Defining an Object-Oriented Architecture for Operations on Genomic Data

Sivert Kronen Hatteberg, sivertkh@ifi.uio.no

Thesis submitted for the degree of Master in Network and system administration 60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2017
Defining an Object-Oriented Architecture for Operations on Genomic Data

Sivert Kronen Hatteberg, sivertkh@ifi.uio.no
© 2017 Sivert Kronen Hatteberg, sivertkh@ifi.uio.no

Defining an Object-Oriented Architecture for Operations on Genomic Data

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo
Abstract

Genomic data is often represented in the form of a genomic track. This track is a set of data with given positions in a genome. With these tracks we can perform different types of calculations. We call these calculations track operations.

The previously proposed file format GTrack introduced a system for categorizing these tracks into distinct track types. The type of a track is dependent on the data it represents.

GTrackCore is the library that handles and pre-processes GTrack files into Numpy arrays. These arrays enable us to perform fast and vectorised calculations.

This thesis introduces an extension to the GTrackCore library. This extension consists of a library of track operations that uses the track type concept. The main usage is centred around an object-oriented implementation. Each operation is defined as a class. For each class we define the track type of the inputs and output. This formal definition enables us to use other operations as input. This is a powerful concept that makes it possible to combine operations into complex trees.

The number of possible track operations is vast, far more than would fit within the scope of this thesis. The framework is written to make the implementation of new operations as easy and effortless as possible.

An operation can also be defined by using existing operations as building blocks. To create such an operation, one would need to define a tree made of existing operations. This tree defines the operation directly and there is no need to do work on the Numpy arrays in this case.

In this thesis we show how this library was designed and implemented. We go into details on the problems we faced and how we solved them. A large focus is on the problems that arise from working with Numpy arrays.

This project has provided such a library. We have extended GTrackCore with a library of track operations. These operations are object-based and use the track type concept introduced by GTrack. We provided multiple interfaces that enables users of different skill level to use the operations.
Contents

I  Introduction  
1  Introduction  
   1.1  Motivation  
   1.2  Goals  
   1.3  Chapter overview  
2  Background  
   2.1  Genomics  
   2.2  Genome reference coordinate system  
   2.3  Genomic Tracks  
      2.3.1  Core informational properties  
      2.3.2  Track type  
   2.4  Track File Formats  
      2.4.1  GTrack  
      2.4.2  BTrack  
   2.5  Track operation  
      2.5.1  Track as result  
      2.5.2  Property as result  
   2.6  Existing tools for genomic track operations  
      2.6.1  BEDTools  
      2.6.2  bioconductor  
      2.6.3  bbcflib  
   2.7  Technology  
      2.7.1  Python  
      2.7.2  Numpy  
      2.7.3  Unittest  
      2.7.4  JSON  
   2.8  GTrackCore  
      2.8.1  The TrackView class  
      2.8.2  The TrackFormat and TrackFormatReq classes  

II  The Project  
3  The design of the library  
   3.1  Who will use the library?  
   3.2  Envisioned usages
3.2.1 From the command line .................................. 26
3.2.2 Raw and object-oriented usage .......................... 27
3.2.3 Raw usage ................................................. 27
3.2.4 Writing a script .......................................... 28
3.2.5 Via a Python console ..................................... 28
3.2.6 Writing custom operations ............................... 30
3.2.7 As module in a large framework ......................... 30

3.3 What does the library do for the end user ............... 30
3.3.1 An operation framework that supports all of the different track types ........................................... 31
3.3.2 Multiple user interfaces .................................. 31
3.3.3 Delayed calculation ...................................... 31
3.3.4 Simplified track handling by using one central container ............................................................ 31
3.3.5 Track validation and result track format generation ... 31
3.3.6 Nesting of operations ..................................... 32

3.4 What does the library do for an operation developer .... 32
3.4.1 Minimizing code writing .................................. 33
3.4.2 A dynamic CLI interface .................................. 33
3.4.3 Dynamic factory for operation objects .................. 34
3.4.4 One simple configuration ................................ 34
3.4.5 Automatic keyword argument handling and validation ................................................................. 34
3.4.6 Track input and output validation ......................... 35
3.4.7 All track data in one object ............................... 35
3.4.8 Reuse of common operations ............................. 35
3.4.9 Encapsulating a nested operation by overloading the calculate method ........................................ 36

4 The implementation of the library .......................... 39
4.1 The operations framework .................................... 39
4.1.1 Raw Operations ........................................... 40
4.1.2 Class representation using the Operator superclass .. 42
4.1.3 Options handling ......................................... 44
4.1.4 A dynamic CLI ............................................. 46
4.1.5 Creating new tracks ....................................... 48
4.1.6 Integration with GTrackCore ......................... 49

4.2 Working with Numpy and ndarray ........................ 51
4.2.1 Array slicing in Numpy ................................... 51
4.2.2 Coding Accumulative cover status trick ................ 52
4.2.3 Handling multiple overlapping .......................... 55
4.2.4 Using fractions instead of a static number .......... 58
4.2.5 Padding of ndarrays ...................................... 58
4.2.6 Handling strings in ndarrays ............................ 60

4.3 Track handling in the library ................................ 61
4.3.1 The TrackContents class ................................. 61
4.3.2 The Genome class ........................................ 63

4.4 Testing ......................................................... 64
4.4.1 Testing operations ........................................ 65
III Conclusion

5 Discussion

5.1 The user interfaces in practice

5.1.1 The raw operations

5.1.2 Object based operations

5.1.3 CLI interface

5.2 Implementing new operations

5.2.1 Simplified implementation by nested operations

5.3 Our usage of Numpy and ndarrays

5.3.1 Common Numpy sub-operations

5.4 Runtime

5.5 Do the operations return correct results?

5.5.1 Testing the framework and its operations

5.5.2 Test template

5.5.3 Possible sources of errors

5.6 Conclusion

6 Future work

6.1 Creating more operations

6.2 Expand existing operations with new features

6.3 Do not run the preCalculation method at object initialization

6.4 Remove the pre/post methods

6.4.1 Creating new layer of operations

6.5 Calculate one region at the time

6.6 Parallelization and optimization

6.7 Making the library portable and easy to install

6.7.1 Ease of installation GTrackCore

6.7.2 Create a BTrack version that is more independent of GTrackCore

6.8 Expand GTools

6.8.1 Improve track handling

6.8.2 Support for running multiple operations

6.8.3 Tab completion

6.9 Automated testing of operations

Appendices

A Operations supported by the library

A.1 Average length

A.2 Average link weight

A.3 Complement

A.4 Count elements

A.5 Coverage

A.6 Expand

A.7 Filter

A.8 Flank

A.9 Intersect
## List of Figures

2.1 The relationship between property and track type 9
2.2 Illustration of the different track types 10

4.1 Finding the overlap 53
4.2 Code assigned to each start and end point 53
4.3 Accumulation of track score 54
4.4 The links in a track represented as a graph 59

A.1 The average length operation 95
A.2 The links of a track visualised as a graph 97
A.3 The complement operation 99
A.4 Complement as a venn diagram 99
A.5 The count elements operation 101
A.6 The coverage operation 102
A.7 The expand operation 104
A.8 The filter operation 106
A.9 The flank operation 108
A.10 The intersect operation 111
A.11 Intersect as a venn diagram 111
A.12 The merge operation 113
A.13 The remove dead links operation 117
A.14 Visualization of the links as a graph 117
A.15 Shifting 119
A.16 The subtract operation 121
A.17 Subtract as a venn diagram 122
A.18 The union operation 123
A.19 Union as a venn diagram 123
A.20 Uniqueifying links, with overlap 126
A.21 Uniqueify links, after merging two tracks. With and without unifying the links 126
A.22 The value select operation 128
List of Tables

2.1 The 15 different track types ............................................. 9
2.2 Supported operations in bedools ................................. 13
2.3 Supported operations in Segtools ................................. 14
2.4 Supported stream operations in bbcflib.gfminer .............. 15
2.5 The track types as described by the properties of the track
    format classes ......................................................... 21
List of Listings

1  GTrack example file ............................................. 11
2  Using the CLI .................................................... 26
3  Using a raw operation to fine the union of two tracks ... 27
4  Simple class example ............................................ 28
5  Nested operation usage .......................................... 29
6  Simple iPython example ......................................... 30
7  Example of a raw operation ...................................... 40
8  Raw operation, use only the needed arrays .................. 41
9  Signature of Union raw operation, segmented track ....... 42
10 Raw operation with track input container ..................... 42
11 Raw operation using multiple arrays ........................... 42
12 The namedtuple KwArgumentInfo ................................. 45
13 Defining the options of an operation ......................... 45
14 Using __getattr__ ............................................... 46
15 Importing operations dynamically ............................... 47
16 Extracting ndarrays from a TrackView object ............... 49
17 Defining a tracks requirements .................................. 50
18 Defining a tracks result TrackFormat ......................... 51
19 Using slicing to compare n against n+1 ....................... 52
20 Using an index to extract results ............................... 52
21 Finding the accumulated cover status ......................... 53
22 Cumulative cover status implementation ...................... 54
23 Cumulative cover status with indexing and encoding ...... 56
24 Operation that can have multiple overlap .................... 56
25 Showing the padded ndarrays of a linked track .............. 59
26 Handling strings in ndarrays .................................... 61
27 Creating a Genome object from a dictionary definition .... 64
28 Creating a Genome object from a JSON definition .......... 64
29 JSON format used to store a genomes regions ............... 64
30 Method for running an operation test ......................... 65
31 Possible extension to GTools. Support for defining the operation as a string ......................... 83
32 Possible extension to GTools. Saving and running saved track strings .................................. 84
33 Possible extension to GTools. Support for pips ................ 84
34 Bash, piping multiple programs into one ...................... 85
35 More control over merging ...................................... 124
36 Template for a new operation .................................. 139
Preface

The target audience for this thesis are master students in the computer science field. The topic is on tools for biomedical informatics. Some knowledge of biomedical informatics will be of help in understanding this thesis, but not a requirement.

During the first semester of my master I got a full time job, working at the University Center for Information Technology. From the second semester I worked on my master part time. Working part time was much more challenging than I first imagined. Finding the time and effort to work on the project after a full work day was not easy. I used much of my vacation time and weekends to work on the project. This allowed me to have periods where I could focus fully on the project.

Acknowledgements

I would first like to thank my supervisors Geir Kjetil Sandve and Sveinung Gundersen for all of their help and guidance. My employer, the University Center for Information Technology, for allowing me the time of needed to complete this project. All of my friends and family for their support and help. A special thanks to David Kristensen, Martin Bore and Marte Svalastoga for their proofreading and feedback. Finally, I would like to thank my dear Malin for her love and support.

Sivert Kronen Hatteberg
University of Oslo
January, 2017
Part I

Introduction
Chapter 1

Introduction

1.1 Motivation

Genomic data is often presented in the form of a genomic track. A genomic track represents our data in a common reference system. This enables us to easily compare different data sets with each other in a consistent and reproducible way. We can visualize this reference system as the DNA molecule stretched into one long line. The element on this line is given an address from zero to the length of the molecule. A track can contain a multitude of different types of data. For example, information about some physical properties, interconnections that can exist in the 3D structure, or to mark areas of interest.

Once we have data defined in a track, we would like to analyse it. We call these types of analyses for genomic track operations. Broadly we can define an operation as any unit of work or calculation performed on one or more tracks. These operations might consist of calculating a simple property to complex operations using multiple tracks. Finding the areas where two tracks overlap is an example of such an operation.

GTrackCore is a library for representing such genomic tracks. It was extracted from the Genomic HyperBrowser[9] system, which is a web based tool for handling and analysing genomic tracks. GTrackCore is the data-model it uses and it is responsible for pre-processing, storing and handling its genomic track data. The Genomic HyperBrowser is capable of performing track operations, but these operations mainly focus on performing statistical analysis on the track data and not on operations that return tracks. HyperBrowser is a large web based system, and will not always be the optimal solution for all user.

The current plan for GTrackCore is to separate it into a standalone tool for performing genomic analysis. This tool is meant to be used directly on a researcher’s laptop or desktop computer, either via a CLI or by writing Python code. GTrackCore itself does not have any operations capabilities. Expanding it with such a library will be a part of this separation.

GTrack introduces a formal model for categorising tracks into different track types [1]. The type of a track is dependent on what type of data it represents. Points and segments are examples of two such track types. A
segment track defines a set of areas on the track. These areas are defined by a start and an end position. A point track defines a set of points, each having a length of zero. GTrack[10] is a tabular separated file format for storing track data in a human readable form. GTrackCore reads and pre-processed these files into a binary representation based on Numpy memmaps. Operations on Numpy arrays are generally very fast and utilizes vector operations. Our operations will use this pre-processed data in their calculations.

The track type formalisation is a powerful and one that we will use extensively throughout the library. It’s based upon the tracks having four possible core informational properties. These properties are gaps, lengths, values and interconnections[1]. A track will have one or more of these properties defined. We go into more detail on this formalization and the core information properties in chapter 2.

With this formalisation we can unambiguously define the input requirements for any operation. In the same manner we can define what the output track type of an operation will be. This enables us to nest operations into complex tree structures. This is a powerful concept that allows the us to use other operations as building blocks. As the number of implemented operations increase, the number of possible operations we can build increase.

As the input track types of an operation is defined, it becomes possible to create operations with mixed inputs. These are not necessarily new operations, but we can now describe them in a more precise way. For example, we might have an operation that finds all points from track A that are covered by a segment from track B. These points are then returned as a new point type track. In this operation we set a requirement for track A and B, in this case point and segment. Operations on interconnected tracks is another major possibility. Support for such 3D linked data is nothing new, but we can now use it in combination with any other track data.

A DNA molecule is not simply a long string of base pairs. It’s folded into complex structures where base pairs can interact which each other independently of their positions. We can visualise these interactions as a graph, where each element is a node, and each interconnect is an edge between two elements. This graph data is easily accessible and can be used in combination with the existing tools and library that python provides.

No existing tool can provide all of this today. The combination of using the track type to formalise operations together with an object base implementation is a very powerful one. Providing such an interface together with a more conventional CLI will make this library versatile and cover a large range of possible uses.

We aim to provide multiple user interfaces that facilitate these different usages. The main implementation will be an object based one. Each operation is defined as its own class. These objects can then be used directly via a Python shell or as part of a more complex Python script. An important user group is scientists with little formal programming experience. We aim to provide a simplified interface where we limit the number of concepts and ideas used. We will avoid the use of objects and rather provide a simple method for each operation. These methods should only take Numpy arrays
as track input and return either Numpy arrays or some other result.

1.2 Goals

In this section we outline the main goals of the project.

- Expand the GTrackCore system with a library of track operations. This library must take advantage of the track type concept introduced by GTrack. The operations should use the pre-processed data provided by GTrackCore.

- The main usage should be based around an object oriented implementation.

- These objects should be able to take both tracks and other operations as inputs. It should be possible to nest operations and tracks into an arbitrary complex tree.

- The library should support a simplified usage. With this usage, we limit the use of more complex ideas and structures like object orienting. Each operation is simple method that takes the appropriate ndarrays as input.

- The library should support multiple different user interfaces. From a command line interface to more advanced usages via interactive Python shells or Python scripts.

- These user interfaces needs to as dynamic as possible.

- The number of possible operations is very large. Implementing new operations must therefore be as simple as possible. As much work as possible should be offloaded to the central framework.

- A simplified way of implementing new operations should be supported. New operations can be defined by using existing ones as building blocks. The library need to support a way of defining operations by such a structure.

- The framework needs to facilitate testing new operations in a simple and consistent way.

- A number of operations are to be implemented. We need to limit the number, as the number of possible ones is far larger than the scope of this theses. In selecting operations to implement we need to choose both common ones that are already implemented and used by existing tools, and others that showcases the new capabilities of our library.
1.3 Chapter overview

**Chapter 2** describes all of the background information needed to understand this thesis. We describe the technology used, look at some existing tools for track operations and discuss the scientific background.

**Chapter 3** presents the design of the library. Here we give an overview of the design of our implementation.

**Chapter 4** gives an overview over our implementation. Here we expand on how we implemented the library, what problems we encountered and how we solved them. We go into detail on some of the main problems, especially the problems related to working with *Numpy*.

**Chapter 5** presents a discussion on the project. Have we reached the goals of the project? What parts worked as expected and what parts did not.

**Chapter 6** is an overview of possible future work and expansions. We give an overview over ideas that we did not have time for or are outside the scope of the project.

**Appendix A** gives an overview of all of the implemented operations. It presents what each operation does, what are its options, inputs and results.
Chapter 2

Background

2.1 Genomics

The genetic instruction for all known living organisms is encoded in a molecule structure we call DNA. DNA is short for deoxyribonucleic acid. The DNA molecule is built up of four different type of units called nucleotides. For DNA, these nucleotides are cytosine (C), guanine (G), adenine (A) and thymine (T). These nucleotides are bound together in long sequences to create the DNA.

DNA is composed of two such long sequences. The nucleotides of each of these are bound together, using hydrogen bonds, creating base pairs. This structure makes up the famous double helix of the DNA molecule.

This DNA code is packaged into a structure called a chromosome. Most of the DNA of a living organism is packaged into one or more of these chromosomes. The human genome consists of 46 such chromosomes. The DNA molecule is coiled tightly together into different shapes. As such they can interact and create interconnection that interact across the molecules.

A genome constitutes the whole of the genetic material of an organism.

2.2 Genome reference coordinate system

A genomic track is a way of representing data associated with a given genome. We can position this data into a common reference system[1]. With such a reference system, we can easily compare it with other tracks on the same genome.

The reference system does simply consist of the length of each region in a given genome. If we try to visualized this process, we unfold the DNA of each chromosome, stretching them from end to end. We can then look at these as line segments, numbered from zero up to the length of each chromosome. Such a reference system can be constructed for any genome.

Each position on these lines represents the position of a base pair in a chromosome. Most genomic data can be represented as data on a line segment like this. A genomic track is simply a series of data with a position on these lines. The common reference system enables us to compare different sets of data with each other in a consistent and reproducible
way. A track can represent a multitude of different data, from a specific physical property of a segment to the interconnections that exists in the folded structure of each chromosome. It can be used to single out areas of interest, like a genomic feature, or to mask out area of disinterest. The important point thing to take away from this is that a track is just data with position in a common reference system. The track itself does not contain the underlying DNA code of the genome.

As the DNA consists of two strands of nucleotides bound together as a double helix, we can have data that is specific to one of the two strands. In these cases, we do not create separate line segment for each strand. To differentiate which of the strands an element is on, we simply tag it as either on the upstream or on the downstream strand.

2.3 Genomic Tracks

GTrackCore is a system for handling genomic track data. It comes with a tabular file format called GTrack. GTrack introduces the concept of tracks having a specific track type[1]. In the following subsections, we will go into detail on what a track type is and how it’s defined.

2.3.1 Core informational properties

GTrackCore use four properties to differentiate the different track types. These are called core informational properties and are gaps, lengths, values and interconnections[1]. The first two describes segments on the track by using the length of the elements and the gaps between them. A track element can have some value associated with it. This value can represent several different properties about an element. The interconnections represent a connection between one element and another element on the track. This interconnection may be used to describe the 3D structure of a chromosome. Elements that are far apart on a the flattened track can be close together in the real 3D structure of a folded DNA-molecule.

2.3.2 Track type

A track does only need to define the core informational properties needed to represent its data. Defining more properties does not add anything to the track. A partitioning of a genome has no gaps, adding gap data does not add anything as all the gaps would have a length of zero. Depending on which of the properties are defined, we get $2^4 = 16$ different track types. Of these, 15 are of interest as the last one have none of the properties defined and can therefore not represent a track[1]. See figure 2.1 for a graphical representation of the relations between the different properties and track type. The 15 different track types are named in table 2.1. For a graphical view of these see figure 2.2.
2.4 Track File Formats

There exists a multitude of file formats for representing a genetic track data. These are either textual formats like tabular and XML, or in some form of binary representation. None of these formats support a track type concept, but they are all capable of representing one or more of the different track types[1].

One of the reasons for creating GTrack, was to create a format that could represent all the data that the different track types uses. We are not going to describe in detail how the existing track file format works. GTrackCore supports converting most of the common formats, and as such we do not have to deal with them in this project.

As part of this project we have studied the capabilities of existing tools. When we talk about their implied track type support, we are in practice talking about which track types the supported file format can represent. Table 2 in the GTrack[1] article gives us this information. This table is an overview of the most popular formats today, and shows which of the track types these formats can represent. As this table is rather large it is not...
This table does however not always tell the whole truth. Some of the track types are special cases of each other. We can therefore still represent the data, but not in a consistent way. The formats do not impose any limits on its data. We would need to check in each case that all our data comply with the requirements of the track type.

### 2.4.1 GTrack

*GTrack*[10] is the file format provided by *GTrackCore*. It’s a type-aware tabular text format. Each *GTrack* file consists of a header and one line for each element of the track.

The header is used to define the track, what its track type is, and what
the different columns in the data fields represents. This header can be quite complex and we will not go into detail on it here. For a complete description on the GTrack header, see its specification[10].

The rest of the file consists of the data elements. This data contains the columns defined in the header. A tab character is used to separate the different values. Listing 1 show one of the example files from the specification.

```plaintext
# GTrack example file 2
#
# Note: tech is a custom column and not part of the GTrack specification
#
##Track type: valued segments
###seqid tech start end value strand
####genome=hg19
chr1 ChIP-seq 1047 1165 0.625 -
chr2 ChIP-chip 2002 2450 . +
chr2 ChIP-chip 3033 3246 0.355 +
```

Listing 1: GTrack example file

### 2.4.2 BTrack

There is a planned binary version of GTrack called BTrack[7, 8]. This format is based on the HDF5[11] library and will store the pre-processed data as a single file. Such a file format would make GTrackCore and this library much more portable. These files could then be created and shared among the users. By saving the pre-processed data as a file, we do not need to pre-process the imported tracks. This will be a real speedup as the pre-processing can take some time.

### 2.5 Track operation

We can broadly define a track operation as any kind of work or calculation done on one or more tracks. Making these operations track specific is the natural extension. This enables us to define what kind of track a specific operation requires and to make assumptions based on this. In the same manner, we can define each operation track output.

The number of possible track operations that can be created within this definition is vast. Some of the more common types are transformation, filtering, genome arithmetic (set operations) and calculations to find some specific property of the track. Set operations are equal to the mathematical set operations. These are one of the most common types of operations. Examples are Union, Intersection and Complement. A set operation will always return a new track as its result. Transformations edits, or modifies,
a track in some way, by expanding, slicing etc. For example, an operation
could, for each segment in a track, expand the segment \(n\) places in each
direction, and if any of the new segments overlap the operation will merge
them. Filtering operations do a conditional select of some sort. These
are somewhat analogous to a simple SQL select statement. Find all points
within a range, find all valued points where the value is larger the \(n\), etc.

As we have defined each operations inputs and outputs, we can nest
them to perform operations that are more complex than a single operation.
This allows for both one-off operations created to solve a specific problem,
and multistep operations like for instance find all segments in track A
with length bigger than \(n\) and then find the intersection with track B.
The nesting of operations also makes it easier to extend the library with
new operations. New operations can use existing operations as building
blocks. This holds true for both wrapping common sequences of operations
into new operations for ease of use, and for introducing completely new
operations.

One last important point to make is that an operation can be valid in a
mathematical sense, but not in a biologically one. We need to take this into
consideration when creating new operations.

We can broadly split the track operations into two main categories.
Operations that gives a new track as result, and operations that give a
property of the track.

### 2.5.1 Track as result

These operations do some work on the input tracks. They compare, filter,
sort, removes or adds elements in some form or another. All of these
operations create new tracks that are returned to the user.

### 2.5.2 Property as result

These are operations that calculate or find some property of a track. The
results are not in the form of a new track but as any other type of data.
They can be a simple number or more detailed data like a graph or plot. The
operations \texttt{AverageLength} and \texttt{Coverage} are examples of such operations.
\texttt{AverageLength} finds the average length of the segments in each region,
and \texttt{Coverage} finds the size of the area that the track covers. More on these
operations and how they work in appendix A.

### 2.6 Existing tools for genomic track operations

The concept of track type was introduced with \texttt{GTrack}. As such this
concept, does not directly translate to the capabilities of any existing tool.

When looking at existing tools, we would like to know what type of
operations they support, and if possible to relate these capabilities to the
different track types. Which track types each tool can perform operations
on is directly related to which track format they support.
Table 2 from the GTrack article[1] gives an overview over which track types existing formats are capable of representing. We will use this table when we examine the existing tools. Some of the track types are special cases of each other. For instance, a segment track can represent a point track, as points are a special case of segments where all lengths are zero, and a partition can be represented by a segment track, as it’s a special case of segments where there are no gaps between any of the segments and the first segment starts at zero and the last end at the region end.

There exists a range of different tools for doing track operations today. These tools differ in what kind of track file formats they support, and in which operations they can perform on these. Some of them are extensively used by researchers today. When writing an operations library for GTrackCore a study of these tools will be a great resource, especially when deciding which operations to implement first.

2.6.1 BEDTools

BEDTools[12] is an extensively used application. At the time of writing, the main bedtools article is cited a total of 1948 times. BEDTools support operations that can do comparison, manipulation and annotation of genomic features in the BED and GFF formats and comparison of sequence alignment in the BAM format to BED and GFF[2].

Using table 2 from the main GTrack[1] article we find the track types that are theoretically supported. In this case the BED and GFF format are both capable of representing the track types of points (P), segment (S), valued points (VP), valued segments (VS) and a special case of linked segments. Although not directly supported it is possible to represent all of the partitions as well. As discussed, partitions are just a special case of segments.

One of bedtools greatest strength is its ability to pipe the result from one operation into another with the Unix pipe. As the pipe is textual, common Unix tools like grep, sed and awk can be used in combination with the tool itself. This makes it possible to solve complex problems without implementing a specific tool for each case. Table 2.2 lists the supported operations[2].

| intersectBed | pairToBed | bamToBed | pairToPair |
| windowBed   | closestBed | subtractBed | MergeBed |
| coverageBed | genomeCoverageBed | fastaFromBed | maskFastaFromBed |
| shuffleBed  | slopBed    | sortBed    | linksBed   |

Table 2.2: Supported operations in bedtools
Segtools

As the name implies Segtools[13] is written to do analysis on genomic segmentations. A segment is defined as a set of non-overlapping regions of a genome, where each segment is assigned one of a set of labels[3]. A label in this context is some data associated with the given segment.

These segments are analogous to the valued segments track type. Segtools supports the BED and GFF formats as input. We use table 2 from the GTrackCore article[1] again. From it we can see that we can theoretically do operations on the track types points (P), segment (S), valued points (VP), valued segments (VS) and a special case of linked segments. As with BEDTool we can in practice support the partitioning types as these are a special case of the segment type. Segtools does not support interconnections so we are left with the first four.

The tools main focus is on creating visualization for use in analysis, with different plots created from the operations. A final HTML-report can be generated.

Table 2.3 lists the supported operations[3].

<table>
<thead>
<tr>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>length-distribution</td>
</tr>
<tr>
<td>nucleotide-frequency</td>
</tr>
<tr>
<td>signal-distribution</td>
</tr>
<tr>
<td>aggregation</td>
</tr>
<tr>
<td>compare</td>
</tr>
<tr>
<td>overlap</td>
</tr>
<tr>
<td>preprocess</td>
</tr>
<tr>
<td>flatten</td>
</tr>
<tr>
<td>feature-distance</td>
</tr>
<tr>
<td>html-report</td>
</tr>
</tbody>
</table>

Table 2.3: Supported operations in Segtools

2.6.2 bioconductor

Bioconductor[14] is a large software package for doing “analysis and comprehension of high-throughput data in genomics and molecular biology”[4]. It consists of a total of 1104 packages. Most of these packages are outside the domain of genomic track operations. Out of these packages GenomicRanges[15] and GenomeIntervals[16] are the most relevant. The GenomeIntervals package provides data structures to represent tracks. These structures are used by the GenomicRanges package[5].

GenomicRanges supports a wide range of operations and they are too numerous to list here. The main categories are Accessors, Ordering, Arithmetic, Set operations, Overlaps, Coverage, Combine, Extraction and Split. For a full list of supported operations see table 1 in the GenomicRanges[5] article.

2.6.3 bbcflib

bbcflib[17], short for Bioinformatics and Biostatistics Core Facility Python Library, is a Python library for doing High-Throughput sequencing analysis[6].

The library is part of the HTSstation web application. This library uses a Python object representation of a track bbcflib.track. These track
objects can be created from a wide array of existing track file formats. The documentation[18] gives the following supported file formats: bed, wig, bedGraph, bigWig, SAM, BAM, sqlite, sga, gff. Again by using table 2 from the GTrakCore article[1], we get that it theoretically can do operations on points (P), segments (S), valued points (VP), valued segments (VS), step functions (SF) and functions (F).

bbcflib provides a module for what it calls track manipulations. This module is called bbcflib.gfminer[19]. Gfminer divides these manipulations or operations into four categories. Common, Stream, Numeric and Figure. Common is mostly helper operations used by other operations, the Plot ones create plots, Numeric returns Numpy matrices and Stream returns streams. Of these the stream operations are of most interest to us. A stream refers to bbcflib.track.FeatrStream[20]. This object is a type of iterator over the fields of a track. A track object can easily be converted to a FeatureStream and visa versa. Table 2.4 lists the supported stream operations.

<table>
<thead>
<tr>
<th>getNearestFeature</th>
<th>concatenate</th>
<th>selection</th>
<th>overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td>neighborhood</td>
<td>intersect</td>
<td>merge_scores</td>
<td>filter_scores</td>
</tr>
<tr>
<td>score_b_feature</td>
<td>window_smoothing</td>
<td>normalize</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.4: Supported stream operations in bbcflib.gfminer

2.7 Technology

2.7.1 Python

Python[21] is an interpreted high-level programming language. It is widely used in scientific computing and it provides many libraries for performing such computing.

Python is the language we use in this project. The main reasoning behind this is that the library is an extension to the GTrackCore system. As this system is implemented in Python, using it in our part is given. That said Python would anyway be one of the natural choices for such a project. Python presents the perfect choice when writing tools that are to be used by non-computer scientists. The Python language is designed with readability in mind. Writing Python code that is easy to understand is simpler than most other programming languages. Python uses dynamic typing, meaning that the type of each variable does not need to be defined in the code. The types are however checked at runtime.

In this section, we introduce some of the Python libraries and concepts that our library uses.

PEP8

PEP8 is the is the style guide of the Python language. It dictates how Python code should be structured, how the different variables and classes
should be named, how long each line should be and much more. We have tried to follow PEP8 and to writing as pythonesque code as possible. The one exception from PEP8 is in the naming of variable, classes, and methods. The GTrackCore system uses camel case and as we are expanding an existing system we have chosen to stick with its naming convention. PEP8 does allow this when expanding an already existing code base.

Factories

Factories is a common design pattern that enable us to create new objects without the need to specify its class. We use multiple factories in our library. The main one is used by the command line interface. This factory is able to dynamically create any operation object given the input from the command line interface.

Abstract methods

Python does not support abstract methods or classes natively. The reasoning for this is that it makes the code less pythonesque. Although we strictly do not need to use abstract methods, they give us more control when then overloading methods directly. To add support for abstract methods we use the library abc[22]. This library enables us to use the decorator abc.abstractmethod on one or more methods in a superclass. Methods with this decorator are then treated as abstract, and any subclass will need to implement this method before the code is able to run. By using abstract methods, we can in a constant way control that all of the required methods are implemented in a new operation.

2.7.2 Numpy

Numpy[23] is an extensive library in Python for doing scientific computing on large arrays and matrices. It's centred around a structure for representing these called ndarray. It provides a large set of methods for performing operation on these.

Python is an interpreted language and its default implementation is not optimised to run such calculations in an efficient way. Numpy enables us to perform these operations fast and efficient. It is able to do this by using a highly optimised C library of vector operation.

ndarray

Ndarrays are the structure that Numpy provides for representing arrays and matrices. The name stands for n-dimensional array. These arrays are homogeneous and of a given type. At creation the size of the arrays is allocated in memory. The size is equal to the length of the array times the size of the type of the elements. This means that they behave more like arrays in C then a normal Python array.
Using such arrays breaks with what we would normally perceive as pythonic. Special care has to be taken when working with these arrays. As the size of the array is allocated at creation, changing its size or type is an operation that involves making a copy. Depending on the size of the array, this might be a costly operation and one which could take time. As such special care should be taken when creating and using these arrays. Our code should avoid operations such as appending elements or changing the type mid-calculation and thus potentially make the computation significantly slower.

When working with ndarrays we do often perform slice and indexing operations on them. These operations create a form of subarray. The important thing to realise about these subarrays is that they are not copies of the original arrays. They are simply a view into them. If we change an element in the slice, then the original element is changed as well. The view makes the slice operations fast as we do not need to copy the array. But we need to be careful when using them in operations. One could inadvertently assume that the view is a separate arrays and change it. Using the original array at a later point could then lead to errors in the calculation.

**The dtype of ndarrays**

As previously mentioned ndarrays have a given type. This type is called a dtype[24]. Each dtype have a given size that defines how much space the ndarray need to allocate per element. When working with ndarrays, we can change its elements as long as we keep within the size of each dtype. If the dtype is np.int16, we are limited by the range of a 16-bit integer. Adding a larger number will overflow the element.

Special care has to be taken when we work with strings in ndarrays. The Python string type does not have a natural delimiting max size. Other types like float or int16 have clearly defined sizes. Python strings on the other hand are as long as defined, and size is allocated in each case.

As ndarrays have a defined size, we need to define the maximum length that strings can have. This is solved by having a dynamic string dtype. This dtype is set to the length of the largest string in the array. As there is only allocated enough space for a string of size n, we can’t change an element to a string of size n + 1 or grater without first changing its dtype. It’s easy to overlook this when writing code as no error is given when assigning a string that’s too long. The element gets updated, but only the first n characters are saved and we end up with corrupt data.

In the following example we show the string type in practice. We first create an array a, this array gets the dtype corresponding to the large string in the arrays. In this case its S11. Where 11 corresponds to the length of the largest string in the array.

```python
1 In [1]: a = numpy.array(['test string', '123'])
2 In [2]: a
3 Out[2]:
```
array(['test string', '123'],
       dtype='|S11'))

We now assign a new string to the array. This works as expected as this
string is smaller than the allocated size.

In [3]: a[1] = '123456789'
In [4]: a
Out[4]:
array(['test string', '123456789'],
       dtype='|S11'))

Next we try to assign a sting with a length of 14 to the array. As we see
only the 11 first characters are saved. No error or exception is given to the
user.

In [5]: a[1] = '123456789abcde'
In [6]: a
Out[6]:
array(['test string', '123456789ab'],
       dtype='|S11'))

If we want to save this string, we need to change the dtype. This involves
making a copy of the array. We use the astype method to simultaneously
make a copy and change its type. This new array will have allocated the
extra space needed to save the new string of length 14.

In [7]: b = a.astype('S14')
In [8]: b[1] = '123456789abcde'
In [9]: b
Out[9]:
array(['test string', '123456789abcde'],
       dtype='|S14'))

Changing the dtype of a ndarray will always create a copy. This copy
will have allocated the correct space for the new dtype. As a copy operation
is an expensive one, it should be avoided wherever possible. If we need to
save longer string, we should first try to find the new maximum string size
and only change the dtype once.

On the other hand, the dtype of an array might end up being bigger then
the largest string in the array. This might happen after an operation that
deletes the elements with the longest string. In cases like these, it might be
advantages to change the dtype to minimize memory usage.

Expanding string arrays might lead to wasted memory space. This
becomes especially important when working on large arrays. If we expand
one string with 50 characters, then the total size of the array will grow with $50 \times n$ where $n$ is the length of the array.

If we only need to change one string, then we should consider if the increase in memory usage is worth it, or if we can solve the problem in some other way. We can calculate a simple metric by comparing the average length of the strings in an array with the dtype. If the dtype is considerable larger than the average size, then we might want to look at ways to limit its size.

**The object dtype** It’s possible to use a *dtype* called object. This *dtype* allows us to use any python object or mix of objects as elements in an ndarray. We can now have sub-arrays of different length. This might sound tempting as it would simplify many problems common with working with ndarrays, but such usage should be avoided as it brakes the optimizations in the Numpy operations. It’s not possible to run the vectorised operations on these ndarrays.

### 2.7.3 Unittest

Unit testing is a method for testing that an isolated unit of code performs as expected. When writing unit test we define the expected results for a given input. If the return results do not match the expected one, the test fails.

*Unittest*[25] is a Python library and framework for writing and running such unit tests. We use this package to test that our operations work as expected. For each operation we define a number of test. In each we create a simple track and an expected result track or number. We then run these tracks through the given operations and compare the result track with the expected one. A template has been provided to make this process easier. This template helps the user by creating the necessary objects and to extract and check the results.

### 2.7.4 JSON

*JSON* is a text based structure for transferring data. This data is structured and can easily interpreted into structures like objects. It’s wildly used and together with *XML* almost an industry standard. Python support the reading and writing of *JSON* files natively. We use *JSON* to store information about a genomes bounding regions.

### 2.8 GTrackCore

*The Genomic HyperBrowser* is a web-based system for performing statistical analysis on genomic data. *GTrackCore* is an extraction of parts of its code. It is the data model and is responsible for the track handling. This project is an expansions of *GTrackCore* where we provide a library for performing track operations on the binary track data provided by *GTrackCore*. 

19
**GTrackCore** is a complex system with a large code base. Describing all of it here in detail is far outside the scope of this thesis. In this section we describe the main structures that our operations library interacts with. In chapter 3 and 4 we go into more detail on how we have used these structures in our implementation.

### 2.8.1 The *TrackView* class

The *TrackView* class is used internally by *GTrackCore* to store and handle all of a region’s track data. When we extract track data from *GTrackCore* we get one of these objects per region. This class is a wrapper for the track data of a region. It provides multiple interfaces for extracting this data. All of the different data columns can be extracted as ndarrays. We can extract either all of the data or just a selection of it.

If we want to save the results of an operation as a new track, then we need to create a *TrackView* object for each region that contains its track data. Each operation has a _calculate_ method that is called for each region of a track. This method needs to return a *TrackView* object if the result is a new track. We will go into more depth on this in chapter 3.

### 2.8.2 The *TrackFormat* and *TrackFormatReq* classes

One very important aspect of track handling, and our usage of tracks in operations, is to be able to consistently identify the track type. Each operation declares a minimum requirement for each of its tracks inputs. In the same manner we want to be able to define the track type for a possible result track. We want our library to be able to easily define and create these objects.

*GTrackCore* does already provide structures for identifying track type and track requirements. It does this via two classes *TrackFormat* and *TrackFormatReq*.

The *TrackFormat* class is used to describe the format of an actual track. Each *Track* object from *GTrackCore*, and its underlying *TrackView* objects, have an associated *TrackFormat* object.

The *TrackFormatReq* class is used to define some minimum requirement which is needed of a track. These two classes are what we use to check the inputs and outputs of our operations. They enable us to stack operations into complex structures where we use other operations as input instead of tracks.

These classes use different names for their properties than what the core informational properties are called. We have included a table that lists the properties that corresponds to each track type. See table 2.5. This table can be used to find the different tracks supported by a given *TrackFormatReq*.

These four properties are *Dense*, *Interval*, *Valued* and *Linked*. All of these are Boolean values. When creating a *TrackFormatReq* we do not need to define all of these properties. This means that we can define a requirement that matches multiple track types. This is very useful in operations where we want to keep any extra data that is not part of the
domain of the calculation. An operation that works on segments can return a linked segment in one case and a valued segment in another.

The *TrackFormatReq* class comes with a method for checking if it’s compatible with a specified *TrackFormat* object. We use this method in our operations to check if a input track is compatible with the operation.

<table>
<thead>
<tr>
<th>Name</th>
<th>Dense</th>
<th>Interval</th>
<th>Valued</th>
<th>Linked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Valued points</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>Linked points</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>Linked valued points</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Segments</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Valued segments</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>Linked segments</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>Linked valued segments</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Genome partition</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>Step function</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>Linked genome partition</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>Linked step function</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Function</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>Linked function</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Linked base pairs</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

Table 2.5: The track types as described by the properties of the track format classes
Part II

The Project
Chapter 3

The design of the library

The overall goal of this project is to create a library that can and will be used. The library needs to be easy to use, but at the same time it should not impose limits on more advanced usage. To achieve this, we first need to take into consideration who the end user is and how they normally use software like this.

3.1 Who will use the library?

Knowing who the user is and their skill level is important when designing an application. Four groups of potential users have been taken into consideration in the design. Common for all of these groups is that they have some advanced computer experience and are familiar working with the Unix shell, using its standard tools and how to write and run small scripts. We do however not expect the users to be expert’s in these topics. The experience gained from something like a software carpentry course will suffice.

The first of the four groups consists of user with little or no formal training in computer science or programming. Their only experience may be using the Unix shell to run programs. These users will be limited to use the CLI interface directly as a tool of its own. The second group consist of users with a bit of programming experience. This experience will enable them to write very simple programs of their own, using the simplified interface. The third group consists of more advanced users who have a better understanding of programing concepts, such as object orientation, and are capable of writing more advanced programs. The last and fourth group are users that have more formal programming expertise. They can implement operations of their own. We have tried to design a library that can be used by all of these user groups.

This is a simplified view of potential user groups. Users do not fall into well defined categories with respects to skill level and experience, but it gives us a simple reference that can be used when designing the library.
3.2 Envisioned usages

In this section we describe the different usages we have envisioned for the library. We have divided the usage into two parts. The first part consists of usages related to performing operations on track data. This is varied, but in large part centred around the user interacting with the library directly on a desktop computer or laptop.

The second part consist of the usages related to the implementation of new operations. As we get more and more track data, there might arise new type of operations that were previously not envisioned. To be ready for such operations, we need to provide the capability for the user to fast and easily implement such operations on their own. The library provides a framework that enable this. The user can write their own operations, ones that interact with the track data directly, or they can nest already implemented operations into a complex structure and use this structure as the basis for a new operation.

3.2.1 From the command line

The simplest usage we provide is to run operations directly on the Unix shell. In this usage we envision that the user just needs to run one or more of the existing operations. The problem is a simple one and there is no need of complex options nor of nesting.

The Unix shell is a powerful and a widely used tool. It comes as standard with a large and powerful set of tools. The shell is very textual in nature and most of its tools uses text as input and output. This enable us to pipe the output from one program directly into another program. Ideally we would have liked our implementation to be compatible with other standard Unix tools like awk, sed and grep, but the object based solution we have chosen does however limit such usages. Existing tools for genomic track analysis like BEDTools are based around a CLI interface working in conjunction with the tools provided on the Unix shell.

The main focus of the CLI interface is to enable a fast and easy usage of implemented operations, and to perform track handling tasks like import and export. The current version is limited in that it only supports running one operation at a time. A future expansion on the CLI might give us the capability to create and run more complex structures. The CLI is designed to be fully dynamic. Any new operations will automatically be added to the interface. The following listing shows how to run an operation on the Unix shell. See listing 2.

```
# python GTools.py union -s -n <trackname-1> <trackname-2> <path to genome JSON>
```

Listing 2: Using the CLI
The two track names reference already pre-possessed tracks in GTrack-Core. The genome definition is given via a JSON file on disk. The two options enables the use of strand information, and how to handle elements with missing strand information.

3.2.2 Raw and object-oriented usage

The framework is divided into two separate parts. The methods that does the calculation on the ndarrays and the object implementation of the operations. There are multiple reasons to split it like this, from a usage standpoint it enables a simplified usage of operations.

3.2.3 Raw usage

We call this usage raw usage, and the operations raw operations. These raw operations do only take ndarrays and options, if any, as input. This simplified usage is meant for users with little programming experience. We are able to run the programs on ndarrays directly without the need to use more advanced programming concepts like objects. In listing 3 we show an example of this usage.

```
from gtrackcore.track_operations.raw_operations.Union import union
import numpy as np

# Two segmented tracks, a and b, as numpy arrays.
aStarts = np.array([2, 10])
aEnds = np.array([6, 15])
bStarts = np.array([50, 89])
bEnds = np.array([67, 1033])

# Find the union
starts, ends = union(aStarts, aEnds, bStarts, bEnds)
```

Listing 3: Using a raw operation to fine the union of two tracks

There are some pros and cons for this usage. It enables a simplified usage that a user with little or no programming expertise might be able to use. However, it removes the capabilities and safety nets provided by the object implementation. It is up to the user to check and fix these. Extraction of additional data is something that the user must do.

Not all operations have a raw operation. In these cases, the user does not have another solution than implementing the raw operations on their own. The simplified nature of these raw operations means that they are very portable. They could easily be used by another application as long as the main concepts from GTrack are used.
Object-oriented usage

Using the class implementation in one of the main usages. This usage provides the most for the user. The user extract tracks as TrackContent objects from GTrackCore and use these as input when creating operation objects. These objects perform a comprehensive set of checks and input validation for the user. The calculation is not performed before the objects calculate method is called. Once the calculation has been performed the results are buffered for later use. A new call to calculate will simply return the buffered result. See listing 4 for a small example of using the class implementation.

```python
from gtrackcore.track_operations.operations.union import Union
from gtrackcore.track_operations.TrackContents import TrackContents

# Given two trackContent objects (trackA and trackB)

# Create the operation object.
operation = Union(trackA, trackB)

# Call the objects calculate method to perform the calculation
result = operation.calculate()
```

Listing 4: Simple class example

Nesting operation objects

The class operations can not only take TrackContents objects as input. Other operations can be used as well. This important feature enables us to create complex tree structures. When the root operations calculate method is called, it will resolve any underlying operations. The nested operations output TrackFormat must be compatible with the other operations Track-FormatReq. See listing 5 for an example of a nested usage.

3.2.4 Writing a script

Both the raw and object implementation can be used in multiple ways. One of the main ones is to write a small script. The operations and options gets defined here and the script get run via the shell. How one chose to write these scripts might vary on what the user wants. Both the raw and class usage described earlier might be used.

3.2.5 Via a Python console

Python is an interpreted language; this means that the code is parsed at run time. As with most interpreted languages Python comes with a console that
from gtrackcore.track_operations.operations.someOperation import SomeOperation
from gtrackcore.track_operations.operations.someNumberOperation import SomeNumberOperation
from gtrackcore.track_operations.operations.union import union
from gtrackcore.track_operations.TrackContents import TrackContents

# Given 4 trackContent objects (track1 to track4)

u1 = Union(track1, track2)
s1 = SomeOperation(track2, u1)
u2 = Union(s1, track3)
s2 = SomeOperation(track4, u2)
u3 = Union(u2, s2)

# Calling the u3 object will resolve s2, u2, s1, u1...
res = u3.calculate()

# We can use a number operation on u3. If not for the call to u3,
# this call will resolve the underlying operations.

n = SomeNumberOperation(u3)
res = n.calculate()

# Another one-line example
res = SomeOperation(Union(track1, track3),
                   SomeOperation(Union(track2, track4), track1)).calculate()

Listing 5: Nested operation usage

enables us to write and run code on the fly. Using such a console is a very
powerful tool when prototyping. The user can see the result for each step
and try a large number of different solutions.

Python comes with a console as standard. This console is a simple one
and it lacks certain features like tab-completion and object and package
discovery. There is an excellent alternative Python console called iPython
that alleviate many of these problems. We make the assumption here that
the best console usage will be through a implementation like iPython, but
this does not mean that the standard console cannot be used.

In this usage we imagine that the user has a task or specific problem
that they need to solve, but they might be uncertain about how. They might
want to combine a couple of operations, or run a simple operation but with
more control over the options then what the CLI provides. Generally, we
expect the problem to either be a ad hoc run or something that later will
be made into a script or in certain cases into its own operation. A Python
console is a useful tool for prototyping new operations. See listing 6 for an
element of this usage.
3.2.6 Writing custom operations

Some users will have problems that are not covered by any of the implemented operations. These problems will range from very simple to complex operations involving thousands of lines of code. We need to provide an environment where the user can easily implement an operation that solves their problem. This process needs to be as simple as possible. We want to offload as much work as possible from the user. The user should not need to write any code related to the operation of the framework itself.

3.2.7 As module in a large framework

The operations library can be used as a module in some larger framework like The Genomic HyperBrowser. There are multiple ways to achieve this. Firstly, one could create a web based portal where a user could create and run operations. The would be analogue to how The Genomic HyperBrowser works today.

Another possibility is to use our concept of operations as objects. These objects can be created locally by the user, defining the tracks and options for each. These objects can then be loaded into a remote system, either via an API call or by saving the object to disk and uploading it. Pickling can be used to save the objects. This possible expansion will provide us with the ability to create and define the operations locally before offloading the actual calculation to a more powerful system. Such a possibility could be integrated directly with the operations framework. A method could be created in the superclass that transferred the object to a remote system.

3.3 What does the library do for the end user

In this section we describe what the library provides for the end user. One of the main goals is to make the library easy to use. A major part of this is to offload as much work as possible from the end user. In this section, we describe the capabilities of the library and what it provides for the user.
3.3.1 An operation framework that supports all of the different track types

The concept of a track type is one that is introduced with the GTrack format. Although it’s not widely used outside of GTrackCore, it’s a powerful one. The existing tools for performing track operations are limited in which track types their underlying track format support. By using GTrackCore we can provide support for all possible track types in one library.

3.3.2 Multiple user interfaces

As described in more detail in section 3.2 we provide a number of different user interfaces. The interfaces vary in complexity, and they provide the user with a choice for each circumstance. Some problems might be straightforward, they can be run directly via the CLI, others might be more complex and requires a solution using a script. The simplified version on the operations enable both advanced and more novice users to write such scripts.

3.3.3 Delayed calculation

The operations objects are based on a concept of delayed calculation execution. None of the calculations will run before the user call the operators calculate method. This gives the user the possibility to first create the operation objects and then stack them together and set options to the desired state. This feature could also be used to schedule the operations at a given time, or to offload the operations to a remove system.

3.3.4 Simplified track handling by using one central container

Tracks are handled via a container class. These objects contain all the information and data that each operation need. The information includes both the track name and the genome definition. The objects will either be created by the user or automatically created when extracting tracks from GTrackCore. The user will not need to interact with the track data in any other form. Each operation will check and extract the correct track data that they require. Any track results will be given as a new track container, that can easily be saved back into GTrackCore.

3.3.5 Track validation and result track format generation

All of our operations have a minimum track requirements defined for each of its inputs. We need to validate that a given input meets these requirements. All of this track validation is handled automatically by the framework.

The framework uses the requirements and check them against the track format of each input track. An error is raised if the requirement and the
input is not compatible. If an operation returns a track, then it needs to have a TrackFormat object corresponding to its track type.

The given track type of such a result track will in some cases be dependent on the track type of the input. In the simplest cases we return the same type of track for any input. In these cases, we can define the TrackFormat statically. In most cases however we need to create the TrackFormat dynamically depending on the inputs.

In such operations, the user should not need to change anything when using different supported track types. This is most relevant when we have extra data that is not used in the calculation itself. If our track has defined values, then the resulting track will have values as well. The developer needs to create a method that generates the correct TrackFormat.

A future expansion would be to create a method that create the TrackFormat object automatically. This might be solved by using a type of configuration where we define the track formats we allow for our results.

### 3.3.6 Nesting of operations

By defining each operations inputs and outputs, we are able to use other operations as inputs. This is a very powerful concept that enable us to stack existing operations into more complex structures. This enables the user to fast and easily create new operations. As the number of operations grows, the number of operations that can be created from other operations grows as well.

This will enable users to create new operations without the hassle and complexity of working with Numpy. The flank operation is such an operation. Flank takes a segmented track and creates flanking segments on the start and end of each segment. These flanking segments are then returned as a new track. We can achieve the same result by using Expand and Subtract. Expand works in the same way as Flank, except that it expands the segments instead. If we first Expand a track and then Subtract the original track, we are left with the flanking segments. We can define this by creating the structure below.

\[
\text{Flank} = \text{Subtract}(\text{Expand}(\text{track}), \text{track})
\]

### 3.4 What does the library do for an operation developer

The other important part of the library is how we implement new operations. In this section we will go into detail on what the framework does for a developer. A major part and focus in this project is to make this process as simple and expedient as possible. We have tried to move as much code and logic as possible up into the main operation super class. Our main thought is to make the number of lines in each operation as small as possible. We provide a default implementation wherever possible. The
user can choose to override these if the operation requires a more specific solution.

### 3.4.1 Minimizing code writing

We have tried to move as much code as possible away from the operation implementations. The developers’ main focus should be on the problem at hand and not on implementing boilerplate code specific to the library. To achieve this, we have tried to make as many parts of the library dynamic. With dynamic, we mean that a solution works for any, current and future operation. The solution needs to work directly for a new implementation without any update or changes to the solution code.

**Overloading methods from the super class** Most methods have a default implementation in the `Operator` superclass. These methods might not always be the desired solution for a given operation. If a different solution is needed, then we simply override the method in question. This approach means that we keep the number of methods that a developer must implement to a minimum. Each operation becomes as small as possible and any deviations from the default is easy to spot in the code.

The default implementations vary in complexity. Some of them do nothing in their default form. These will be overloaded if they are needed and simply ignored if they are not. One such example is the `_setResultTrackFormat` method. This method is used to set and resolve the `TrackFormat` of the result track. If an operation does not return a track, there is no need to define the `TrackFormat`. If the result is a track, we overload this method in such a way that it provides us with the correct `TrackFormat` for the operation.

Others have an actual implementation that, in theory, can be used by all operations. It’s only if the user has specialized needs that these will be overloaded. The `createTrackName` is one such example. It generates a track name to use by `GTools`, or other interfaces, when saving a tracks into `GTrackCore`. The track name is on the form `<operation-name>-<epoch-time>`. If a developer has other needs for the track name, then the method can be overloaded with a new one.

### 3.4.2 A dynamic CLI interface

The library provides a command line interface called `GTools`. The interface is designed to be totally dynamic from the developer’s perspective. New operations are automatically added to the CLI without any need for configuration, interaction or new code on the developer’s part. When an operation is defined in the operations package, it shows up automatically in the CLI, with the correct inputs and outputs defined.

`GTools` loads any modules in the operations package. Modules that start with an underscore are ignored. This means that the operation is available to the user as soon as the class implementation is defined. `GTools` is based on the argparse[26] packages. This packages provides us with
an interface to the command line. It writes help text to the terminal and handles the appropriate options and track inputs for each operation. We can define multiple commands in *argparser* by dividing each command into a separate *subparser*. For each operation we define such a subparser. These subparsers are dynamically created for each operation using its configuration, options and help strings.

### 3.4.3 Dynamic factory for operation objects

The *Operator* super class provides a dynamic operation factory. The factory is capable of generating objects of any current or future operation. This factory is written for use with the CLI *GTools*. It takes the arguments provided by *argparse* as input and parse these to create the correct operator object. Although it's written for *GTools* it can easily be used in other settings such as a script. See the chapter 6 for a possible solutions to making this factory more generic.

### 3.4.4 One simple configuration

Each operation has one configuration that is used by all parts of the library. The configuration is used as a basis when performing input and output validation and in creating the subparser. Each operation is responsible for its configuration. If we need to change it, we only need to do it in one place. This change will affect all relevant code. This means that the user only needs to write the configuration once. Each part of the library will automatically use this configuration.

### 3.4.5 Automatic keyword argument handling and validation

An operation will almost always need additional options apart from the input tracks themselves. These options define the different parameters of each operation, from some value to Boolean flags used to enable or disable options. These options can both be required and optional. All of these options are defined using keyword arguments.

We use keyword arguments to distinguish them from the track inputs and to be able to provide a default value of each. Normally one would define keyword arguments in the *init* method. This way is a bit lacking for our use. Firstly, our operations do not have a *init* method defined and adding one would only add unnecessary code to the class. Secondly we need to define these options in multiple places. The CLI needs both the options names and extra variables like help text. We solve this by defining a *namedtuple* called *KwArgumentInfo*. This *namedtuple* contains all the information we need about each option. This includes the name, help text, the default value and other variables used by *argparser*. All that the user needs to do is to implement a class method that returns a dictionary of these *namedtuples*.

There is no need to implement a *init* method. As we have no *init* method, we need to find another way of access our options. We solve
this by implementing the __getattr__ method in the super class. This class is Python's last resort when looking up a variable of an object. By implementing this method, we can intercept these variable lookups and return the result we want. All of the options defined in each operation can be accessed by calling self._<optionname>. The getattr method then checks if such an option is defined and returns its value. If no value for the option is defined, then its default value is returned. Using this method means that we only need to define the options once. There is no need to define the keyword arguments in the init method of each operation. This means that we can eliminate this method altogether. We can use the info to dynamically create the subparser for each operation. This eliminates the need for the user to define this themselves.

### 3.4.6 Track input and output validation

For each operation we define what the minimum requirements of each input track is and if it returns a track, what the format of the output is. Once these are defined the rest is handled by the framework. Input tracks will be checked against the requirements of the operation. If they are not meet, an error is raised. If the operations are nested, then the framework will iterate recursively through the tree to resolve the requirements, by checking the result TrackFormat of each operation with the corresponding TrackFormatReq of the inner operation. As discussed in section 2.8.2, these two classes are what GTrackCore uses to define track type and track type requirements. The first one is used to define the requirements of a track input, and the second one defines the result track.

### 3.4.7 All track data in one object

All operations in the library use a common track container. This object contains all the information that each operation needs. From a developer perspective this means simplified track data handling. This container hides most of the track handling from the user. The track data and region information is automatically extracted and provided to the _calculate method for each track. The developer’s responsibilities are limited to extracting the needed ndarrays from the TrackView and, if applicable, to create a new TrackView from the results. In some cases, we need to create the result track format given the format of the input tracks. To simplify this process, we provide each tracks TrackFormat as a property in the TrackContainer class.

### 3.4.8 Reuse of common operations

When writing operations, we are often faced common problems. We might want to change the inputs in some way, or to remove inconsistencies from the results.

Merge is an example of such a problem. Depending on the operation we might want to merge any overlapping features in a track. Merging
overlap is a task that quickly becomes difficult and complex, especially when working with all of the different data columns. We want to avoid multiple implementations of such code.

When working on linked data, an operation can often create inconsistencies. For instance, if our operation uses multiple track, we have no guarantee that the node ids are unique across both tracks. We can fix this if we first run an operation that make the ids unique. Another example is related to how GTrack stores links. All links are directional from one node to another. An undirected graph is represented by using a directed link in each direction. If our operation removes a node, we need to check that none of the edges in the graph points to it. If they do, then we would need to remove them.

The framework supports reusing such code. We implement these common problems as normal operations. These operations can then be applied to the tracks before and after the main calculation. We call these operations pre- and post-operations. This is a very powerful simplification that enables the developer to focus on the main part of the problem. We do not differentiate between these help operations and the normal ones. They are for all practical purposes normal operations and can be used as such.

There are multiple advantages with this approach. Firstly, the developer does not need to rewrite or copy existing code. This means that there are fewer places for bugs to be introduced. Future fixes and improvements can be done in one place. As the developer can ignore these common problems we get a faster development time.

Code that uses ndarrays can easily become difficult to understand and debug. Especially when working on code written by other developers. By dividing this code into smaller units we get simpler and more readable code.

There are some drawbacks to this approach. What we in principle do is to run operations in series. We first resolve the pre-operation, then the main operation and finally the post-operation. There is some overhead associated with this. If speed is of the essence, one might want implement the code directly as a raw operation.

Pre-operations need to run at the object initializing. The reasoning for this is that we need to resolve the track type of the input track. As the pre-operation work on these tracks, it might change their type. Object creation can therefore take some time. This breaks the delayed execution use case. We go into more one these problems and how we could solve them in chapter 6.

3.4.9 Encapsulating a nested operation by overloading the calculate method

The library enables us to create new operation by nesting existing operations into more complex tree structures. Some operations can be defined directly by such a tree. Our framework supports the creation of operations by simply defining such a tree. These operations are very simple and fast to implement. To enable such operations, we simply overload the main calcu-
late method from the Operator superclass, define our operation and return its result. We define the number of tracks and options as normal and use these in the calculate method.

This method is simply an encapsulating of a nested operations. We can get the same results by writing a small program that defines the same structure. But the encapsulation provides us the all of the features the framework gives us, as e.g. direct use of the encapsulated operation through the CLI.

As the number of available operations grow, this encapsulation method gets more and more relevant. This method can enable users with less programming experience to create operations that can easily be shared, without the need to delve into the complexity of ndarrays.
Chapter 4

The implementation of the library

In this chapter we describe in more detail how we have solved and implemented the various aspects and problems of the library. The focus of this chapter is to show the problems we have encountered and to show how we have solved them. This chapter is not meant to be a total documentation of every aspect of the library. For more more specific details, one should look to the code directly. In appendix A, we give a more detail description on how the different operations work and how they are implemented.

We have split this chapter into a few main parts. In the first part we talk about how the framework is implemented, how the different parts interact, the distinction between the different types of operations and how the library interacts with the GTrackCore system. We then talk about how tracks are handled. We go into detail on how we use the different track type objects to check and determine the inputs and output tracks for each operation. This in the basis for making the nesting of operations possible.

In the next part we talk about how the library uses indexes from the raw operations to extract any other data from the original tracks. Next we have a quite large section that focuses on the problems we have encountered when working with ndarrays. Here we go into more details on the specifics of the Numpy operations, there peculiarities and how we have solved a few specific problems. The final part is about the operation testing framework is implemented.

4.1 The operations framework

The library is build around a main framework. This framework enables us to offload much of the work from the user. It provides track type checking, nesting of tracks into complex trees, operations that can extract extra data, input validation, delayed calculation, different user interfaces and much more. In this section we will show how this framework is implemented and how it interacts with GTrackCore. This is not complete description of the implementation. We focus on the main parts and the problems they solve.
The framework consists of two main parts. The methods that do the main calculation, we call this methods raw operations, and the class representations of the operations. This classes will always be subclasses of the main Operator class.

As described in chapter 3, we have created this division to enable a simplified usage where the user is able to work with ndarrays directly without the added complexity of using more complex concepts like objects.

4.1.1 Raw Operations

A raw operation is the calculating part of an operations. It contains only one method that calculates a given result for a single region. By design we keep these methods simple and avoid using concepts or objects from GTrackCore. To make the direct usage of these method simple we have split them into their own python packages. These packages contain only one raw operation method per file. Each operator implementation need to imports its own raw operation method. Another part of making these methods simple is to limit the types of inputs they can have. All track data input is to be in the form of ndarrays. Where each type of track data is given as its own array. These methods should only take the track data they need in their calculation as input. It’s up to the user or class representation to extract any additional data from the base tracks.

Given an operation that compare two segmented tracks. The corresponding raw operations should then only take the start and end ndarrays of the two tracks as input. Listing 7 gives an example of a raw operation.

1
2
3
def rawOperation(t1Starts, t2Starts, t1Ends, t2Ends)
# Perform some calculation
return resStarts, resEnds, index, encoding

Listing 7: Example of a raw operation

Our tracks might have more data, but we want to avoid adding them as optional keyword arguments. Doing this complicates the method and makes it less usable on it own. If we add options for all of the possible data arrays, then the resulting inputs gets way to big and complicated. In listing 8 we gives an example of such a version.

Depending on the type of operation we might want to use these add these extra data in the result track. To be able to extract them the raw operation method needs to return an index and encoding array. These two arrays give us the track and original position of each element in the result. We can use these to extract the data corresponding to the results.

In other types of operations, we might not care about any extra data or it simply does not make sense to add it. In these cases, we can simply return the arrays of our results without any index or encoding arrays.

Finally, in some operations we will need to perform calculations or comparisons on the extra data columns if they are present. We want to
Keep this calculation to the raw operation method. In these cases, we have no other choice then to define the extra columns. The *Merge* operation is an example of such an operation. In it we combine any overlapping segments. Any extra data these segments have need to be combined as part of this operation.

Operations that take uses strands are another example. If the strand information is present, then this is something the we will most likely need as part of or calculation.

For all of the main options we should try to limit our usage to simple types like numbers, strings and Booleans. That said we need to balance the simplicity of the raw operation methods with the needs of the class implementation. For example, we use functions as inputs in many raw operations. These are used to overload default behaviour like how values are combined.

The results of a *raw operation* method should be one of two things. Either a new track or as some sort of property of the input track or tracks.

The result of a property operations can in theory be of any type. As long as the result describes some sort of property about a given track. Whoever in practise it’s usually in the form of one or a set of numbers or Booleans. The operations *AverageLength* is such an operation. It returns the average length of the segments in a track.

If the result of a *raw operation* is a track, we give it in the form of a set of *ndarrays*. These *ndarrays* can then be used to create a new track. If keeping extra data makes sense for a given operation, then we can optionally return the index and encoding arrays that we previously discussed. This information can be used to extract and add other data to the result track. See section 4.2.2 for more information on how the data gets extracted.

The complexity of these methods may vary. They may be as small as one or two lines to upwards of thousands of lines of code. In listing 9 we give the signature of a *raw_operation* that takes two segmented tracks as input.

**Changes** How the raw operations work have stayed much the same during the project. At one part of the implementation we experimented with using a separate container class for the raw operations track inputs. This container would be a simple version of the *TrackContents* class,
Listing 9: Signature of Union raw operation, segmented track

containing only the ndarrays of on region. Using such a container would make the inputs for each raw operations simple, especially when operation used many arrays. This would have solved many of the problems we have with the signatures of the raw operation methods getting to big. Using a container, we get a signature on the following form.

Listing 10: Raw operation with track input container

As we can see from listing 10, this is a much more compact and readable then the example in listing 11.

Listing 11: Raw operation using multiple arrays

This usage would also make the implementation of each class representation easier as we could automate the creation of these container objects. On the other hand, we would need extra code to extract all of the required ndarrays in each raw operation. We could use this container class to store result tracks as well. This would make the methods we use to create new tracks simpler and easier to use for the developer of new operations. In the end we felt that the usage of a container object would have made the raw operation usage to complicated. It would have broken the idea behind having this simplified usage. If we were to use this method, we would be better off with removing the concept of a raw operation altogether.

4.1.2 Class representation using the Operator superclass

The other part of the framework is the class representation of the operations. This part constitutes of the main bulk of the library. Usage these objects are the primary way of using the library.

The class representations can to some extent be thought of as a form of wrapper for the raw operation methods. They do both perform the same calculation on the same data. This is where the comparison stops. We now work on a whole track at the time and not on one region. The class
representation performs a multitude of tasks for the user. These range from input validation, data extraction to the creation of new tracks from the calculated results. For each region we extract the correct ndarrays and pass them on to the raw operation. The results are then used to create a new track, adding any extra data from the original tracks.

This class representation means that we can use more advanced usages such as operation nesting. When nesting operations, we use other operations as inputs. When the operations run we first resolve any underlying operations and using their results as an input.

All class representations are subclasses of the main Operator superclass. This class is the core of the library and it’s responsible for the main control logic and input validation. It checks that all input tracks are compatible with the requirements, that the options, supplied keyword argument are correct and ones that are defined for the operation. To make the creation of new operations as easy as possible we have tried to make this superclass as general and dynamic as possible. By doing this we simplify the process of developing new operations as much as possible. When developing new operations, the focus for the developer should be on the calculation itself and not in writing boilerplate code.

Each class implementation should be as simple as possible and only contain the code needed. Any common task should be performed by the superclass. When writing a new operation one should only need to define the properties and configurations of the operations and the calculation method.

The simplest class implementation of an operation contains only two things. The implementation of the abstract method _calculate and the operations configurations in the form of its options and inputs and output requirements.

The method _calculate is the interface between the super class method calculate and the corresponding raw_operation. No calculation is executed before a call to the calculate method. These operations do not only support TrackContents objects as input. The inputs can themselves be other operations.

The Operator class is large and performs many tasks, but the main checking and run logic is based around two methods, these are _checkArgs and calculate. The first one check that the inputs track matches the given requirement and the second is executed when the user starts the calculation. It’s the operations main control loop and is responsible for the iteration over all the regions of a track, and for combining all the region results back into a new track. If the operations are nested it will resolve any underlying operations requirements.

Changes done during the course of the implementation

There have been multiple and extensive changes to the class representation throughout the project. In the initial prototype, much of the work done by each operation was performed by implementations in each implementation. This meant that developing new operations was a more
strenuous task. Each implementation contained a lot of boilerplate code and the options where defined in multiple places.

Example of such code are the creation of the subparser object used by GTools, the parsing and checking of user options, track name generation and factory methods for generating objects.

As we started to write more operations it become clear that this was not the best solution. Many of these methods in each operation were equal across the operations. Options were defined multiple time. Both in the method for parsing options and when creating the subparser. There was also a fairly complex __init__ method in each operator. The early versions were based around using abstract methods, via the ABC[22] package. By using this package, we could control in that the developer had implemented all the required methods. The solution we went with was to move as much code and logic from the implementations into the superclass. Left in the implementations are the calculation and its options. Although this simplifies the development, it locks the user into a default solution. This might not always be the solution we required. For example, the superclass provides a method for generating track-names. This method can be used when saving results. This method retunes the name of the operations combined with the current epoch time. The CLI GTools use this method when it saves result tracks to GTrackCore. In some operations we might want to change how these methods work. To do this one can simply overload the method with one that returns the desired result. This restructuring involved a fair amount of work as all of the methods needed to be generalised. In the end we ended up with an implementation where almost all of the methods have a general or default implementation in the superclass. If the developer needs a specific solution to a problem, then we simply overload the method from the superclass.

4.1.3 Options handling

Each operation can have a number of different options. These options can both be required or optional. With options we mean anything that’s not a track input. These options can be numerous and be used in multiple places in the operations. As we want to be able to present the options to the users automatically, we do also need a textual definition of what each option does, what is possible values are and it default value.

Options handling is designed to be as simple and automatic as possible. All options to an operation are defined by using keyword arguments. As we do not define the __init__ method in the operations, we need to handle and check them ourselves.

To achieve this, we have created a namedtuple called KwArgumentInfo. This namedtuple defines all the information each part of the library needs to know about each options. A namedtuple is a lightweight object type structure in python. It’s perfect for simple structures where we want some object like structure but without the hassle of writing a separate class.

The listing 12 shows the KwArgumentInfo definition.

Many of the values here are meant to be used by the argparser library.
KwArgumentInfo = namedtuple('KwArgumentInfo', ['key', 'shortkey', 'help', 'contentType', 'defaultValue'])

Listing 12: The namedtuple KwArgumentInfo

Key is the name of the options. Shortkey is a shorted version of the name. Help is a string that defines the options and what is does. ContentType is the type used by argparser. Final default value is the value that will be used if the options is not defined by the user. This value is analogues to the default value given a keyword argument.

For each operation we need to define one of these namedtuples. This is done by implementing the method _getKwArgumentInfoDict. This method should return a OrderedDict of these namedtuples. The name of each option is used as key.

In listing 13 we show a simple example of an operations with two options.

@classmethod
def _getKwArgumentInfoDict(self):
    return OrderedDict([
        ('resultAllowOverlap', KwArgumentInfo('resultAllowOverlap', 'o', 'Allow overlap in the result track.', bool, False)),
        ('removeExtras', KwArgumentInfo('removeExtras', 'e', 'Remove all extra information, if present.', bool, False))
    ])

Listing 13: Defining the options of an operation

Now that we have the options defined, we need be able to use them. The first problem is to check that the keyword arguments provided by the user matches the ones that we have defined. Secondly we need to provide a default value for the options that are undefined by the user. Finally, we need to provide all of these values to our code in a dynamic way. The way we solved this was in providing the options as class variables. More on how we solved this later.

Parsing and checking the keyword arguments provided by the user

When we initialize a new operation object, we provide it with a set of keyword argument. As each operation does not define its keyword arguments directly, we need to parse and check them ourselves. This is done automatically by the Operator superclass using the options definition provided by each operation.

The Operators __init__ method takes all the keyword arguments as input via the keyword argument dictionary. This dictionary is then passed to the _parseKwargs method. This method does two things. It checks
that all the given options are valid for the given operation and it creates a dictionary called \_kwargs. This dictionary contains a value for all of operations options. If an option is defined, then this value will be used. If no value is given, then we use the default value specified in the namedtuple. The current version raises a TypeError if the user tries to specify an options that is not defined.

**Providing the options as variables** After the options have been parsed and checked we need to provide them back to the user. This is done by implementing the \_getattr\_[27] method.

The \_getattr\_ method is a special built-in method that python calls as a last resort when looking up a variable of an object.

By implementing this method, we can intercept variable calls and provide the correct value for each options back to the user. We use the \_kwargs dictionary we created while parsing keyword arguments. All options are provided as hidden python variables. For example, if our operation have a options useStands, then we can excess it by calling the self \_useStands variable.

If the variable is not one of the options, we raise a AttributeError. The values we return are from the dictionary we created in the \_parserKwargs method.

Listing 14 shows a simplified version of the one used in the library.

```python
1 def __getattr__(self, name):
2     if name.startswith('_'):
3         try:
4             # If in kwargs, return it
5             return self._kwargs[name[1:]]
6         except KeyError:
7             raise AttributeError("{} not found in kwarg".format(name))
8     else:
9         raise AttributeError
```

Listing 14: Using \_getattr\_

### 4.1.4 A dynamic CLI

The library provides a CLI called GTools. This CLI is written to be fully automatic and dynamic. Any new operations added to the library will automatically show up in the CLI. In this subsection we will show how this was solved.

**Automatically adding new operations**

New operations are automatically added by the CLI interface. This is achieved by using a library called importlib[28]. Importlib is a back port
of the Python 3 `importlib` packages. This package is the implementation of the import functionality from Python 3.1 and onwards.

With this package we can import modules as variables. This makes it easy to create a dynamic method that reads through a package folder and loads all possible modules. We extract all of the operator classes, using the build in method `getattr[27]`, and store these in a dictionary. When creating the subparser we iterate over this dictionary.

The following listing show how we use `importlib`. The method `getOperation` is one that the operations super class provides. It reads the files on disk in the operator package and return these as a list of module and operation names. Listing 15 shows a simplified version of the code used.

```python
module, operations = getOperation()

self._importedOperations = {}

for operation in operations:
    m = "{0}.{1}".format(module, operation)
    mod = importlib.import_module(m)
    cls = getattr(mod, operation)
    self._importedOperations[operation] = cls
```

Listing 15: Importing operations dynamically

The correct operation can then be selected from the user’s choice. Some of the operations provided with the CLI are native to GTools. Import and export of tracks are examples of such operations.

**Automatic creation of subparser objects**

The provided CLI is created using the argparser[26] package.

This package is a framework for creating CLIs. It takes care of usage printing and argument handling. For each operation we need to specify a subparser that defines its inputs, options and help text. Previously this was done by defining each subparser as a method in each operations. This was not a good solution as we ended up by defining the options multiple times. A generic solution was created in the operator super class to fix this. Each operation defines its options, number of tracks, help text for each of the track and for the operation. When GTools creates its parser, it iterates through all of the loaded operations. For each of the operations we use the provided information to create a subparser. New operations in the operations packages will be added to the CLI automatically. We read the operations and create there corresponding subparser in each run.
Dynamic operation factory

When a user starts an operation through GTools, we need to create the appropriate operation object. In earlier versions we provided this by implementing a factory method in each operation. As with subparser this is now solved generally in the Operator superclass by using a factory method. This method takes the results of the argparser as input. From this we get the name of the track, the path to the genome definition file and any options that can be supported via the CLI. This method will then extract the track from GTrackCore and create the correct operation object with these tracks and options.

Result track name generation

The CLI will automatically save any result track that gets generated back into GTrackCore. To be able to do this we need a track name for each new track. The operator superclass has a method for generating such track names. In its default implementation this track name is simply the name of the operation combined with the current epoch time.

4.1.5 Creating new tracks

When an operation returns a track as its result, we first start with a set of numpy arrays that defines this track. This arrays have been created by the operations raw operation method. We need to converted these arrays into a TrackView to be able to save the track back into GTrackCore. As a part of this we often want to extract any extract extra data from the inputs tracks. A raw operation can, as an option, also return an index and an encoding arrays that identifies where in the original track or tracks the new elements originated. This can then be used to extract data that was not part of the operation itself.

The _calculate should return its results as a TrackView object. We have created a helper method createRawResultTrackView, that makes the creation of these objects simpler. This method is located in the package gtrackcore.track_operations.utils.TrackHandling.

This method takes the index, encoding, the original tracks and any new ndarrays as input. All defined ndarrays are used in the new track. If the base tracks contain any other data columns, then we extract the information from those using the index and encoding. All of these together is then used to create the new TrackView object. In some cases, we might want to limit the number of extra data columns we extract. This is possible by specifying the TrackFormat that the result track should have. The extracted columns are then checked against the TrackFormat. Any columns outside of the track type will then be discarded. In general, we should always provide a TrackFormat to ensure that the TrackView we create has the correct TrackFormat. When we create a TrackView it generates a TrackFormat that describes its data.
Due to the way track storage has been implemented in the original \textit{GTrackCore} we need to be especially careful when creating points and partitions tracks. If the \textit{TrackFormat} from the operation is used, we can guarantee that the \textit{TrackView} created is of the correct type.

4.1.6 Integration with \textit{GTrackCore}

The operations library is an extension of the \textit{GTrackCore} system. \textit{GTrackCore} is an integral part of our operations library and we use many of its classes and concepts. All track data are provided to us via \textit{GTrackCore}.

Although our library is an extension of \textit{GTrackCore}, using it will not require an insight or knowledge of how \textit{GTrackCore} work and how it’s implemented. The concepts of track types and what they represent is the main thing that the user would need to know about. Most of the interaction we have with \textit{GTrackCore} is centred around extracting pre-processed track data, and then save a new track back into \textit{GTrackCore}. Although the direct interactions are limited we use many of the objects and concepts from \textit{GTrackCore} extensively throughout the library.

In this subsections we will describe how we interact with \textit{GTrackCore} and what objects we use and how we use them.

The \textit{TrackView} class

The \textit{TrackView}[29] class is a container class that contains all of the pre-processed data for a given region. When we extract a track from \textit{GTrackCore} we get one \textit{TrackView} object per region of a track. The underlying ndarrays containing the regions data is then extracted from these. Listing 16 shows the extracting of the start array from a \textit{TrackView} object.

```python
# Given a TrackView object (tv) for a given region
# We can from this trackView extract the required ndarrays
# Each data type has its own method
starts = tv.startsAsNumpyArray()
```

Listing 16: Extracting ndarrays from a \textit{TrackView} object

After the operations has completed its operation, we use our new ndarrays to create new \textit{TrackView} objects. These objects will be the basis for a new \textit{TrackContent} object. As we work with ndarrays our data is already pre-processed, saving these back into \textit{GTrackCore} is a fast operation. We use the \textit{TrackView} class extensively throughout the library.

The \textit{TrackFormat} and \textit{TrackFormatReq} classes

As discussed in section 2.8.2, \textit{GTrackCore} uses two classes to describe what the format of a track is. The first one \textit{TrackFormat} defines the format
of an actual track. Each Track, and by extension each TrackView, have an associated TrackFormat object that defines the data in the Track. We use the same concept in the TrackContainer class. Each object will have a TrackFormat object that defines its track format. These objects are created and extracted from GTrackCore when we create a TrackContents object. As such the user does not need to deal with them directly. The user will only need to know that the track has a given format and that the operations have a minimum requirement to there input tracks.

In the same way as the TrackFormat describes the format of a track, the TrackFormatReq class describes a set of requirements for a given track. This set of requirements is not attached to a set of data, but is rather used to define what type of track we need in a given situation.

We use these objects both in the operations themselves and in each TrackContent object. Each operation has one or more TrackFormatReq objects that define the requirements of each track input.

Our main usage of these objects is to check that tracks given as input of an operation is compatible with the operation. These two classes together are also what provides the basis for making nesting of operations possible. As we in a consistent way can define what the inputs and outputs of each operation is, checking that the output of one operation matches the input of another is trivial. This enable us to create complex structures of operations, and to easily check that this structure is a valid one.

To test if a track is compatible with an operation we use the method isCompatibleWith from the TrackFormatReq class. This method takes a TrackFormat objects as input and checks if its compatible with the TrackFormatReq class. When we initiate an operation object, all of its input tracks will be check against the corresponding TrackFormatReq object. If they are not compatible, then an exception will be raised.

In each operation we create at least one TrackFormatReq object. This object defines the operations input track. Listing 17 shows how we define these objects in our operations.

```
trackRequirements = [TrackFormatReq(dense=False)]
```

Listing 17: Defining a tracks requirements

If the result is a track, then we need to create the corresponding TrackFormat. Creating this TrackFormat is a more complicated task than creating the requirement and is dependent on what the operation does. Some operations will always return the same type of track. In these operations, we can be defined the track format statically. Others are able to carry over any other data a track might have. In these operation the type of the output track will be dependent on the type of the input tracks.

Expand is an operation that have a minimum requirement of one segmented track. If we have a linked segment track that we want to expand, we want to keep the link data. If we do, then the TrackFormat of the output
track will be that of a linked segment track. If we expand a segmented track, we want the result TrackFormat to be that of a segmented track. This means that for these operations we can not set the resultTrackFormat before the operation object is initiated. The ResultTrackFormat is set by overriding the _setResultTrackFormat method.

In listing 18, we define a static resultTrackFormat.

```python
def _setResultTrackFormat(self):
    # The result will always be a segment track.
    self._resultTrackFormat = TrackFormat(startList=[], endList=[])```

Listing 18: Defining a tracks result TrackFormat

### 4.2 Working with Numpy and ndarray

The GTrackCore system is build to use Numpy[23] and its ndarrays[30] for all of its data storage. Our operations are based around doing calculations and operations on these ndarrays. Working with ndarrays can be challenging. Numpy code with slicing, indexing and padding can fast become non-pythonic and difficult to read and understand.

A large part of the work done in the library has been working with and interacting with different types of ndarrays and operations on them. In this section we will describe some of the problems we have encountered.

The main premise for using ndarrays is that we can use them to run our calculations as vector operations. Using vector operations gives us a considerable increase the run speed to the operations.

#### 4.2.1 Array slicing in Numpy

Array slicing is a concept we use extensively throughout the library. Array slicing is a complex subject and we will not go into detail on every usage here. What we will do is go give a small overview and explanation on that the common slice operations in our operations.

Common for most of the slice operations is that they do not copy the array they slice. They create a view of the array. This view is simply a window into the old array. Any changes to the view is reflected in the original array.

**Checking element \( n \) against element \( n + 1 \)**

One operations that we often perform is to compare \( n \) with \( n + 1 \) in an array. This can be done with array slicing. We create two new arrays. One that contains the whole arrays except for the last element and one that contains the whole element without the first one. We can then check these arrays against each other directly. Listing 19 shows how we do this.
# Given a ndarray

```python
ourArray = np.array([.....])
```

# We then create two slices.
# The first one without the last element
```python
first = ourArray[:-1]
```

# And the second one without the first
```python
second = ourArray[1:]
```

# We can then compare these new arrays
```python
index = np.where((first > second))
```

# We usually do the slicing the the operations itself.
```python
Index = np.where((ourArray[:-1] > ourArray[1:]))
```

Listing 19: Using slicing to compare $n$ against $n + 1$

Using indexing to select a set of array elements

We use a *Numpy* operation called *where* extensively throughout the operations. This and operations like it creates an index we can use to extract a set of elements. In the case of *where* these are the elements that fulfilled the requirements. This indexing operations does not create a new copy of the array. It creates a *Numpy* view. Any changes to the view will change the original array as well. Such an index is capable of extracting an arbitrary set of elements from the original ndarray. When writing operations, we use *Numpy* indexing extensively. Listing 20 shows a commonly used code where we use the index from a *where* operation to extract the results.

```python
# Given the index from the last example.
index = np.where((ourArray[:-1] > ourArray[1:]))

# We can then use this array to extract the result from the original array
result = ourArray[index]
```

Listing 20: Using an index to extract results

Using these indexes is a very powerful tool for our usage. We do often work with multiple arrays that represents different parts of the track. One calculation on one of these arrays can then be used to select the correct sets from the other arrays.

4.2.2 Coding Accumulative cover status trick

When creating operations that combined multiple tracks in some way, we want to know how the elements of these tracks interact. By using a trick
from *The Genomic HyperBrowser*, we can find the cumulative cover status for each element in a region. This enables us to find things as overlap, union, and intersection in a simple way. Although this method is a fast one, the code is a bit difficult to read and understand. We will in this section try to explain how the code by visualizing what it does.

In the library we use this trick in a number of operations. The original code from *The Genomic HyperBrowser* does only find the starts and ends of the new segments, it does not keep track of where the segment originated in the original tracks. We have extended this code to also save which track, and where in it, the track element originated. This enables us to extract and keep any extra data from the original tracks.

We start by having two track $t_1$ and $t_2$. We want to find where these tracks overlap.

![Figure 4.1: Finding the overlap](image)

We can achieve this by coding the starts and ends of each track. The starts with a positive number, and the ends with a negative number. In this example we use 1 and $-1$ for track one, and 2 and $-2$ for track two. See figure 4.2 for a visualization of the codes.

![Figure 4.2: Code assigned to each start and end point](image)

In the code these *eventCodes* are stored as *ndarrays*. *Numpy* provides us with an operation for accumulation. What this operation does is to update each element $e$ to $e_{n} = e_{n} + e_{n+1}$.

```python
# The event codes
eventCodes = np.array([1, 2, -1, -2, 1, -2])
# Find the cover status for each event.
coverStatus = np.add.accumulate(eventCodes)
# The cover status of our track
# array([1, 3, 2, 0, 2, 3, 1, 0])
```

Listing 21: Finding the accumulated cover status

Listing 21 shows how we create the accumulated cover status. We can then use this cover status to select the part of the track we are interested in.

A more intuitive way of looking at it is to assign the numbers 1 and 2 to the segments. We can then add the number together to create a new array.
The parts with a score of 1 is only present in track one, parts with a score of 2 only in track two and finally parts with a score of 3 is present in both tracks. See figure 4.3.

![Figure 4.3: Accumulation of track score](image)

In listing 22 we show a simple implementation. Note the we save both the position value and the event code in the same array. Using division and modulo to extract the position and code respectively.

```python
# First we encode each array,
# We want to store both the position and event code in one array
# To achieve this we multiply each array with 8 as well.
# Dividing with 8 will get the position.
# A modulo with 8 - 4 gives us the event code

# Event code of 1 (5-1)
t1CodedStarts = t1Starts * 8 + 5
# Event code of -1 (3-4)
t1CodedEnds = t1Ends * 8 + 3
# Event code of 2 (6-4)
t2CodedStarts = t2Starts * 8 + 6
# Event code of -2 (2-4)
t2CodedEnds = t2Ends * 8 + 2

# Combine and sort all events into one big array
allCodedEvents = np.concatenate((t1CodedStarts, t1CodedEnds,
t2CodedStarts, t2CodedEnds))
allSortedEvents = allCodedEvents.sort()

# Get all of the event codes, and there corresponding event
allEventCodes = (allSortedEvents % 8) - 4
allSortedDecodedEvents = allSortedEvents / 8

# We create the cumulative cover status for each element.
cumulativeCoverStatus = np.add.accumulate(allEventCodes)

# Get all elements that have a cover status of 3 or more
newStartIndexes = np.where(cumulativeCoverStatus[:-1] >= 3)

# Calculate the length
allEventLengths = allSortedDecodedEvents[1:] - allSortedDecodedEvents[:-1]

# Extract the new starts and ends
starts = allSortedDecodedEvents[newStartIndexes]
ends = allEventLengths[newStartIndexes]
```

Listing 22: Cumulative cover status implementation
Index for extracting extra data from the origin tracks

In the *Merge* operation we use and change all of the different arrays directly. This is because we need combine or select one of them in each case. We then return all of the new arrays. In other operations this is not always the case. In general, we do not want to use more arrays in the raw operations then what the operation requires. But if our track has one or more of the other data columns defined we would to add these to the resulting track. We solve this by retrieving the data from one or more of the input tracks after the calculation.

To be able to do this we need to know which track and where in it to retrieve this data. As introduced in section 4.1.1, this is done by creating and returning two arrays in the raw operations. These are an index array, that gives us the position, and an encoding array that gives us the track. These arrays are created at the beginning of the raw operations. We combine them with the other arrays and in doing so addressing each element. These are then combined and sorted. The rest of the calculation is equal to the original code except that we need to change the slicing operations to 2-dimensional ones. This change will triple the size of the arrays. This might be a problem if size of our tracks starts to approach the size of the system’s memory. For most track operations this will not be a problem.

The example here are on the *Merge* operation, but the principles are mostly the same for any operations. Assign each element its originating “address” and return these as two new arrays. In some cases, it might not be given which track to retrieve data from. If we merge two elements from two different track, is it track A or B we want to use? There is no simple solution to this problem as this extra data is hidden from the raw operation. In most of or operations we have solved this the naïve way by defaulting to track A.

See listing 23 for an example usage of indexing. The rest of the code is equal with listing 22 except for the slicing operations.

4.2.3 Handling multiple overlapping

A problem that often occurs in our operations, and one that we need to deal with, is how to handle multiple overlapping elements.

With multiple overlap, we mean the situation where three or more track elements overlap. Most of our operations are based on using *Numpy* slicing and vector operations. These operations are dependent on the same case being true for all elements in an array. A common usage is to create two new views where we remove the first and last element respectively. When we compare these elements we are in fact comparing $n$ to $n+1$ for all elements in the array.

The problem arises when we want to compare more elements then these two. This is not a problem for all operations. It does often crops up in operations where we remove elements from a track and we want to combine or select one of its data values.

We want to use vector operations as much as possible. By using vector
Listing 23: Cumulative cover status with indexing and encoding

t1Index = np.arange(0, len(t1Starts), 1)
t2Index = np.arange(0, len(t2Starts), 1)
t1Encode = np.zeros(len(t1Starts), dtype=np.int) + 1
t2Encode = np.zeros(len(t2Starts), dtype=np.int) + 2
t1CodedStarts = t1Starts * 8 + 5
t1CodedEnds = t1Ends * 8 + 3
t2CodedStarts = t2Starts * 8 + 6
t2CodedEnds = t2Ends * 8 + 2
allCodedEvents = np.concatenate((t1CodedStarts, t1CodedEnds,
t2CodedStarts, t2CodedEnds))
index = np.concatenate((t1Index,t1Index,t2Index,t2Index))
encode = np.concatenate((t1Encode,t1Encode,t2Encode,t2Encode))
combined = np.column_stack((allCodedEvents, index, encode))
combined = combined[np.lexsort((combined[:, -1], combined[:, 0]))]

Listing 24: Operation that can have multiple overlap

a = np.array([1,2,1,4])
# create n and n+1 by removing the last and first element.
n = a[:-1]
n1 = a[1:]
# n = array([1, 2, 1])
# n1 = array([2, 1, 4])
# We can the use these to compare n with n+1
res = np.where(n > n1)

operation, we are often limited in what exceptions we can make for special cases. We need one simple expression that can be applied to the hole array. These problems can’t normally be fixed directly without adding a lot of complexity and a much longer runtime. The solution we use is to find and resolve any multiple overlap in advance of the main calculation. This adds some runtime to the operation but less than what a non vector operation would be. By doing it in this way we can still use a vector operation in our main calculation.

We show here how we solved this problem in the Merge operations. This example is from the first part of the operation where we combine elements that overlap totally with the previous one. In these, the end of the element at $n + 1$ is smaller or equal to the end at the element at $n$. As the elements are sorted or start and then ends. We can then assume that this
element is totally inside the previous one.

We start by finding the index to all elements of these overlapping elements. Here we find all ends where $n$ is smaller or equal to $n+1$.

\[
\text{overlapIndex} = \text{np.where}((\text{end}[1:] \leq \text{end}[:-1]))
\]

As the element at $n+1$ is inside the one at $n$, we can resolve it by removing the element at $n+1$, merging any values and links it might have.

We do this by creating a removeIndex of all n+1 elements.

\[
\text{removeIndex} = \text{overlapIndex} + 1
\]

This is where our problem arises. If we have multiple overlap, we will then both update and delete the element at index 3.

\[
\begin{align*}
\# & \text{ Indexes of features to keep and to remove.} \\
\text{overlapIndex} &= [2, 3] \\
\text{removeIndex} &= [3, 4]
\end{align*}
\]

To fix this we must need to first merge these element such that there is no overlap between the two index arrays. We repeat this until all overlap has been resolved. Although we introduce an extra for loop, it should not affect the total run time much. The first iterations will remove all of the normal overlap. Each iteration after this will only work on the multiple overlap. The maximum number of iterations for this for-loop is equal to the nr of features in the largest multiple overlap.

The worst case would be a track that consists of one long string of small overlapping segments. In this case the runtime to resolve the multiple overlap would approach $\Theta(n)$

We check for multiple overlap in the start out our loop. We we find any we then remove the overlapping index.

\[
\begin{align*}
\text{inRemove} &= \text{np.in1d}((\text{removeIndex}, \text{overlapIndex}, \text{invert}=\text{True}) \\
\text{if} & \text{ len(inRemove) > 0:} \\
& \# \text{ We have multiple overlap} \\
\text{inOverlap} &= \text{np.in1d}(\text{overlapIndex}, \text{removeIndex}, \text{invert}=\text{True}) \\
& \# \text{ Remove multiple overlap for now:} \\
\text{removeIndex} &= \text{removeIndex}[\text{inRemove}] \\
\text{totalOverlapIndex} &= \text{totalOverlapIndex}[\text{inOverlap}]
\end{align*}
\]

After removing the overlap, we are left with 2 and 4. We then merge these as normal.

\[
\begin{align*}
\text{overlapIndex} &= [2] \\
\text{removeIndex} &= [4]
\end{align*}
\]
The loop then recalculates the overlap for the whole track. As we have merge 4 with 2, and removed 4 in the process, we are now only left with 2 and 3 which we now can merge as normal.

```plaintext
# We then merge 2 and 4, removing 4 in the process
# In the next iteration we are only left with the two last features.
# We then merge these as normal.
overlapIndex = [2]
removeIndex = [3]
```

Using this method assure us that all of the values gets checked or combined with each other. If we are merging multiple partially overlapping segments, we need to save the end value of the largest element as well. We do this by pushing it down the elements as we merge and remove them.

### 4.2.4 Using fractions instead of a static number

Operations that expand or move segments in a track do often have two different modes for specifying the length of this change. Either as a static number that is equal for all segments or as a fraction of each segments length. Using fractions is a common type of options and is used extensively by tools like BEDTools[12].

We solve this problem by first finding the length of all the segments in a track. The array then gets multiplied by the fraction given by the user. We then end up with an ndarray that contains a specific length for each element. As we do only work on hole base pairs in our operations the length array gets rounded and converted to an integer ndarray

```plaintext
shiftLengths = ends - starts
shiftLengths = shiftLengths * lengths
lengths = np.around(shiftLengths).astype(int)
```

In most of our operations this is all that we need to do. We simply overload the lengths variable, the later calculation work for both an integer or ndarray version of lengths. If lengths are an integer, each element gets it added on. If length is a ndarray of equal shape, then each corresponding element in the two ndarrays gets combined.

```plaintext
starts = starts + lengths
ends = ends + lengths
```

### 4.2.5 Padding of ndarrays

When the track data is in the form of arrays, we have a $n \times m$ ndarray. The size of this arrays is as always fixed. This presents us with some problems when our data is not uniform. One element might have zero or one items in
this array and another will be full. Any unused space in these arrays needs
to be padded with some space holder that represents an empty element.
When we create and change such arrays, we need to ensure that all of the
arrays are padded correctly. In practice this means that we need to pad any
array that are smaller than the largest array.

Padding is frequently used when working with the linked data ndarrays.
When our operations do work on the edge and weight ndarrays, we
often need to remove or create new padding. There can be multiple edges
originating from a given node. The edge and weights are therefore often
in the form of a $n \times m$. The linked data form a track will seldom be a com-
plete graph. The nodes will in most cases not have the same number of
edges, but as we work with ndarray then all of our arrays need to be of
the same length. This length is always equal to the the node with the most
edges. It’s technically possible to create ndarrays that have sub-arrays of
different length, but this should be avoided as it will cause a considerable
increase in run time. With arrays of different length, Numpy is no longer
capable of using vector operations. We can achieve this by using the dttype
object. With this dttype, we can use any Python object as an element.

Figure 4.4: The links in a track represented as a graph

Given a track with the graph in figure 4.4, we have the following link
ndarrays as shown in listing 25.

```python
ids = np.array(['1', '2', '3', '4'])
edges = np.array([['2', '3', '4'],
                  ['3', '', ''],
                  ['3', '', ''],
                  ['3', '', '']])
```

Listing 25: Showing the padded ndarrays of a linked track

As we can see the arrays are padded with empty strings. When we want
to change these arrays, we need to add the appropriate padding. This can
fast become very complicated as we may want to add edges to an already
padded edge. To avoid having to deal with this we simply add the new
edges and padding at the end and remove the extra padding later.

```python
e1 = edges[0]
e2 = edges[1]
```
# Adding the edges from 2 to 1
newEdges = np.concatenate((e1, e2))

# We now need to extend the edges array to the same size as newEdges.
padding = np.zeros((len(a), len(a[0])), dtype=newEdges.dtype)
edges = np.c_[edges, padding]

# The updated edge can now be added.
edges[0] = newEdges

Each node does now have the correct edges, but we have unnecessary padding that we need to removed. The result we want is arrays with a length equal the maximum number of edges in a node.

```python
# Edges after merge and padding, Note the excessive padding
np.array([["2", "3", "4", "3", ",", ","]
          [["3", ",", ",", ",", ","]
          [[",", ",", ",", ",", ","]
          [["3", ",", ",", ",", ","]]])

# First we create a temporary array to store the array
# Note that this array will be an object one, but since we only
# use it for storage and not for any vector operations, the added
# runtime will be small.
newEdges = np.array([None] * len(edges))

# For all edges, select only the edges and not the padding.
# This removes any padding inside the array as well.
for i, edge in enumerate(edges):
    ind = np.where(edge != ",")
    newEdges[i] = edge[ind[0]]

# Find the longest edge array
maxLength = max(len(x) for x in newEdges)

# Recreate the edges by adding the correct padding to each array.
edges = np.array([np.pad(x, (0, maxLength-len(x)), mode='constant')
                  for x in newEdges])

# Edges after the padding is fixed,
np.array([["2", "3", "4", "3"]
          [["3", ",", ",", ","]
          [[",", ",", ","]
          [["3", ",", ",", ","]]])
```

### 4.2.6 Handling strings in ndarrays

As discussed, working with strings in ndarrays can pose some problems. Each element of a string ndarray have a given allocated size $n$. This size is defined by its dtype. If the dtype of a ndarray is 'S5' than there is allocated
space for string of length 5 in each element. If we try to assigning a string with size $n+1$, then only the first $n$ characters will be saved in the \textit{ndarray}.

When creating operations, we often come across cases where we would like to save strings that are longer than what the \textit{dtype} of the current \textit{ndarrays} allows. When doing operations on links we often update the ids of one or more elements. In listing 26, we show an example where we change the \textit{dtype} to allow longer strings.

```python
# Get the ids to update
newIds = ids[updateIndex]

# Update them in some way
newIds = np.array(['updated-{}'.format(x) for x in range(len(newIds))])

# If the updated ids are large, we need to recreate the ids array with a bigger dtype.
if newIds.dtype > ids.dtype:
    ids = ids.astype(newIds.dtype)

ids[updateIndex] = newIds
```

Listing 26: Handling strings in \textit{ndarrays}

Note that this creates a new copy of the array. This is an expensive operation and we want to limit it usage. When used, care should be taken to find the size of the largest possible string first, and then do the copy once. As we allocate enough space for a string of size $n$ at each element, we should avoid using long strings for only a few elements, as this can make the array unnecessary large.

### 4.3 Track handling in the library

The creation and handling of track makes up a large portion of the interactions the user have with the library. One of the main objects we have for the library is to make it as user friendly as possible. An important step in this is how we create, handling, and ultimately save tracks. The user should not need be acquainted with a plethora of different classes and concepts from \textit{GTrackCore} to be able to perform these tasks. To facilitate this, we have created a main track container called \textit{TrackContents}, and a genome definition container called \textit{Genome}. A user of the library will only need to use these objects when handling tracks. All track operations in the library is based around this container class. The API of \textit{GTrackCore} supports exporting and importing tracks in this container.

#### 4.3.1 The \textit{TrackContents} class

The Track concept encompasses multiple types of data, definitions and names. Managing all of these will fast become a cumbersome task for the
user. To alleviate this, we created a container class called TrackContent. This class stores all of the track data, genome definition, name and any other supporting data. We use this container class in all track interactions in the library. All operations take TrackContent objects as inputs and all operations, that returns a track as result, will return it as a TrackContent object. The following list describes the main properties of the class.

**TrackViews** A OrderedDict of the tracks TrackViews, with the region as key.

**Genome definition** A Genome definition object. This object gives us the name of the genome and describes the regions of the track. What the regions are, and what their bounding regions are.

**TrackFormat** A TrackFormat object describing the format of the track.

The content of the container is available as multiple properties. These are used by the super class of the operations to extract the track data.

- **trackViews** The TrackViews as an OrderedDict, region is used as key.
- **trackFormat** The TrackFormat object that describe the format of the track. Extracted from the TrackViews.
- **allowOverlap** If the track allows overlapping elements.
- **regions** The regions of the track. The keys from the TrackViews dictionary.
- **genome** The genome definition as a Genome object.

**Saving results**

The TrackContents objects are located in memory until they are saved in some manner. Saving these tracks back into GTrackCore is a common operation that the user will perform often. For this reason, we want to make it as simple as possible. To achieve this, we have implemented a save method in the TrackContainer class. This method takes a name and automatically saves the track for the user. As the resulting track from an operation is already pre-processed, saving it to GTrackCore is a fast process involving no new pre-processing. When the user calls the save method we initiate a API call to GTrackCore. The specific method used in the API[31] is importTrackFromTrackContents. This API call creates and runs a PreProcessTracksJob to save the track.

**Changes**

**The genome definition as separate container class** In the start of the project, we save the genome definition as a dictionary in the TrackContent class. As work progressed we soon discovered that this approach would not be the best solution. A single Genome definitions can be used by many different tracks. Saving these as a dictionary could lead to problems for the user. We might end up with errors in the different versions used. What we ended up with was to move all of the genome definition into a separate genome container class. This approach is more modular.
and allows a greater reuse of the genome definitions. We can now easily check that two tracks are on the same genome by comparing their genome objects.

**Integrating saving** There have been multiple iterations on how to save tracks back into GTrackCore. The first iteration we used a `preprocessTrackJob` directly. This was later simplified by adding a API call to `GTrackCore`. This API call creates and runs the `preprocessTrackJob`. During the implementation it did fast become evident that this saving operations is one that will be reformed often. The final solution was to integrate saving into the `TrackContent` object itself. This is done via a save method. This method takes a name provided by the user and saves the track using the API.

**Variables as properties** In the first iteration of the `TrackContent` class, we used normal variables with getter and setter methods. These methods were used to checked that the new variables were valid. As part of a general code tidying we converted these to Python properties. This made the code more Pythonesque as we got rid of all of the getters and setters methods. At the same time, we still kept the added protection of not modifying the variables directly.

### 4.3.2 The Genome class

For each track, we need a definition of its genome. This definition describes the bounding regions of each of the genomes region. The bounding region is simply the length of each region. As an element can’t be outside its region, we use this information to check that the data is correct and remove any overflow created by an operation. In `GTrackCore` these definitions are stored as a Python dictionary. To make these definitions easier to work with and more portable, we have created a container class called `Genome`. This class contains the genome definition as well as the name of the genome. A future expansion would be to make a system for handling these genome definitions.

**Generating genome objects** The genome definition is often given as a set of regions-id, region-length pairs. There are multiple ways one could represent these pairs. In our implementation we have added support for two such structures. Firstly, we have the data structure that is used internally. This structure is a simple python dictionary with the region id as key and the length as value. The other one is a `JSON` version of this same structure. This `JSON` structure contains the genome name as well. To be able to easily create `Genome` objects from these types of structures we have provided two provided methods. The first one, `generateGenomeRegions`, takes a dictionary and genome name as input. An example of the dictionary can be seen in listing 27.
```python
gdef = {
    "chr1": 247249719,
    "chr2": 242951149,
    "chr3": 199501827,
    "chr4": 191273063,
    ...
    "chrM": 16571
}
g = Genome.generateGenomeRegions("HG19", gdef)
```

Listing 27: Creating a *Genome* object from a dictionary definition

The second one, `createFromJson`, is almost identical, except here we give the file path to a *JSON* file instead of a python dict. The *JSON* contains the name as well, so we do not need to supply it. Listing 28 shows this method in use.

```python
g = Genome.createFromJson("~/tracks/genomes/hg19.json")
```

Listing 28: Creating a *Genome* object from a *JSON* definition

The *JSON* file needs to be on the format shown in listing 29.

```json
{
    "name": "HG19",
    "size": {
        "chr1": 249250621,
        "chr2": 243199373,
        "chr3": 198022430,
        ...
        ...
        "chrM": 16571
    }
}
```

Listing 29: *JSON* format used to store a genomes regions

### 4.4 Testing

An important part of any software project is to test the code and verify that it works as expected. The main focus of this section is on testing the implemented operations. We have written many test that test the various aspects of the library and its framework. These test are not that complex and difficult so we do not talk in length about them here.
When testing the library and its operations, we have used a package called `unittest`[25]. This package provides us with a framework for performing unit testing. Unit testing is a technique for testing software. With unit testing we divide a program into parts or units and test that these unit performs as expected. The size of these units will depend on the code being tested. The important part is that this unit can be isolated.

### 4.4.1 Testing operations

When we write new operations, we want to be able to test that they give us the desired results. When writing scientific software, it’s important to be able to prove that our solutions are correct.

When we test the operations we test the operation as a whole. We create a set of small tracks that contains the different situation we want to test. Keeping these tracks small and rather create many of them makes the tests more transparent. We then run these operations and check that the results match the expected results.

One of our main goal is to write a library that is easily extendable with new operations. As a part of this we need to make sure that testing them is easy as well.

We have created a template, and a set of help methods, that help the user with testing the operations. This template can show in full in appendix C.

This template makes testing simpler in multiple ways. The `unittest` framework is based around implementing a sub class of `unittest.TestCase`. In this class we create one method for each of our unit tests. Every method with a name that starts with test will automatically be added as a test.

All of our test performs the same task, they run an operation on one or more tracks and compare the result to the expected one. This means that we can create one method that perform all of these tasks. Listing 30 show such a method from the tests for the merge operation.

```python
def testPointsWithStrandsPositive(self):
    self._runTest(starts=[10,10], strands=['+','+'], expStarts=[10],
                  expStrands=['+'], expTrackFormatType="Points")
```

Listing 30: Method for running an operation test

As we can see this make each test makes each of the test methods simple and straightforward. We can fast implement new tests as all we need to do is to define the input and output. There is no need to create the structures and objects we would normally need. The `_runTest` method creates the correct tracks, run then through the operations and compare the result with the expected one. An optional expected track format can also be supplied. This can be used to test that the track format of the result tracks matches the one we expect. As each operation is not equal, we need to make some
changes to this method in each case. We have moved some of it into separate a separate test utilities package. This package contains methods for creating the test tracks.
Part III

Conclusion
Chapter 5

Discussion

In this chapter we discuss the project and the implementation of the operations library. Have we fulfilled the goals for the project? Which parts worked in practice and which did not?

5.1 The user interfaces in practice

The implemented library provides multiple user interfaces. These interfaces work in different ways and are designed for users of different skill level. In this section we will discuss their implementation and if they worked as expected.

5.1.1 The raw operations

The raw operation concept was envisioned to provide a simplified usage of the library. This usage would only use a limited set of programming concepts and thus make the library more accessible to users without a formal programming background.

But does this concept really make for a simplified usage? On the one hand we have provide the users with simple methods that only uses ndarrays and options as input, but are the ndarrays simple to work with, and are the user able to get their track data in the form of these arrays?

To be able to answer the, we need to know the problems that these users have, and how they usually solve them. These are users how write and use simple scripts in their daily work. They are used to working with and manipulating textual data, but not in more advanced concepts like object-oriented programming. The tracks data these users have is probably limited in complexity. They will in most cases be of the simpler track types like segments or points.

The simplified interface will of useful for these users, but there are some challenges that needs to be solved.

The user will most probably have track data defined as a file in some common file format. This file would first need to be parsed into the arrays. Implementing such a parser is of course not something we could expect of the user. A possible solution would be to create a simplified track parser.
This parser could be based on \textit{GTrackCore} and generate the required \textit{ndarrays} for the user. This parser should be able to take the commonly used file formats and return track as \textit{ndarrays}.

Many of the track types can be quite complex. For example, the arrays that represents links are almost always padded. Our raw operation expects these arrays to be on the form that \textit{GTrackCore} provides. Working with these track types will be more challenging.

In some of these cases one could make the argument that the command line interface provides a better solution. The \textit{CLI} uses the class implementation and the users will then get all of the benefits of the object in a simplified form. But this might be more a question of user preference.

Starting to use a new tool takes time. If the tool is too complex or too different from the current workflow, then they might not give the tool a chance. This is an important point to make. We want to provide a solution that's our users can relate to and understand.

The class representation provides the users with multiple safety nets, input validation and simplified track data handling. All of which much be handled by the user when using only the raw operations.

\subsection{Object based operations}

The library's main interface is via the object implementation of each operation. They work much as expected and provide the user with new capabilities like nesting. The operations perform track validation and sanity checks for the user. Most importantly these operations extend the track type concept to the operations. Each operation is defined by the minimum track type of its input and the track type of its output.

Many of the classes we use from \textit{GTrackCore} are themselves wrappers around the things like the track data. Many of our structures and classes are new wrappers around these objects. We could have used these objects directly and avoided the need for an extra level of wrappers. This would have reduced the runtime, but it might have made the library more complex to use. Another possibility would have been to change these classes to suit our needs, but this would involve a major restructuring in \textit{GTrackCore}.

\subsection{Pre and post operations}

We provide two methods that can be implemented in an operation. These methods enable us to run one or more operations on our track, both before and after the calculation. These methods were added to enable the reuse of common operations such as Merge.

These two methods create a sort of middle ground between a normal operations and the simplified operations. With these methods it becomes simpler to implement new operations. We can use other operations to fix common problems before the raw operations. This means that we can make the raw operations much simpler and we can focus on the problem at hand.

On the other hand, we add an overhead in the runtime for the operation. In some cases, it might be better to fix these problems in the raw operations.
Nesting of operations

The operations support nesting of operations into complex structures. We do this by allowing operations as input of other operations. When we run the outermost operations, the underlying tree gets resolved recursively. We use the track type definition and required classes supplied by GTrackCore to check and validate that all of these operations are compatible with each other.

Nesting of operations will add an overhead in the calculation. The operations run one at a time and return their results to the next operation. In each step the track or tracks gets copied. A better solution would have been to pass this information more directly and as soon as it was ready. Multiple operations could then run in parallel.

5.1.3 CLI interface

We have implemented a command line interface called GTool. This interface is meant to be used on the Unix shell. This is the simplest user interface we provide. It does not require any insight or understanding into the inner workings of the library.

The CLI is fully dynamic. It automatically loads all of the operations that are defined in the operations package.

This interface should be the go to solution for simply running an operation. There are however some weaknesses to it, and we will go into more detail on these now.

Genome definition handling

Handling of genome definitions is a bit cumbersome. In each operation we need to provide the genome definition for its tracks. This definition is currently in the form of a JSON file on disk. In each operation we need to provide the file path to the correct definition.

This is both a problem with the CLI and with GTrackCore. GTrackCore does not provide a structure for storing and retrieving these definitions. This genome handling could have been done in a better way. We could have created a common repository for such definitions. We would then only need to provide the genome name to the operation. Using this name, we could have retrieved the correct definition. We could still use the JSON files for the definition, but only importing them once.

Only one calculation at a time

The CLI does only support running one operation at a time. Ideally we would have liked to support a usage like what BEDTools provide. It supports piping its results to another operations or to one of the standard UNIX shell tools.
These tools that user them are generally very fast. The pipes run in parallel, with the output of the first program being used by the second program as fast as it arrives.

As we work with binary data, such piping becomes more difficult. We would need to wait until we have the full track object before returning it to the piping. This would mean that the first operation would finish completely before the second one could start. None of the standard UNIX tools could be used either as these tools are almost exclusively text based.

Our CLI does not provides any such capabilities. We run an operations and its results track gets saved into GTrackCore. If we want to perform a new operation on the result, we would first need to extract it. We could then start a new operation using this track as input. This make the CLI cumbersome to use, especially if we want to perform a series of operations.

A possible solution would be to support some type of string definition. The user would specify a string of operations that we then parse and turned into the corresponding tree structure. The following listing show an example of how such a string could be defined.

```python
GTools.py -s "subtract(flank('track1', s=10), 'track2')"
```

Another possibility is to create a type of operation builder in the CLI. Here we could define the operation by adding the on to each other and at the end running this structure.

### 5.2 Implementing new operations

One of the main goal was to make the writing of new operations as simple as possible.

The library supports two ways of implementing operations. The normal one that uses code that work on the ndarrays, and a simplified version that uses the fact that we can nest operations together to create new operations. We discuss this simplified implementations in subsection 5.2.1.

Our main focus has been in making the code necessary for each operations as small as possible. The Operator super class provides dynamic methods that can be used by almost all operations. This means that we do not need to implement them in each case. The operations define a configuration that these methods use. The main focus for the developer can then be on the difficult part, which is the Numpy code.

These general methods do however not always provide the correct results. They are in most cases hidden from the user, and this might lead to confusion when debugging. Some of these methods are used by other parts of the library. We need to better document these and provide the developer with a good guide on how to implement operations.

We do not provide the user with much help when it comes to the raw operations. More work should have been done in making these easier to implement. We could have created a library of common operations and used it to divide the problems into smaller parts.
5.2.1 Simplified implementation by nested operations

We have created a simplified way of implementing new operations. These operations work by using the nesting properties of or object implementation.

We use already implemented operations and arrange them into a tree structure. We can then wrap this structure into an operation class. This class will take the required options as input or simply define the operations statically. This can be done by simply overriding the calculate method from the Operator super class.

These operations can be implemented and change fast and easily. The developer does not need to handle the ndarrays in any way. The track format will also be provided by the tree. Users who normally would not be able to create a operation can now create operations on there own. This opens the library up to a large number of new users.

These operations will however have a longer runtime then the the Numpy ones. Each operation in the tree is completed before it returns its result to the next track. This involves copying the track between the different operations. If the tree is large or runtime is of the essence, then a normal implementation will be preferred.

These operations do not, for obvious reasons, have a raw operations method. As such we have no simplified version of the operations to provide our users.

5.3 Our usage of Numpy and ndarrays

Our calculations are mostly based on performing operations with ndarrays. These operations are can be complex and the code difficult to understand. We have tried to document this code to the best of our abilities, but the nature of the code means that it will not be trivial for a new developer to understand all of it.

Parts of the code could have been structured in different ways to avoid this. We could have divided the parts into multiple methods to compartmentalise what the different parts do. The merge operation is divided into multiple methods. But these are there mostly because we need to perform the same operation multiple times.

On the other hand, one of the main reasons for using Numpy operations is to get an improved runtime. We can’t sacrifice the to much just for readability’s sake. If readability was the main goal, we would have implemented the operations with normal linear loops.

5.3.1 Common Numpy sub-operations

Many of the operations use blocks of Numpy code that performs similar tasks. With many different implementation of the same code, we are bound to get errors or inconsistencies.

As the library stands today, we have not move these operations into a package of common packages. Such a package would have made the process
of implementing new raw operations easier. If we want to improve the code, we would only need to do it in one place. Operations for handling links is a good candidate. These are often complex and with many exceptions and problems associated with array padding.

5.4 Runtime

The runtime of the operations has not been a focus for this thesis. That said have not written our code oblivious to its runtime.

Pre-processing tracks into its binary representation can take some time, especially when the tracks are large.

As this is a task that our user would often perform we need to discuss it here. The proposed introduction of the binary file representation BTrack, would alleviate this as tracks could be distributed in their binary form.

It might be possible to improve runtime of the pre-processing, but this is way outside the scope of this theses. When considering the runtime of the operations we assume that the tracks are already pre-processed.

The work performed by the framework is small compared to the raw operations. The operations in the framework that might take some time do all related to the copying of track data. As long as the size of the tracks is does not approach the size of the systems memory, then these operations should not impact the overall runtime. If we have very large tracks, then we will get into problems from paging. This would also impact the raw operation so we can ignore it in this context.

The raw operation constitutes the bulk of the work done by each operation. It’s this method that performs the calculation and transformations on the track data. Optimizing these methods will provide the larges improvements in runtime. When implementing operations, our main focus has been to always use vectorised operations. We have tried to avoid code that iterate linearly over the arrays. Using vector operations can be a strenuous task. We are often limited to operations that work on the array as a whole, without the possibility of adding exceptions.

We do often need to divide the operations into multiple steps. In some cases, we can solve these problems easier by using a normal loop. When using a loop, we need to be sure that it works on the exceptions only and not the array as a whole.

Many of the operations require us to make copies of the data arrays. Copying large arrays can take time and we should avoid it if possible. Some of the implemented operations rely to heavily on copying arrays. Rewriting these would improve their runtime.

As a general rule we try to avoid the use of the object dtype. By using this dtype we can create arrays of arrays with different length. This might sound tempting when working with the inflexible ndarrays. But using such a structure makes it impossible to run most vector operations. They can however be useful in some cases. In the merge operation we use such a structure to save temporary results. We have a set of arrays of different length that we want to store. As we do not perform any operations on this
array, we should not get an excessive runtime.

As it was not a part of this project, we have not tested or compared the runtime of the different operations. We have not used much time on optimizing the individual operation. The calculating part of the operations is separated into its own method. This separation makes it easier to optimize the operations at a later date. When implementing the raw operations and working with the Numpy operations we have tried to avoid the usual pitfalls and used vector operations whenever possible. There are probably many cases where the runtime of the operations could be improved easily.

5.5 Do the operations return correct results?

Providing results that are consistent and correct is an essential part of any application. This becomes, if possible, even more essential for a tool that performs scientific calculation. The users of such software need to be able to trust their tools and the results they return.

5.5.1 Testing the framework and its operations

When testing the library, we have split the tests into two different parts. First, we test the framework part of the library. That is any part that is not directly related to the specific operations. Secondly we test the operations themselves. Although we make this divide, the tests are not unrelated. All of the operations tests use the framework. As such we test the framework when testing the operations. The tests of the operations are much larger and more comprehensive. At the time of writing there were 474 different operation tests. All of these running a different track on one of the implemented operations. We have based much of our testing on these tests.

We have elected to not test the raw operations directly. Each operation uses its corresponding raw operation. The class operation is a wrapper for this method. With the exception of operations that have pre- or post-calculate methods, we do not change the data before or after they are returned. Testing the raw operations themselves would one add an already created test. For the operations that changes to the data we need to create tests that disable these. There are some operations that have such methods that we cannot disable. In these cases, it might have been an advantageous to test the raw operation directly. We have not made any such tests, but we think that the operation tests should be sufficient. Many of the operation tests could easily be converted to test the raw operations directly. One would simply need to change the run test method to use the raw operations instead of the class representation. The options used are in most cases the same, and the arrays could be used directly without the need to package them in a TrackContents. There might be cases where errors have been overlooked either because of typing errors or in misunderstanding how a given operation work.
5.5.2 Test template

We provide an extensive test template for testing each operation. With this template we try to make it as easy and effortless to implement test for a new operation. In these tests we simply define the input tracks, any options and the expected result and result type. All tracks and expected results are given as simple Python arrays. By doing this we avoid the need to create both the TrackContents for the tracks and operator object in each test. Both of these tasks is offloaded to a run test method. This method creates all of the required objects, runs the test and compare the result with the expect one.

Testing code can be a tedious and boring task, often left as a last minute task if not forgotten totally. This library is meant to be expanded with more operations, as a part of this it’s important to make testing as simple as possible. With the provided test template, we provide the building blocks for a test driven development of new operations. The user can early in the process start to define tests for an operation. As the operation gets more complex, all that the user needs to do is to update the run method.

This template is far from perfect. The current version is still just a template. The user will still need to partially implement the run method for each operation. This could have been done in a way where the user would only need to define the tracks, options and expected results.

Ideally we could have made part of the testing automatically. In each operation we define its input, output and options. An automatic test suit could have used this information to create and run a set of test on each operation. These automated tests would have been limited as we have no way of knowing what the test does and from this what the expected results would be. But we could have tested that the operation returns the declared type of track etc.

What we could have provided was a more automatic test framework. We could have created a dynamic method for running the tests. The user would then only need to define the tracks, options and expected results for each test. The options went through a major restructuring at a late stage in the project. The possibility of such automated tests did not become clear before this restructuring. At that point we did not have the time to implement it.

5.5.3 Possible sources of errors

We have identified tree main possible sources of errors. The first one is errors that are introduced from the operations. These errors will be limited to one single operation. Error in the operations are the most probable ones. Part of this code is often complex and involves using Numpy operations. Numpy code can easily become difficult to understand. With multiple slicing, combination and encoding tricks. This type of code is prone to errors. Especially when changing an existing code base.

The second source of possible errors is the framework itself. We have created a set of test for testing various parts and concepts introduced by the library. Possible sources of error are, how we handle the tracks, in validating inputs, extracting the correct data and the creating of new tracks.
from the results. Much of this is not trivial to test on their own. We have base much of the testing of these part on the testing of the operations themselves. As the operations test uses these parts we can from them conclude that they work as expected in the scope of each test.

The third source is would be from some bug in \textit{GTrackCore}. Through this project we have not found any bugs or errors that originate from \textit{GTrackCore}. We have not created any tests specific to the part of \textit{GTrackCore} that the library uses. \textit{GTrackCore} is a big system with a large and complex code base. There is already implemented a large set of tests for it. To be able to extend these test in any meaningful way, we would need to have a much more comprehensive understanding of \textit{GTrackCore} than what the scope of the project accommodates.

Error could of course be introduced from other sources. The data in each track might have inconsistencies that are not handled when pre-processed by \textit{GTrackCore}. One of the numerous external libraries might have bugs that none of our test catch. The library will run on multiple Unix type systems and on multiple versions of these. The version of the external libraries used might differ across these systems. These different versions might contain bugs not present on the version used to implementing this project. We have elected to not take these sources for error into consideration. Most of these are either out of our control or outside the scope of this theses. The object of this section was to discuss if the code we introduced have such errors.

\section*{5.6 Conclusion}

We have expanded the \textit{GTrackCore} system with a library of track operations. This library introduces object based operations that use the track type concept. We use the \textit{Numpy} arrays that \textit{GTrackCore} provides and get the speed improvements of using vectorised operations.

The library supports multiple uses and have multiple user interfaces.

One of our main goals was to provide an object-oriented implementation that used the track type concept. The library provides such a structure. Each operation is implemented as its own class using a common super class.

These operations define the track type of there inputs and outputs, this enable us to use operation as input for other operations. This enable us to create trees of operations.

We wanted to provide a simplified usage that enabled users without formal programing experience to use the library. This is provided trough the CLI, \textit{GTool}, and with a simplified version of each operations. We call these for raw operations. These are the methods that do the actual calculation in each operation. They are used by the object implementations as well. These methods provide a simplified usage where we limit the number of concepts used. The CLI does also provide a simplified usage as it can be used as normal Unit shell tool.

We have implemented as set of operations in this framework. These
operations are mix both of common operations and new ones that show the capabilities of our solution.

As the number of possible operations is vast, one of our main goals was to make the implementation of new operations as simple as possible. We have moved as much work as possible to the super class. This class provides dynamic methods and factories that each operation uses. When implementing a new operation, we do only need to define the configuration and the methods that perform the actual calculation. This enables the user to focus on the problem at hand and not on writing code related to the framework.

Secondly we can define a new operation by using existing operations as building blocks. We use the fact that operations can have other operations as input. These operations can be combined into new operations. We create a new operation class that wraps around this tree. This enables the user to create new operations very fast and easily. These operations do not require any code that works on the ndarrays. This is a big simplification and enable more users to create new operations.
Chapter 6

Future work

In this chapter we expand on some of the possible future expansion to the library. This work encompasses some of the ideas we did not have the time to implement and some thoughts on the long term lines of the library.

6.1 Creating more operations

Most of the future work will be in creating more operations. The current version of the library contains a mix of common operations and new ones that showcases its capabilities. The number of possible operations is quite large. We need to identify which operations are needed by the community and have these implemented.

One of the main goal of this project has been to make the implementation of new operations as easy as possible. This will alleviate some of this work. The capability to combine existing operations into new ones will also make this work easier. As the number of implemented operations increases, so does the number of these complex operations. The community at large could then write and share the operations they create.

6.2 Expand existing operations with new features

There is possible extension that could be added to the implemented operations. These extensions encompass support for new operations and options to optimizing the calculations themselves. Not all operations have complete support for using strands. Adding this and other feature should be relatively straightforward and would improve the library.

6.3 Do not run the preCalculation method at object initialization

In the current implementation, we run the preCalculation method at the initiation of each operation object. This brakes the design goal of delaying
the execution of the calculations. With this usage the user can first create the operation, or tree of operations, and then run them. Running this method in the __init__ method brakes the usage as it might involve a big operation that can take time.

The reasoning for running this method at initiation is that we need to resolve the TrackFormat of both the input tracks and the operations result track at creation. The current way of validating the track inputs is to extract the TrackFormat from them. A preCalculation works by running one or more operations on the input tracks and replacing them with the result tracks. These operations might change the TrackFormat of these tracks. This applies to the TrackFormat of the results as well. In some operations the track type of the result is dependent on the input tracks. It's imperative to know the correct TrackType of the track inputs and output. These are used to validate the input and as the basis for track nesting.

A future expansion would enable the resolving of the TrackType without the need to first run the PreOperation. Another possibility is to only allow operations that does not change TrackType.

### 6.4 Remove the pre/post methods

We use two methods, preCalculation and postCalculation, to apply other operations to our track at the start and end of the calculations. This is a powerful usage that enables the reuse of common operations instead of implementing them in each case. But this usage creates some problems. Firstly, it makes the operations more complex to understand. Which parts run when might be difficult to understand at first. Secondly we now run the pre method in the initialisation of the object. This means that the object creation could take time. This brakes the delayed calculation usage. We do not want to perform any calculation before the calculate call. A better way might be to define each operation as a nested tree. This would make for much more readable operations. The current Union operation can be defined as the following expression.

\[
\text{Union}(t_1, t_2) = \text{Merge}(\text{Union}(\text{UniquifyLinks}(t_1), \text{UniquifyLinks}(t_2)))
\]

This could then be defined in the operation.

```python
class Union(Operator):
    def defineOperation(self):
        operation = Merge(Union(UniquifyLinks(t1), UniquifyLinks(t1)))
```

This is similar to how we defined complex operations today. This expansion would expand on this and allow running a raw operation as well.
6.4.1 Creating new layer of operations

One possible solution is to create a new layer or type of operation. Instead of the current hybrid version, we could divide them into two different types. One for complex operations that have its structure defined as a tree and a raw version that that do only work directly on the ndarrays. This new type will mostly not be used by the end user, but instead be called from the complex operations.

```python
class RawUnion(RawOperation):
    # The new type of operation
    # Does only do work on ndarrays, no pre/post methods etc.
    def _calculate(self, region, tv1, tv2):
        # Extract ndarrays, call operation method, create and return
        # a new TrackView
        return tv

class Union(Operation):
    # The complex operation that contains the tree
    tree = Merge(RawUnion(UniquifyLinks(t1), UniquifyLinks(t1)))
```

6.5 Calculate one region at the time

We currently finish the calculation of the hole track in each operation before returning any data. This is not ideal and something we would want to improve. This problem is a two fold one. When working with track that's are large then the system memory, we would get into performance issue with paging. In these situations, we would like run the calculations on one region at the time. For this to work we would also need to create a version of TrackContainer that only load one region at the time to memory.

Secondly we want to build the basis for more optimized and parallelized operations. The main premise for this improvement is that we want each operation to return the result for each region as soon as it is calculated. This will enable the next operation to start its calculation as soon as the first operations has finished one region. This will enable to run multiple parts of the operation at the same time, but without the need to write parallelized code for each operation. This problem is also one that need to be solved if we want to create a for pipe for the operations.

6.6 Parallelization and optimization

The current version does not support parallelization. Adding this would be a big plus. The simplest way of achieving this would be to run multiple regions at the same time. Operations are for the most part region independent. Running multiple region calculations at the same time should then for most operations not present any parallelization problems. As
discussed passing the regions to a new operation would also enable us to run multiple operations at the same time.

6.7 Making the library portable and easy to install

6.7.1 Ease of installation GTrackCore

Today the installation of GTrackCore is a bit complicated and requires the installation of multiple other dependencies. This is less of a problem when we have a couple of servers but if we want scientist to use the library, and by proxy the GTrackCore system on their personal computers, then we need to make this process easier. The ideal would be to use a package system of some sort and use it to resolve the dependencies. The first and simplest solution would be to create a Python packages and add it to PyPi[32].

6.7.2 Create a BTrack version that is more independent of GTrackCore

BTracks is a proposed binary file format that will contain the pre-processed data of a track. One possible direction in making it easier to use and install the library is to create a simplified version of GTrackCore that only supports a limited number of operation on BTrack files. Extracting data, making new BTrack files etcetera.

6.8 Expand GTools

The current version of GTools is a simple one that is mostly limited to loading tracks and running operations. There are multiple directions in which it should be extended.

6.8.1 Improve track handling

Track handling in GTools is a bit cumbersome and needs to be improved. The current version is based around importing and extracting track into GTrackCore.

Improve Genome definition handling

For each track we need a genome definition that defines the tracks bounding regions. In the track operations library this is done via the Genome class. These Genome objects can be created directly from a JSON file. In the current version of GTools we need to supply the corresponding JSON file for each operation. This usage is prone to error as one could easily supply the wrong definition for the given track. There are multiple ways we could fix this problem. The simplest we would be to make a sort of library of genome definitions. When we load a track of a given genome
then we lookup the definition in this library. If the definition is missing, then and only then would we need to define it. This could be done via a importGenome method. The other possibility is to make the genome definition a more integral part of GTrackCore. This is a more complex solution, that will involve more work, but it's probably the better solution in the long run as it solves this problem generally and not only for GTools.

**Temporary storage of tracks**

The current version saves the result of any operation directly back into GTrackCore. This is not always desired. Some result will only be used as a step in a larger calculation. The saving, retrieving and ultimately deletion of this track is unnecessary work that the user should not need to perform.

To fix this we could add some form of temporary track storage buffer. This buffer would store the tracks until we chose to save them. Adding this buffer is an important step in supporting multiple operations in GTools.

There are multiple challenges in implementing such a buffer. As we would like to use tracks in this buffer in a later operation, it can't live in memory. Saving them back to GTrackCore takes time and add unnecessary complexity. We want to make it possible to save the results to this buffer instead of in GTrackCore. Add support for using this tracks in input of a new operation. Support for this buffer will make supporting multiple operations simpler.

### 6.8.2 Support for running multiple operations

The current version of GTools does only support running the operations one at a time. This is quite limiting for the general usage of the CLI. If we want to run two operations is series on a track, we will first need to run operation 1, find the result track, and then run operation 2 on it. This does not make for a very good or fast workflow. There is no way of nesting operations, or piping one after another. Adding this feature would make the CLI much more powerful.

**Tree structure as a string**

One possible solution would be to supply the CLI with a string that defines a tree of operations. To achieve this, we need to implement parser and define a syntax for these strings. Listing 31 gives an example of how one would run such a string. Any options are defined in the tree.

```python
GTools.py parse flank
(union(track1, filter(track2, removeLinks=True)))
```

Listing 31: Possible extension to GTools. Support for defining the operation as a string
Building on this we could add support for saving and reusing this strings. To achieve this, we would need to extend the parser to support a placeholder track. It would need to identify these tracks and dynamically find the number of required tracks in each tree.

See listing 32 for an example on how such operations could work. One would also need to add support for manipulating these trees. Listing, deleting, copying, etcetera.

```
$ python GTools.py parse -s <name> flank\(\text{union}\{\text{track1, filter}\}
  (\text{track2, removeLinks=True})\})

$ python GTools.py parse -r <name> <track1> <track2>
```

Listing 32: Possible extension to GTools. Saving and running saved track strings

**Support piping**

Another possible solution would be to support *Unix* type pipes. Pipes enable us to send the output of a program to the input of another program. This is a very powerful concept and an integral part of most terminal implementation. *BEDTools* support this today, adding support for pipes would make the two tools more compatible. See listing 33 for an example of such a pipe.

```
$ python GTools.py flank track1 | python GTools.py filter --removeLinks
```

Listing 33: Possible extension to GTools. Support for pips

There are many problems in adding support for pipes, and it may be that implementing it is not feasible within the constraints of the framework. We give an overview of the main problems here.

**Tracks as objects** We represent tracks as objects. *Unix* pips on the other hand are designed to use streams of data, usually text. It can handle streams of binary data, but these are more complex to handle. This presents us with a problem. How do we transfer the track data between the different operations in the pipe? We could wait for operation one to finish and transfer the whole track, but ideally we would like operation one to pass each region on as soon as they are finished.

**Returning results as soon as we have them** The pipes work as a stream, the outputs of program one is immediately available to program two. As long as each program returns there results continuously the all
run in parallel. The second program will begin to use the inputs as soon as they arrive. To take advantage of this we would need to review parts of the framework. It is probably not feasible to return each element as it is ready. Our calculations are in large part based on vector operations. These are performed on whole region at the time. A simple solution and one we probably want to implement anyway is to return each region as soon as the calculation on it is completed.

Operation 2 in the pipe could then start work on the regions as soon as they are ready.

**Multiple inputs** Operations can support multiple tracks as input. If we want to support piping, how do we first run two different operations and combine them into a new one? We can solve this natively in bash with parenthesis, but this might be a bit too complex for some users. See listing 34 for an example.

```bash
(echo 'b'; echo 'a') | sort
```

Listing 34: Bash, piping multiple programs into one

### 6.8.3 Tab completion

Many operations have long names and even longer options. Adding tab completing would improve the usability of the tool. There are multiple tools and libraries that can be used to achieve this. Argcomplete[33] looks like a natural place to start as we are already using the argparser[26] library to create the command line interface.

### 6.9 Automated testing of operations

Operations are currently tested with manually defined tests in a test framework. We would like to expand this test framework with a set of automated tests. These automated tests should work on any present or new operations. As its difficult to define how an operations work, we need to limit these test that are common for all operations. Such as testing if the declared result TrackFormat matches the TrackFormat of the output track.
Journal references


Other written references


Online references

9. The Genomic HyperBrowser. URL: https://hyperbrowser.uio.no.
27. Python getattr. URL: https://docs.python.org/2/reference/datamodel.html#object.%5C_%5C_getattr%5C_%5C_ (visited on 2016).
29. GTrackCore - The TrackView class. URL: https://github.com/sivertkh/gtrackcore/blob/master/gtrackcore/track/core/TrackView.py (visited on 2016).
32. The Python Package Index. URL: https://pypi.python.org/pypi (visited on 2016).
33. The argcomplete package - Bash tab completion for argparse. URL: https://argcomplete.readthedocs.io (visited on 2016).
40. VirtualBox, a general-purpose full virtualizer for x86 hardware. URL: https://www.virtualbox.org/ (visited on 2016).
41. The fork of GTrackCore containing the code for this project. URL: https://github.com/sivertkh/gtrackcore (visited on 2016).
Appendices
Appendix A

Operations supported by the library

In this appendix we give an overview over the operations that are currently implemented in the library. This appendix is meant to be read as documentation and a user guide. We provide a description on what each operation does and how to use them. For each operation we give a detailed description on what its track requirements are, what options are and how these options change the operations.

We have tried to make this description and names match the variable names used in the configurations of each operation as possible.

The operations can be divided into two main parts. Ones that give a new track as the result, and the ones that return some other type of information. This information is usually some form of information about the track. Although this division is useful in some settings we have chosen to order all the operations in alphabetical order. We have done this to make this overview simpler to read, and easier to lookup a given operation.

We base our describing of the track requirements on the four properties from the TrackFormat class. These properties are dense, interval, valued and linked. For a detailed descriptions of which track types these properties corresponds to, see table 2.5.

We only define the properties needed. If we only define one property, we do not care about the other three. For example, if the operations only require that the input track is not dense, then it support both valued and linked tracks. In these cases, these properties are in often just carried over from the input or ignored in operations that do not return a track.

In some cases, we need to restrict the track requirements further. We might need a certain type of values, or the links need to be weighted. In the cases where this is relevant we write these requirements in the same form as the 4 main properties.

**Debug printout** Most of the operations have an undocumented debug options. This options is a Boolean flag that enable some debug printout. It varies to what extent, if any, that this options prints debug information. We have for this reason elected to exclude it from the documentation, and only
mention it here. In a later version this option might become a standardized one or remove altogether.
**A.1 Average length**

![Example track](image)

(a) Average length  
(b) Average length ignores overlap

(c) Custom average function

Figure A.1: The average length operation

---

**Description**

This operation takes a none dese track and finds the average length of its elements. The length is given as number of base pairs. The result is returned as a `OrderedDict` with the region as key and the average length per region as the value.

The length can be in the form of a fraction. Although it might not make sense to talk about a fraction of a base pair, we might miss important information by rounding this number. We leave it to the user to decide how to best interpret this result.

This operation does not distinguish between overlapping and non overlapping tracks. It finds the average length of each element, overlapping or not and returns the average length. If one does not want this overlap in the average, then the track needs to be merged in advance.

By default, this operation uses the package `numpy.average`[34] to find the average. The operation supports overloading of this average function. In figure A.1c we show this usage with a custom average function that ignores small segments.
### Inputs and output

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input tracks</td>
<td>1</td>
</tr>
<tr>
<td>Input minimum requirements</td>
<td>Non dense</td>
</tr>
<tr>
<td>Overlap allowed in inputs</td>
<td>Yes</td>
</tr>
<tr>
<td>Result is track</td>
<td>No</td>
</tr>
<tr>
<td>Output track format</td>
<td>n/a</td>
</tr>
<tr>
<td>Overlap allowed in result</td>
<td>n/a</td>
</tr>
<tr>
<td>Other results</td>
<td>The average length of the elements in each region</td>
</tr>
</tbody>
</table>

### Options

**customAverageFunction** Override the default average function with a custom one. The default is `numpy.average`[34].

A.2 Average link weight

![Image of a graph with vertices 1, 2, 3, and 10, connecting with edges labeled 1, 2, and 3.]

Figure A.2: The links of a track visualised as a graph

Given a track that describes the graph in figure A.2. This operation will then return the average of the weights which in this case is \((3 + 10 + 2)/3 = 5\).

Description

This operation finds the average weights of all links in a track region. The result is returned as a `OrderedDict` with the region as key and the average length per region as the value value.

The operation does not make any simplification to the graph created by the links. We count all links event if there are multiple ones from a node to another. If a more complex average is required, then a set of transforming operations needs to be performed in advance. For example, we might want to look at the minimum spanning tree of our graph and find its average. To do this we would first need to run an operation on the track that transformed its links into the minimum spanning tree. For simpler problems a custom average function might suffice.

The average function in the operation can be overloaded with a custom one. The default average method used is `numpy.nanmean[35]`. We use `numpy.nanmean` and not `numpy.average` here as there can be multiple weights per element and the padding cause `np.nan` entries that we need to handle.

Inputs and output

<table>
<thead>
<tr>
<th>Number of input tracks</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked with weights.</td>
<td>The default average function will only work on numbered weights.</td>
</tr>
<tr>
<td>Input minimum requirements</td>
<td></td>
</tr>
<tr>
<td>Overlap allowed in inputs</td>
<td>Yes</td>
</tr>
<tr>
<td>Result is track</td>
<td>No</td>
</tr>
<tr>
<td>Output track format</td>
<td>n/a</td>
</tr>
<tr>
<td>Overlap allowed in result</td>
<td>n/a</td>
</tr>
<tr>
<td>Other results</td>
<td>The average weight of the links per region</td>
</tr>
</tbody>
</table>
Options

customAverageFunction Override the default average function with a custom one. The default is `numpy.nanmean[35]`. When using a custom average function, one needs to take the shape of the weights into consideration. As per the GTrack definition the weight can be on a number of different formats, and with different types. We assume that the user is aware about this when using this function. The weights array is passed to this function without any testing or sanity checking.
A.3 Complement

Description

The complement operation takes a gaped track as input and create the complementing track. The complementing track is the track that covers all of the area that the input track does not cover. There is no overlap between a track and its complement.

As we look on the coverage of the track, any overlapping segments will be merge in advance of the operation run. The merge is done via the *Merge* operation. See section A.10 for more information on the *Merge* operation. There will never be any overlap in the resulting track and it will always be a segment type track.

Any other data in the input track will not be used in the result. We have no way of knowing who this data would map across the complementing track. A possible future expansion to this operation would be to give the user a way to map this data.

This operation supports the use of *strands*. If strand information is available, then the operation will follow it. It can be disabled by the user. When using strand information, we divide the track into two parts and treat these separately. We create a complementary track for the upstream segments and one for the downstream. These are then combined into a single track. The strand information is of course saved in this process. We treat any elements with missing strand information as on the upstream by default. This can be change via the options.

In figure A.4 visualise the track as a set using a venn diagram. The circle represents the area the track cover and the box the whole area of the
genome. When finding the complement, we return the total area of the genome minus the area of the track.

**Inputs and output**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input tracks</td>
<td>1</td>
</tr>
<tr>
<td>Input minimum requirements</td>
<td>Non dense</td>
</tr>
<tr>
<td>Overlap allowed in inputs</td>
<td>No</td>
</tr>
<tr>
<td>Result is track</td>
<td>Yes</td>
</tr>
<tr>
<td>Output track format</td>
<td>Non dense, intervalled, no other data is carried over</td>
</tr>
<tr>
<td>Overlap allowed in result</td>
<td>no</td>
</tr>
<tr>
<td>Other results</td>
<td>n/a</td>
</tr>
</tbody>
</table>

**Options**

**useStrands** Follow the strand information when creating the complementing track. When following the strand, we divide the features on strand and create two complementing tracks, one for each of the strands. The resulting features will have the corresponding strand in the new track.

**treatMissingAsNegative** Treat features with missing strand info as if they have negative strand. The default is to treat them as if they are positive. To ignore element with missing strand information, we need to remove them in advance.
A.4 Count elements

![Diagram](image_url)

- Nr of elements: 3
- Nr of elements: 4

(a) Normal count
(b) Overlap does not influence the result

Figure A.5: The count elements operation

**Description**

Count the number of elements in a non dense track. The result is returned as an *OrderedDict* with the region as key and the number of elements per region as the value.

This operation does not distinguish between overlapping and non overlapping tracks in the input. All defined elements of a track will be counted. If a non overlapped result is required, then the track needs to be merged in advance.

**Inputs and output**

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input tracks</td>
<td>1</td>
</tr>
<tr>
<td>Input minimum requirements</td>
<td>Non dense</td>
</tr>
<tr>
<td>Overlap allowed in inputs</td>
<td>Yes</td>
</tr>
<tr>
<td>Result is track</td>
<td>No</td>
</tr>
<tr>
<td>Output track format</td>
<td>n/a</td>
</tr>
<tr>
<td>Overlap allowed in result</td>
<td>n/a</td>
</tr>
<tr>
<td>Other results</td>
<td>The number of elements in each region</td>
</tr>
</tbody>
</table>
A.5 Coverage

Description

This operation calculate the coverage of a track. Coverage is simply a measurement of how much of the region our data covers. In detail we mean how many base pairs the elements of the tracks covers. This gives us a measurement on how large a portion of the genome the track "covers". The result is returned as a OrderedDict with the region as key and the coverage per region as the value.

This operation does not handle overlapping segments. If there are overlapping segments in the track then the overlap will be counted twice. To alleviate this, use the Merge operation to merge the segments in advance. See section A.10 for more information on the Merge operation. Figure A.6b and A.6c visualises this problem. In the first figure two of the base pairs are counted twice giving us a total coverage of 13. In the second figure the elements are merged and we get the correct coverage of 11.

This operation will only work on non dense tracks. Dense tracks covers the hole region by definition, and can be ignored here. The region sizes will equal the coverage for these tracks.
**Inputs and output**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input tracks</td>
<td>1</td>
</tr>
<tr>
<td>Input minimum requirements</td>
<td>Non dense</td>
</tr>
<tr>
<td>Overlap allowed in inputs</td>
<td>Yes</td>
</tr>
<tr>
<td>Result is track</td>
<td>No</td>
</tr>
<tr>
<td>Output track format</td>
<td>n/a</td>
</tr>
<tr>
<td>Overlap allowed in result</td>
<td>n/a</td>
</tr>
<tr>
<td>Other results</td>
<td>The coverage in number of base pairs per region</td>
</tr>
</tbody>
</table>
A.6 Expand

The **Expand** operation, as the name implies, is used to expand each element of a track a given number of base pairs. This can be done in either both or one of the directions. This operation is the analogous to the **Flank** operation, except that we do not remove the part covered by the input track. **Flank** can in fact be defined using the **Expand** and **Subtract** operations.

\[
\text{Flank} = \text{Subtract}(\text{Expand}(\text{track}), \text{track})
\]

For more info on the **Flank** and **Subtract** operations see section A.8 and A.14. We can expand the elements either equally with a given number of base pairs or as a fraction of each element’s length. Overlaps in the input track is supported. The calculations do only look at each element and is not dependent on a merged track. The finished track can be merged if the user wishes. The operation handles any over or underflow of the track region that might be introduced. We cut the segments if they are expanded below 0 or above the bounding region.

This operation supports using **strand** information. If the track has strand information, then the operation will use it. If this is not wanted, then the user can disable it via the options. When using the strand information, the expanding will be in the direction of the strand. If we expand equally in both directions the strand information can be ignored. See figure A.7b for an example on how strand changes the workings of the operation. Elements with missing strand information will by default be treated as on the upstream. This can be changed to on the downstream by the user.

**Post calculation**

This operation has a **postCalculation** method defined. This method will merge any overlap that was introduced if we do not allow overlap in the result track. Overlap in the result can occur when we expand into the neighbouring element.

\[
\text{Merge}(\text{Expand}(\text{track}))
\]
**Inputs and output**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input tracks</td>
<td>1</td>
</tr>
<tr>
<td>Input minimum requirements</td>
<td>None dense</td>
</tr>
<tr>
<td>Overlap allowed in inputs</td>
<td>Yes</td>
</tr>
<tr>
<td>Result is track</td>
<td>Yes</td>
</tr>
<tr>
<td>Output track format</td>
<td>None dense, any other data is carried over</td>
</tr>
<tr>
<td>Overlap allowed in result</td>
<td>Yes</td>
</tr>
<tr>
<td>Other results</td>
<td>n/a</td>
</tr>
</tbody>
</table>

**Options**

- **resultAllowOverlap** Allow overlapping segments/points in the result track. The default is *False*. Any overlapping feature are merged in *postCalculate*.

- **both** Expand both the start and ends an equal amount of base pairs. This option can not be used at the same time as *upstream* and *downstream*.

- **downstream** Expand the starts a given number of base pairs. This option can be used in conjunction with *upstream*. If we use the strand information this option will follow the strand direction.

- **upstream** Expand the ends a given number of base pairs. This option can be used in conjunction with *downstream*. If we use the strand information this option will follow the strand direction.

- **useFraction** If set we treat the inputs of *both*, *downstream* and *upstream* as a fraction of each features length. The default value is *False*.

- **useStrands** Follow the strand information if present in the track. Only affects the *downstream* and *upstream* options. Default is *True*.

- **treatMissingAsNegative** Treat features with missing strand info as if they have negative strand. The default is to treat them as if they are positive. To ignore elements with missing strand information, one would need to remove them before running the expand operation.
A.7 Filter

The *Filter* operation removes one or more of a track’s extra data columns. This can be useful if we do not need this data in our calculations, removing them first can improve the overall runtime. This operation does not change the main type of the track; it only removes data. We can remove the values from a valued segments track and get a segment track as result. What it does not support is converting a segments track to a points track, or a genome partitioning track to a segment track.

This operation is input safe. That is, you do not need to check if the track used as input have the data column you want to remove. The operation will do a best effort attempt to remove the chosen columns. If they are present they will be remove if not, then the option is ignored.

*Link* removal is split into two parts, *removeLink* and *removeWeights*. The first one removes all link information and the second one removes any weights. We have divided it this way, as just removing the *ids* or *edges* columns by themselves make little sense. When removing *ids*, we would need to remove *edges* and *weights* as well.
**Inputs and output**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input tracks</td>
<td>1</td>
</tr>
<tr>
<td>Input minimum requirements</td>
<td>No requirements</td>
</tr>
<tr>
<td>Overlap allowed in inputs</td>
<td>Yes</td>
</tr>
<tr>
<td>Result is track</td>
<td>Yes</td>
</tr>
<tr>
<td>Output track format</td>
<td>The track format that corresponds to the input track with the chosen columns removed. If we removed links from a linked points track, the result track format will be that of a points type track.</td>
</tr>
<tr>
<td>Overlap allowed in result</td>
<td>Yes</td>
</tr>
<tr>
<td>Other results</td>
<td>n/a</td>
</tr>
</tbody>
</table>

**Options**

- **removeStrands** Removes any strand information from the track.
- **removeValues** Removes any value information from the track.
- **removeLinks** Remove any link information from the track. This will remove weights as well.
- **removeWeights** Remove any weight information from the track, leaving the rest of the link information alone.
- **removeExtras** Remove any extra information from the track.
A.8 Flank

The Flank operation is used to create flanking segments for each feature of a track. The flank can be created in one or both directions. A new track is created that contains these flank segments. No part of the original track is kept. This new track will always be a simple segment track. All other data columns are removed. We think that most normal usages would not need them. If they are needed in some corner case, the other implementation mentioned below will create the flanks with the extra data columns.

The flanks can either be defined statically as a number of base pairs used for all flanks, or as a fraction of each element length. Any over or underflow of the track region is handled by the operation. If these occur the flank will be cut at either zero or the bounding region. Under or overflow might be introduced when we have elements close to either the start or end of a region.

This operation is analogous to the Expand operation, except here we do only keep the expanded part. For more information on the Expand operation see section A.6.

The current implementation of this operation is based on a raw operation that performs calculation directly on the ndarrays. But this is not the only way to express this operation. It can also be created by nesting the Expand and Subtract operation.

\[ \text{Flank} = \text{Subtract}(\text{track}, \text{Expand}(\text{track})) \]

Here we first expand our track to match the flanks, we then subtract the original track from the expanded one to get the flank track. This implementation will give each track any other data columns from their corresponding element. These can then be removed using a Filter operation. For more information on Expand, Filter, Flank and Subtract see sections A.6, A.7, A.8 and A.14 respectively.

This operation supports using strand information. If the track has strand information, then we will use it. If this is not wanted, then it can disable it via the options.

When using strand information, we create flanks in the direction of the strand. If we flank is equal in both directions the strand information can

Figure A.9: The flank operation
be ignored. Elements with missing strand information will by default be treated as on the upstream. This can be changed to on the downstream by the user.

**Post calculations**

This operation has a postCalculation method defined. This method will merge any overlap that was introduced if we do not allow overlap in the result track. Overlap in the result can occur when we have neighbouring segment where the length of the gap between them is smaller than the flank size.

**Inputs and output**

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input tracks</td>
<td>1</td>
</tr>
<tr>
<td>Input minimum requirements</td>
<td>None dense</td>
</tr>
<tr>
<td>Overlap allowed in inputs</td>
<td>Yes</td>
</tr>
<tr>
<td>Result is track</td>
<td>Yes</td>
</tr>
<tr>
<td>Output track format</td>
<td>None dense and inversed, no other data is carried over.</td>
</tr>
<tr>
<td>Overlap allowed in result</td>
<td>Yes</td>
</tr>
<tr>
<td>Other results</td>
<td>n/a</td>
</tr>
</tbody>
</table>

**Options**

- **resultAllowOverlap** Allow overlapping segments/points in the result track. The default is False. Any overlapping feature are merged into one.
- **both** Create a flank segment, of equal length, at both the stars and the ends. This option can not be used at the same time as downstream and upstream.
- **downstream** Create a flank segment at the start of each feature. This option can be used in conjunction with upstream. If we use the strand information this option will follow the strand direction.
- **upstream** Create a flank segment at the end of each feature. This option can be used in conjunction with start. If we use the strand information this option will follow the strand direction.
- **useFraction** If set we treat the inputs of both, downstream and upstream as a fraction of the features length. The default value is False.
- **useStrands** Follow the strand information if present in the track. This only affects the downstream and upstream options. The default value is True.
- **treatMissingAsNegative** Treat features with missing strand info as if they have negative strand. The default is to treat them as if they
are positive. If one wants to ignore elements with missing strand information, then one needs to remove them from the track in advance.
A.9  Intersect

![Image of intersect operation](image)

(a) Intersect, segments  
(b) Intersect, with values

Figure A.10: The intersect operation

Description

The *Intersect* operation takes two tracks and finds their intersection. With intersect we mean the any area where both tracks have an element defined. This intersection is returned as a new track. Any part of the two tracks that are not present in both will be removed. The resulting track will always be of the same type as track A. The results will be as one segment, not as two totally overlapping segments.

If the track A have any other data columns, then these will be carried over. It’s only data from track A that will be saved to the new track. Any data from track B will be discarded. This might not be the best solution to this problem, but it’s difficult to create a general solution that covers all cases. Does keeping this data even make sense when we only look at the overlap. The data might not even be compatible. Track A contains a values column of numbers for some physical property and track B only characters.

The current version of intersect does not support using strand data, but this could easily be added in the future.

![Figure A.11: Intersect as a venn diagram](image)

Figure A.11: Intersect as a venn diagram

Figure A.11 shows the operation as a set operation. We visualised it here as a venn diagram. This operation is equal to the set operation $A \cap B$. 

111
**Inputs and output**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input tracks</td>
<td>2</td>
</tr>
<tr>
<td>Input minimum requirements</td>
<td>None dense</td>
</tr>
<tr>
<td>Overlap allowed in inputs</td>
<td>Yes</td>
</tr>
<tr>
<td>Result is track</td>
<td>Yes</td>
</tr>
<tr>
<td>Output track format</td>
<td>None dense, any other data is carried over.</td>
</tr>
<tr>
<td>Overlap allowed in result</td>
<td>Yes</td>
</tr>
<tr>
<td>Other results</td>
<td>n/a</td>
</tr>
</tbody>
</table>

**Options**

allowOverlaps Allow overlap in the input tracks.

resultAllowOverlaps Allow overlap in the result track. The default is False. Any overlapping feature are merged into one.
### A.10 Merge

(a) Merge of overlap

(b) With values

(c) Using strand

(d) Naive strand merge

(e) With links

Figure A.12: The merge operation

#### Description

The *Merge* operation merges any overlapping features in a track. Any available data will be merged into a single feature. If the data is not needed, then it can be filtered out in advance by using the *Filter* operation. See section A.7 for more information on the filter operation.

This operation transforms a track that allows overlap and converts into one that does not. Multiple other operations use this operation to remove any overlap that gets created by the operation.

The value data column can have a range of different data in a number of different formats. Merging the data from two elements might not always make sense. This operation will make a best effort attempt, but the default is based on the assumption that the data is numerical. To enable merging on other types of data, the user can overload the function used to merge values with a custom one. We do not get the same problem when merging links and their weights as we simply keep all links.

This operation will follow the strand direction if strand information is present. When using strands, we only merge elements in the same
direction. Any elements with missing strand information will be treated as if they are on the upstream. This can be changed by the user. If one wants to ignore the elements with missing strand info, then these needs to be filtered out in advance.

**Merging values** If the track has values, then these will be merge automatically. The default behaviour uses the function `np.maximum` to simply choose the largest of the values. This behaviour can be overridden by supplying the operation a custom merge value function. By using a custom function, we can support merging of any type of data.

**Merging strands** If a track has strand information we can merge it in two different ways. The default one is to only merge element in the same strand direction. With this method we only merge elements that are on the same strand. The other possibility is to do a naive merge where we ignore the strand direction and merge each element directly. If the strand is equal, then we use it for the new element, if there is a mismatch then we set the strand to unknown. Merging in this way can be achieved by setting `useStrands` to `False`.

**Merging links** If the track have links, then these will be automatically merge. When merging a linked track, we keep all of the links from the original track. We take all links from the elements to be merged and combine them into one. Each merge element gets a new id. This id will be on the form `merge-<merge-number>`, where `merge-number` is the number of merge performed. We need to create new ids as any incoming links need to updated as well. Before the merge they point to two different nodes. As we now only have one node, a new id is needed. All links in the track that points to the old ids gets updated to point to the new merge element. Any weights on the links will be saved as well.

**Inputs and output**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input tracks</td>
<td>1</td>
</tr>
<tr>
<td>Input minimum requirements</td>
<td>Non dense, or dense and interveled</td>
</tr>
<tr>
<td>Overlap allowed in inputs</td>
<td>Yes</td>
</tr>
<tr>
<td>Result is track</td>
<td>Yes</td>
</tr>
<tr>
<td>Output track format</td>
<td>Equal to the TrackFormat of the input track, all other data is carried over</td>
</tr>
<tr>
<td>Overlap allowed in result</td>
<td>No</td>
</tr>
<tr>
<td>Other results</td>
<td>n/a</td>
</tr>
</tbody>
</table>

**Options**

**useStrands** Follow the strand information when merging the track. When following the strand, we divide the features on strand and...
merge these two separately combining them into one track. The resulting features will have the corresponding strand in the new track.

**treatMissingAsNegative** Treat features with missing strand info as if they are on the downstream. The default is to treat them as if they are on the upstream. To ignore missing one needs to remove them before the merge.

**mergeValuesFunction** Supply a custom function that will be used to merge values. Use this function if your data is not numerical or you want some other result then the maximum.
A.11 Print track

Description

The Print track is a debug operation. For each region in a track it dumps all of the ndarrays and there dtype to the terminal. As one might deduce this get impractical when the size of at track gets bigger then a couple of elements. The thought behind this to enable the user to easily look into how the ndarrays of a track gets shaped by other operations. This can be useful when implementing new operations. This operation is not meant to be used to export tracks from the library or GTrackCore. To export data one should use one of the built in features of GTrackCore.

Inputs and output

<table>
<thead>
<tr>
<th></th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input tracks</td>
<td>1</td>
</tr>
<tr>
<td>Input minimum requirements</td>
<td>None, supports all tracks</td>
</tr>
<tr>
<td>Overlap allowed in inputs</td>
<td>Yes</td>
</tr>
<tr>
<td>Result is track</td>
<td>No</td>
</tr>
<tr>
<td>Output track format</td>
<td>n/a</td>
</tr>
<tr>
<td>Overlap allowed in result</td>
<td>n/a</td>
</tr>
<tr>
<td>Other results</td>
<td>All of the ndarrays of each region in a track printed to the terminal</td>
</tr>
</tbody>
</table>
A.12 Remove dead links

The `Remove dead links` operation is used to remove dead links. These can be introduced by operation that works on a tracks links. With dead link we mean an edge that points to a non existing node. There are multiple ways that dead links can end up in a track. Firstly, there might be errors in the original track data. This operation can be used to confirm that our track data have a valid graph structure. Secondly errors can be introduced by other operations when they remove a feature. This second usage is something that all operations that can remove elements need to take into account. We have created this operation to avoid duplicate code in each operation that can work on linked tracks.

This operation checks that all edges points to a valid id. With valid we mean that there is a node with this id in the region. Any edge that do not point to a valid id is removed or updated to point to a predefined termination id. This id can either be an id that exists in the track or a placeholder name. By using a placeholder name, we can easily identify these links later.

![Figure A.13: The remove dead links operation](image)

**Description**

Figure A.14a and A.14b shows the links of a track visualised as a graph and how the graph looks like after a dead link have been removed.

**Id scope** The operation support two different scopes. When searching for valid id the default is to limit the search to just the region where the
edge is located. This operation supports a second scope where we look globally for the id in all regions of the track. Which scopes to use depend upon how the track data is structured. If the user knows that there are links between regions, then the global scope can be activated. When using this global scope, we need to be able to guarantee that the ids are unique across all regions. The operation do not perform any check of global uniqueness. This must be checked by the user in advance.

**Inputs and output**

<table>
<thead>
<tr>
<th>Number of input tracks</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input minimum requirements</td>
<td>Linked</td>
</tr>
<tr>
<td>Overlap allowed in inputs</td>
<td>Yes</td>
</tr>
<tr>
<td>Result is track</td>
<td>Yes</td>
</tr>
<tr>
<td>Output track format</td>
<td>Equal to the TrackFormat of the input track, all other data is carried over</td>
</tr>
<tr>
<td>Overlap allowed in result</td>
<td>yes</td>
</tr>
<tr>
<td>Other results</td>
<td>n/a</td>
</tr>
</tbody>
</table>

**Options**

- **newId** Instead of removing the dead edges, we can move them to another id or a bogus placeholder id. This can be useful if we want to identify the dead links, but for some reason we want to save the information they contain.

- **useGlobalIds** Use the global scope for finding valid ids. In this mode we assume that links can cross the regions and check all ids in the track when checking if an edge id is valid. This mode requires that ids are unique across the regions.
A.13 Shift

\[ t \quad S_2 \quad t \quad S_{0.5} \]

(a) Shift, simple  \hspace{1cm} (b) Shifting using a fraction

Figure A.15: Shifting

Description

The Shift operation move all elements a given number of base pairs along the track. We can shift in two different ways. Either by moving all elements a specific number of base pairs or by moving each element a given fraction of its length. In the latter we need to first find the length of all elements and then calculate the shift length for each. These lengths will be rounded to the closest integer. There is only one option for supplying the shift length \textit{shiftLength}. If this option is a positive number, then we move the elements upwards on the track, if negative we move them downwards. The same is true when using fractions.

This operation requires that the input track is non dense. Shifting a dense track makes little sense as all of its area is already defined. The resulting track will be of the same type as the input track.

We check for and fix any over or underflow that might be instructed by the shift. This will be fixed by cutting the elements to the bounding region. If an element is shifted outside of the region, then it will be removed altogether.

This operation supports overlapping tracks as input. The operation look at each element separately. If they overlap or not makes no difference for the calculation.

This operation supports using \textit{strand} information. If the track has strand information, then we will use it. This can be disable by the user. If disabled, then this operation will run as if the track had no strand information. When using strand information, we shift each element in the direction of its strand. Elements with missing strand information will by default be treated as if they are on upstream. This can be changed to on the downstream by the user.

Post calculation  This operation has a post calculation method. This method will run a Merge operation on the result if we do not allow overlap in it. The strand options from this operation will be passed on to the Merge operation. All other options are the default one.

\[ \text{Shift} = \text{Merge(Shift(track))} \]
Inputs and output

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input tracks</td>
<td>1</td>
</tr>
<tr>
<td>Input minimum requirements</td>
<td>Non dense</td>
</tr>
<tr>
<td>Overlap allowed in inputs</td>
<td>Yes</td>
</tr>
<tr>
<td>Result is track</td>
<td>Yes</td>
</tr>
<tr>
<td>Output track format</td>
<td>Equal to the TrackFormat of the input track</td>
</tr>
<tr>
<td>Overlap allowed in result</td>
<td>Yes</td>
</tr>
<tr>
<td>Other results</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Options

resultAllowOverlap Allow overlapping segments/points in the result track. The default is False. Any overlapping feature are merged into one.

shiftLength The length of the shift. Either as a number of base pair or as a fraction of each element’s length.

useFraction Treat the shift length as a fraction of each element’s length. The default value is False.

useStrands Follow the strand information if present in the track. The elements will then be shifted in their correct direction. The default setting is True.

treatMissingAsNegative Treat features with missing strand info as if they are on the downstream. The default is to treat them as if they are on the upstream. To ignore missing one needs to remove them before the union.
A.14 Subtract

\[ t_1 - t_2 \]

(a) Normal subtract

\[ s \]

(b) Using strands

Figure A.16: The subtract operation

Description

The Subtract operation is used to subtract one track from another. What we in practice do is to find any overlap between two tracks and remove them from the first track.

The ordering of the track inputs matter for this operation. We subtract track B from track A. Track A can be on any non dense format. Any data columns will be kept in the new track. The operation does not change this data in any way. If a segment gets sliced in two, the same data will be save in the new smaller segments. Track B needs to be non dese as well, but any other data columns will be ignored. This operation does not subtract any values or other data from each other. It does not support overlapping tracks as input. Any overlap in the tracks will be merge before the subtract.

If the tracks have strand information, then it will be used. Both track need to have strands. If one of the track is without strands, then the strands will be ignored. When we use the strand information we do only subtract strands in the same direction.

Elements with missing or unknown strand information will be treated as if they are on the upstream by default. This can be change to the downstream by the user.

As the tracks get merge before the start of the calculation there can’t be any overlap in the result track.

Figure A.17 shows the operation as a set operation. We visualised it here as a venn diagram. This operation is equal to the set operation \( A - B \).

Pre and post calculation  This operation has both a pre and a post calculation method. In the preCalculation method we merge any overlap
that might exist in the two tracks. In the postCalculation we remove any dead links that might have been introduced.

The full expression for the operation will then be.

\[
\text{Subtract} = \text{RemovDeadLinks}(\text{Subtract}((\text{Merge}(	ext{Track A})), (\text{Merge}(	ext{Track B}))))
\]

**Inputs and output**

<table>
<thead>
<tr>
<th>Number of input tracks</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input minimum requirements</td>
<td>Non dense</td>
</tr>
<tr>
<td>Overlap allowed in inputs</td>
<td>No, will be merge at the start</td>
</tr>
<tr>
<td>Result is track</td>
<td>Yes</td>
</tr>
<tr>
<td>Output track format</td>
<td>Equal to the TrackFormat of track A</td>
</tr>
<tr>
<td>Overlap allowed in result</td>
<td>Yes</td>
</tr>
<tr>
<td>Other results</td>
<td>n/a</td>
</tr>
</tbody>
</table>

**Options**

- **resultAllowOverlap** Allow overlapping segments/points in the result track. The default is False. Any overlapping feature are merged into one.

- **useStrands** Follow the strand information if present in the track. Default is True

- **treatMissingAsNegative** Treat features with missing strand info as if they have negative strand. The default is to treat them as if they are positive. To ignore missing one needs to remove them before the union.
## A.15 Union

The **Union** operation takes two tracks and combines them into one. The default behaviour is to also merge any overlapping elements. This merge can be disabled by the user by setting the `resultAllowOverlaps` option to `False`.

Merging of overlap is done via the **Merge** operation. All strand options are passed onto the merge operation. All other options are the defaults for **Merge**.

When merging two different linked tracks, we have no guarantee that the link ids are unique across both tracks. To mitigate this, we make the ids of both tracks unique. This is done via the **UniquifyLinks** operation.

![A B](image)

**Figure A.19:** Union as a venn diagram

Figure A.19 shows the operation as a set operation. We visualised it here as a venn diagram. This operation is equal to the set operation $A \cup B$. 

123
Pre and post calculation  This operation has defined both a pre and post calculation method. In the pre calculate method we uniquify the ids of each track via the UniquifyLinks operation. Each id in both of the tracks get a new prefix, which is \textit{track-1} and \textit{track-2} respectively. For more information about the UniquifyLinks operation see section A.16.

In the post calculate we use the Merge operation to merge any possible overlap that might have been introduced. This merge can be disabled by using the \textit{allowResultOverlap} setting. For more information on the Merge operation see section A.10.

The full expression for the operation:

\[
\text{Merge(Union(UniquifyLinks(Track 1), UniquifyLinks(Track 2)))}
\]

More control over merge  Is some cases we would like more control over how the merge is done. We can achieve this by splitting this operation in two. First we combine the two tracks and allowing overlap in the result. When use a separate Merge operation to get full control over the options. See listing 35 for a example.

```
1   u = Union(track1, track2, resultAllowOverlaps=False)
2   track3 = u.calculate()
3   m = Merge(track3, <custom options>)
4   result = m.calculate()
```

Listing 35: More control over merging

Inputs and output

<table>
<thead>
<tr>
<th>Number of input tracks</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input minimum requirements</td>
<td>None gaped or gaped and intervalled</td>
</tr>
<tr>
<td>Overlap allowed in inputs</td>
<td>Yes</td>
</tr>
<tr>
<td>Result is track</td>
<td>Yes</td>
</tr>
<tr>
<td>Output track format</td>
<td>Equal to the \textit{TrackFormat} of the input tracks. Any other data is carried over.</td>
</tr>
<tr>
<td>Overlap allowed in result</td>
<td>Yes</td>
</tr>
<tr>
<td>Other results</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Options

\textit{resultAllowOverlaps}  Allow overlapping elements in the result track. The default is \textit{False}. Any overlapping feature are merged into one.
**useStrands** Follow the strand information when merging overlapping features. Only used when `resultAllowOverlap` is set to `False`. When following the strand, we only merge features on the same strand. Ignored if the tracks are missing strands.

**treatMissingAsNegative** Treat features with missing strand info as if they have negative strand. The default is to treat them as if they are positive. To ignore missing one needs to remove them before the union.
A.16 UniquifyLinks

![Diagram of UniquifyLinks operation](image)

**Description**

The *UniquifyLinks* operation is used when we want to combine multiple linked tracks in some way using an operation. As the tracks can have multiple origins we cannot give a general guarantee that the ids are unique across the tracks. This can lead to inconsistencies in result of an operation as the links now points to multiple nodes.

Links in *GTrackCore* are represented with a minimum of two ndarrays. A *ids* array which identify all node, and a *edges* array that for each node gives its outgoing edges. If two nodes have the same id, we have no way of knowing which of the two nodes an edge is pointing to. See the figures in figure A.21 for a graph representation.

This operation fixes this problem by adding a new identifier in the front of each id. When applied to all input track in an operation, with different identifier, we can guaranty that none of the ids overlap. Note that this operation does not check or fix any overlapping ids within each track. We assume that ids are unique within each track.

![Incorrect and Correct Graphs](image)

(a) The incorrect graph, where the same edge points to multiple nodes  
(b) The correct, uniquified graph

**Figure A.21: Uniquify links, after merging two tracks. With and without uniquifying the links**
### Inputs and output

<table>
<thead>
<tr>
<th>Number of input tracks</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input minimum requirements</td>
<td>Linked</td>
</tr>
<tr>
<td>Overlap allowed in inputs</td>
<td>Yes</td>
</tr>
<tr>
<td>Result is track</td>
<td>Yes</td>
</tr>
<tr>
<td>Output track format</td>
<td>Linked, equal to the TrackFormat of the input track. Any data is carried over.</td>
</tr>
<tr>
<td>Overlap allowed in result</td>
<td>Yes</td>
</tr>
<tr>
<td>Other results</td>
<td>n/a</td>
</tr>
</tbody>
</table>

### Options

**identifier**  Specify the tag added to each id's unique tag for track 1.
A.17 Value select

(a) Selection on numerical values

(b) Selection on boolean values

(c) Selecting using a custom compare function, in this example we compare characters

Figure A.22: The value select operation

Description

The Value select operation is used to make a selection of elements in a track based on their given values. This operation is used to filter out elements that might not be of interest. The operation supports numbers and Boolean values, but by using a custom compare function any value type can be supported.

Any valued, non dense track can be used as input. Each value gets compared with the limit and removed if below it. By default, this operation does only remove elements below the given limit. A custom compare function is needed to enable a reverse usage where we remove elements above the limit.

Inputs and output

<table>
<thead>
<tr>
<th>Number of input tracks</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input minimum requirements</td>
<td>Non dense with values</td>
</tr>
<tr>
<td>Overlap allowed in inputs</td>
<td>Yes</td>
</tr>
<tr>
<td>Result is track</td>
<td>Yes</td>
</tr>
<tr>
<td>Output track format</td>
<td>Corresponding to the TrackFormat of the input track, all other data is carried over</td>
</tr>
<tr>
<td>Overlap allowed in result</td>
<td>yes</td>
</tr>
<tr>
<td>Other results</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Options

limit The value limit to select on. Any element with a numeric value lower then this will be removed.
**compareFunction** Custom compare function that overrides the default behaviour. This function takes the value ndarray and the limit as input. The result need to be an index that gives us the corresponding start and end arrays of the new track.
Appendix B

Installing the library

The operations library is implemented as a fork of GTrackCore. This fork can be found at: https://github.com/sivertkh/gtrackcore. This fork is at the time of writing current with all changes in the main GTrackCore repository on GitHub.

In general, one simply needs to clone the repository and run the install script provided. In practice the process it is not that simple. The GTrackCore library have some dependencies that we need to resolve first. These dependencies are not handled by the install script.

In this chapter we provide install guides for some common Linux distributions. Ubuntu is a widely used Linux distribution and should cover a large swath of the potential user base. We have provided a guide on Fedora and RedHat Enterprise as these are the supported client and server distributions at the University of Oslo.

The install process will be very similar on most type of modern Linux distributions. Use the distributions package manager to install the following packages: python-setuptools, python-nose, python-tables, python-yaml. Installing python-tables will in most package managers also install the HDF-5[11] library.

If not included, then this package needs to be installed as well. After installing the packages, we clone the repo and run the easy install script. If it fails, we need to look at the error message and see if they point to some missing package. We do this until all of the dependencies are met and the install script finishes.

As root In all of these guide we install the library as the root user. The library can be installed without root privileges but this is not covered in detail here. The python packages can be installed locally via the pip[37] package manager. In combination with virtualenv[38] one should be able to get the library running.

B.1 Ubuntu

This guide is written for Ubuntu 16.04.1 LTS. Installation on other version should be very similar.
**GTrackCore** have several dependencies. We install these first.

```
# apt-get install python-setuptools python-nose python-tables libhdf5-10
```

The operations library needs the `yaml` package. We install it now:

```
# sudo apt-get install python-yaml
```

Clone **GTrackCore** from **GitHub** to an appropriate folder, here we use the fork that contains the operations library. In time this will be merged into the main **GTrackCore** repository.

```
# git clone https://github.com/sivertkh/gtrackcore.git
```

Enter the folder and run the install.

```
# cd gtrackcore
# python setup.py install
```

### B.2 Redhat/CentOS

This install guide was created for **RedHat/CentOS 7**. Installation on older versions should be similar.

Some of the packages might not be in the default repositories. To get them we need to enable the **EPEL** repository. This repository is enabled as default on the University of Oslo’s **RedHat** install.

To enable **EPEL** run the following command.

```
# yum install epel-release
```

**GTrackCore** have several dependencies. We need to install them first. The **HDF5** library gets installed as a requirement of python-tables.

```
# yum install python-nose python-tables git
```

The operations library needs the `yaml` package. We install it now:

```
# yum install python-yaml
```

Clone **GTrackCore** from **GitHub** to an appropriate folder, here we use the fork that contains the operations library. In time this will be merged into the main **GTrackCore** repository.
Enter the folder and run the install.

```bash
# cd gtrackcore
# python setup.py install
```

## B.3 Fedora

This install guide was created for *Fedora* 24. Installing on the other supported versions should be similar. On older unsupported version one would not need to install Python 2.7. *Fedora* 24 does not come with Python 2.7 installed as standard. We first need to install it.

```bash
# dnf install python27
```

*GTrackCore* have several dependencies. We install these first. On Fedora *HDF5* get installed as a requirement of python-tables.

```bash
# dnf install python-nose python-tables
```

The operations library needs the *yaml* package. We install it now:

```bash
# sudo dnf install python-yaml
```

Clone *GTrackCore* from *GitHub* to an appropriate folder, here we use the fork that contains the operations library. In time this will be merged into the main *GTrackCore* repository.

```bash
# git clone https://github.com/sivertkh/gtrackcore.git
```

Enter the repository folder and run the install script.

```bash
# cd gtrackcore
# python setup.py install
```
B.4 Windows

The current version of GTrackCore does unfortunately not run on Windows based systems. The reason for this is the use of system specific calls like fcntl. This system call is one that the is not provided on Windows. To fix this one would need to rewrite the code to avoid such calls or to make the code system aware and use the win32api when running on Windows.

One possibility for users on Windows systems is to use some type of virtual machine. VirtualBox[40] is a good, open source, virtual machine that can be used to run a Linux machine on top of Windows. There will of course be some overhead in using virtual machine, but for testing or normal usage on not extremely large tracks, probably adequate.
Appendix C

Templates for creating new operation

In this appendix we provide three templates that can be used as a basis when creation of new operations. These templates are provided with the library. They can also be downloaded directly from the Github repository[41]. Listing C shows a template for a new operator class. This template gives the pattern the library uses. It can also be of help to look at the implemented operations. Listing C shows a template for a new raw operation. There are less restrictions on how these are created, but we have provided a small example. The only requirement is that the inputs are one or more ndarrays. Listing C show a framework that can be used as a basis for testing new operations.
from gtrackcore.track.core.TrackView import TrackView
from gtrackcore.track.format.TrackFormat import TrackFormat
from gtrackcore.track_operations.operations.Operator import Operator
from gtrackcore.track_operations.operations Operator import
createRawResultTrackView

from gtrackcore.track_operations.operations.Operator import
KwArgumentInfo

from collections import OrderedDict

# Import the appropriate raw operation
from gtrackcore.track_operations.raw_operations.NewOperation import newOperation

# Import operations used in the pre and post methods
from gtrackcore.track_operations.operations.Merge import Merge
from gtrackcore.track_operations.operations.UniquifyLinks import UniquifyLinks
from gtrackcore.track_operations.operations.RemoveDeadLinks import RemoveDeadLinks

# Create a new class of the Operator super class
class NewOperation(Operator):
    # The configuration of the operation.
    _trackHelpList = ['Track 1 description', 'Track 2 description']
    _operationHelp = "Description of the operation"
    _numTracks = 2
    _resultIsTrack = True
    _trackRequirements = [TrackFormatReq(dense=False),
                           TrackFormatReq(dense=False)]

    def _calculate(self, region, tv1, tv2):
        ""
        This method is called for each region in the track. Get the
data one need, call the raw operation and create a new track
view from the result and return.
        ""

        :param region: Current region
        :param tv1: Track view from track 1
        :param tv2: Track view from track 2
        :return: TrackView with a new track, some other results or None.
        ""

        # First we need to extract the numpy arrays we need in our
        # operation. The track requirements for each track need to
        # mach what we require.
t1Starts = tv1.startsAsNumpyArray()
t1Ends = tv1.endsAsNumpyArray()
t1Strands = tv1.strandsAsNumpyArray()
t2Starts = tv2.startsAsNumpyArray()
t2Ends = tv2.endsAsNumpyArray()
t2Strands = tv2.strandsAsNumpyArray()

if self._someOtherOption:
    # One might want to use the options to change the raw
    # operation inputs. It's often easier to do this here.
    t1Strands = None
t2Strands = None

    # Call the raw operation. A raw operation will only take
    # Numpy arrays and options as input
    result = newOperation(t1Starts, t1Ends, t1Strands, t2Strands,
                          t2Ends, t2Strands, self._someOption)

    # Raw operations will always return new Numpy arrays or
    # None if there is no result.
    if result is not None and len(result[0]) != 0:
        # Split the result up for readability
        starts = result[0]
        ends = result[1]
        strands = result[2]
        index = result[3]
        encoding = result[4]

        # Depending on what the operations does, and how complex
        # it is, you might just want to use create the result
        # TrackView directly.
        tv = TrackView(region, starts, ends, None, strands, None,
                        None, None, borderHandling='crop',
                        allowOverlaps=True)

        # If we want to keep other data from the input tracks, we
        # can use a function called createRawResultTrackView to
        # create the trackView. To use this out rawOperation need
        # to return two extra arrays, index and encoding. Index is
        # where in the original track and encoding is which of the
        # input track (if more then one). Any new* variables are
        # used, else they are created from the original track
        tv = createRawResultTrackView(index, region, [tv1,tv2],
                                       self._resultAllowOverlaps, encoding=encoding,
                                       newStarts=starts, newEnds=ends)

        return tv

else:
    # No result or no data at region
    return None

def preCalculation(self):
    """
    This method can be implement if one want to run operations on
    the track before the start of the main operation.

    Note that this method will run at the initialization of the
    object.
    :return: None
    """
Retrieve the tracks.

```python
t1 = self._tracks[0]
t2 = self._tracks[1]
```

Run the calculation on the track. This is often dependent on the track type. Here we uniquify the links if the tracks have them.

```python
if t1.trackFormat.isLinked():
    u = UniquifyLinks(t1, identifier="track-1")
    t1 = u.calculate()

if t2.trackFormat.isLinked():
    u = UniquifyLinks(t2, identifier="track-2")
    t2 = u.calculate()
```

Store the updated track.

```python
self._tracks = [t1, t2]
```

def postCalculation(self):
    
    This method can be implement if one want to run the results through one or more operations after the main calculation

    Run the operations using the track in self._result and replace it with the new results.

    :return: None

    # Run some operation on the result
    r = RemoveDeadLinks(self._result).calculate()
    self._result = r

    if not self._resultAllowOverlap:
        # It is often useful to run cleanup operations here.
        # One we use often is to merge overlap if we do not allow it.
        m = Merge(self._result).calculate()
        self._result = m

def _setResultTrackFormat(self):
    
    Use this method to create the TrackFormat that will correspond to the TrackFormat of the result track.

    It is important that the output requirements matches the actual output.

    :return: None

    # Here we give some of the common types.

    # Case 1: Result TrackFormat match the TrackFormat of one of the input tracks.
    self._resultTrackFormat = self._tracks[0].trackFormat
# Case 2: Some form of combination of the different properties

t1TrackFormat = self._tracks[0].trackFormat

t2TrackFormat = self._tracks[1].trackFormat

# Here the resultTrackFormat is stranded only if track 1
# is stranded and track 2 is linked.
if t1TrackFormat.hasStrand() and t2TrackFormat.isLinked():
    strands = []
else:
    strands = None

self._resultTrackFormat = TrackFormat(startList=[], endList=[],
    strandList=strands)

# Case 3: Static type.
# Here the result is always a segmented track
self._resultTrackFormat = TrackFormat(startList=[], endList=[])

@classmethod
def _getKwArgumentInfoDict(self):
    ""
    This class defines the operations options
    The KwArgumentInfo contains the following fields.
    key: option key used by argparser
    shortkey: short version of key, used by argparser
    help: help text that describes the option
    contentType: Content type from argparser
    defaultValue: The default value, will be supplied if the
    options is not defined.
    ""
    return OrderedDict(
        ('debug', KwArgumentInfo('debug', 'd', 'Print debug info', bool, False)),
        ('resultAllowOverlap', KwArgumentInfo('resultAllowOverlap', 'o',
            'Allow overlap in the result track.', bool, False)),
        ('useStrands', KwArgumentInfo('useStrands', 's',
            'Follow the strand direction', bool, True)),
        ('treatMissingAsNegative', KwArgumentInfo('someOption', 's',
            'A description of the option',
            bool, False))))

Listing 36: Template for a new operation
import numpy as np

def someOperation(starts, ends, someOption=None, someOtherOption=False, debug=False):
    # Each raw operation should be as simple as possible.
    # We take numpy arrays as inputs, do some calculation on them,
    # and return either the arrays that constitutes a new track
    # or some type calculated value.

    if debug:
        # Debug printing is often useful.
        print("Start of someOperation")
        print("t1Starts: {}".format(starts))

    if someOtherOption:
        # If a option is given we may want to change some of the
        # arrays or may provide different calculation
        starts = starts + 1

    # Tracks may have additional data columns, as these are
    # not necessary here we need to extract them later.
    # To make this possible we need to create a index array that
    # tells us where in the original array the elements are.
    index = np.arange(0, len(starts), 1, dtype='int32')

    # If the operation uses more then one track, we do also
    # need to know which track to get the value from.
    # As we have only one track here, we do not need it.
    encoding = np.zeros(len(starts), dtype='int32') + 1

    # In this simple example we find all elements that
    # have a start bigger then 1000, and an end smaller then
    # 2000.
    # The np.where method is very powerful one and can
    # be used in many situations
    selectIndex = np.where((starts > 1000) | (ends < 2000))

    # We then use the index to select the data for our new track.
    starts = starts[selectIndex]
    ends = ends[selectIndex]
    index = index[selectIndex]

    # The return value is always ndarrays. Its the
    # responsibility of the class operator to create the
    # new TrackView
    return starts, ends, index

Listing 37: Template for a new raw operation
import unittest
import numpy as np

from gtrackcore.metadata import GenomeInfo
from gtrackcore.track.core.GenomeRegion import GenomeRegion

from gtrackcore.test.track_operations.TestUtils import createSimpleTestTrackContent

# Import the appropriate Operation
from gtrackcore.track_operations.operations import SomeOperation

class TemplateTest(unittest.TestCase):

    def setUp(self):
        # Define some chromosomes to use
        # A small one makes it easier to test partitions etc
        self.chr1Small = (GenomeRegion('hg19', 'chr1', 0, 3))

        # Here we do most of the test on chr1
        self.chr1 = (GenomeRegion('hg19', 'chr1', 0,
                                  GenomeInfo.GENOMES['hg19']['size']['chr1']))
        self.chromosomes = 
        (GenomeRegion('hg19', c, 0, l)
         for c, l in GenomeInfo.GENOMES['hg19']['size'].iteritems())

    def _runTest(self, starts=None, ends=None, values=None,
              strands=None, ids=None, edges=None, weights=None,
              extras=None, expStarts=None, expEnds=None,
              expValues=None, expStrands=None, expIds=None,
              expEdges=None, expWeights=None, expExtras=None,
              expNoResult=False, customChrLength=None,
              allowOverlap=True, resultAllowOverlap=False,
              someOperationOption=None, someOtherOption=None,
              debug=False, expTrackFormatType=None):

        # gtrackcore.test.track_operations.TestUtils contains a useful
        # methods. We use one to create a simple test TrackContent
        track = 
        createSimpleTestTrackContent(startList=starts, endList=ends,
                                      valList=values,
                                      strandList=strands,
                                      idList=ids, edgelist=edges,
                                      weightsList=weights,
                                      extraLists=extras,
                                      customChrLength=customChrLength)

        # Create the operation
        s = SomeOperation(track,
                          someOperationOption=someOperationOption,
                          someOtherOption=someOtherOption, debug=debug)

        # Run the calculation
self.assertTrue(s is not None)
result = s.calculate()
resFound = False
for (k, v) in result.getTrackViews().items():
    if cmp(k, self.chr1) == 0 or cmp(k, self.chr1Small) == 0:
        # All test tracks are in chr1
        resFound = True
        # Get all the arrays
        newStarts = v.startsAsNumpyArray()
        newEnds = v.endsAsNumpyArray()
        newValues = v.valsAsNumpyArray()
        newStrands = v.strandsAsNumpyArray()
        newIds = v.idsAsNumpyArray()
        newEdges = v.edgesAsNumpyArray()
        newWeights = v.weightsAsNumpyArray()
        newExtras = v.allExtrasAsDictOfNumpyArrays()
        if debug:
            # A debug print is useful to have
            print("newStarts: {}".format(newStarts))
            print("expStarts: {}".format(expStarts))
            print("newEnds: {}".format(newEnds))
            print("expEnds: {}".format(expEnds))
            print("newStrands: {}".format(newStrands))
            print("expStrands: {}".format(expStrands))
            print("newIds: {}".format(newIds))
            print("expIds: {}".format(expIds))
            print("newEdges: {}".format(newEdges))
            print("expEdges: {}".format(expEdges))
        if expTrackFormatType is not None:
            # Check that the track is of the expected type.
            self.assertTrue(v.trackFormat.getName() == expTrackFormatType)
        if expEnds is None and expStarts is not None:
            # Assuming a point type track. Creating the expected
            # end. We need to do this as gtrackcore create a
            # virtual ends array for point tracks
            expEnds = np.array(expStarts) + 1
        # Check that the different arrays are correct or empty
        if expStarts is not None:
            self.assertTrue(newStarts is not None)
            self.assertTrue(np.array_equal(newStarts, expStarts))
        else:
            self.assertTrue(newStarts is None)
        if expEnds is not None:
            self.assertTrue(newEnds is not None)
            self.assertTrue(np.array_equal(newEnds, expEnds))
        else:
self.assertTrue(newEnds is None)

if expValues is not None:
    self.assertTrue(newValues is not None)
    self.assertTrue(np.array_equal(newValues, expValues))
else:
    self.assertNull(newValues)

if expStrands is not None:
    self.assertTrue(newStrands is not None)
    self.assertTrue(np.array_equal(newStrands, expStrands))
else:
    self.assertNull(newStrands)

if expIds is not None:
    self.assertTrue(newIds is not None)
    self.assertTrue(np.array_equal(newIds, expIds))
else:
    self.assertNull(newIds)

if expEdges is not None:
    self.assertTrue(newEdges is not None)
    self.assertTrue(np.array_equal(newEdges, expEdges))
else:
    self.assertNull(newEdges)

if expWeights is not None:
    self.assertTrue(newWeights is not None)
    self.assertTrue(np.array_equal(newWeights, expWeights))
else:
    self.assertNull(newWeights)

if expExtras is not None:
    for key in expExtras.keys():
        if v.hasExtra(key):
            expExtra = expExtras[key]
            newExtra = newExtras[key]
            self.assertTrue(np.array_equal(newExtra, expExtra))
        else:
            self.assertTrue(len(newExtras) == 0)
    else:
        self.assertFalse(resFound)
else:
    self.assertTrue(resFound)

# We then create a test method for each thing we want to test
def testSegments(self):

Segments, test
:return: None

```python
self._runTest(starts=[10], ends=[32], expStarts=[5,11],
              expEnds=[10,16], someOtherOption=123,
              expTrackFormatType="Segments")
```

```python
def testLinkedBasePairs(self):
    
    LBP example, note the use of customChrLength
    :return:
    
    self._runTest(ids=['1','2','3'], edges=['2','3','1'],
                   weights=[[1],[1],[1]], someOperationOption=True,
                   expIds=['1','2','3'], expEdges=['2','3','run'],
                   customChrLength=3,
                   expTrackFormatType="Linked base pairs")
```

```python
if __name__ == '__main__':
    unittest.main()
```

Listing 38: Template for a new operation test