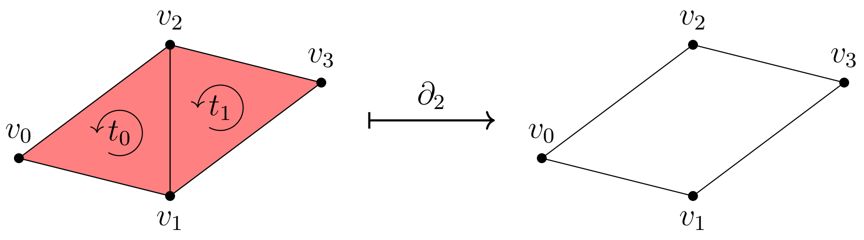


Figure 3.2: Example of the boundary map on a 2-chain. Left: the 2-chain $t_0 + t_1 = [v_0, v_1, v_2] + [v_1, v_3, v_2]$. Right: the 1-chain $\partial_2(t_1 + t_2) = [v_0, v_1] + [v_1, v_3] + [v_3, v_2] - [v_0, v_2]$.

We now define two special cases of chains, that we will then use to define homology.

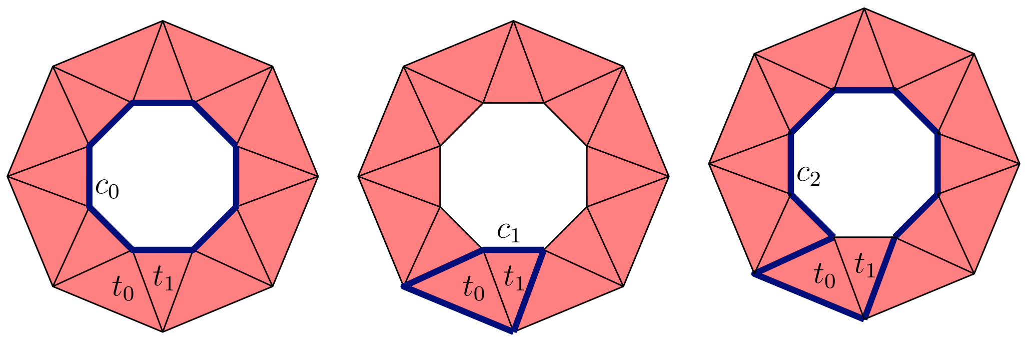
Definition 3.3.3 (Cycles)**.** A *p-cycle* is a p-chain whose boundary is zero. The *group of p-cycles* is $\mathrm{Z}_p = \mathrm{Z}_p(K) := \ker \partial_p$.

As shown in example 3.3.1, if v_0, v_1, v_2 are distinct, the boundary of $c := [v_0, v_1] + [v_1, v_2]$ is not 0, thus c not a cycle. However, the boundary of $d := [v_0, v_1] + [v_1, v_2] + [v_2, v_0]$ *is* 0, and so d is a cycle. Compare this to the fact that the underlying set $[v_0, v_1] \cup [v_1, v_2]$ is a non-closed curve, while $[v_0, v_1] \cup [v_1, v_2] \cup [v_2, v_0]$ is closed. In the two dimensional case, $[v_0, v_1, v_2] + [v_1, v_3, v_2]$ is not a cycle, but any 2-chain whose underlying set is homeomorphic to the 2-sphere is a 2-cycle, and similarly in higher dimensions.

Definition 3.3.4 (Boundaries)**.** A *p-boundary* is a p-chain that is the boundary of some $(p+1)$-chain, i.e. $\sigma \in K$ is a p-boundary if and only if there exists a simplex τ such that $\partial_{p+1}\tau = \sigma$. The *group of p-boundaries* is $\mathrm{B}_p = \mathrm{B}_p(K) := \operatorname{im} \partial_{p+1}$.

Note that all p-boundaries are p-cycles because of theorem 3.3.1, so B_p is a subgroup of Z_p. Below follows an example of a cycle that is *not* a boundary.

Example 3.3.2. Let K have three vertices v_0, v_1, v_2, three edges $[v_0, v_1]$, $[v_0, v_2]$, $[v_1, v_2]$, and one triangle $[v_0, v_1, v_2]$. Then the 2-chain $[v_0, v_1] + [v_1, v_2] - [v_0, v_2]$ is a cycle and also a boundary (of the triangle). Now consider the same complex but without the triangle; in this case, the above 2-chain is a cycle, but it is not a boundary.



(a) *A 1-cycle that is not the boundary of any 2-chain.*

(b) *A 1-cycle that is a boundary of a 2-chain.*

(c) *A 1-cycle that is homologous to the one in the left-most subfigure.*

Figure 3.3: Examples of 1-cycles, boundaries, and homologous cycles.

This example suggests that cycles that are not boundaries capture some notion of *holes* in the given simplicial complex. Removing the triangle from the complex introduced a hole, or a "tunnel" through the complex, and likewise caused a boundary to become a non-bounding cycle. The following definitions makes this notion precise.

Definition 3.3.5 (Homology groups)**.** The *pth (simplicial) homology group* of a simplicial complex K is the quotient group

$$\mathrm{H}_p(K) \coloneqq \mathrm{Z}_p(K)/\mathrm{B}_p(K).$$

If the complex is clear from context, we write H_p instead of $\mathrm{H}_p(K)$. The elements of H_p are called *homology classes*, and any two cycles in the same homology class are said to be *homologous*. The rank of H_p is the *pth Betti number*, denoted β_p.

Thus, two cycles are homologous if and only if their difference is a boundary. Usually, H_p intuitively captures a notion of "p-dimensional holes" in the complex. This can be made precise in some cases via abelianisations of homotopy groups, which we do not describe in detail. With this intuitive perspective, we have that β_1 is the number of holes whose "boundary" is a 1-cycle, such as the hole in a circle or the middle of a torus, and β_2 is the number of holes whose "boundary" is a 2-cycle, such as the empty interior of a sphere. β_0, however, does not count holes, but can instead be seen to be the number of connected components.

Example 3.3.3. Figure 3.3 shows a triangulation of the annulus, and highlights the 1-cycles c_0, c_1, and c_2, as well as the triangles t_0 and t_1. The cycle c_0 is not a boundary while c_1 is the boundary of the 2-chain $t_0 + t_1$. Note that $c_2 = c_0 + c_1$, and $c_1 \in \mathrm{B}_1$ so c_2 lies in the coset $c_0 + \mathrm{B}_1$. Of course $c_0 \in c_0 + \mathrm{B}_1$ also; thus c_0 and c_2 are homologous. Intuitively, these figures suggest that homologous cycles correspond to the same hole.

We now present a theorem that will help us motivate the use of persistent homology in the next section.

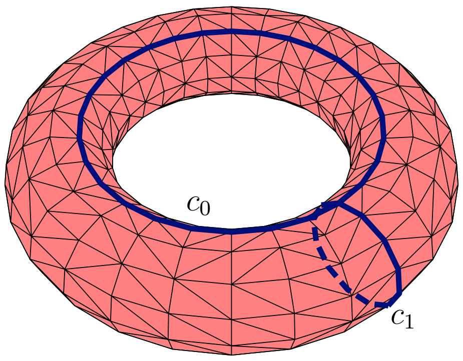


Figure 3.4: A triangulated torus, with two highlighted 1-cycles c_0 and c_1.

Theorem 3.3.2. *If K and L are simplicial complexes with homotopy equivalent underlying spaces, then $\mathrm{H}_p(K) \cong \mathrm{H}_p(L)$ for all p.*

A proof of this theorem can be found in the book Algebraic Topology by Hatcher [14, Corollary 2.11]. As a consequence, all complexes that triangulate a triangulable space have the same homology groups, and so it makes sense to speak of the homology of the space itself without respect to any specific triangulation. In this text, we implicitly assume that all spaces we consider are triangulable, unless otherwise specified.

To exemplify Betti numbers, we end this section by presenting those of the torus and relating them to a triangulation.

Example 3.3.4. Figure 3.4 shows a triangulation of the torus together with two 1-cycles c_0 and c_1 on it. It is well-known that the Betti numbers of the torus are $\beta_0 = 1$, $\beta_1 = 2$, $\beta_2 = 1$, and $\beta_p = 0$ for $p > 2$, and we can convince us of this by considering the aforementioned figure. Since the torus has one connected component, we must have $\beta_0 = 1$. Next, both c_0 and c_1 are 1-cycles that are not the boundary of any 2-chain, and further, c_0 and c_1 do *not* differ by a 1-boundary — they are not homologous — so there are at least two distinct generators of H_1. In fact, there are precisely two generators, so $\beta_1 = 2$. Lastly, there is only one non-zero 2-cycle, namely the sum of all triangles in the triangulation, and since there are no 3-simplices in this complex, this 2-cycle is not a boundary. Thus it generates H_2, and $\beta_2 = 1$.

Of course, since there are no p-simplices for $p > 2$, $\beta_p = 0$ for all such p.

3.4 Persistent homology

Our goal is to give a formal definition of persistent homology, as well as to link this description with intuition. Consider the sets $X \subset \overline{X} \subseteq \mathbb{R}^d$ where X is finite and $\overline{X}$ is some triangulable space. This situation often arises in practice when the points in X are points of data measured in some experiment, and $\overline{X}$ is the true domain of the data. Ideally, we can construct a simplicial complex with vertices in X that is homotopy equivalent to $\overline{X}$, for instance by constructing a Čech or Vietoris-Rips complex with a specific radius ε. If so, we are able to compute the homology of $\overline{X}$ only using X, which may be useful for further analysis of the data. This is however too idealistic, so we describe *persistent homology*, a formalism that aims to solve the problem of constructing the homology of $\overline{X}$ given only X.

The key idea of persistent homology is that tracking the homology groups of a sequence of simplicial complexes on X can reveal key properties of $\overline{X}$. As a concrete example, we can choose a sequence of nonnegative real numbers

$$\varepsilon_0 < \varepsilon_1 < \cdots < \varepsilon_n$$

and get a corresponding sequence of abstract simplicial complexes such as

$$\check{C}_{\varepsilon_0}(X) \subseteq \check{C}_{\varepsilon_1}(X) \subseteq \cdots \subseteq \check{C}_{\varepsilon_n}(X)$$

or

$$V_{\varepsilon_0}(X) \subseteq V_{\varepsilon_1}(X) \subseteq \cdots \subseteq V_{\varepsilon_n}(X).$$

We can then construct a sequence of homology groups on (realisations of) these complexes and study their behaviour. The sequence of complexes needs not be Čech or Vietoris-Rips, but can be any sequence. We call such a sequence a *filtered simplicial complex.*

Definition 3.4.1 (Filtered simplicial complex)**.** A *filtered simplicial complex* is a pair $(K, \mathcal{F})$ where K is a simplicial complex and $\mathcal{F} = (K_i)_{i=0}^n$ is a finite increasing filtration of K, that is, $K_i \subseteq K\ \forall i$ and $K_i \subseteq K_j$ iff $i \leq j$. Given a simplex $\sigma \in K$, we define its *degree* $\deg(\sigma)$ to be the index i such that $\sigma \in K_i \setminus K_{i-1}$, or 0 if $\sigma \in K_0$. We will often denote a filtered simplicial complex $(K, \mathcal{F})$ simply as K.

For a filtered complex K we obtain for $i \leq j$ the inclusion maps

$$\widetilde{\eta}_p^{i,j} : \mathrm{Z}_p(K_i) \hookrightarrow \mathrm{Z}_p(K_j).$$

These induce the homomorphisms

$$\eta_p^{i,j} : \mathrm{H}_p(K_i) \to \mathrm{H}_p(K_j) \tag{3.3}$$

such that the following diagram commutes:

$$\begin{tikzcd}
\mathrm{Z}_p(K_i) \arrow[r, hook, "\widetilde{\eta}_p^{i,j}"] \arrow[d, "\pi_i"'] & \mathrm{Z}_p(K_j) \arrow[d, "\pi_j"] \\
\mathrm{H}_p(K_i) \arrow[r, "\eta_p^{i,j}"] & \mathrm{H}_p(K_j)
\end{tikzcd}$$

Here, π_i and π_j are the canonical projections mapping cycles to their corresponding equivalence classes. That is, $\eta_p^{i,j}$ maps a homology class in $\mathrm{H}_p(K_i)$ to the class in $\mathrm{H}_p(K_j)$ that contains it.

By combining these maps we get a sequence of homology groups associated with the filtration of K,

$$\mathrm{H}_p(K_0) \xrightarrow{\eta_p^{0,1}} \mathrm{H}_p(K_1) \xrightarrow{\eta_p^{1,2}} \dots \xrightarrow{\eta_p^{n-1,n}} \mathrm{H}_p(K_n).$$

The idea of persistent homology is to track homology classes through this sequence. To this end we define *persistent homology groups* and their corresponding Betti numbers.

Definition 3.4.2 (Persistent homology groups)**.** Let K be a filtered simplicial complex, and let $\eta_p^{i,j} \colon \mathrm{H}_p(K_i) \to \mathrm{H}_p(K_j)$ be the homomorphism induced by the inclusion $K_i \hookrightarrow K_j$. The *pth persistent homology group* and *pth persistent Betti number* for $i \leq j$ is

$$\mathrm{H}_p^{i,j} = \mathrm{H}_p^{i,j}(K) := \operatorname{im} \eta_p^{i,j} \qquad \text{and} \qquad \beta_p^{i,j} := \operatorname{rank} \mathrm{H}_p^{i,j}.$$

Persistent homology groups, like the homology groups of the individual subcomplexes K_i, do not necessarily reflect the topological features of the original space X. It is rather the properties of $\mathrm{H}_p^{i,j}$ together with the maps $\eta_p^{i,j}$ that are relevant. This leads us to the concept of *barcodes*, and to define this we need to explain what it means for a homology class to be born and to die.

Definition 3.4.3 (Birth and death)**.** Let K be a filtered simplicial complex, $\mathrm{H}_p^{i,j}$ its persistent homology groups, and $\eta_p^{i,j}$ the induced homomorphisms as in equation 3.3. We say that an element $\gamma \in \mathrm{H}_p(K_i)$ is *born* at index i if

$$\gamma \in \mathrm{H}_p(K_i) \setminus \mathrm{H}_p^{i-1,i}(K). \tag{3.4}$$

further, we say that an element γ that is born at index i *dies* at index $j > i$ if

$$\eta_p^{i,j-1}(\gamma) \notin \mathrm{H}_p^{i-1,j-1} \quad \text{and} \quad \eta_p^{i,j}(\gamma) \in \mathrm{H}_p^{i-1,j}. \tag{3.5}$$

The birth and death of an element can be visualised with the following diagram:

$$\begin{tikzcd}
\mathrm{H}_p(K_{i-1}) \arrow[r] & \mathrm{H}_p(K_i) \arrow[r] & \mathrm{H}_p(K_{j-1}) \arrow[r] & \mathrm{H}_p(K_j) \\
 & \gamma \arrow[u, phantom, sloped, "\in"] \arrow[d, phantom, sloped, "\notin"] \arrow[r, mapsto] & \eta_p^{i,j-1}(\gamma) \arrow[u, phantom, sloped, "\in"] \arrow[d, phantom, sloped, "\notin"] \arrow[r, mapsto] & \eta_p^{i,j}(\gamma) \arrow[u, phantom, sloped, "\in"] \arrow[d, phantom, sloped, "\in"] \\
\mathrm{H}_p^{i-1,i-1} \arrow[r] & \mathrm{H}_p^{i-1,i} \arrow[r] & \mathrm{H}_p^{i-1,j-1} \arrow[r] & \mathrm{H}_p^{i-1,j}
\end{tikzcd}$$

Note that $\mathrm{H}_p^{i-1,i-1} = \mathrm{H}_p(K_{i-1})$. The element γ is born at index i since $\gamma \in \mathrm{H}_p(K_i) \setminus \mathrm{H}_p^{i-1,i}$. In other words, γ is a class at filtration step i, but it is not in the image of the homology group at step $i-1$, so it must be a *new* class in this sense. Next, $\eta_p^{i,j-1}(\gamma) \in \mathrm{H}_p(K_{j-1}) \setminus \mathrm{H}_p^{i-1,j-1}$ means that, at index $j-1$, the image of γ still is distinct from the image of any other element of $\mathrm{H}_p(K_{i-1})$. In other words, γ is still distinct from all classes that were alive before γ was born. Finally, $\eta_p^{i,j}(\gamma) \in \mathrm{H}_p^{i-1,j}$ means that, at index j, the image of γ is now equal to the image of some element existing already at index $i-1$, so now γ has merged with some class that was alive before γ was born.

Definition 3.4.4 (Persistence)**.** For a given filtered simplicial complex, if a class γ is born at index i and dies at index j, we say that the *persistence* of γ is $j-i$. If there is no j such that γ dies at index j, we say that the persistence of γ is ∞.

Intuitively, the birth of a class at index i means that as we add new simplices when going from K_{i-1} to K_i a new homology class, i.e. a new class of non-bounding cycles (or a p-dimensional hole), is formed in the complex. The death of a class at index j means that as we add new simplices going from K_{j-1} to K_j, the class merges with another homology class that existed already in $\mathrm{H}_p(K_{i-1})$. A class with high persistence is one that exists in a large part of the filtration while one with low persistence is short-lived.

The lifetime of a homology class can more precisely be represented using persistence intervals.

Definition 3.4.5 (Persistence intervals)**.** A *persistence interval* is a pair (i, j) with $0 \leq i < j \in \mathbb{Z} \cup \{\infty\}$.

A homology class in a filtered simplicial complex born at i that dies at j can then be represented with the persistence interval (i, j), where a class that does not die corresponds to the interval (i, ∞). The idea of a *barcode* is to represent the persistent homology of a filtered complex with such intervals. In general, however, when a class γ is born, so are its multiples, and of course these also die at the same index. Thus we get an infinite number of persistence intervals if the ring of coefficients is infinite. Under certain conditions, however, the persistent homology of a filtered complex can be described with a finite number of persistence intervals. This was shown by Zomorodian and Carlsson [15], and we present their theory below.

Definition 3.4.6 (Persistence modules)**.** Let R be a ring. A *persistence module $\mathcal{M}$ over R* is a family of pairs $\mathcal{M} = (M_i, \varphi_i)_{i=0}^{\infty}$ where each M_i is an R-module and each $\varphi_i : M_i \to M_{i+1}$ is a module homomorphism.

Of special interest, in order to define barcodes, are persistence modules of *finite type*.

Definition 3.4.7 (Finite type)**.** A persistence module $\mathcal{M}$ is of *finite type* if each component module M_i is finitely generated, and the maps φ_i are isomorphisms for all $i \geq m$ for some $m \in \mathbb{N}$.

Given a persistence module $\mathcal{M}$ over a ring R we define a graded module over $R[t]$,

$$\alpha(\mathcal{M}) \coloneqq \bigoplus_{i=0}^{\infty} M_i. \tag{3.6}$$

Here the action of t on an element of the module is defined as

$$t \cdot (m_0, m_1, m_2, \dots) = (0,\ \varphi_0(m_0),\ \varphi_1(m_1),\ \varphi_2(m_2),\ \dots). \tag{3.7}$$

This action "shifts" module elements one step up in the grading as defined by the direct sum.

If the ring of coefficients R is not a field, there is no simple classification of persistence modules [15]. If we instead consider homology over a field of coefficients F, persistence modules admit a simple decomposition. For a persistence interval (i, j), define

$$q(i,j) \coloneqq \begin{cases} t^i F[t]/\langle t^{j-i}\rangle & \text{if } j < \infty, \\ t^i F[t] & \text{otherwise,} \end{cases}$$

where $\langle t^{j-i}\rangle$ is the principal ideal generated by t^{j-i}. Finally, for a finite multiset of persistence intervals $\mathcal{B} = ((i_0, j_0), (i_1, j_1), \dots, (i_k, j_k))$, where the same persistence interval may appear multiple times, let

$$Q(\mathcal{B}) \coloneqq \bigoplus_{l=0}^{k} q(i_l, j_l).$$

Then we have the following result by Zomorodian and Carlsson.

Theorem 3.4.1 (Correspondence)**.** *The map $\mathcal{B} \mapsto Q(\mathcal{B})$ defines a bijection between finite multisets of persistence intervals and isomorphism classes of persistence modules of finite type over the graded ring $F[t]$.*

We are now ready to define barcodes.

Definition 3.4.8 (Barcodes of persistence modules of finite type)**.** Let $\mathcal{M}$ be a persistence module of finite type. Then the *barcode* of $\mathcal{M}$ is the finite multiset $\mathcal{B}$ of persistence intervals such that

$$\alpha(\mathcal{M}) \cong Q(\mathcal{B})$$

with α and Q defined as above.

We can naturally construct persistence modules of finite type from the homology of a filtered simplicial complex.

Definition 3.4.9 (Barcodes of filtered simplicial complexes)**.** Let K be a filtered simplicial complex with a finite number of simplices, and $\mathrm{H}_p(K_i)$ its homology groups with coefficients in a field F, and let $p \geq 0$. Assume that the filtration has $n+1$ levels $(K_i)_{i=0}^{n}$. Define a persistence module $\mathcal{H}_p \coloneqq (\mathrm{H}_p^i,\ \eta_p^i)_{i=0}^{\infty}$ where

$$\mathrm{H}_p^i \coloneqq \begin{cases} \mathrm{H}_p(K_i) & \text{if } i \leq n, \\ \mathrm{H}_p(K_n) & \text{if } i > n, \end{cases}$$

and

$$\eta_p^i := \begin{cases} \eta_p^{i,i+1} & \text{if } i \leq n, \\ \mathrm{id}_{\mathrm{H}_p(K_n)} & \text{if } i > n. \end{cases}$$

Since K has a finite number of simplices, $\mathcal{H}_p$ is a persistence module of finite type. Then the *barcode of K in dimension p* is the barcode of $\mathcal{H}_p$, which we denote $\mathcal{B}_p$.

In the graded module $\alpha(\mathcal{H}_p)$, the action of t is

$$t \cdot (\gamma_0, \gamma_1, \gamma_2, \dots) = (0,\ \eta_p^{0,1}(\gamma_0),\ \eta_p^{1,2}(\gamma_1),\ \eta_p^{2,3}(\gamma_2),\ \dots) \tag{3.8}$$

and corresponds to moving all homology classes one step forward in the filtration. Note that each persistence interval $(i, j) \in \mathcal{B}_p$ corresponds to a basis element of $\alpha(\mathcal{H}_p)$, and in fact, this basis element generates all homology classes that are born at index i and die at index j. Thus, barcodes describe the persistent homology of a filtered complex as long as homology is considered over a field and the complex is finite. Barcodes are often visualised by plotting each persistence interval as a horizontal line segment with start and end positions in the x-coordinate being the birth and death indices of the interval. See figure 3.6 and example 3.4.1.

We end this section with an example of a filtration for which we detail the theory in this section.

Example 3.4.1. Consider the set X of ten points in figure 3.5a. The points are placed uniformly on a circle with radius 1, offset by additive normally distributed noise. We construct a filtered Vietoris-Rips complex $(K, \mathcal{F})$ where $K = V_1(X)$ and $\mathcal{F} = (V_{0.01k}(X))_{k=0}^{100}$. Figure 3.6 shows the barcodes in dimensions 0 and 1 for this filtration; $\mathcal{B}_0$ is represented with solid lines, and $\mathcal{B}_1$ with dashed lines. The barcode $\mathcal{B}_0$ has ten intervals born at radius 0, indicating that $\mathrm{H}_0(V_0(X))$ has rank 10, which is indeed the case since it consists of ten connected components. At radius $\varepsilon = 0.45$, the complex (figure 3.5b) has one component and one whole, so we have $\beta_0 = \beta_1 = 1$. We get the same information from the barcodes, since both $\mathcal{B}_0$ and $\mathcal{B}_1$ have a single interval crossing the point $\varepsilon = 0.45$, indicating a single generator in each of the two dimensions. Finally, at radius $\varepsilon = 1$ the complex (figure 3.5d) has a 2-chain covering the hole, so $\mathrm{H}_1(V_1(X))$ is trivial, while $\mathrm{H}_0(V_1(X))$ has a single generator. This can be seen in the barcode since the dashed line has died at a smaller radius, and the only class still persisting is in $\mathcal{B}_0$.

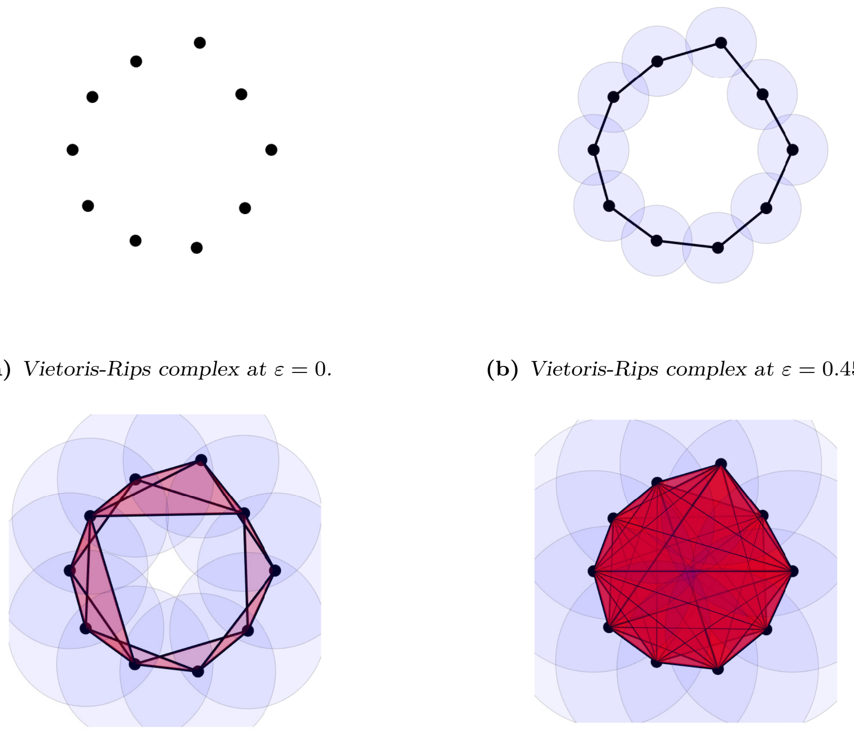

(a) *Vietoris-Rips complex at* $\varepsilon = 0$. **(b)** *Vietoris-Rips complex at* $\varepsilon = 0.45$. **(c)** *Vietoris-Rips complex at* $\varepsilon = 0.75$. **(d)** *Vietoris-Rips complex at* $\varepsilon = 1$.

Figure 3.5: Vietoris-Rips complexes built on a set of 10 points, at various radii ε. Also shown are disks of radius ε centered at each point.

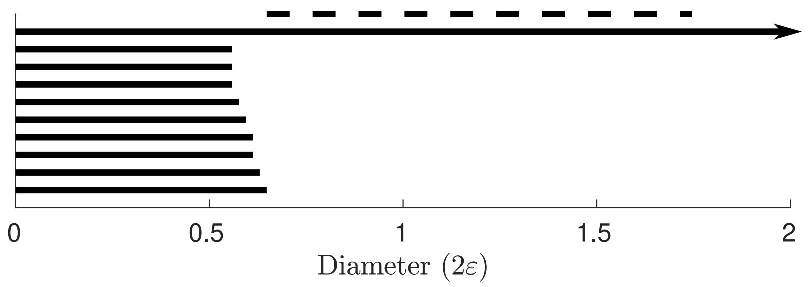


Figure 3.6: The barcodes of a Vietoris-Rips complex on the ten points shown in figure 3.5a in dimensions 0 and 1. In total, the the filtration consists of 100 subcomplexes with values of ε in the interval $[0, 2]$. Solid lines represent $\mathcal{B}_0$ while the dashed line represents the only interval in $\mathcal{B}_1$. The arrowhead indicates that the interval is infinite, not dying at any index.

4. Computing persistent homology

While homology can be defined with coefficients in any ring, if we restrict the coefficients to a field, persistent homology can computed efficiently. The standard algorithm for computing persistent homology (algorithm 2) was first developed by Edelsbrunner and Zomorodian [16] for homology over $\mathbb{F}_2$, and then extended to arbitrary fields by Carlsson and Zomorodian [15].

Given a filtered simplicial complex K, one orders the simplices $\sigma_i \in K$ such that $\sigma_i < \sigma_j$ implies that $\deg(\sigma_i) \leq \deg(\sigma_j)$, and that σ_j is not a face of σ_i. With the ordered simplices, one then defines the boundary matrix ∂.

Definition 4.0.10 (Boundary matrix)**.** Given a filtered simplicial complex and an ordering on the simplices σ_i, the *boundary matrix* of the complex is the matrix ∂ with elements

$$\partial_{i,j} = \begin{cases} 1 & \text{if } \sigma_i \text{ is a face of } \sigma_j \text{ of one dimension less,} \\ 0 & \text{otherwise.} \end{cases}$$

Note that, if the simplices are ordered as required above, boundary matrices are strictly upper triangular. On a boundary matrix ∂, we define the mapping $j \mapsto \text{low}_\partial(j)$ as follows:

Definition 4.0.11 ($\text{low}_\partial(\cdot)$)**.** Let ∂ be an n-by-n matrix. We define the function $j \mapsto \text{low}_\partial(j)$ for $j = 0, \ldots, n-1$ by

$$\text{low}_\partial(j) := \begin{cases} \max \{i \mid \partial_{i,j} \neq 0\} & \text{if } \partial_j \neq 0, \\ -1 & \text{otherwise.} \end{cases}$$

If the matrix is clear from context, we may omit the index parameter and just write $\text{low}(\cdot)$.

Here we follow the convention adopted in persistent homology literature, and write ∂_j to mean the jth column of ∂. Persistent homology can then be computed by reducing a boundary matrix with column additions until it is *reduced*.

Definition 4.0.12 (Reduced matrix)**.** A matrix ∂ is *reduced* if $\text{low}_\partial(\cdot)$ is injective on the non-zero columns of ∂.

Algorithm 2 is the typical example of how to compute persistent homology. This algorithm is guaranteed to converge and to result in a valid reduced matrix, from which the persistence intervals can be read. The column additions performed by the standard algorithm are usually called *left-to-right* operations, since we only ever add column k to column j if $k < j$, i.e. if column k is to the left of column j.

Algorithm 2: Standard algorithm over arbitrary fields

```
  Input: An $n \times n$ boundary matrix $\partial$
1 for $j = 0, \dots, n-1$ do
2     while $\exists k < j$ such that $\text{low}_\partial(k) = \text{low}_\partial(j) > -1$ do
3         $i \leftarrow low_\partial(j)$
4         $\alpha \leftarrow \partial_{i,j} / \partial_{i,k}$
5         $\partial_j \leftarrow \partial_j - \alpha \partial_k$
6     end while
7 end for
8 return $\partial$
```

Often, persistent homology is computed over $\mathbb{F}_2$, in which case the standard algorithm simplifies to algorithm 3.

Algorithm 3: Standard algorithm over $\mathbb{F}_2$

```
  Input: An $n \times n$ boundary matrix $\partial$ over $\mathbb{F}_2$
1 for $j = 0, \dots, n-1$ do
2     while $\exists k < j$ such that $\text{low}_\partial(k) = \text{low}_\partial(j) > -1$ do
3         $\partial_j \leftarrow \partial_j + \partial_k$
4     end while
5 end for
6 return $\partial$
```

From a reduced matrix r, we can read the persistence intervals by means of the function $\text{low}_r(\cdot)$.

Definition 4.0.13 (Simplex classes)**.** Let r be a reduced boundary matrix and consider the map $\text{low}_r(\cdot)$ defined on it. If r_j is nonzero, we call σ_j *negative*, and $\sigma_{\text{low}(j)}$ *positive*. Simplices that are neither positive nor negative are called *essential*.

Positive simplices are those that, when born, create a new persistent homology class. The corresponding negative simplex, when born, causes the death of that class. Essential simplices cause the creation of a homology class that never dies later in the filtration. So for every j such that $r_j \neq 0$, we get the interval $[\deg(\sigma_{\text{low}_r(j)}), \deg(\sigma_j))$, and for every j such that $r_j = 0$ and $\nexists k : \text{low}_r(k) = j$, we get $[\deg(\sigma_j), \infty)$. Further, a persistence interval with minimum $\deg(\sigma_i)$ corresponds to a homology class of the same dimension as σ_i.

Example 4.0.2. Consider a pointcloud of three points arranged in a triangle and let K contain the triangle between the points, its edges, and its vertices. Then we may define a filtration $K_0 \subseteq K_1 \subseteq K_2 = K$ as in figure 4.1.

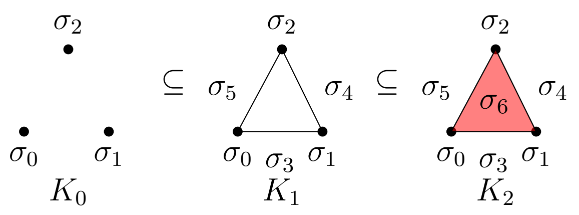


Figure 4.1: A filtration of a simplicial complex consisting of one triangle.

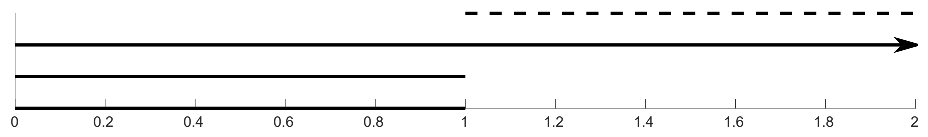


Figure 4.2: The barcode of the filtration in figure 4.1. Solid lines are connected components, the dashed line is a one-dimensional hole.

Note that the ordering $\sigma_0 < \cdots < \sigma_6$ respects the filtration: simplices introduced in K_i are smaller than those introduced in K_{i+1} (e.g. $\sigma_0 < \sigma_3$). The ordering also respects the faces of the simplices: every simplex is larger than its faces (e.g. $\sigma_3 < \sigma_6$). With this ordering, this filtration has the boundary matrix

$$\partial = \begin{pmatrix} 0&0&0&1&0&1&0 \\ 0&0&0&1&1&0&0 \\ 0&0&0&0&1&1&0 \\ 0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0 \end{pmatrix}.$$

The standard algorithm gives the reduced matrix r in two iterations as follows:

$$\partial = \begin{pmatrix} 0&0&0&1&0&1&0 \\ 0&0&0&1&1&0&0 \\ 0&0&0&0&1&1&0 \\ 0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0 \end{pmatrix} \mapsto \begin{pmatrix} 0&0&0&1&0&1&0 \\ 0&0&0&1&1&1&0 \\ 0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0 \end{pmatrix} \mapsto \begin{pmatrix} 0&0&0&1&0&0&0 \\ 0&0&0&1&1&0&0 \\ 0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0 \end{pmatrix} = r.$$

Note that, in r, lowest nonzeroes only exist in rows 1, 2, and 5, and no two columns have their lowest nonzero in the same row, so $\text{low}_r(\cdot)$ is injective on the nonzero columns, as required.

We can now read the persistence intervals from the reduced matrix. We get four intervals: $[\deg(\sigma_1), \deg(\sigma_3)) = [0, 1)$, $[\deg(\sigma_2), \deg(\sigma_4)) = [0, 1)$, $[\deg(\sigma_5), \deg(\sigma_6)) = [1, 2)$, and $[\deg(\sigma_0), \infty) = [0, \infty)$. The intervals starting at $\deg(\sigma_0) = \deg(\sigma_1) = \deg(\sigma_2)$ measure connected components; the interval starting at $\deg(\sigma_5)$ measures a 1-dimensional hole, since σ_5 is 1-dimensional. With all this, we can construct the barcode in figure 4.2. We see that we have three connected components initially, but only one persists infinitely, and we have one hole that is born in K_1 and killed in K_2. Intuitively, the edge σ_5 closes the loop, creating the hole, while the triangle σ_6 fills the hole, killing it.

4.1 Previous work

Chen and Kerber improved on the standard algorithm by introducing the clearing lemma [17], and constructed the algorithm now commonly known as *twist*. The twist algorithm loops over the dimensions of the simplices in decreasing order, and employs the clearing technique, but is otherwise similar to the standard algorithm, reducing one column at a time. Bauer, Kerber and Reininghaus constructed a parallelisable algorithm [18], sometimes called *chunk*, that partitions the columns into chunks, reduces some columns within each chunk, and then performs standard reduction on the remaining unreduced columns.

Mendoza-Smith and Tanner constructed the `pms` algorithm [8], that builds on the concepts by Chen and Kerber [17] and Bauer et al. [18]. The `pms` algorithm differs from the aforementioned algorithms in that it iteratively reduces matrices, and instead of reducing one column at a time, it aims to operate on many columns each iteration, partially reducing them every step. Informally, it aims to make $\text{low}_\partial(\cdot)$ converge to $\text{low}_r(\cdot)$ of the reduced matrix r uniformly instead of pointwise. While column additions are plentiful and can be done in parallel, `pms` only uses fully reduced columns for additions, which it identifies in each iteration with sequential loops over the matrices' columns. Our algorithm instead identifies leftmost columns, which are a superset of the reduced columns, exposing more column additions per iteration. Mendoza-Smith and Tanner show how information about known reduced columns can be used to estimate the set of essential simplices, that is columns that lead to infinite persistence intervals $[\deg(\sigma_j), \infty)$. Because our algorithm does not know which columns are reduced, we cannot directly use the same method.

Many problems involving sparse matrices can be difficult to implement. In the article by Elsman et al. that introduces the `expand` function, they show how expansion can be used to compute products between sparse matrices and dense vectors or dense matrices, while acknowledging that multiplication between two sparse matrices is a more difficult problem. Other authors, e.g. Pizzuti et al. [19], further investigate the generation of efficient sparse matrix vector multiplication from high-level code. Apart from these, there seems to be a lack of more general sparse matrix computations in similar high-level functional languages.

5. A parallel algorithm

The standard algorithm for boundary matrix reduction (algorithm 2) sweeps the matrix from left to right, reducing the columns one at a time. Since it only performs left-to-right column additions, when it adds ∂_k to ∂_j, we have that $k < j$. Thus, it *only adds reduced columns to other columns.*

In truth however, we do not need any knowledge of which columns are reduced to perform additions. In 2006, Cohen-Steiner et al. proved the *pairing uniqueness lemma* [20] (lemma 5.1.1 below), but it was not until 2020 that Morozov and Nigmetov pointed out the lemma's possible application in exposing new parallelism in the reduction of boundary matrices [21]. The lemma shows that columns can be added in any order, rather than reducing them one at a time, and this allows us to operate on many columns simultaneously. We use this result, together with a simple observation about a commonly used technique by Chen and Kerber [17], to construct a massively parallel boundary matrix reduction algorithm. Below, we construct our algorithm for computing persistent homology over $\mathbb{F}_2$, although it is generalisable to arbitrary fields.

5.1 Adding columns in parallel

The standard algorithm is often seen as defining a matrix factorisation $r = \partial\phi$ of the input boundary matrix ∂, where r is reduced. Indeed, adding column ∂_i to ∂_j is equivalent to setting $\phi_{i,j} = 1$. More precisely, we have $r_j = \sum_{i=0}^{n-1} \phi_{i,j}\partial_i$. Note then that the standard algorithm gives a factorisation where ϕ is invertible upper triangular: start with $\phi = I$ the identity matrix, and for each column addition $\partial_j \leftarrow \partial_j + \partial_k$, set $\phi_{k,j} = 1$. Since additions are left-to-right we have $k < j$ always, so ϕ is upper triangular, and since its diagonal entries are nonzero, it is invertible. It turns out that such a factorisation is all we need to compute persistent homology, which is captured by the following lemma by Cohen-Steiner et al. [20].

Lemma 5.1.1 (Pairing uniqueness)**.** *Let $\partial\phi$ be a reduced matrix where ϕ is invertible upper triangular. Then* $\text{low}_{\partial\phi}(\cdot)$ *is independent of ϕ.*

Thus the notewothy implication of lemma 5.1.1 is, as Morozov and Nigmetov put it, that *it does not matter in what order we add the columns.* Any combination of left-to-right column additions that leads to a reduced matrix (i.e. $\partial\phi$ such that $\text{low}_{\partial\phi}(\cdot)$ is injective on nonzero columns) will give the same correct persistence intervals. In particular, we do not need to add reduced columns to other columns, as in the standard algorithm, but we can choose columns to add more freely.

There are many ways to use this freedom. In their paper, Morozov and Nigmetov use the lemma by proposing a partially lockfree reduction algorithm for shared memory multicore systems, based on dividing the matrix into one chunk per thread, each chunk being reduced from left to right as in the standard algorithm. We instead propose a chunkless approach, operating on a very large number of columns in parallel. To this end, we partition the nonzero columns in ∂ into *leftmost* columns and their corresponding *neighours*.

Definition 5.1.1. For a matrix ∂, the set of *leftmost* columns is

$$\mathcal{L} \coloneqq \{j \mid \nexists k < j : \mathrm{low}_\partial(k) = \mathrm{low}_\partial(j) > -1\}.$$

Given $\ell \in \mathcal{L}$, its *neighbours* is the set

$$\mathcal{N}(\ell) \coloneqq \{j > \ell \mid \mathrm{low}_\partial(j) = \mathrm{low}_\partial(\ell)\}.$$

The algorithm we propose is to iteratively add ∂_ℓ to ∂_j for all $\ell \in \mathcal{L}$ and $j \in \mathcal{N}(\ell)$ until the matrix is reduced. See algorithm 4 below for imperative pseudocode.

Algorithm 4: Parallel column additions

```
  Input: An $n \times n$ boundary matrix $\partial$ over $\mathbb{Z}_2$
1 while $\partial$ is not reduced do
2     $\mathcal{L} \leftarrow \{j \mid \nexists k < j : \mathrm{low}_\partial(k) = \mathrm{low}_\partial(j) > -1\}$
3     for $\ell \in \mathcal{L}$ do
4         for $j \in \mathcal{N}(\ell)$ do
5             $\partial_j \leftarrow \partial_j + \partial_\ell$
6         end for
7     end for
8 end while
9 return $\partial$
```

Note that both computing $\mathcal{L}$ and adding columns can be done in parallel; finding leftmost columns can be done independently for each row, and column additions are independent of each other.

5.2 Incorporating the clearing lemma

The *clearing* technique is an often used optimisation in boundary matrix reduction which allows us to set some columns to zero without explicitly reducing them [17]. This result, commonly called the clearing lemma, was originally stated informally as an observation of persistent homology. Here we instead present and prove the lemma without any reference to homology.

Lemma 5.2.1 (Clearing). *Let ∂ be a strictly upper triangular matrix such that $\partial^2 = 0$. Let r be a reduced form of ∂, and let j be the index of a nonzero column in r. Then it holds that the column with index* $\mathrm{low}_r(j)$ *in r is zero.*

Proof. First assume that ∂ is reduced. Then, column j in ∂^2 equals $\sum_{i=0}^{n-1} \partial_{i,j}\partial_i$, which is zero since $\partial^2 = 0$. Further, since ∂ is reduced, $\mathrm{low}_\partial(\cdot)$ is injective on the nonzero columns, implying that the nonzero columns in ∂ are linearly independent. Thus, if $\partial_{i,j} \neq 0$, have $\partial_i = 0$. In particular, if $\partial_j \neq 0$, we have $\partial_{\mathrm{low}(j),j} \neq 0$, and so $\partial_{\mathrm{low}(j)} = 0$.

Now assume that ∂ is not reduced. Then a reduction of ∂ is of the form $\partial\phi$ where ϕ is an invertible upper triangular matrix, and so ϕ^{-1} is also invertible upper triangular. Note that, analogously to the way that multiplying with ϕ on the right corresponds to left-to-right column additions, multiplying with ϕ^{-1} from the left corresponds to down-to-up row additions. Note also that such row additions must preserve low_∂, i.e. $\mathrm{low}_{\phi^{-1}\partial} = \mathrm{low}_\partial$. Let now $e := \phi^{-1}\partial\phi$. Then $\mathrm{low}_{\partial\phi} = \mathrm{low}_e$, so e is reduced; further, $e^2 = 0$, so the lemma holds by the previous argument. □

The clearing lemma is a standard optimisation; many existing algorithms invoke the clearing lemma once they have guaranteed a column to be reduced. We propose, however, that the lemma can be invoked at any time, without knowledge about which columns have already been reduced.

Corollary 5.2.1. *Let ∂ be a strictly upper triangular matrix such that $\partial^2 = 0$, and let j be the index of a nonzero column in ∂. Let r be a reduced form of ∂. Then it holds that the column with index $\mathrm{low}_\partial(j)$ in r is zero.*

Proof. Consider the standard algorithm (algorithm 2) and a column index j. First assume that ∂_j is already reduced, i.e. $\partial_j = r_j$. Then, by the clearing lemma, the column in r with index $\mathrm{low}_\partial(j)$ is zero. Now assume instead that ∂_j is not reduced. Then, according to the standard algorithm, there must exist $k < j$ such that $\partial_k = r_k$ and $\mathrm{low}_\partial(k) = \mathrm{low}_\partial(j)$. Since ∂_k is reduced, it holds by the clearing lemma that the column in r with index $\mathrm{low}_\partial(j)$ is zero. Finally, the pairing uniqueness lemma tells us that this result is independent of the reduction algorithm, since the result is only about the map $\mathrm{low}_r(\cdot)$, which must be uniquely determined. □

This implies a strong form of the clearing technique: at any stage during reduction, we may simply set column $\mathrm{low}_\partial(j)$ to zero for *every* column index j, if $\mathrm{low}_\partial(j) \neq -1$. We do this at every iteration in our algorithm, so incorporating this into algorithm 4 gives us algorithm 5 below.

This is the final form of the algorithm that we have implemented. We now give a proof that algorithm 5 converges.

Theorem 5.2.1. *Algorithm 5 converges and yields a reduced matrix.*

Proof. Corollary 5.2.1 guarantees that the clearing loop at line 2 is valid.

Now, assume we have a nonreduced matrix ∂. Then there is some $\ell \in \mathcal{L}$ and $j \in \mathcal{N}(\ell)$

Algorithm 5: The final form of our implemented algorithm

```
   Input: An $n \times n$ boundary matrix $\partial$ over $\mathbb{F}_2$
1  while $\partial$ is not reduced do
2      for $j = 0, \dots, n-1$ do
3          Set column $\mathrm{low}_\partial(j)$ to zero
4      end for
5      $\mathcal{L} \leftarrow \{j \mid \nexists k < j : \mathrm{low}_\partial(k) = \mathrm{low}_\partial(j) > -1\}$
6      for $\ell \in \mathcal{L}$ do
7          for $j \in \mathcal{N}(\ell)$ do
8              $\partial_j \leftarrow \partial_j + \partial_\ell$
9          end for
10     end for
11 end while
12 return $\partial$
```

and we know that j is an unreduced column. Thus, the algorithm will perform the column addition $\partial_j \leftarrow \partial_j + \partial_\ell$, meaning that $\mathrm{low}_\partial(j)$ will strictly decrease during this iteration. In general, $\mathrm{low}_\partial(j)$ *must* strictly decrease for some j at every iteration. But $\mathrm{low}_\partial(j) \geq -1\ \forall j$, so after a finite number of iterations, $\mathrm{low}_\partial(j)$ must have reached its minimum for all j, at which point the matrix must be reduced. Since all additions performed are left-to-right, lemma 5.1.1 guarantees correctness. □

5.3 A functional perspective

The above pseudocode is useful for comparing our algorithm with other boundary matrix algorithms in the literature, but before we give an implementation in Futhark we present (an example of) algorithm 5 in conceptual Futhark code. Assume we have a type `matrix`. Then we may do the clearing with a function `clear` of type `matrix` $\rightarrow$ `matrix` that sets the appropriate columns to zero.

For representing $\mathcal{L}$, we may define a lookup table `arglows` such that `arglows`$[i] = j$ iff j is the minimum column index such that $\mathrm{low}_\partial(j) = i$, otherwise `arglows`$[i] = -1$; we may build `arglows` with a function `leftmost_lookup` of type `matrix` $\rightarrow$ `[]i64`. This way, assuming $\partial_j \neq 0$, if we have `arglows`$[\mathrm{low}_\partial(j)] = j$, then we know that $j \in \mathcal{L}$; otherwise, `arglows`$[\mathrm{low}_\partial(j)] < j$ and $j \in \mathcal{N}($`arglows`$[\mathrm{low}_\partial(j)])$. With this in mind, convergence can be checked by verifying that every column is either zero or in $\mathcal{L}$.

Finally, we assume a function `add_columns` of type `matrix` $\rightarrow$ `[]i64` $\rightarrow$ `matrix` that, given ∂ and `arglows`, performs all left-to-right column additions available. Then we get the following representation of algorithm 5:

```
let reduce_matrix (∂: matrix): matrix =
  loop ∂ while !(is_reduced ∂) do
    let ∂' = clear ∂
    let arglows = leftmost_lookup ∂'
    in add_columns ∂' arglows
```

5.4 Implementation

5.4.1 Matrix data structure

The boundary matrix of a d-dimensional complex with n simplices will have a total of n^2 entries but at most $n(d+1)$ nonzero entries. So if n is significantly larger than d, which is a reasonable assumption if we wish to analyse large pointclouds, the boundary matrix is *sparse*; a large portion of its elements are zero. If we then use a sparse matrix data structure that only stores the nonzero entries, we may get $O(n)$ space complexity instead of $O(n^2)$. We will often mention the *number of nonzero elements* $\mathrm{nnz}(\partial)$ of a matrix ∂, and similarly the number of nonzero elements $\mathrm{nnz}(\partial_j)$ of a single column.

Many numerical problems are sparse in nature, so a number of different sparse matrix data structures have been developed. In our implementation we use a variant of the *sparse compressed column* (CSC) format. In a typical CSC structure, one stores three arrays to represent a matrix ∂ of size $m \times n$: arrays `column_offsets` of length n, and `row_indices` and `values` both of length $\mathrm{nnz}(\partial)$. We now give an example of a matrix in the standard CSC format, storing offsets but not lengths.

Example 5.4.1. The matrix

$$\partial = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 6 \\ 0 & 0 & 3 & 5 \\ 7 & 0 & 0 & 7 \\ 1 & 0 & 8 & 9 \end{pmatrix}$$

has a CSC representation

$$\begin{aligned} \texttt{column\_offsets} &= [0, 3, 3, 6] \\ \texttt{row\_indices} &= [0, 3, 4, 1, 2, 4, 1, 2, 3, 4] \\ \texttt{values} &= [2, 7, 1, 1, 3, 8, 6, 5, 7, 9] \end{aligned}$$

assuming zero-based indexing. To extract a specific column ∂_j, we first find its offset $p = \texttt{column\_offsets}[j]$, and its number of nonzeroes $\mathrm{nnz}(\partial_j) = \texttt{column\_offsets}[j+1] - \texttt{column\_offsets}[j]$ (although for the last column we instead have $\mathrm{nnz}(\partial_3) = \mathrm{nnz}(\partial) - \texttt{column\_offsets}[3]$). Then, the column is described by `row_indices` and `values` at the indices p through $p + \mathrm{nnz}(\partial_j) - 1$. For instance, the first column has offset 0 and number of nonzeroes $3 - 0 = 3$. Its row indices are then $[0, 3, 4]$ and its values are $[2, 7, 1]$. The number of nonzeroes of the second column is $3 - 3 = 0$, meaning it is empty.

To do matrix reduction over $\mathbb{F}_2$, our matrices are binary and so we can omit the `values` array if we ensure that `row_indices` only correponds to true nonzero elements.

As demonstrated, with the standard CSC format the number of nonzeroes in a column can be found by simply subtracting column offsets. However, we modify the CSC format and explicitly store an array `column_lengths` in our structure. This is because we need the possibility for the array `row_indices` to have a length of more than $\mathrm{nnz}(\partial)$; why we need this will be explained later.

Example 5.4.2. With our variant of the CSC format, the matrix

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

may be represented as

$$\begin{aligned} \texttt{column\_offsets} &= [0, 0, 3, 7] \\ \texttt{column\_lengths} &= [0, 2, 2, 3] \\ \texttt{row\_indices} &= [0, 1, \varepsilon, 0, 2, \varepsilon, \varepsilon, 1, 2, 3] \end{aligned}$$

where ε is any dummy value that will never be accessed.

In our variant of the CSC format, `column_lengths` stores the number of nonzero elements for each column. Note that, in example 5.4.2, `row_indices` contains more elements than the total number of nonzeroes. We also require that, for all j, column j is described by the `column_lengths`$[j]$ *first* elements of `row_indices` starting at index `column_offsets`$[j]$. The elements with value ε are simply padding at the end of the column. For instance, the sequence of row indices $(0, 1, \varepsilon)$ in the example above is a valid description of the first column, while the sequence $(0, \varepsilon, 1)$ is not valid. We represent this with a Futhark type:

```
type csc_mat =
  { col_offsets: []i64
  , col_lengths: []i64
  , row_idxs:    []i32
  }
```

We can find $\mathrm{low}_\partial(j)$ in constant time if we also keep the row index array sorted column-wise. Then $\mathrm{low}_\partial(j)$, being the largest row index of a nonzero element, is simply the last entry in `row_idxs` for column j:

```
let low (∂: csc_mat) (j: i64): i64 =
  if ∂.col_lengths[j] == 0
    then -1
    else i64.i32 ∂.row_idxs[ ∂.col_offsets[j] +
                             ∂.col_lengths[j] - 1 ]
```

Keeping the row indices sorted column-wise is also essential for our approach to column additions, see section 5.4.4.

5.4.2 Clearing

To avoid recomputing often, we explicitly store the array `lows` describing $\text{low}_\partial(\cdot)$ separate from the matrix, and we do the same with the lookup table `arglows` that describes $\mathcal{L}$. To compress type signatures, then, we define a record type:

```
type~ state [n] =
  { matrix:   csc_mat
  , lows:     [n]i64
  , arglows:  [n]i64
  }
```

The first stage of the main loop in our algorithm is to invoke the clearing lemma. In our implementation, to check if a column j is zero or not, we simply check whether `lows`$[j] = -1$. Thus, to clear a column we simply need to ensure that this holds; we do not need to modify the column in the actual matrix. We then do clearing as follows:

```
let clear [n] (s: state[n]): state[n] =
  let lows' = scatter (copy s.lows)
                      s.lows
                      (replicate n (-1))
  in s with lows = lows'
```

In the application of `scatter` above, we create an index-value pair $(\text{low}_\partial(j), -1)$ for each j, thus setting `lows`$[\text{low}_\partial(j)]$ to -1 as required. Recall that `scatter` ignores out-of-bounds indices.

5.4.3 Identifying pairs of columns to add

Once clearing has been done, we build the set $\mathcal{L}$ of leftmost columns; that is, we identify all indices j such that $\nexists k < j : \text{low}_\partial(k) = \text{low}_\partial(j) > -1$. A convenient way of finding one minimum per row in Futhark is to use `reduce_by_index`:

```
let update_lookup [n] (s: state[n]): state[n] =
  let arglows' =
    reduce_by_index (replicate n i64.highest)
                    i64.min
                    i64.highest
                    s.lows
                    (iota n)
  in s with arglows = arglows'
```

In this function, we use the minimum function on 64-bit integers `i64.min` as reduction function, with neutral element `i64.highest` (equal to $2^{63}-1$). For every index i into the destination `replicate n i64.highest`, we are finding the minimum value in `iota n` such that its corresponding index in `s.lows` equals i. In other words, we are finding the smallest column index j such that `s.lows`$[j] = i$. For instance, if we have a matrix with 4 columns and `s.lows` $= [2, 1, 3, 2]$ we have that

```
reduce_by_index (replicate 4 i64.highest)
                i64.min
                i64.highest
```

```
                    [2,1,3,2]
                    [0,1,2,3]
```

evaluates to $[2^{63}-1,1,0,2]$. Note that no column has its lowest nonzero in row 0, so the first element is $2^{63}-1$. This is fine; we will never attempt to access the lookup table at this index in this case.

5.4.4 Pairwise column additions

We begin by partitioning the indices of nonzero columns into those that are leftmost columns (i.e. that will not be added to) and those that are neighbours of some leftmost column (i.e. those that *will* be added to).

```
let (neighbour_idxs, leftmost_idxs) =
  (iota n) |> filter (λj → s.lows[j] != −1)
           |> partition (λj → s.arglows[j]] != j)
```

Before performing column additions, we use `s.lows` and `s.arglows` to identify all pairs of columns that should be summed. We define the array `left_right_pairs` of type `[](i64,i64)` with (k,j) an element iff the addition $\partial_j \leftarrow \partial_j + \partial_k$ will be performed in the current iteration.

```
let left_right_pairs =
  map (λj → (s.arglows[s.lows[j]], j))
      neighbour_idxs
```

The next step is to set $\partial_j \leftarrow \partial_j + \partial_k$ for every (k,j)-pair in `left_right_pairs`. This is made nontrivial by using a sparse matrix structure together with the fact that we cannot know the number of nonzeroes in the column $\partial_j + \partial_k$ before computing it — a problem also encountered in sparse matrix-matrix multiplication. We cannot do column additions in-place since there may not be enough space. Instead, we take the approach of calculating upper bounds on the new number of nonzeroes and creating a new matrix with the capacity to hold that many elements. Then we populate its columns, possibly leaving some unused space. Note that if $(k,j) \in$ `left_right_pairs`, then $\mathrm{low}_\partial(j) = \mathrm{low}_\partial(k)$, so

$$\mathrm{nnz}(\partial_j + \partial_k) \leq \mathrm{nnz}(\partial_j) + \mathrm{nnz}(\partial_k) - 2$$

and we can use this as the size of the new columns. When creating a new matrix, we gives its array of row indices a length of

$$\sum_{(k,j)\in\texttt{left\_right\_pairs}} (\mathrm{nnz}(\partial_j) + \mathrm{nnz}(\partial_k) - 2) + \sum_{j\in\texttt{leftmost\_idxs}} \mathrm{nnz}(\partial_j).$$

We initialise this array with elements equal to some unspecified value ε before populating it with the new columns.

Example 5.4.3. Consider the following matrix and corresponding CSC representation:

$$\partial = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix}$$

$$\begin{aligned} \texttt{column\_offsets} &= (0,2,5) \\ \texttt{column\_lengths} &= (2,3,1) \\ \texttt{row\_indices} &= (0,2,0,1,2,0) \end{aligned}$$

Suppose we wish to compute the matrix

$$\partial' = \begin{pmatrix} \partial_0 & \partial_0 + \partial_1 & \partial_2 \end{pmatrix}.$$

Since $\text{nnz}(\partial_0) = 2$ and $\text{nnz}(\partial_1) = 3$, $\text{nnz}(\partial_0 + \partial_1) \leq 2 + 3 - 2 = 3$, and we may initialise the new matrix as

$$\begin{aligned} \texttt{column\_offsets'} &= (0, 2, 5) \\ \texttt{row\_indices'} &= (0, 2, \varepsilon, \varepsilon, \varepsilon, 0). \end{aligned}$$

After computing the column sum we see that $\text{nnz}(\partial_0 + \partial_1) = 1$ and its only nonzero row index is 1; we write this into the new matrix and get

$$\begin{aligned} \texttt{column\_offsets'} &= (0, 2, 5) \\ \texttt{row\_indices'} &= (0, 2, 1, \varepsilon, \varepsilon, 0). \end{aligned}$$

Now we know the lengths of each column, so we can set

$$\texttt{column\_lengths'} = (2, 1, 1).$$

Note how there are two unused elements with value ε, that aren't contained in any column as indicated by the lengths and offsets.

We divide the process of computing the new matrix, given `left_right_pairs`, into three distinct steps: first we create a new CSC matrix ∂' with an array `row_indices'` large enough to store the results of column additions; second, we copy the leftmost columns into this new matrix, since they won't change; third, we add all (j, k)-pairs and write the results into the new matrix.

5.4.4.1 Initialising the new matrix

Given the initial matrix ∂ and the array `left_right_pairs` we create a new CSC matrix with the following function:

```
let init_new_matrix (∂: csc_mat)
                    (left_right_pairs: [](i64,i64))
                  : csc_mat =
  let neighbour_idxs = (unzip left_right_pairs).1
  let new_col_lengths_bounds =
    scatter (copy ∂.col_lengths)
            neighbour_idxs
            (map (λ(k,j) → ∂.col_lengths[k] +
                            ∂.col_lengths[j] − 2)
                 left_right_pairs)
  let new_col_offsets =
    [0] ++ init (scan (+) 0 new_col_lengths_bounds)
  let new_row_idxs =
    replicate (i64.sum new_col_lengths_bounds) (−1)
  in { col_offsets = new_col_offsets
     , col_lengths = new_col_lengths_bounds
     , row_idxs    = new_row_idxs
     }
```

`scatter` pairs each $j \in$ `neighbour_idxs` with $\mathrm{nnz}(\partial_k) + \mathrm{nnz}(\partial_j) - 2$, as described above, and writes these values into a copy of ∂`.col_lengths`. That is,

$$\begin{aligned}&\texttt{new\_col\_lengths\_bounds}[j]\\ &= \begin{cases}\mathrm{nnz}(\partial_j) + \mathrm{nnz}(\partial_k) - 2 & \text{if } (k, j) \in \texttt{left\_right\_pairs},\\ \mathrm{nnz}(\partial_j) & \text{otherwise.}\end{cases}\end{aligned}$$

This gives an array of the upper bounds of the number of nonzeroes in the new matrix. A prefix sum on this array then gives the new matrix' column offsets, and the sum of the bounds gives the total number of possible row indices that can be written, so this is the length of the new array of row indices.

5.4.4.2 Copying the leftmost columns

Now we define a function that takes the initial matrix ∂, the newly initialised matrix ∂', and the array `leftmost_idxs`, and copies all leftmost columns from ∂ to ∂'.

```
let copy_columns (leftmost_idxs: []i64)
                 (∂: csc_mat)
                 (∂': csc_mat)
               : csc_mat =
  let (is,as) =
    expand
      (λj → ∂.col_lengths[j])
      (λj α → (∂'.col_offsets[j] + α
              , ∂.row_idxs[∂.col_offsets[j] + α]))
      leftmost_idxs
    |> unzip
  let new_row_idxs =
    scatter (copy ∂'.row_idxs) is as
  in ∂' with row_idxs = new_row_idxs
```

In the above code, we use `expand` to create index-value-pairs (with indices in `is` and values in `as`), that we then scatter into the row index array. Each $j \in$ `leftmost_idxs` is expanded into a value array containing the nonzero row indices of ∂_j, and an index array containing the indices of ∂'`.row_idxs` that corresponds to ∂'_j.

Example 5.4.4. Assume we have

$$\partial = \begin{pmatrix}1 & 1 & 1\\ 0 & 1 & 0\\ 1 & 1 & 0\end{pmatrix} \qquad \begin{aligned}\partial\texttt{.col\_offsets} &= (0, 2, 5)\\ \partial\texttt{.col\_lengths} &= (2, 3, 1)\\ \partial\texttt{.row\_idxs} &= (0, 2, 0, 1, 2, 0)\end{aligned}$$

and a newly initialised matrix

$$\partial' = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \qquad \begin{aligned} \partial'\texttt{.col\_offsets} &= (0,2,5) \\ \partial'\texttt{.col\_lengths} &= (2,3,1) \\ \partial'\texttt{.row\_idxs} &= (\varepsilon,\varepsilon,\varepsilon,\varepsilon,\varepsilon,\varepsilon) \end{aligned}$$

and that $\texttt{leftmost\_idxs} = [0,2]$, so that we wish to copy the first and last columns of ∂ into ∂'. Then `expand` works as follows: The first element $j = \texttt{leftmost\_idxs}[0] = 0$ is expanded into $\partial\texttt{.col\_lengths}[j] = 2$ index-value-pairs, namely

$$(\partial'\texttt{.col\_offsets}[j] + 0,\ \partial\texttt{.row\_idxs}[\partial\texttt{.col\_offsets}[j] + 0]) = (0,0)$$

and

$$(\partial'\texttt{.col\_offsets}[j] + 1,\ \partial\texttt{.row\_idxs}[\partial\texttt{.col\_offsets}[j] + 1]) = (1,2).$$

In other words, we will scatter the row index values 0 and 2 into $\partial'\texttt{.row\_idxs}$ at indices 0 and 1, respectively; this amounts to copying the first column. Similarly, the second leftmost index $j = \texttt{leftmost\_idxs}[1] = 2$ is expanded into $\partial\texttt{.col\_lengths}[j] = 1$ index-value-pair, namely

$$(\partial'\texttt{.col\_offsets}[j] + 0,\ \partial\texttt{.row\_idxs}[\partial\texttt{.col\_offsets}[j] + 0]) = (5,0)$$

meaning we will copy the single nonzero element of the third column. In total, `expand` gives us the index-value-pairs $[(0,0),(1,2),(5,0)]$. Scattering these index-value-pairs finally gives us $\texttt{new\_row\_idxs} = [0,2,\varepsilon,\varepsilon,\varepsilon,0]$.

5.4.4.3 Adding columns

Finally, we need to compute the column sums indicated by `left_right_pairs`. As mentioned, we keep the invariant that the row indices in the CSC matrix are sorted column-wise, and we utilise this in our approach to adding columns. Given two columns, i.e. two arrays of row indices, possibly of different lengths, we use a sequential merge algorithm to find the sum of the columns. While a normal merge algorithm simply computes a sorted union of two arrays, we compute the sorted XOR of two arrays, since we are summing columns over $\mathbb{F}_2$. For each pair $(k,j) \in \texttt{left\_right\_pairs}$, we keep a pointer `px` into the nonzero row indices of ∂_k, and likewise the pointers `py` and `pz` into ∂_j and ∂'_j respectively. If we let `cx` be the sorted array containing the nonzero row indices of ∂_k and similarly `cy` and `cz` for ∂_j and ∂'_j, the process of merging `cx` and `cy` into `cz` is described by the imperative pseudocode in algorithm 6 below.

Note that, in algorithm 6, when a pointer `px` or `py` is pointing past the bounds of the corresponding array, we set the row index `rx` or `ry` to ∞, so that the merging always writes the other, finite index. The effect is that once one row index becomes infinite, we simply write the remainder of the other column into `cz`. We do so because it simplifies the implmentation in Futhark, as will be seen further down. We also return the final value of `pz`, since this will equal the number of nonzero elements in the column sum, which we can use to update the `col_lengths` array.

Algorithm 6: Computing the XOR of sorted columns by sequential merging

Input: Two columns cx and cy over $\mathbb{F}_2$ to sum, and a mutable column cz to write the sum into. All columns are in the form of arrays containing the row indices of nonzero elements.

Output: The modified column cz, and the final state of the pointer pz.

```
1  px ← 0, py ← 0, pz ← 0
2  while px < length(cx) or py < length(cy) do
3      rx ← if px < length(cx) then cx[px] else ∞
4      ry ← if py < length(cy) then cy[py] else ∞
5      if rx < ry then
6          cz[pz] ← rx
7          px ← px + 1, pz ← pz + 1
8      else if ry < rx then
9          cz[pz] ← ry
10         py ← py + 1, pz ← pz + 1
11     else
12         px ← px + 1, py ← py + 1
13 end while
14 return cz and pz
```

To sum all column pairs in parallel, we would have to run many instances of algorithm 6 in parallel, one for each pair in `left_right_pairs`. These would then have to either write into the same array, namely ∂'`.row_idxs` but at different positions, or they would have to return their individual `cz` columns which would then be concatenated to form the final row index array of the new matrix. Neither of these approaches are currently possible in Futhark, however. Instead, we exchange the order of the outer parallel pass over the left-right-pairs and the inner sequential merge loop; we use an outer sequential loop that, in each iteration, performs a single step of the merge process for every left-right-pair in parallel. Since all columns may have a different number of nonzero elements, the outer loop then has to perform enough iterations to guarantee that every left-right-pair can be fully merged.

We describe the body of our parallel merge function in several steps, starting with the name and type:

```
let add_pairs [n0] (left_right_pairs: [n0](i64,i64))
                   (∂: csc_mat)
                   (∂': csc_mat)
                 : csc_mat =
```

In the following, then, `n0` is the number of left-right pairs. We start by extracting the neighbour column indices (i.e. those that will be written into) and copy the row indices of ∂' so that we can scatter into the array,

```
let neighbour_idxs = (unzip left_right_pairs).1
let row_idxs = copy ∂'.row_idxs
```

after which we define two useful arrays:

```
let offsets =
  map (λj → ∂'.col_offsets[j]) neighbour_idxs
let bounds =
  map (λj → ∂'.col_lengths[j]) neighbour_idxs
```

Recall that $j \in$ `neighbour_idxs` iff ∂'_j will be written into. If $j =$ `neighbour_idxs`$[\alpha]$, `offsets`$[\alpha]$ is simply the offset of column j in `row_idxs`, and `bounds`$[\alpha]$ is now the maximum number of nonzeroes that can possibly be written into the column. Then, the body of `add_pairs` is as follows:

```
let pxs = replicate n0 0
let pys = replicate n0 0
let pzs = replicate n0 0
let (row_idxs, _, _, pzs_final) =
  loop (row_idxs, pxs, pys, pzs)
  for i < i64.maximum bounds do
    let xs = tabulate n0 <| λα →
      get_elem_in_col left_right_pairs[α].0 pxs[α]
    let ys = tabulate n0 <| λα →
      get_elem_in_col left_right_pairs[α].1 pys[α]
    let row_idxs' =
      update_row_idxs i row_idxs pzs xs ys
    let (pxs', pys', pzs') =
      advance_pointers pxs pys pzs xs ys
    in (row_idxs', pxs', pys', pzs')

  let col_lengths =
    scatter (copy ∂'.col_lengths)
            neighbour_idxs
            pzs_final
  in ∂' with col_lengths = col_lengths
        with row_idxs    = row_idxs
```

To compare the loop in `add_pairs` with algorithm 6, let $j =$ `neighbour_idxs`$[\alpha]$ be the index of a column that will be written into. Then, `pxs`$[\alpha]$, `pys`$[\alpha]$, and `pzs`$[\alpha]$ correspond to the pointers `px`, `py`, and `pz` (pointers into ∂_k, ∂_j, and ∂'_j), while `xs`$[\alpha]$ and `ys`$[\alpha]$ correspond to `rx` and `ry` (nonzero row indices in ∂_k and ∂_j).

Before looping we initialise the arrays of pointers `pxs`, `pys`, and `pzs`. Then we loop over these pointers together with `row_idxs`, for i ranging from 0 to the largest length bound of all neighbour columns. At every iteration, for every $j \in$ `neighbour_idxs`, we either write a single element into ∂'_j, or do nothing, depending on the state of the pointers. If we write an element, we write it at position `pzs`$[\alpha]$ relative to the column's offset.

At the start of every iteration, we use `pxs` and `pys` to look up the elements they point to. We do this with the function

```
let get_elem_in_col (j: i64) (nz_idx: i64): i32 =
  if nz_idx < ∂.col_lengths[j]
  then ∂.row_idxs[∂.col_offsets[j] + nz_idx]
  else i32.highest
```

so that `xs`[α] is the row index of column `neighbour_idxs`[α], as long as the pointer `pxs`[α] is still within the length of that column — if it is not, we set `xs`[α] $\leftarrow$ `i32.highest`, which corresponds to setting `rx` $\leftarrow \infty$ in algorithm 6.

Once we have the elements `xs` and `ys`, for each α, we check whether `xs`[α] = `ys`[α]. If so, ∂_k and ∂_j have this row index in common, so it will not be in the column sum; thus we do nothing in this case. Otherwise, we write the minimum of `xs`[α] and `ys`[α] into ∂'_j: However we only do this if $i <$ `bounds`[α], since if this isn't the case, then the loop has iterated fully through column ∂'_j and there cannot be anything more to write:

```
let update_row_idxs [m] (i: i64)
                         (row_idxs: *[m]i32)
                         (pzs: [n0]i64)
                         (xs: [n0]i32)
                         (ys: [n0]i32)
                       : *[m]i32 =
  scatter' row_idxs (tabulate n0 <| λα →
    if bounds[α] <= i
    then (−1,−1)
    else let idx = offsets[α] + pzs[α]
         let v = if xs[α] == ys[α]
                 then −1 else i32.min xs[α] ys[α]
         in (idx,v))
```

Here, `scatter'` is a variant of `scatter` with type $[n]t \to ($`i64`$, t) \to [n]t$, that takes a zipped list of index-value pairs.

The last step of the loop is to increment all pointers:

```
let advance_pointers (pxs: [n0]i64)
                     (pys: [n0]i64)
                     (pzs: [n0]i64)
                     (xs: [n0]i32)
                     (ys: [n0]i32)
                   : ([n0]i64, [n0]i64, [n0]i64) =
  tabulate n0 (λα →
    if xs[α] == ys[α]
      then (pxs[α]+1, pys[α]+1, pzs[α])
    else if xs[α] < ys[α]
      then (pxs[α]+1, pys[α], pzs[α]+1)
    else  (pxs[α], pys[α]+1, pzs[α]+1) )
  |> unzip3
```

Once the loop has finished, we use the final state of the `pzs` pointers, now called `pzs_final` for emphasis, to set the actual column lengths of the resulting matrix ∂'. Recall that, in `init_new_matrix`, we set the column lengths of the neighbouring columns to the maximum upper bound on their sizes. After the loop, `pzs` indicates how many elements were actually written per column, and these numbers equal the final column lengths. Thus, we copy the initial column lengths of ∂', and scatter `pzs_final` into it.

5.4.5 Checking for convergence

Checking whether the matrix is reduced or not is simple: we just check whether, for every nonzero column ∂_j,

$$\texttt{s.arglows}[\texttt{s.lows}[j]] = j,$$

because if so, every nonzero column is a leftmost column. Thus we define the following:

```
let is_reduced [n] (s: state[n]): bool =
  all (λj → s.lows[j] == −1 ||
             s.arglows[s.lows[j]] == j)
      (iota n)
```

6. Experiments

We benchmark our implementation on some of the pointclouds used by Otter et al. [7]. For each pointcloud, we use Javaplex [22] to generate Vietoris-Rips complexes and their boundary matrices. We measure only the time taken to reduce the boundary matrices. All benchmarks are run on an Nvidia GTX 1070 GPU, and all runtimes presented are averages over 10 runs. The Futhark code is compiled to OpenCL. On the same datasets, we also benchmark the software OpenPH, which implements `pms` [8], an existing GPU-based persistent homology algorithm. OpenPH is written in CUDA, and we run all benchmarks on the same hardware.

We use six pointclouds (as collected and prepared by Otter et al.): **klein**: 400 points in 3 dimensions sampled from a figure-8 immersion of the Klein bottle; **dragon**: 1000 points in 3 dimensions sampled from the Stanford dragon 3D model; **fractal r**: 512 points in 259 dimensions generated from a certain self-similar network used to study brain connectivity; **hiv**: 1088 points in 673 dimensions corresponding to genomic sequences of the HIV virus genome; **celegans**: 297 points in 202 dimensions corresponding to neurons in the connectome of *c. elegans*; **genome**: 1397 points in 688 dimensions representing a sample of the human genome; and **h3n2**: 2722 points in 1173 dimensions corresponding to genomic sequences of the H3N2 virus.

For each pointcloud, we generate boundary matrices with about 10^6 columns. Table 6.1 shows the runtimes of our algorithm and OpenPH on these matrices. Further, for each pointcloud, we generate a number of smaller matrices; figure 6.1 shows, for each pointcloud, the runtime as a function of the number of columns in the matrix. The number of columns in each matrix is controlled by varying the radius used in constructing the Vietoris-Rips complexes. The largest matrix sizes for each dataset are the largest that Javaplex was able to generate on the hardware used.

Dataset	**klein**	**dragon**	**fractal r**	**hiv**
Number of columns	$2.1 \cdot 10^6$	$2.5 \cdot 10^6$	$1.5 \cdot 10^6$	$1.4 \cdot 10^6$
Runtime OpenPH [s]	7.53	10.22	2.68	1.33
Runtime ours [s]	2.16	2.03	0.58	0.51
Speedup	3.49	5.03	4.62	2.61

Dataset	**celegans**	**genome**	**h3n2**
Number of columns	$1.4 \cdot 10^6$	$3.8 \cdot 10^5$	$1.3 \cdot 10^6$
Runtime OpenPH [s]	1.37	0.52	1.57
Runtime ours [s]	0.60	0.13	0.47
Speedup	2.28	4	3.34

Table 6.1: Average runtime over 10 runs on Vietoris-Rips complexes generated from chosen pointclouds.

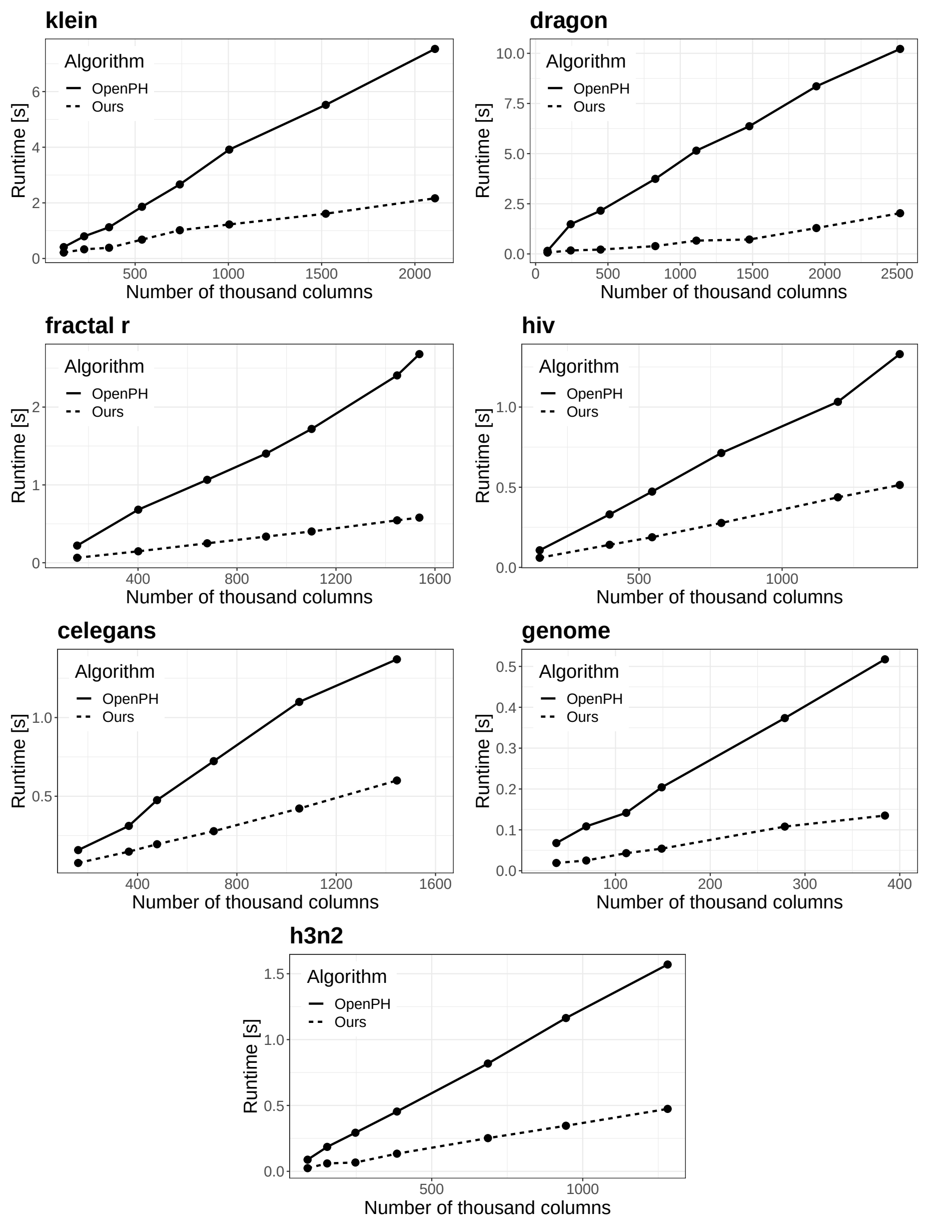


Figure 6.1: Average runtimes as a function of the number of columns for both OpenPH (solid lines) and our algorithm (dashed lines). One subfigure for each pointcloud. For each pointcloud, table 6.1 shows the runtimes in seconds for the largest matrix.

6.1 Early stopping

As defined by Mendoza-Smith and Tanner, let $\mathrm{low}_\partial^\ell$ be the map low as on the matrix yielded after reducing ∂ for ℓ iterations. If r is a reduction of ∂, we define the relative error

$$E^\ell = \frac{\left\|\mathrm{low}_\partial^\ell - \mathrm{low}_r\right\|_1}{\left\|\mathrm{low}_r\right\|_1} \tag{6.1}$$

between the approximated low after ℓ iterations and the true low. We run our algorithm on the datasets described above, for a varying number of iterations. Figure 6.2 then shows E^ℓ as a function of ℓ on the same boundary matrices as used in table 6.1. Further, table 6.2 presents the time and number of iterations required to reach $E^\ell \leq 10^{-k}$, and compares the times to those of table 6.1.

Table 6.2: The number of iterations required to reach $E^\ell \leq 10^{-2}$ on each of the boundary matrices used in table 6.1. ℓ and t are the minimum number of iterations and the time required, respectively, to reach $E^\ell \leq 10^{-2}$. ℓ_0 and t_0 are the number of iterations required and the time required, respectively, to fully reduce the matrix.

Dataset	**klein**	**dragon**	**fractal r**	**hiv**	**celegans**	**genome**	**h3n2**
ℓ	67	44	1	1	1	3	1
ℓ_0	252	195	16	6	3	9	4
t	1.22	1.12	0.41	0.37	0.43	0.091	0.35
t_0	2.16	2.03	0.58	0.51	0.60	0.13	0.47

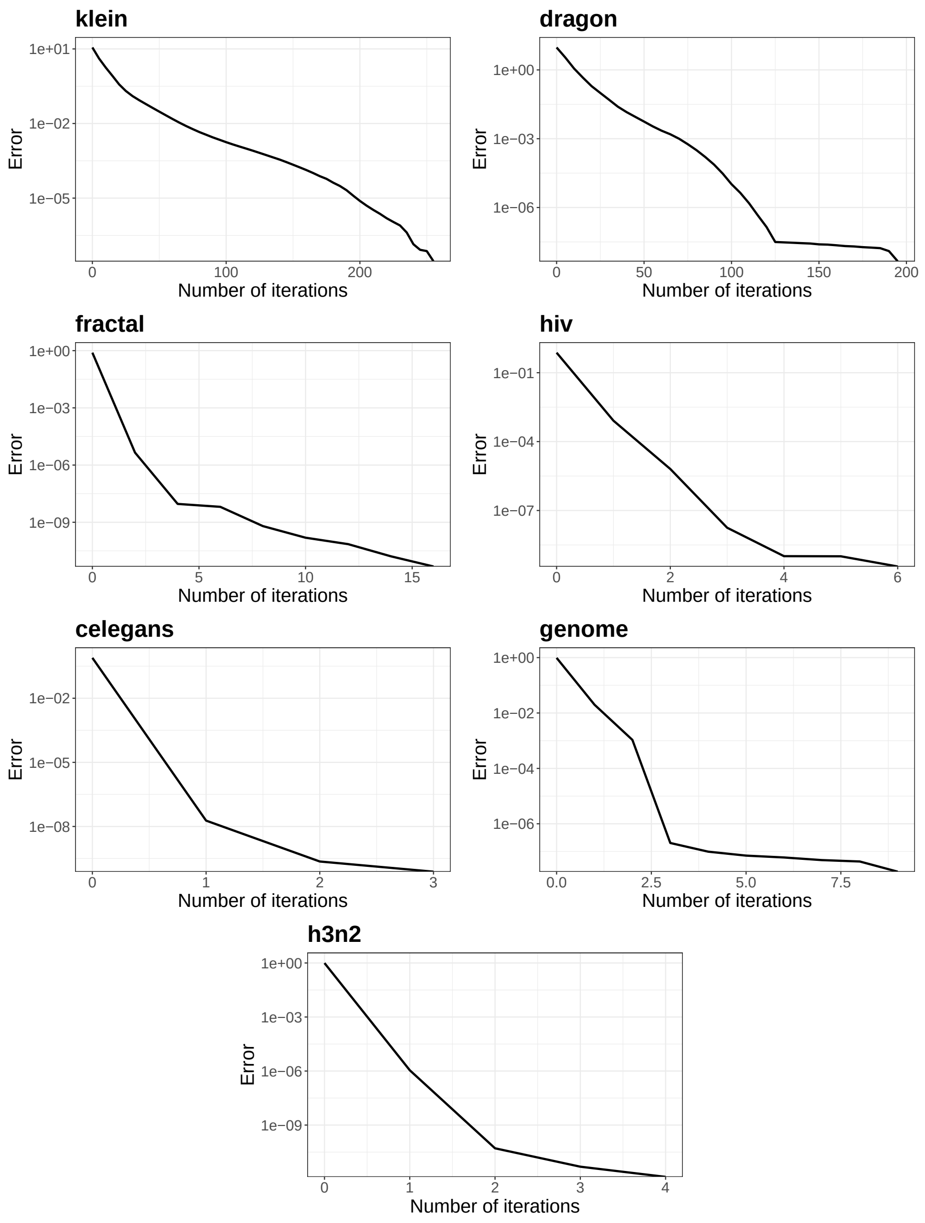


Figure 6.2: The error E^{ℓ} as a function of the number of iterations ℓ performed. One subfigure for each pointcloud. Note the log scale on the y-axes.

7. Discussion

The work presented in this thesis forms the basis for a paper to be presented at the ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing (FHPNC 2021) [23]. The source code of our implementation can be found at `https://github.com/erikvonb/futhark-ph`.

7.1 Our experiments

We have presented a massively parallel algorithm for boundary matrix reduction on GPU hardware. We have shown that it can be implemented in the functional language Futhark, with most effort going into parallel column additions in a sparse matrix structure. Our experiments show that this implementation is faster than `pms`, an existing boundary matrix reduction algorithm for GPUs. Apart from successfully reducing boundary matrices on GPU hardware, we have also shown that Futhark is a viable language to use for this kind of sparse matrix operation.

Figure 6.2 shows that the relative ℓ_1 error of low_∂ decreases very quickly during the first few iterations. Although these convergence rates are experimental rather than theoretical, the consistency across the seven different datasets suggests that fast convergence in early iterations may be a general feature of our algorithm. For a more thorough experimental study of convergence, other norms can be used. There also exist metrics on spaces of barcodes, such as the *Wasserstein distance* and its special case the *bottleneck distance* [7]. Such metrics can be used to define an error between the approximate low_∂^ℓ map and the true low_r.

7.2 Futher work

While our implementation is relatively simple, we would like to investigate whether it is possible to improve it further. The standard, twist and chunk algorithms have been efficiently implemented in the software package PHAT [24], and its authors have shown that matrix data structure has a major impact on performance. For instance, we can attempt to represent columns with bit sets, allowing the use of the bitwise XOR operation to add columns. Furthermore, future versions of Futhark may support *accumulators*, a feature that would allow us to map a column addition function over the left-right pairs, allowing us to more directly implement algorithm 6 function in section 5.4.4.3. While this would likely lead to simpler, more readable code, whether or not it would lead to performance improvements remains to be seen.

7.3 Using Futhark

Futhark is a high-level hardware-agnostic language, and as such allows us to write code without thinking about the details of the hardware representations, nor do we have to think about parallel computing primitives such as threads and synchronisation, as is the case if we write e.g. OpenCL or CUDA by hand. We have instead approached the problem entirely in terms of regular arrays and second-order array combinators.

While knowledge about GPU hardware is certainly beneficial in order to know what kind of behaviour to expect and to know which problems can be feasibly solved on a GPU, the hardware agnosticism of Futhark has allowed us to write our code without directly using any such knowledge. Note how neither chapter 2 nor 5 mention how arrays are laid out in memory, accessed, cached, or traversed, nor do we mention how computation is spread across threads, how and when threads are synced, or even how many threads are used. Indeed, our implementation is presented without any mention of threads at all. Instead, these things are all handled by the Futhark compiler. The compiler analyses code and applies aggressive optimisation techniques do decide all of this for us.

Futhark being purely functional gives us a very simple top-level description of the algorithm: representing the CSC matrix is done with a simple record type, and the functions `low`, `clear`, and `update_lookup` are arguably very simple to both write and understand. The functions `init_new_matrix` and `copy_columns` are defined only using a few SOACs, namely `scatter` and `scan`, and `expand`. While their definitions are perhaps less understandable for a new reader, they are quite small and, as mentioned, written without any reference to threads or regard for memory access patterns, for instance.

The main challenge is adding column pairs in parallel. With access to fully irregular arrays we could describe a sparse matrix with an array of arrays, each inner array containing the nonzero row indices, and we could add columns by simply computing and replacing the required inner array, even if its length has changed. However, managing irregular data introduces many problems and possibly performance penalties [13], and so this is not something that Futhark supports.

As mentioned, we are not able to map a column addition function over the array `left_right_indices` to add all column pairs (recall the pseudocode in algorithm 6). Instead we had to swap the outer parallel loop over left-right-pairs with the inner sequential loop over row indices, leading to a more complicated algorithm description. The former of these approaches requires in-place updates in a manner that is not possible to express in Futhark. This is due to current limitations that may not exist in the future, as the upcoming accumulator feature should give enough flexibility. While this indicates that Futhark will continue to evolve and become more flexible, the fact that we were able to implement column addition in a reasonable manner shows us that Futhark is already in its current state a usable language.

References

[1] H. Edelsbrunner and J. Harer, *Computational Topology: An Introduction.* 2010-01, ISBN: 978-0-8218-4925-5. DOI: `10.1007/978-3-540-33259-6_7`.

[2] M. Li, M. H. Frank, V. Coneva, W. Mio, D. H. Chitwood, and C. N. Topp, "The persistent homology mathematical framework provides enhanced genotype-to-phenotype associations for plant morphology," *Plant Physiology*, vol. 177, no. 4, pp. 1382–1395, 2018, ISSN: 0032-0889. DOI: `10.1104/pp.18.00104`. eprint: `http://www.plantphysiol.org/content/177/4/1382.full.pdf`. [Online]. Available: `http://www.plantphysiol.org/content/177/4/1382`.

[3] S. M. S. Syed Musa, M. S. Md Noorani, F. Abdul Razak, M. Ismail, M. A. Alias, and S. I. Hussain, "Using persistent homology as preprocessing of early warning signals for critical transition in flood," *Scientific Reports*, vol. 11, no. 1, p. 7234, 2021-03, ISSN: 2045-2322. DOI: `10.1038/s41598-021-86739-5`. [Online]. Available: `https://doi.org/10.1038/s41598-021-86739-5`.

[4] B. Olsthoorn, J. Hellsvik, and A. V. Balatsky, "Finding hidden order in spin models with persistent homology," *Phys. Rev. Research*, vol. 2, p. 043 308, 4 2020-12. DOI: `10.1103/PhysRevResearch.2.043308`. [Online]. Available: `https://link.aps.org/doi/10.1103/PhysRevResearch.2.043308`.

[5] M. K. Chung, J. L. Hanson, J. Ye, R. J. Davidson, and S. D. Pollak, "Persistent homology in sparse regression and its application to brain morphometry," *IEEE Transactions on Medical Imaging*, vol. 34, no. 9, pp. 1928–1939, 2015. DOI: `10.1109/TMI.2015.2416271`.

[6] W. K. Chong and S. Rudkin, "Persistent homology in tourism: Unlocking the possibilities," *Tourism Management*, vol. 81, p. 104 132, 2020, ISSN: 0261-5177. DOI: `https://doi.org/10.1016/j.tourman.2020.104132`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0261517720300583`.

[7] N. Otter, M. A. Porter, U. Tillmann, P. Grindrod, and H. A. Harrington, "A roadmap for the computation of persistent homology," *EPJ Data Science*, vol. 6, 2017-08. DOI: `10.1140/epjds/s13688-017-0109-5`.

[8] R. Mendoza-Smith and J. Tanner, "Parallel multi-scale reduction of persistent homology filtrations," *arXiv preprint arXiv:1708.04710*, 2017.

[9] M. Elsman, T. Henriksen, and C. E. Oancea, *Parallel Programming in Futhark.* 2018. [Online]. Available: `https://futhark-book.readthedocs.io`.

[10] "OpenCL." (Accessed 2021-05-31.), [Online]. Available: `https://www.khronos.org/opencl/`.

[11] "CUDA." (Accessed 2021-05-31.), [Online]. Available: https://developer.nvidia.com/about-cuda.

[12] T. Henriksen. "Futhark." (Accessed 2021-05-31.), [Online]. Available: https://futhark-lang.org/.

[13] M. Elsman, T. Henriksen, and N. G. W. Serup, "Data-parallel flattening by expansion," in *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ser. ARRAY 2019, Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 14–24, ISBN: 9781450367172. DOI: 10.1145/3315454.3329955. [Online]. Available: https://doi.org/10.1145/3315454.3329955.

[14] A. Hatcher, *Algebraic Topology.* 2002.

[15] A. Zomorodian and G. Carlsson, "Computing persistent homology," in *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, ser. SCG '04, Brooklyn, New York, USA: Association for Computing Machinery, 2004, pp. 347–356, ISBN: 1581138857. DOI: 10.1145/997817.997870. [Online]. Available: https://doi.org/10.1145/997817.997870.

[16] H. Edelsbrunner, D. Letscher, and A. Zomorodian, "Topological persistence and simplification," in *Proceedings 41st Annual Symposium on Foundations of Computer Science*, 2000, pp. 454–463. DOI: 10.1109/SFCS.2000.892133.

[17] C. Chen and M. Kerber, "Persistent homology computation with a twist," *Proceedings of the 27th European Workshop on Computational Geometry*, 2011.

[18] U. Bauer, M. Kerber, and J. Reininghaus, "Clear and compress: Computing persistent homology in chunks," in *Topological Methods in Data Analysis and Visualization III*, P.-T. Bremer, I. Hotz, V. Pascucci, and R. Peikert, Eds., Cham: Springer International Publishing, 2014, pp. 103–117, ISBN: 978-3-319-04099-8.

[19] F. Pizzuti, M. Steuwer, and C. Dubach, "Generating fast sparse matrix vector multiplication from a high level generic functional ir," in *Proceedings of the 29th International Conference on Compiler Construction*, ser. CC 2020, San Diego, CA, USA: Association for Computing Machinery, 2020, pp. 85–95, ISBN: 9781450371209. DOI: 10.1145/3377555.3377896. [Online]. Available: https://doi.org/10.1145/3377555.3377896.

[20] D. Cohen-Steiner, H. Edelsbrunner, and D. Morozov, "Vines and vineyards by updating persistence in linear time," in *Proceedings of the Twenty-Second Annual Symposium on Computational Geometry*, ser. SCG '06, Sedona, Arizona, USA: Association for Computing Machinery, 2006, pp. 119–126, ISBN: 1595933409. DOI: 10.1145/1137856.1137877. [Online]. Available: https://doi.org/10.1145/1137856.1137877.

[21] D. Morozov and A. Nigmetov, "Towards lockfree persistent homology," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 555–557, ISBN: 9781450369350. DOI: 10.1145/3350755.3400244. [Online]. Available: https://doi.org/10.1145/3350755.3400244.

[22] H. Adams, A. Tausz, and M. Vejdemo-Johansson, "Javaplex: A research software package for persistent (co)homology," in *Mathematical Software – ICMS 2014*, H. Hong and C. Yap, Eds., Software available at `http://appliedtopology.github.io/javaplex/`, Berlin, Heidelberg: Springer, 2014, pp. 129–136, ISBN: 978-3-662-44199-2.

[23] E. von Brömssen, "Computing persistent homology in futhark," in *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*, ser. FHPNC 2021, Association for Computing Machinery, 2021.

[24] U. Bauer, M. Kerber, J. Reininghaus, and H. Wagner, "Phat – persistent homology algorithms toolbox," *Journal of Symbolic Computation*, vol. 78, pp. 76–90, 2017, Algorithms and Software for Computational Topology, ISSN: 0747-7171. DOI: `https://doi.org/10.1016/j.jsc.2016.03.008`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0747717116300098`.