**UNIVERSITY OF OSLO**
**Department of Informatics**

# An Extensible Framework for Comparative Analysis of Annotations

Master's thesis

Eivind Gard Lund
Institute of Informatics
University of Oslo

**May 5, 2011**

**Abstract**

Recent progressions in highly specific sequencing technologies generates high-resolution genomic data. Comparative analysis of these data is a source of further insight into genomic mechanisms, but the domain remains largely unexplored. Prototypical programs require flexible and scalable solutions as the requirements are expected to change. Array programming has good abstractions for operations on large data sets and the resulting performance is often excellent. No published efforts have, as far as we know, previously been made to assess the suitability of array programming to non-numerical problems.

We present two methods for comparative annotation analysis called *projection* and *quantitative comparison*. We have furthermore developed a range of array programming algorithms for numerous annotation track operations. All algorithms are implemented as part of a framework for comparative annotation analysis.

Both methods for comparative analysis have promising properties, but further work on verification and analysis of the biological interpretation is needed. Array programming have been proved applicable to a wide range of problems. A serious limitation with array programming is that the model of a problem domain must fit perfectly with the restricted set of available operations.

# Acknowledgements

I would like to thank my supervisor Geir Kjetil Sandve for his guidance and inspiring attitude. I would also like to thank my co-supervisors Eivind Hovig and TorbjÃÿrn Rognes for having made themselves available.

Secondly, I would like to thank Finn DrablÃÿs and HÃěvard Aanes for assistance with some of the analyses.

On a more personal level I would like to thank my girlfriend Julie for her support and indulgence and my brothers in arms at toppetasjen.

Eivind Gard Lund
University of Oslo
May 2011

# Contents


**1 Introduction** **4**
1.1 Motivation . . . 4
1.1.1 Annotation Analysis . . . 4
1.1.2 Array Programming . . . 5
1.2 Research Context . . . 6
1.3 Research Questions . . . 6

**2 Background** **7**
2.1 Biology . . . 7
2.1.1 Galaxy . . . 7
2.1.2 BEDTools . . . 8
2.1.3 BioPerl and BioPython . . . 9
2.1.4 The Genomic Hyperbrowser . . . 9
2.1.5 Storing Biological Data . . . 9
2.2 Computer Science . . . 10
2.2.1 Computational Science and Python . . . 10
2.2.2 Python interacts with Optimized Compiled Libraries . 10
2.2.3 Array Programming . . . 11
2.2.4 Numpy - Array Programming in Python . . . 11
2.2.5 Type Systems . . . 15
2.2.6 The relational model for database management . . . 16
2.2.7 MapReduce . . . 17
2.2.8 Hash Table . . . 18
2.2.9 Software Bugs . . . 18

**3 Methods** **20**
3.1 Annotations . . . 20
3.1.1 Chromosomes Divides Annotations into Disjoint Sets . 21
3.2 Coupled Track Elements are the Basis for Comparative Operations . . . 21
3.3 Modeling Intervals Suitably for Array Programming . . . 22

3.3.1 Every Position has a State . . . . . . . . . . . . . . . . 23
3.4 Operations on Intervals using Array Programming . . . . . . . 24
3.4.1 Set Operations . . . . . . . . . . . . . . . . . . . . . . 24
3.4.2 Counting Occurrences Within an Interval . . . . . . . . 25
3.4.3 Connecting Closest Intervals From Different Tracks . . 27
3.5 Inter-Genome Comparisons . . . . . . . . . . . . . . . . . . . . 28
3.5.1 Projection of Track Elements . . . . . . . . . . . . . . 29
3.5.2 Quantitative Comparisons . . . . . . . . . . . . . . . . 29
3.5.3 Accounting for Strand Direction . . . . . . . . . . . . . 29
3.5.4 Coupled Track Elements may Overlap . . . . . . . . . 30
3.5.5 Mapping Coupled Annotation Tracks . . . . . . . . . . 31
3.6 Other . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
3.6.1 Representing Sets Numerically . . . . . . . . . . . . . . 33

**4 Design and Implementation 35**
4.1 Operations on Chromosome Annotation Tracks . . . . . . . . 35
4.1.1 Chromosome Tracks are Represented by Numpy Arrays 36
4.1.2 Generating Number Line States . . . . . . . . . . . . . 36
4.1.3 The Union of Two Chromosome Annotation Tracks . . 37
4.1.4 The Intersection of Two Chromosome Annotation Tracks 38
4.1.5 Assuring that Chromosome Annotation Track Elements are Non-overlapping and Ordered . . . . . . . . . . . . 38
4.1.6 Indices if Sorted . . . . . . . . . . . . . . . . . . . . . . 39
4.1.7 Counting Points Separated by Pivot Elements . . . . . 39
4.1.8 Counting the Number of Points Closest to each Track Element . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.1.9 Counting the Number of Points within each Track Element . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.1.10 Counting Intervals inside Regions . . . . . . . . . . . . 41
4.1.11 Counting Closest Intervals for Each Region . . . . . . . 42
4.1.12 Computing the Distiance Between a Point and its Closest Region . . . . . . . . . . . . . . . . . . . . . . . . 42
4.1.13 Filter Points Inside Track Elements . . . . . . . . . . . 43
4.2 Scaling Chromosome Level Operations Genome-wide . . . . . 44
4.2.1 Encapsulating Annotation Tracks . . . . . . . . . . . . 44
4.2.2 Organizing Operations on Annotations . . . . . . . . . 45
4.2.3 Representing an Annotation Track . . . . . . . . . . . 46
4.3 Organizing Coupled Annotation Tracks . . . . . . . . . . . . . 48
4.3.1 Coupled Annotation Tracks . . . . . . . . . . . . . . . 48
4.4 Operations on Coupled Annotation Tracks . . . . . . . . . . . 51
4.4.1 Count Overlapping Track Elements . . . . . . . . . . . 52

**5 Use Cases** **54**
5.1 Five-Vertebrate Article . . . . . . . . . . . . . . . . . . . . . . . . . 54
5.1.1 Conservation and divergence of TF Binding . . . . . . 55
5.1.2 Classifying and Counting Turnovers . . . . . . . . . . . 59

**6 Discussion** **67**
6.1 Inter-Genome Comparisons . . . . . . . . . . . . . . . . . . . . . . . 67
6.1.1 Distance . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
6.1.2 Projection . . . . . . . . . . . . . . . . . . . . . . . . . . 68
6.1.3 Quantitative Comparisons . . . . . . . . . . . . . . . . . 70
6.2 Array Programming . . . . . . . . . . . . . . . . . . . . . . . . . . 72
6.2.1 Guidelines for Interval Problems . . . . . . . . . . . . . 72
6.2.2 Encapsulation and Reuse . . . . . . . . . . . . . . . . . . 74
6.3 On Reproducibility of Analyses . . . . . . . . . . . . . . . . . . . 75
6.4 Encapsulation and Typing . . . . . . . . . . . . . . . . . . . . . . 76
6.5 Evaluation of Numpy as a Programming API . . . . . . . . . . . 78
6.5.1 Best Effort Approach . . . . . . . . . . . . . . . . . . . . 78
6.5.2 Missing Features . . . . . . . . . . . . . . . . . . . . . . 79
6.6 Trusting the Results . . . . . . . . . . . . . . . . . . . . . . . . . . 79
6.7 Architecture and Organization . . . . . . . . . . . . . . . . . . . . 80

**7 Conclusion** **82**
7.1 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
7.2 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
7.3 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84

# Chapter 1

# Introduction

The advent of advanced sequencing technologies have changed the landscape of bioinformatics where genomic data are now produced at a resolution and of a quality that far exceeds prior methods. This rapid rate of change have not yet been fully reflected in the available programs for analysis of genomic data. We've formalized two methods for meaningful comparative annotation analysis and in addition developed a sound methodology for operations on annotations using array programming.

## 1.1 Motivation

### 1.1.1 Annotation Analysis

The DNA sequence alone is not very valuable with some interpretation. An annotation track is a collection of empirical properties on a genome, such as the locations of all known genes or the experimentally determined binding sites for a transcription factor. Much can be learned from analyzing relationships between annotation tracks.

The Galaxy platform[9] is a web service that provides a large set of basic operations that the user composes to a complex query on one or more annotation tracks.

The Hyperbrowser is an system for statistical annotation track analysis. It has a very different approach to user interaction than other comparable systems. A user selects or supplies input data and is presented with a series of questions inferred from the input. A selected question initiates a large series of computations with a resulting suitable statistical test that answers the question at hand.

Comparative sequence analysis have historically been a valuable method

for finding genes. Mutations occur randomly during evolution, but a mutation within an important functional element is likely to harm an organism and thereby decrease its chances of survival. Sequences conserved between multiple species have been protected during evolution because of their importance to the various species.

Genomic sequence analysis have been supplemented by annotation analysis within a single genome. Comparative annotation analysis is similarly a logical expansion of comparative sequence analysis. Neither the Galaxy platform, the Genomic Hyperbrowser nor any other known annotation analysis system supports any form of comparative annotation analysis.

A recent article[22] compared the distribution of two types Transcription Factor Binding Sites (TFBSs) within and around the conserved sequences of five species. The paper did not delve on the methods used for comparisons, it's implication, usability or if alternatives existed. We see a clear need for an exploration of the possibilities for comparative annotation analysis with the goal of developing methods for meaningful comparisons and development of generic functionality for this purpose.

### 1.1.2 Array Programming

Array programming is a technique that generalizes operations on scalars to apply transparently to arrays. Array programming primitives concisely express broad ideas about data manipulation. It's a popular choice in many scientific environments, as solutions to many numerical problems are easily expressible in a form that fits with the programming model. These generically applicable operations have a very efficient implementation in the environments that provide them. Array programming applied to suitable problems often have a succinct implementation that is easy to reason about even without much prior programming experience. Interestingly enough, no articles we are aware of discusses use of array programming outside the field purely numerical applications.

Initial work on comparative annotation analysis is prototypical and failures are expected. We therefore need a programming environment that allows for rapid development and fast turnovers. An annotation track is in essence a multitude of locations on a genome with a natural numerical representation . The abstract way that array programming express operations on large data sets are therefore promising. We want to further explore the possibilities for solving problems related to annotation analysis using array programming.

## 1.2 Research Context

My supervisor, Geir Kjetil Sandve, is one of the core members of the Genomic Hyperbrowser project[21]. The Hyperbrowser is at present only able to perform analyses within a single genome. The ability to perform inter-genome comparisons would greatly benefit the project. The Hyperbrowser is written in the Python programming language and uses the Numpy library for array programming. Array programming operations are orders of magnitude faster than similar functionality implemented in Python. To extend the usage of Numpy would reduce the overall run-time for analyses, which often are performed on huge data sets.

## 1.3 Research Questions

- Biological:
  - Is it possible to unambiguously compare annotation tracks between genomes?
  - Can formal methods for meaningful comparison of annotation tracks from different genomes be developed?
- Array Programming
  - Is it possible to solve non-numeric problems, such as interval comparison problems, using array programming?
  - For which operations on annotation tracks can we develop array programming algorithms?

# Chapter 2

# Background

## 2.1 Biology

### 2.1.1 Galaxy

Galaxy[9] is a web-plattform featuring many basic operations and a tight integration to the UCSC Genome Browser[13]. Their target user is a life science researcher without any or little programming experience. The authors of Galaxy have put a lot of thought and major research efforts into making analysis and results transparent and reproducible[10]. The system provides a large set of basic operations that the user composes to solve a complex problem. Data are provided through integration with the UCSC Genome Browser or uploaded from the user.

Galaxy offers a *Workflow Editor* where the user may easily combine operations through a graphical editor. The resulting complex operation may be given a name for later reuse. Such workflows are to Galaxy what functions are to programming. The ability to create, save and share such workflows is a major feature of the Galaxy platform. Regrettably, at the time of writing, it's not possible to combine existing workflows in a new workflow. This is a major limitation, but we expect this functionality to appear in a future release.

The galaxy platform is described by the authors as[2]:

> High-throughput data production has revolutionized molecular biology. However, massive increases in data generation capacity require analysis approaches that are more sophisticated, and often very computationally intensive. Galaxy is a software system that provides this support through a framework that gives experimentalists simple interfaces to powerful tools, while automatically

managing the computational details.

The main purpose of the framework is to provide standardized graphical interfaces to tools and let the user connect and compose them in meaningful ways. Adding new tools is made very simple because of the complete separation between the Galaxy platform and the tools it provides. A tool is an independent program that must be installed on the server. The tools themselves may be written in any programming language and are embedded into a galaxy system by adding a small XML file with configurations. All tools reads input from the standard input and writes its results to the standard output. This is very similar to one of the fundamental design principals of the Unix operating system[19]:

> Much of the power of the UNIX operating system comes from a style of program design that makes programs easy to use and, more important, easy to combine with other programs.

The Galaxy Platform is attractive to the end user because of its usability, flexibility and the low barrier entry. Usage requires no local installation and the interaction with tools is standardized and quite intuitive. The galaxy platform is the environment that connects small standalone tools, data and user history. It's the intention that users create their own tools to solve specific issues and ideally share those extension with the community. The great freedom given in crafting autonomous tools have many benefits. It does not restrict potential contributors by tying them to a specific programming language. This freedom is, however, a double edged sword. Many of these tools must have partly overlapping functionality and solve some of the same sub-problems. Even so, the Galaxy platform provides no assistance in solving the commonalities present in many tools.

### 2.1.2 BEDTools

BEDTools[20] are a collection of programs for annotation analysis. It has the same design principle as Galaxy where complex problems are solved by the composition of many smaller tools. The tools are composed using ordinary operating system concepts such as pipes..[19]. BEDTools package offers operations such as sorting, intersubsection, union and computing complementary regions. Different operations are easily combined as part of a script together with other common unix-tools. The BEDTools package is a collection of small independent programs and therefore provides no means or help in extending or supplementing the available operations. Missing operations or functionality must be created from scratch as autonomous programs.

Many of the prebundled tools in the Galaxy platform are in fact the BEDTools package.[1]

### 2.1.3 BioPerl and BioPython

Language specific APIs such as BioPython[4] and BioPerl[23] are well known and popular. They provide a common interface to format conversions, querying of large biological databases, machine learning and many sequence alignment programs. It's not possible to analyze or operate on annotations using these packages.

### 2.1.4 The Genomic Hyperbrowser

*The Genomic Hyperbrowser*[21] is an inferential analytical system for life science researchers. The Hyperbrowser has a rather unique approach to user interaction. It's declarative in the sense that the user states what he wants to achieve and the system will figure out the gritty details of how to achieve it. This in contrast to the Galaxy system where users have to compose a chain of simple low-level operations to compute their end-result. Possible questions are presented to the user as a consequence of the selected input data. This design choice makes the Hyperbrowser a very user-friendly and powerful tool for problems of the specific target domain.

### 2.1.5 Storing Biological Data

There is an abundance of different file formats used to store biological data in bioinformatics. The *BED*[2] file format has over time become quite ubiquitous. The Browser Extensible Data (BED) format is a file format used by the UCSC genome browser for defining genomic regions. One genomic region is encoded per line. All of the genomic regions described by a single BED file are of the same type. For example they denote the different experimentally determined binding sites for a specific TFBS or they could describe regions affected by a specific methylation.

[1] http://groups.google.com/group/bedtools-discuss/browse_thread/thread/56f4be5fbad5b86/e6d0dab6370c4bbb?lnk=gst&q=galaxy

[2] http://genome.ucsc.edu/FAQ/FAQformat.html#format1

| chrom | chromStart | chromEnd |
|---|---|---|
| chr1 | 4 | 100 |
| chr1 | 300 | 330 |
| chr2 | 1 | 10 |
| chr2 | 50 | 60 |

Table 2.1: BED file-format example

## 2.2 Computer Science

### 2.2.1 Computational Science and Python

Many scientific computing environments, such as Mathematica, Matlab and Octave, have become very popular over the last decades. The terse syntax and operations tailored for the domain simply makes scientists more effective in such environments[14]. A problem with such packages is that they interoperate poorly with external systems and the functionality provided are sometimes too simple.

Python is a general purpose programming language. It's very popular within the scientific computing and research domain.It has the capability to operate with external software and libraries. What it lacks in domain specific specialization can for most applications be mediated by molding the language and environment to fit the domain. Other areas of widespread adoption are web application programming and systems programming.

It's relatively easy to learn the basics of the language and quickly start producing non-trivial programs. A drawback with Python for solving scientific problems is its execution speed.

### 2.2.2 Python interacts with Optimized Compiled Libraries

There exists a large number of well tested, specialized and optimized libraries and functions written in other languages for scientific applications. It is possible to create interfaces for these external modules to integrate them into a Python environment. An external library is interchangeable from a native library once a proper interface has been created.

Two very common tools used to generate such interfaces semi-automatically are f2py[18] and Swig[1]. F2py is a tool that connects Fortran[3] programs to

[3] An old programming language especially suited for numeric computation and scientific computing. http://en.wikipedia.org/wiki/Fortran

a Python environment. Swig has been developed to automate the task of integrating compiled code with scripting language interpreters.

These tools allows Python to cheat the speed issues often associated with scripting languages. There's a certain overhead involved with calling a function from a compiled library. The shortest execution times are experienced if most of the computation is performed by a compiled library using as few calls into the library as possible. Many computational problems fits well with this approach, which have made the combination of Python and the Numpy library a popular choice for computationally intensive tasks.

### 2.2.3 Array Programming

*Array programming*[4] languages generalize operations on scalars to apply transparently to vectors, matrices, and higher dimensional arrays. This is a high-level language construct for data manipulation instead of the data modeling that object-oriented languages offers[16]. The fundamental idea behind array programming is that operations are applied to an entire set of values. This makes it a high-level programming model as it allows the programmer to think and operate on whole aggregates of data, without having to resort to explicit loops of individual scalar operations. Array programming is typically used to solve mathematical equations for scientific applications. The examples in listing 2.1 and 2.2 serves to illustrates the difference.

Listing 2.1: Multiplying every element of a multi-dimensional array by a scalar in an Algol-like language

```
Line 1  for (i = 0; i < n; i++)
   -        for (j = 0; j < n; j++)
   -            x[i][j] *= 3;
```

Listing 2.2: Multiplying every element of a multi-dimensional array by a scalar in an Array Programming language

```
Line 1  x *= 3
```

### 2.2.4 Numpy - Array Programming in Python

Numpy provides a homogeneous, multidimensional array of a particular data type[17]. The extension also provides universal functions that operate rapidly

[4]sometimes called vector programming or vector operations

over the multidimensional array. A universal function in Numpy is a function that operates on arrays in an element-by-element fashion. There are at least two compelling arguments for using Numpy for scientific computing in Python. The first reason is the drastically improved execution speed when used correctly on applicable problems. The second is that many common domain specific operations are easily expressible using the library. Many commonly used operations in the scientific domain requires complex algorithms for efficient computations. Matrix multiplication[7] is an example that's already present in the Numpy Library. Numpy offers a significant gain in execution speed together with concise implementation in ideal circumstances. Any problem that can be solved primarily as a series of operations on whole datasets and matrices can perform comparable to an equivalent implementation in the C programming language.

### Basic Operations

Some examples of basic array operations in Numpy are given below:

- Pairwise addition of the elements of two arrays:

```
Line 1  In [8]: array([1,2,3]) + array([10,20,30])
     -  Out[8]: array([11, 22, 33])
```

- Scaling an array by a scalar:

```
Line 1  In [9]: array([1,2,3]) * 3
     -  Out[9]: array([3, 6, 9])
```

- Adding an offset to every element of an array:

```
Line 1  In [10]: array([1,2,3]) + 5
     -  Out[10]: array([6, 7, 8])
```

- Sorting an array[5]:

```
Line 1  In [11]: np.sort([2,3,1])
     -  Out[11]: array([1, 2, 3])
```

- Basic aggregation functions:

[5] The namespace `np` is used to avoid confusion between the `sort` function in the Numpy library and the regular Python `sort` function

```
In [12]: np.sum([2,3,1])
Out[12]: 6

In [13]: np.max([2,3,1])
Out[13]: 3

In [14]: np.min([2,3,1])
Out[14]: 1
```

### Array Indexing

Numpy arrays can be indexed in three ways, where the first two are identical to how ordinary Python arrays may be indexed.

**By Number** Indexing by number returns the value at the position of the number.

**By Slice** Indexing by slicing returns, explained very simply, a view of the part specified by the slice. A slice is typically a combination of one of the following:

- The first N items.
- The last N items.
- Every second item.
- The whole array except the first N items.
- The whole array except the last N items.

**By Array** The last and Numpy specific way of indexing arrays is by another array. The indexing array must either be a boolean or an integer array. Indexing with a boolean array will return those elements where the corresponding boolean value is true. Integer indexing allows selection of arbitrary items in the array based on their position. The main difference between boolean and integer indexing is the fact that integer indexing may change the original order. These concepts are best illustrated by a few examples.

1. The first example illustrates indexing by boolean arrays:

```
In [12]: x = array([4,2,1])
In [13]: bidx = array([True,False,True])

In [14]: x[bidx]
Out[14]: array([4, 1])
```

2. The second example illustrates indexing by integer arrays:

```
In [12]: x = array([4,2,1])
In [15]: iidx = [0,2]

In [16]: x[iidx]
Out[16]: array([4, 1])

In [17]: x[[2,1,0]]
Out[17]: array([1, 2, 4])
```

### The **argsort** Function

It's common to represent a table as multiple arrays of equal length. Each array represents a column of the table and all the values at equal positions in the arrays represents a row. The consistency of the table is broken if only one of the column-arrays is reordered. The `argsort` function mends this problem by computing the shuffling of elements that would have resulted in a sorted array. A table is sorted by a column if all arrays of the table are reordered by the shuffling that would have sorted the column. The shuffling is just an ordinary integer array whose values identifies the order of positions in the original array. The previous subsection(2.2.4) explained how it in Numpy is possible to sort an array by another integer array.

```
In [18]: x = array([3,1,2])

In [19]: idx = x.argsort()

In [20]: idx
Out[20]: array([1, 2, 0])

In [21]: x[idx]
Out[21]: array([1, 2, 3])
```

**Recreating Original Order** In some instances it is very useful to recreate the original unsorted ordering of an array that has been sorted. The application of the `argsort` function to an array returns the indices that sorts the

array if the original array is index by the index array. We compute the reversible indices by argsorting the result of a call to `argsort`. If the reversible indices are computed from a unsorted array that is subsequently sorted, then it is possible to recreate the original array by indexing the sorted array by the reversible index array. This is best illustrated by an example:

```
In [18]: x = array([3,1,2])

In [19]: idx = x.argsort()

In [22]: sorted_x = x[idx]

In [23]: rev_idx = idx.argsort()

In [26]: rev_idx
Out[26]: array([2, 0, 1])

In [24]: sorted_x
Out[24]: array([1, 2, 3])

In [25]: sorted_x[rev_idx]
Out[25]: array([3, 1, 2])
```

### 2.2.5 Type Systems

Every value in a programming language has a type. Integers, strings and class instances are all examples of values with a type. A type system attempts to prove that no type errors can occur. What constitutes as errors varies depending on the specific type system, but it generally tries to guarantee that the type of an operation and the type of the value being operated on makes sense. A type system is said to be *static* when the type checking is performed at compile time and *dynamic* when the type checking occurs at run-time. Some of the advantages of static type checking are a restricted form of program verification, slightly faster execution speed and documentation. The major disadvantage of static type checkers is that they reject some programs that do not have type errors and are well-behaved at runtime, but this can not be proven at compile time and the programs are therefore rejected. Dynamic type systems are more flexible, but lacks the advantages gained by compile time type checking. Variables in a statically typed language have a type, whereas variables in a dynamically typed language do not.

### 2.2.6 The relational model for database management

All relational database management systems are based on a model published in a paper by E.F.Codd in 1969[5]. The term relation, as in *relational model* and *relational database*, is used here in its accepted mathematical sense. Given sets $S_1, S_2, \ldots, S_n$ (not necessarily distinct), $R$ is a relation on these $n$ sets if it is a set of $n$- tuples each of which has its first element from $S_1$, its second element from $S_2$, and so on[6]. More concisely, $R$ is a subset of the Cartesian product $S_1 \times S_2 \times \ldots \times S_n$. This translates in layman's term to a table with a subset of the possible combinations of the sets. Each column in the table contains values from one set and is called a *domain* of $R$. Each row of the table represents a combination of the set values. The table has the following more formal properties[6]:

1. Each row represents an $n$-duple in $R$.
2. The ordering of rows is immaterial.
3. All rows are distinct.
4. The ordering of columns is significant–it corresponds to the ordering $S_1, S_2, \ldots, S_n$ of the domains on which R is defined.
5. The significance of each column is partially conveyed by labeling it with the name of the corresponding domain.

Every row is unique and consequently a single domain (or combination of domains) of a table forms a primary key. A primary key is the smallest combination of domains that uniquely identifies a row. It's a common requirement for elements of a relation to cross-reference other elements of the same relation or elements of a different relation. A domain (or a combination of domains) of a relation $R$ is a *foreign key* if its not the primary key of R but its elements are values of the primary key of some relation $S$.

Imagine a table of students and another table of schools. It would clearly be useful reference the school a student attended and such a reference between the two tables is a cross-reference. Notice that we for this example assume that each student attend exactly one school. The school primary key may then be a foreign key in the students table.

The process of organizing data to minimize redundancy is called normalization. There are well defined rules to guide the design of a well-formed database schema. There are many levels of normalization, but the early level proposed by Codd in 1969[5] had, according to the paper, the following advantages which also holds true for later more fine grained normalizations:

1. It would be devoid of pointers (address-valued or displacement-valued).
2. It would avoid all dependence on hash addressing schemes.
3. It would contain no indices or ordering lists.

We've seen how the relational model supports one-to-many relationships in our trivial example where each student belonged to a school. The relational model does not, however, directly support many-to-many relationships.

A database model completely unable to express many-to-many relationships would hardly been useful for real world applications. The relational model overcomes this limitation by constructing a special table to express many-to-many relationships, called a junction table. The junction table contains domains to describe the foreign keys for both of the tables that it connects and the primary key is the combination of those domains. Each row in the junction table connects two entities from the two tables with a many-to-many relationships.

Lets illustrate this by expanding the previous student example. If we wanted to model the relationship between students and courses we would expect a student to attend multiple courses and a course to be attended by multiple students. This is an example of a many-to-many relationship. This relation is impossible to describe be augmenting the existing tables. A table consists of a fixed number of columns. Every student attends a variable amount of courses and every course is attended by a variable amount of students. We're therefore unable to store the variable number of courses every student attends in the *student* table and we're also unable to store the variable number of students attending each course in the *course* table.

### 2.2.7 MapReduce

`Map` and `reduce` are important functions in many programming languages but originates from functional programming paradigm.

Common for all functional languages is that functions are first class objects. This means that functions are values as any other value of the language. They may be bound to names, gives as arguments to functions or returned from functions.

Complex functions are ideally composed of multiple simpler ones. Functions should be as general as possible for better reusability.

Another important concept is the absence or restriction of mutable/destructive operations. It's clearly easier to test and reason about a function if it does not depend upon or change external state. This also creates a loose coupling between the different components of a program or module.

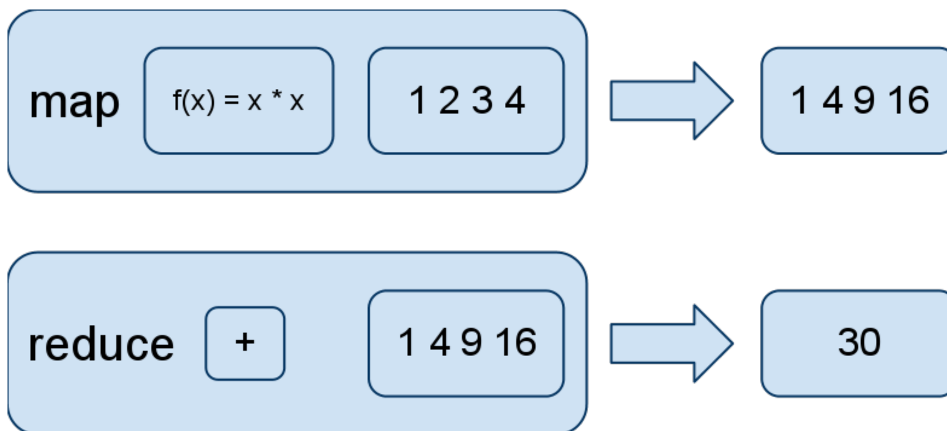



Figure 2.1: Illustration of the two functions map and reduce

**Map** applies a function to every element of a sequence and returns a new sequence of results.

**Reduce** combines its arguments using a supplied combination function.

Google introduced *MapReduce* as a framework for distributed data processing[8]. Google's framework is suitable for processing huge datasets on certain kinds of distributable problems i.e. to problems that are divisible into multiple sub-problems whose solutions are combined in the reduction step. This model is desirable in the distributed domain because the implementor of an algorithm writes only a reduce function and a map function. The framework handles all other issues such as load balancing, hardware failures and job distribution.

### 2.2.8 Hash Table

A hash table is a data structure for arbitrary retrieval of data. The hash table maps keys to their associated values by using a hash function to compute a unique key for each value. Maps are very useful for many programming problems and almost ubiquitous in many languages. Many algorithms have been developed for efficient storage of key/value pairs and succeeding retrieval based on a the key.

### 2.2.9 Software Bugs

All programs should ideally be bug free, but the complexity involved with software development makes nontrivial programs completely devoid of bugs nearly impossible to create. There are many sources of bugs, but many can

be classified as unintended interactions between different parts of a software system. This frequently occurs because software systems are complex in their nature, so programmers are unable to mentally track every possible way in which parts can interact and therefore make false assumptions, misunderstands or perform a logical error.

Serious bugs stops further execution of the program or in other ways make their presence very clear. Serious bugs are always preferable because they immediately call attention to the error and are often also easier to track and hence resolve. Bugs that only have a subtle effect on a system or only occurs in very unusual but legal input are often the hardest to locate and track. The importance of a bug free system varies. It's annoying when a web browser crashes, but possibly fatal for other systems such as a railway signaling control system.

# Chapter 3

# Methods

## 3.1 Annotations

The genomic sequence of an organism provides a great resource for understanding and learning about living organisms such as ourselves. The DNA sequence alone provides little or no information without understanding it in a deeper context. Annotations helps bridge the gap from the sequence to the biology of an organism. Annotations are the product of the layers of analysis and interpretation of raw DNA data necessary to extract its biological significance and place it into the context of our understanding of biological processes[24]. Annotations marks empirical properties of a genome — in particular proteins and their products.An annotation track is illustrated in figure 3.1 on the following page.

We'll define the following terms when discussing annotations:

**Track Element** A single location on a genome with possible meta-information. For example a specific gene or a specific transcription factor binding site on the human genome.

**Chromosome Annotation Track** A collection of related track elements on a genome, all located on the same chromosome. For example all known genes located on the first human chromosome.

**Annotation Track** All related track elements on a genome. For example all known human genes.

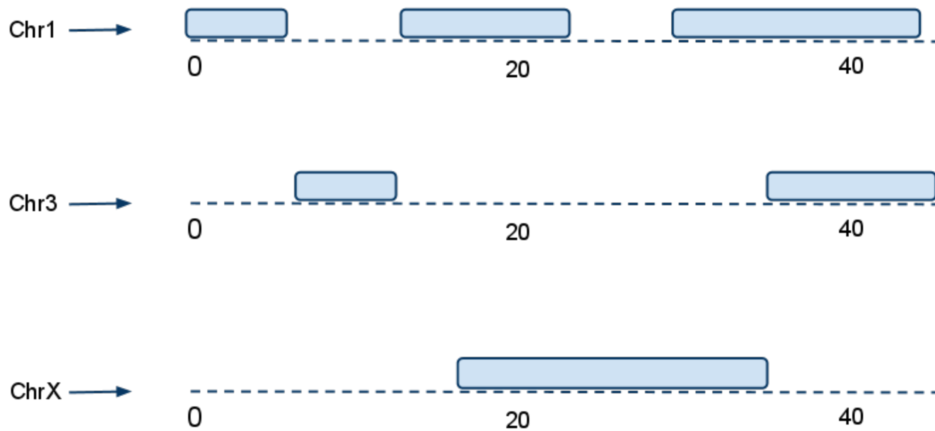



Figure 3.1: Illustration of an annotation track.

### 3.1.1 Chromosomes Divides Annotations into Disjoint Sets

The genome of an organism is the whole of its hereditary information encoded in its DNA. A genome consists of chromosomes that are organized structures of DNA. The DNA within a chromosome is viewed as a sequence with defined start and end positions. Every nucleotide is therefore addressable as an offset from the start position. An track element marks a location within a chromosome. An annotation track is therefore naturally divided into disjoint sets by the chromosome boundaries of the species, as illustrated in figure 3.1.

A simplified model of track elements are as intervals on a number line. This has the implication that most operations on annotations can be simplified to operations on intervals.

## 3.2 Coupled Track Elements are the Basis for Comparative Operations

A very important type of meta-information for some track elements are the locations of similar or identical track elements in different genomes. This knowledge glues the genomes together and establishes the basis for comparisons across genome boundaries. These track elements will become very important when discussing inter-genome comparisons and we'll therefore need to define some proper terminology:

**Coupled Track Element** A special type of track elements that are connected to one or more track elements in different genomes.

**Coupled Annotation Track** A collection of coupled track elements on a single genome that all are connected to coupled track elements from the same coupled annotation tracks. Coupled track elements within a coupled annotation track may overlap, which is different from regular annotation tracks.

An example of a coupled annotation track could be the location of all human genes with known mouse orthologs or the locations of regions in the dog genome conserved between the mouse genome and the human genome.

**Connected Coupled Track Elements (CCTEs)** A set of coupled track elements from different genomes that are connected.

An example is the location of a human gene and its corresponding orthologs location in the mouse genome.

**Connected Coupled Annotation Tracks** A regular coupled annotation track describes a series of related coupled track elements on a single genome. Connected coupled annotation tracks are the set of coupled annotation tracks that are connected.

An example of this is a coupled annotation track describing human genes with mouse orthologs that is connected with the coupled annotation track describing mouse genes with human orthologs. The two of them together form a connected coupled annotation track.

## 3.3 Modeling Intervals Suitably for Array Programming

Section 3.1 explained how it was possible to view track elements on a chromosome level as intervals on a number line. Any methodology concerning intervals is therefore transferable to track element. This is useful since intervals on a number line constitutes a simpler model of much of the problem domain.

We'll in the following discussion refer to a collection of related intervals as an interval track. This is similar to how we in section 3.3 defined a chromosome annotation track to be a collection of track elements of a similar type within a chromosome. An interval track consists of zero or more non-overlapping intervals on a number line. Intervals from different tracks may overlap.

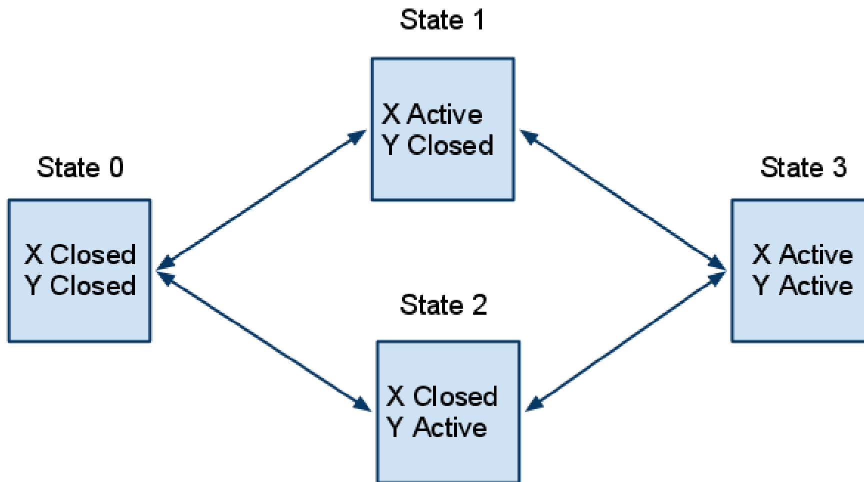



Figure 3.2: Transitions Between States generated for two interval tracks, X and Y. *Closed* represent the absence of intervals of that type.

### 3.3.1 Every Position has a State

Intervals from different tracks may overlap and it follows from this that any position on the number line may be covered by zero or more intervals. Keeping track of multiple overlapping intervals has proven difficult using array programming. Instead we reason about the same problem from the point of view of positions on the number line. Each position has a state, i.e. zero or more intervals covers that position. We're thus able to describe all the interval tracks and their interaction through a chain of adjacent non-overlapping states that precisely describes the active interval types for the span of each state. The trivial case is a number line without intervals where every position has the same state because none of them are covered by intervals. A line with a single interval track has two states, positions are either inside or outside an interval. A line with two interval tracks, as illustrated in figure 3.2, has four states and so on.

#### Intervals as Transition Events

Every interval generates two transition events. The first transition occurs at the start position of the interval and denotes a transition from the context dependent current state to the state representing what was the current state including the interval track that began at that point. The second transition occurs at the end of the interval and denotes a transition from the context dependent current state to what was the current state excluding the interval

track that just ended. There are only as many transitions from each state as there are sets of intervals. Every transition event has a position and belongs to an interval track.

#### The State Sequence is Deductible from the Transition Events

All transition events from every interval track are merged in ascending order. The start state of any system is always the empty state, that's the state representing uncovered positions. The following states are deductible from the starting state and the merged transition events.

#### Multiple Transition Events at a Single Position

An interval is defined by a pair of start and end positions. It's common in bioinformatics to define intervals using an excluding end point. This means that every position from the start position up to, but not including, the end position are part of the interval. We'll follow this practice and our methodology must be sensitive to this.

We've previously discussed how to view intervals as a series of events possible different interval tracks with a position. It's possible that multiple events occur at the same position. This happens when, for example, an interval ends and another starts at the same position. The order of these two events are not interchangeable. If the start event occurs before the end event, then this would indicate an overlap between the two intervals which is false. The end event must therefore occur before the start event at the same position.

## 3.4 Operations on Intervals using Array Programming

### 3.4.1 Set Operations

Set operations such as intersection, union and complement are well understood concepts that also applies to intervals. The union of two intervals are all areas covered by one or both of the intervals, the intersection is the area covered by both and the complement is the area covered by none of the intervals.

Example figure 3.3 on the next page depicts two interval tracks on a line. A corresponding state sequence generated from the interval tracks is illustrated in figure 3.4. Extracting the union or the intersection is both visually

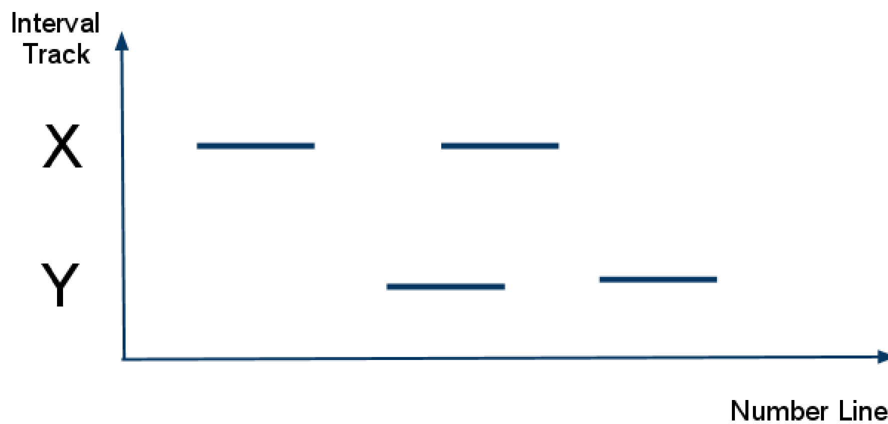



Figure 3.3: An example where the two interval tracks X and Y mark two locations each.

and computationally quite simple given a state sequence. The intersection are those areas with an y-axis value of 3, indicating that both *State X* and *State Y* are active. The union is the complement of those areas with an y-axis value of 0, that is every position where at least one interval set was active.

### 3.4.2 Counting Occurrences Within an Interval

It's often very useful to be able to count how many intervals from a interval track that falls inside every interval from another interval track. In the discussion below we'll call the interval track that we're counting occurrences within for the host interval track. The interval track that is being counted will be called the target interval track. Intervals from the host interval track are called host intervals and similarly intervals from the target interval track are called target intervals. The rest of this discussion assumes the host intervals to be as large or larger than the target intervals.

Lets first establish how we define 'inside an interval'. A single position is clearly inside an interval if it's within the boundaries of the interval. An interval is also undoubtedly inside another interval if it's completely covered by the other interval. Depending on use case, it's sometimes too strict to require complete coverage. This is especially true when comparing intervals of approximately the same size or when one of the interval track only has estimated locations. The opposite extreme is to treat any overlap as sufficient for acceptance. If the intervals compared are quite large, a single overlapping position should hardly be enough.

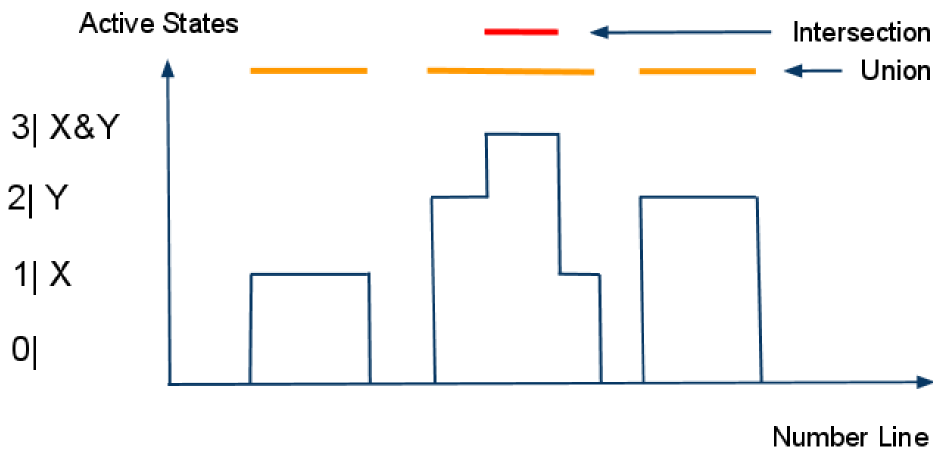



Figure 3.4: State representation of the interval tracks in figure 3.3 on the preceding page. The y-axis represent the state type (also depicted in figure 3.2 on page 23). The union are the areas covered by all states except *State 0*. The intersection are those areas covered by *State 3*

A less strict requirement of 'inside' is to only require a predefined ratio of a target interval to overlap a host interval. The ratio required may vary for different usages. There is always a trade-off between false positives and false negatives when performing an analysis. A too weak requirement will be more likely to increase the number of false positives, i.e. falsely detected matches. A too strict requirement is more likely to exclude many occurrences that really should have been detected, formally named false negatives. The ideal refinement depends on use case.

It's evident from the above discussion that different use cases require different definitions of 'inside'. We'll now discuss slightly different methods for counting occurrences that fits with the array programming model. We'll continue to use the transition event model described in section 3.3 on page 22. The model viewed each interval as two transition events, a start event and an end event. Every event has a position on the number line and a belongs to a specific interval track. Events from different interval tracks are merged and ordered by their position on the number line.

The strictest definition of 'inside', requiring a whole target interval to be covered by a host interval, is computable by counting the number of complete start and end event pairs from the target track between the host interval's start and end events.

The perhaps most intuitive way to implement support for a weaker ratio requirement is to scale the target intervals before the actual computation. An initial requirement was for the host intervals to be at least as large as the

target intervals. All host intervals must therefore be at least twice as big as a target interval scaled to half its original size. A host interval that completely covers an target interval scaled to half its size actually covers at least three quarters of the original target interval. Similarly a host interval that covers the middle position of a target interval must at least cover half of the actual target interval. A consequence of this is that this scaling method does not allow for a weaker ratio than 50%. A slight variation of the scaling of target intervals is to use the midpoint of target intervals as a measure for 'inside'. If the midpoint of a target interval is inside a host interval, then it follows that at least 50% of the target interval is covered by the host interval. This is even true when the host interval is as small as half the size of the target interval. It hardly makes sense to talk about 'inside' if the target interval is a lot larger (more than two times) than the host interval.

### 3.4.3 Connecting Closest Intervals From Different Tracks

This section discusses the problem of locating the closest neighbors between two interval tracks. For every interval in one interval track, we want to find the closest interval in another interval track. The interval tracks are as before assumed to be internally ordered and non-overlapping. Lets first reason about the simplest possible solution to the problem:

- An unfeasible naive solution with a runtime of $O(N^2)$ is to compute the distance between each point and every interval and selecting the interval with the shortest distance, for all points. This easily implemented using both array and imperative programming.
- An optimized solution is to selectively iterate over an ordered input set and only compare against a few relevant intervals. This is quite simple using imperative programming, but the dependence on selective iteration makes this approach impossible using array programming.
- Minor modifications to our model of the problem domain enables us to solve the problem efficiently using array programming.

Before we start the actual discussion, note the following:

- If the interval $x$ is closer to the interval $y$ than any other interval, then it follows that the distance from the midpoint of $x$ to $y$ is the shortest distance between the midpoint of $x$ and any other interval.
- The distance between the midpoint of $x$ and the midpoint of $y$ must **not** be the shortest distance between the midpoint of $x$ and any other midpoint.

- Any position between two intervals is closest to the interval with the shortest distance between that position and the interval.
- The distances from a position to the intervals are equally long at the midpoint position between the intervals.

Every position above the midpoint between two intervals is therefore closer to the interval placed higher on the number line. Every position below the midpoint position is consequently closer to the interval placed lower on the number line. The midpoints between all adjacent target intervals generates a series of what we have coined as *pivot positions*. These pivot positions are used to generate a new set of intervals, called *pivot intervals*. The first pivot interval spans the region from the start of the number line to the first pivot position. The second interval spans the region from the first pivot position to the second pivot position. This continues until the last interval that spans the region from the last pivot position and the rest of the number line. These pivot intervals are non-overlapping and they completely cover the number line.

Every target interval has a corresponding pivot interval of at least an equal size. Pivot intervals are useful because every position within them are closer to their corresponding target interval than any other target interval.

The generated pivot intervals combined with the ability to count intervals within intervals of another track, as discussed in section 3.4.2 on page 25, almost solves our initial goal of connecting the closest intervals. The methods so far enables us to count the number of host intervals that are closest to each target interval. The final result is deductible from this partial result. The interval tracks are as we remember internally ordered. The first target interval must therefore also be the leftmost target interval on the number line. Furthermore, if the first pivot interval encompasses three host intervals, then we know that those intervals are the three leftmost and therefore the first three host intervals. This final conversion step is trivially computable and concludes this method.

## 3.5 Inter-Genome Comparisons

Section 3.2 on page 21 described how coupled annotation tracks create connections between genomes. We still need to discuss how to perform meaningful comparisons utilizing these coupled annotation tracks. Our main contributions are two methods for comparisons. The first approach is to transfer track elements from a source genome, through a coupled annotation track,

to a target genome. The second approach is to compute results local to each coupled track element and compare the mapped results between genomes.

### 3.5.1 Projection of Track Elements

An interesting application of coupled annotation tracks is to treat them as transition locations between the genomes. Imagine tubes connected to corresponding coupled track elements on different genomes. Regular track elements can be funneled through those tubes from one genome to another. Regular track elements completely covered by a coupled track element have an unambiguous projection between the genomes. Coupled track elements may overlap. A regular track element may therefore consequently be projected to multiple locations in another genome if the track element is covered by multiple coupled track elements. Projecting only those track elements within coupled track elements on the source genome, creates a new annotation track on the target genome. It's then interesting to compare the track to related annotation tracks already present on the target genome.

### 3.5.2 Quantitative Comparisons

Coupled annotation tracks identify somehow related locations in different genomes. We previously, in section 3.5.1, discussed how to project track elements through these coupled annotation tracks. A serious issue with projection was that only positions within mapped annotations were transferable. A different approach is to compare partial results between mapped coupled track elements in different genomes. The partial results are computed relative to each coupled track element locally within each genome. The term relative in this setting is important. This has the implication that the location whose partial result was deducted from does not have to be the location of the coupled track element. It might also be from a defined vicinity of the coupled track element, the closest gene or for example the area between the coupled track element and the closest gene. The only restriction is that the coupled track elements must the points of reference, everything else is otherwise possible. The partial results are typically computed by a combination of the operations previously mentioned e.g. counting occurrences or computing distances.

### 3.5.3 Accounting for Strand Direction

Corresponding coupled track elements in different genomes may be inverted in relation to each other. It's important to adjust for this when projecting

points or computing distances, especially if the mapped annotations span a large interval. The first position in a coupled track element should be projected to the last position of the corresponding track element in the other genome if the mapped annotations are inverted. The second to last position should analogously be projected to the second position in the corresponding track element.

### 3.5.4 Coupled Track Elements may Overlap

We previously discussed operations on interval tracks, in 3.4, where we required that intervals from the same track did not overlap. Coupled track elements were introduced in 3.2 and one of the properties of coupled track elements is that they may overlap. The discussion on quantitative comparisons between mapped coupled track elements in section 3.5.2 on the previous page clearly showed the need for additional operations that allowed for overlaps. This section provides methods to handle such questions using array programming. Note all questions are sensible when overlapping track elements must be preserved. The intersection is for example non-sensible when the original track elements must be contained.

#### Counting Overlaps

Slight modifications to the previous event model discussed in section 3.3 on page 22 facilitates counting the number of intervals overlapping intervals from a *reference track*. We'll call the interval track we count from the reference track to avoid confusion. Both interval tracks may overlap and they're also otherwise equal. We treat both interval tracks as multiple transition events and, as before, and merge them into a common sequence.

The actual computation is easiest if it's performed in two steps. We'll make a distinction between two types of overlaps:

1. A regular interval may start before the reference interval it overlaps and end inside it or completely span it.

2. A regular interval may also start within a reference interval and end before or after it.

It's difficult, if at all possible, to perform both counts simultaneously using array programming. It's, however, quite simple to sum the results of two partly independent computations.

Our initial step is to count the number of open regular intervals at the beginning of each reference interval. We'll interpret the transition event sequence and the following generated state sequence slightly different to enable this. We'll let each state represent the number of open regular intervals. So a series of non-overlapping adjacent states spans the number line and encodes the number of open or active regular intervals for the duration of the state. The first partial computation is completed by extracting the number of open regular intervals for the states associated with each reference interval start event.

The second step is quite trivial. It's just a matter of counting the number of regular start events between every reference start and end events.

The total number of regular intervals overlapping each reference interval is computable by adding the two partial results discussed above. Some extra complications arose because of the possibly internally overlapping interval tracks, but a proper solution is still not too complex to be practically usable.

### 3.5.5 Mapping Coupled Annotation Tracks

Coupled track elements differ from ordinary track elements in that they contain additional information about similar locations in other species. Coupled track elements are introduced in section 3.2.

Both of the two proposed methodologies for inter-genome comparisons, projection(3.5.1) and quantitative comparisons (3.5.2), are based on coupled annotation tracks. We therefore need to discuss and develop proper methods for array programming operations on coupled track elements. Corresponding coupled track elements often cross chromosome boundaries. This section discusses methodology needed for connecting the chromosome level operations together with inter-genome comparisons.

An intuitive attempt would be to construct and operate on the mapping using a hash table, a data structure discussed in section 2.2.8. It's entirely possible to perform an implementation based on such a data structure. This would, however, break our attempt at creating a complete methodology for solving comparative problems using array programming. A map is specialized for storage and arbitrary retrieval of data. Arbitrary lookup is not important for our use of mappable track elements. We want to operate on all annotations and the particular order is not important as long as it's consistent. Hash maps are therefore specialized for usages irrelevant to the problem at hand.

Section 2.2.6 discussed the relational model that most databases are built

upon. The original paper[5] claimed some advantages to the model. Especially one point is interesting in the context of mappable track elements;

> The relational model avoids all dependence on hash addressing schemes.

This statement does not imply that hash tables and relational databases are equivalent, it simply states that the functionality provided by a relational model does not depend on hashing. The relational model is created for organizing and subsequently retrieving data.

Lets see how this relates to coupled track elements. Coupled track elements essentially encode two distinct types of information:

1. Locations on genomes.
2. Connections between locations.

A table in the relational model that describes locations must at least contain columns for the following types of information to unambiguously describe a location:

- A genome identifier
- A chromosome identifier
- The start position
- The end position

Every row in such a table describes a location on a genome. We'll additionally need to extend this model with information about mappings or connections between locations. If every coupled track element mapped to exactly one other track element, then this information could have been encoded by augmenting the table with an additional column describing mappings. Every coupled track, element, however, may map to arbitrary many other track elements. It's therefore not possible to augment the table so it additionally describes mappings between coupled track elements. The coupled track elements form a many-to-many relationship. The relational model describes many-to-many relationships by creating an additional table that describes mappings. Such a table is called a junction table.

## 3.6 Other

### 3.6.1 Representing Sets Numerically

Every state represents a combination of zero or more interval sets. Each state is in other words a subset of the superset representing all states. Every subset must be given a unique representation to be computable using array programming. For practicality and efficiency it's important to find a succinct and informative representation. The number of possible subsets of a set grows exponentially with the size of the set[15]. The exponential growth of the set size makes a binary number system representation especially suitable. Every subset of a set of N elements can be represented by a n-digit binary number. The value at every position in the binary number represents an element of the set. If an element is present in a subset, then the value at its position is 1. Conversely the value is 0 if the element isn't present in the subset.

It's naturally also possible to represent subsets using numbers of base 10 instead of base 2. Every element of the set is uniquely identified by an exponentiation of 2. A subset is uniquely identified as the sum of the integer representations of its elements.

We'll illustrate this by creating a unique integer representation of every subset of the set Red,Green,Blue:

1. Identify each element of the set by a exponentiation of 2.

| Color | Exponentiation | base 10 | base 2 |
|---|---|---|---|
| Red | $2^0$ | 1 | 001 |
| Green | $2^1$ | 2 | 010 |
| Blue | $2^2$ | 4 | 100 |

Table 3.1: caption-test

2. Every subset is then identified as the sum of its elements:

| Subset | Exponentiation | base 10 | base 2 |
|---|---|---|---|
| $\emptyset$ | 0 | 0 | 000 |
| Red | $2^0$ | 1 | 001 |
| Green | $2^1$ | 2 | 010 |
| Green,Red | $2^1 + 2^0$ | 3 | 011 |
| Blue | $2^2$ | 4 | 100 |
| Blue,Red | $2^2 + 2^0$ | 5 | 101 |
| Blue,Green | $2^2 + 2^1$ | 6 | 110 |
| Blue,Green,Red | $2^2 + 2^1 + 2^0$ | 7 | 111 |

Table 3.2: caption-test

# Chapter 4

# Design and Implementation

The implementation must not only adhere to the methods and algorithms described, it must also fit with the target language Python in conjunction with the Numpy library.

We've made major efforts to keep the complexity, concepts and usage as simple as possible. We belive that any program and especially a framework must be understandable and intuitive in order to be extendable and really usable.

The higher abstraction levels of the system were implemented using an object-oriented design. Low level operations are implemented using array programming through the Numpy library. The implementation is presented in a bottom up approach.

Intra-genome operations operate on one or more annotation tracks within the same genome. The corresponding low-level implementations, which are presented in 4.1, operates on chromosome annotation tracks. We'll later, in section 4.2, see how the framework combines the partial results produced by the low-level operations. Chromosome annotation tracks are introduced in section 3.1, but are in essence all track elements from an annotation track that resides on a specific chromosome.

## 4.1 Operations on Chromosome Annotation Tracks

Low level operations are implemented with array programming using the Numpy library. The operations solve problems such as computing overlap, union, intersection, connecting the closest track elements from a corresponding annotation tracks and more. The methodology needed to solve such problems using array programming is presented in chapter 3.

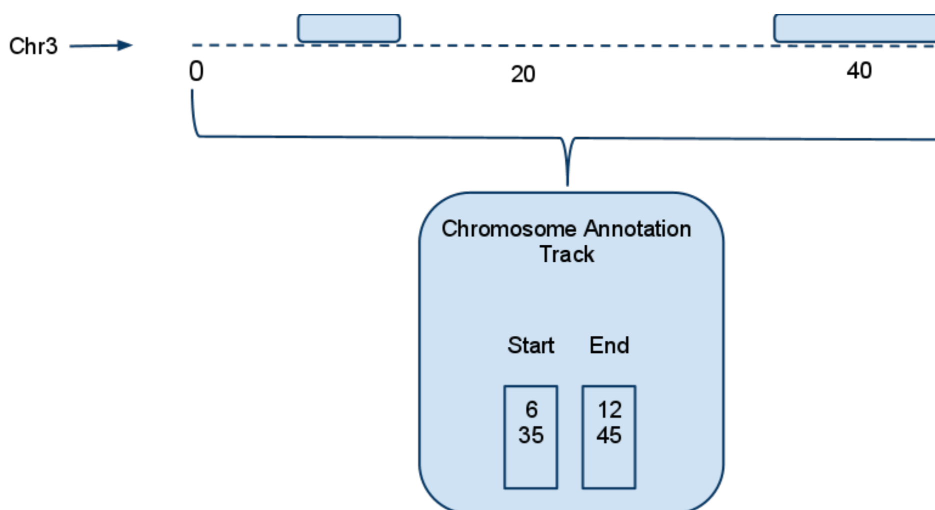



Figure 4.1: Track elements on a chromosome annotation track with a visual representation compared to an actual representation.

### 4.1.1 Chromosome Tracks are Represented by Numpy Arrays

Python has some overhead in both time and space when creating objects. The abundance of biological data quickly ruled out the possibility of a completely object oriented design.

Track elements on a chromosome level are therefore represented by Numpy arrays. Numpy arrays are comparable to more efficient languages in both space and time usage, this is explained in detail in section 2.2.2 on page 10. A chromosome annotation track is represented by two arrays of *start* and *end* positions. The rows of the two arrays correspond. This relationship is illustrated in figure 4.1.

### 4.1.2 Generating Number Line States

Section 3.3.1 described how track elements on a chromosome could be viewed as a series of adjacent non-overlapping states. Section 3.3.1 explained how track elements each generate two transition events between states. It was also outlined how a state set could be derived from a series of transition events. Annotation tracks are given a unique flag value, as discussed in 3.6.1. Every transition start event is given the flag value of its annotation track. Correspondingly, every transition end event is given the negative of the flag value of its annotation track. Every state represents a combination annotation tracks and is unambiguously represented as the sum of flag values

from the annotation tracks it represents. We'll define a few terms to make the following discussion easier.

- A state representing an absence of annotation tracks is called an **empty state**.
- A state representing all annotation tracks is called a **complete state**.
- All other states are referred to as **middle states**.

We saw in section 3.4.1 how the state model made it easy to formulate methodology for computing the union and intersection of annotation tracks using array programming. We'll first discuss the implementation of a function that computes the state sequence from two chromosome annotation tracks. We refer to the code listing in 4.1.

Listing 4.1: A function common to both union and intersection

```
Line 1 def statesAndTransitionPositions(x,y):
     -     positions = concatenateAnnotations(x,y)
     -     flags = getFlags(x,y)
     -     idx = positions.argsort()
     5     flags = flags[idx]
     -     positions = positions[idx]
     -     states = flags.cumsum()
     -     return states,positions
```

The `positions` variable is an array of all transition event positions. The `flags` variable is an array identifying the transition event type occurring at that position. Both the `flags` and `positions` are reordered relative to the `positions` array (line 4-6). The `states` sequence is derived from the cumulative sum of the `flags` array (line 7). The series of states and the starting position for each state are returned from the function.

### 4.1.3 The Union of Two Chromosome Annotation Tracks

We concluded in 3.4.1 that set operations would easily be implementable if we were able to generate a series of states from the annotation tracks. Lets see how to compute the union of two chromosome annotation tracks now that we're able to produce the state series. The union of two chromosome annotation tracks are the areas covered by one or more track elements. In other words, all areas except those covered by none.

Listing 4.2: Union function at a chromosome level

```
Line 1 def union(x,y):
```

```
-     states,positions = statesAndTransitionPositions(x,y)
-     noneActive = states == 0
-     firstStart = np.roll(noneActive,1)
5     return Annotations(positions[firstStart],positions[noneActive])
```

The second line computes the series of states and positions by utilizing the function presented above in 4.1.2. The third line extracts the empty states. The state following an empty state must by definition be a middle state, see the figure of legal transitions in 3.2 on page 23. The forth line identifies the middle states following an empty state. The returned positions are the complement intervals of the empty states, which by definition is the union.

### 4.1.4 The Intersection of Two Chromosome Annotation Tracks

The intersection is computed very similarly to the union. Instead of identifying the empty states, we identify the complete states. The intersection is by definition only the complete states. Every state is internally only represented by its start position. It's therefore necessary to identify the starting position of the state following a complete state as this is also the end position of the complete state.

Listing 4.3: Intersection function at a chromosome level

```
Line 1  def intersection(x,y):
-           states,positions = statesAndTransitionPositions(x,y)
-           bothActive = states == 3
-           oneEnds = np.roll(bothActive,1)
5           return  Annotations(positions[bothStart],positions[firstEnd])
```

### 4.1.5 Assuring that Chromosome Annotation Track Elements are Non-overlapping and Ordered

Union and intersection are important building blocks for many other low-level operations. A very useful feature is the ability to remove overlaps and order a chromosome annotation track. This is very easy implemented by computing the union of a chromosome annotation track against itself.

Listing 4.4: Removing internal overlaps made easy by union.

```
Line 1  def removeOverlaps(x):
-           return union(x,x)
```

### 4.1.6 Indices if Sorted

We will soon need a function that computes the position every value of an array would have had if the array was sorted. Section 2.2.4 discusses how the Numpy `argsort` function returns the reordering that will sort the array. The reordering is actually an array of indices that index the values of the array in sorted order. This allows us to reorder multiple related arrays according to the sorted order of a single array. If we in a second call to `argsort` use the returned value of the first call to argsort as a parameter, we end up with the positions an element would have had in a sorted array. Another useful property of the second `argsort` call is that these indices will reorder the array in sorted order back to the original unordered order. This is a property that we'll exploit when performing inter-genome comparisons. The implementation of `indicesIfSorted` therefore is as follows:

Listing 4.5: A function that returns the the positions the elements of the input arrays would have had if they where concatenated and sorted

```
Line 1  def indicesIfSorted(*args):
   -        x = np.concatenate(args)
   -        return x.argsort().argsort()
```

### 4.1.7 Counting Points Separated by Pivot Elements

Much of the methodology for comparisons between intervals using array programming as discussed in chapter 3 simplified the problems to counting the distribution of points separated by pivot elements. The term *pivot elements* is coined in this thesis and explained in 4.1.11. This section discusses the implementation of the function *countItems* which counts the distribution of points separated by pivot elements. The application of the function will first become apparent in the following sections.

Listing 4.6: A function that counts the number of points between each pivot element.

```
Line 1  def countItems(pivots,items):
   -        allPositions = indicesIfSorted(pivots,items)
   -        pivotPositions = allPositions[:len(pivots)]
   -        start,end = [-1],[len(allPositions)]
   5        pivotPositions = np.concatenate((start,pivotPositions,end))
   -        diffs = np.diff(pivotPositions)
   -        return diffs - 1
```

The pivots positions divides the chromosome into distinct parts. An absence of pivot positions represents a single part spanning the whole chromosome. A single pivot position divides the chromosome into two distinct parts. One part up to the pivot position and one part including the position and the rest of the chromosome. Two pivot positions creates three parts, and so on... The general pattern is therefore:

$$N \text{ pivot positions} = N + 1 \text{ parts}$$

The function accepts an array of pivot positions and an array of points. The second line computes the position every element would have had if the two input arrays were concatenated and sorted and stores the result in the *allPositions* variable. The positions the pivot elements would have had are listed first and the positions the items would have had are then subsequently listed. For example if the first pivot element would have been at the third position in a concatenated and ordered array, then this translates to that at exactly 3 items are located left of the first (smallest) pivot element and belongs to the first part[1]. If the next pivot element would have been at position 5, then this means there are exactly one item between the first and the second pivot elements. This is one less than the difference between the two positions.

blah blah ferdig. husk aa bedre forklare det med start=-1 og end=len(x).

### 4.1.8 Counting the Number of Points Closest to each Track Element

Section 4.1.11 discussed methodology for connecting intervals from different chromosome annotations tracks using array programming. The midpoint between two adjacent track elements is, obviously, the only point where both track elements are equally distant. We furthermore defined, for completeness, that the exact midpoint was closer to the rightmost track element. The midpoints between adjacent track elements therefore form pivot elements grouping a chromosome into parts. Each part is the area closest to its corresponding track element.

Listing 4.7: A function that counts the points closest to each region

```
Line 1  def countClosestPointsPerTrackElement(trackElements, points):
     -      pivots = getTrackElementsMidpoints(trackElements)
     -      return countClosest(pivots,points)
```

[1]assuming the array to be zero-indexed, as is the case with numpy.

Listing 4.8: A function that computes the midpoints between track elements

```
def getTrackElementsMidpoints(trackElements):
    leftSide = trackElements.end[:-1]
    rightSide = trackElements.start[1:]
    means = elementwiseMean(leftSide,rightSide)
    return np.ceil(means)
```

### 4.1.9 Counting the Number of Points within each Track Element

Counting the number of points within each track element is quite similar to the problem of counting the closest points for each track element, as discussed in 4.1.8. The only deviation is how we select the pivot positions and also we process the results of the `countItems`(4.6) function.

If we simply used the start and end positions of the track elements as pivot positions we would get a count of points within each track element. This additionally give us a count of points between each track element which is not interesting for this computation. The uninteresting counts are removed from the final result.

Listing 4.9: Function that counts the number of points inside each region

```
def countPointsPerTrackElement(trackElements,points):
    pivots = np.concatenate((trackElements.start,trackElements.end))
    pivots.sort()
    counts = countItems(pivots,points)
    return counts[1::2]
```

### 4.1.10 Counting Intervals inside Regions

We've seen how to count if the number of points inside track elements in 4.1.9. A perhaps more interesting question to answer is how many track elements from one annotation track are within each track element from another annotation track. To answer this question we first need to be precise about what is meant by 'inside'. This was the topic of 3.4.2, where it was argued that at least 50% overlap was required. This coincided with checking if the midpoint of a track element was inside the other track element type given that the track elements were approximately equally large or that reference track element is larger than the counted track element.

So to count the number of regions inside another type of regions we first compute the middlepoints for each region and then utilize the existing functionality.

Listing 4.10: Function that counts the number of intervals (smaller regions) inside regions

```
Line 1  def countIntervalsPerRegion(regions,intervals):
   -        midPoints = elementwiseMean(intervals.start,intervals.end)
   -        return countPointsPerRegion(regions,midPoints)
```

### 4.1.11 Counting Closest Intervals for Each Region

The solution to this problem follows the same logic and simple step as the previous section 4.1.10 on the previous page. We compute the middle point for each interval and apply the previously defined function for points.

```
Line 1  def countRegionForClosestIntervals(regions,intervals):
   -        midPoints = elementwiseMean(intervals.start,intervals.end)
   -        return countRegionsForClosestPoints(regions,midPoints)
```

### 4.1.12 Computing the Distiance Between a Point and its Closest Region

It's often of high intrest to be able to compute the shortest distance between the track elements of two annotation tracks. This problem might be further decomposed as two distinct, but related, problems:

1. Which corresponding track element is closest
2. what's the extact distance.

We've already discussed a solution to something similar to the first sub-problem in 4.1.8 where we were able able to compute how many points each track element is closest to. Since both the track elements and the points are non-overlapping and in ascending order we're able to deduct the following facts:

- If the first track element is closest to $N$ points where $N > 0$, then it follows that this must be the first $N$ points. Formally the interval [0,N) of the points sequence.
- If then also the second region is closest to $M$ points where $M > 0$, then it follows this must be the points in the interval $[N, N + M)$

Knowing this we're able to create a table of points and its closest track element by repeating each track element as many times as it has closest points.

Our remaining problem is to compute the distance between the pairwise aligned points and track elements. The shortest distance between a point and a track element is the minimum of the distance from a point to the start of the track element or the distance to the end of the track element. Except of the points inside a track element, where the distance is 0.

Listing 4.11: A function that computes the the distance between a point and its closest region

```
Line 1 def distanceFromPointToClosestRegion(src,dst):
     -     cnt = countRegionForClosestPoints(src, dst)
     -     src = src.toAll(lambda x: x.repeat(cnt))
     -     return distanceBetweenRegionAndPoint(src, dst)
```

Listing 4.12: A function that computes the distance between regions and points elementwise

```
Line 1 def distanceBetweenRegionAndPoint(regions,points):
     -     distanceFromStart = np.abs(regions.start - points)
     -     distanceFromEnd = np.abs(regions.end - points - 1)
     -     inside = np.logical_and(regions.start <= points, regions.end > points)
     5     ans = np.minimum(distanceFromStart,distanceFromEnd)
     -     ans[inside] = 0
     -     return ans
```

### 4.1.13 Filter Points Inside Track Elements

We've previously, in 4.1.9, discussed how to count the number of points inside each track element. A twist to this question is to compute what points are inside track elements. Section 4.1.7 discussed how to count the distribution of points separated by pivot elements. We'll utilize this functionality to count the number of points inside and between track elements and use that information to deduct exactly which points that are within track elements.

Listing 4.13: A function that computes a mask of what points are within any of the supplied track elements

```
Line 1 def pointsInsideRegionMask(regions,points):
     -     pivots = np.sort(np.concatenate((regions.start,regions.end)))
     -     counts = _pointsPerPivotInterval(pivots,points)
     -     mask = np.zeros(len(counts),dtype='bool')
     5     mask[1::2] = True
     -     return mask.repeat(counts)
```

The input data must as usual appear in ascending order and the track elements must not overlap. The second line generates pivot elements from the

start and end positions of the track elements and succeeding line counts the number of points within each pivot interval[2]. Every odd pivot interval (the first, third and so on) tells us how many points there are up to the first track element, between the following track elements and after the last track element. Every even pivot interval tells (the second, fourth and so on) tells us how many points there are inside each track element. The total number of points inside every pivot interval equals the total number of input points. The fourth and fifth line initializes a boolean mask corresponding with the pivot intervals. The mask has the boolean value `true` for every other position, which reflects a pivot interval for the area inside a track element. The every element of the mask is repeated as many times as there are points inside that pivot interval. This is best illustrated by an example. If there are three points inside the first pivot interval then the first mask value of `false` is repeated three times. If the next pivot interval contains two points, then the second mask value, which is `true`, is repeated two times and so on. This example reflects a situation where the first three points occurs before the first track element and the next two are inside the fist track element. Only the last two of the first five points are inside track elements and this is reflected by the mask.

## 4.2 Scaling Chromosome Level Operations Genome-wide

The choice of implementation language, libraries and tools are reflected in the design of the program. This implies that a good design is relative to both the the domain and the tools at hand. The design is thefore not a general principle or methodology.

Software design is therefore the middle tier between method and implementation. The design should consist of as general parts as possible but is ultimately bound by the available tools and details of the domain.

### 4.2.1 Encapsulating Annotation Tracks

Annotation tracks describe empirical properties of genomes and are discussed in 3.1. Annotation tracks consists of multiple track elements. Track elements describe locations within a chromosome. Section 3.1.1 discussed how the chromosome boundaries may be used to group track elements by chromosome type. See figure 4.2 on the next page for an illustration of this

[2]Pivot intervals are explained in detail in 4.1.11 on page 42

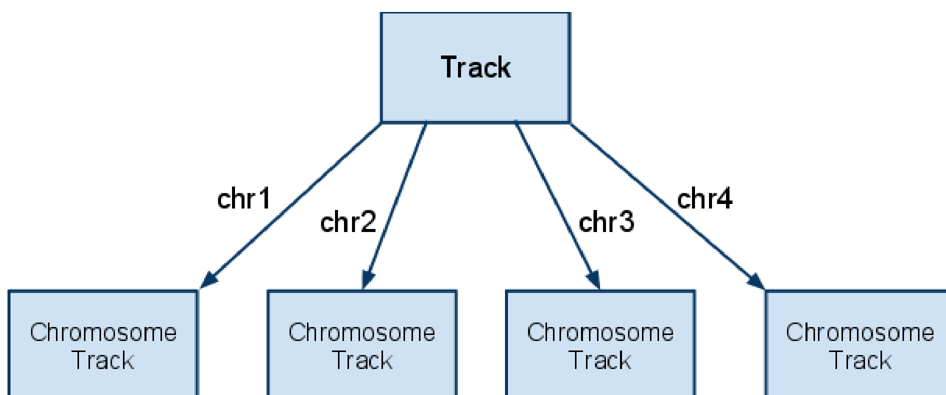



Figure 4.2: Track elements of an annotation track are grouped by the chromosome boundaries.

relationship. This hierarchy also provides a good model to represent annotation tracks using an object-orientated model. We'll refer to the hierarchical object-oriented representation of an annotation track as a Hierarchical Track Representation (HTR) in this text.

### 4.2.2 Organizing Operations on Annotations

There's a huge amount of interesting operations to perform on one or more annotation tracks. It's important to organize and structure the operations properly for the framework to be practically usable. We have classified all operations as one of three distinct types based on the input and output of the operations:

**Transformation** An operation on a single annotation track that produces a new modified annotation track. An example is something as simple as scaling all track elements by a factor.

**Aggregation** An operation on a single annotation track that produces a single, typically numeric, result. An example is computing the total number of positions covered by an annotation track.

**Combinations** Operations on multiple annotation tracks producing a new annotation track as a result. Examples are computing the intersection of two annotation tracks or counting the number of points within each track elements of an annotation track.

The previous section (4.2.1) sketched an intuitive hierarchical representation of an annotation track called a HTR. Comparisons between track

elements are based on location. It's not possible or meaningful to compare locations between different chromosomes and it follows that operations on chromosome tracks may be performed isolated. This is why we've so thoroughly discussed and explored the possibilities of chromosome level operations and methodology in section 3.4 and section 4.1. A chromosome level operations may therefore be applied to all chromosome annotation tracks of a HTR and the partial results are subsequently combined. An operation on multiple annotation tracks, such as an intersection, may similarly operate on tuples of corresponding chromosome annotation tracks and combine the partial results at the end.

### 4.2.3 Representing an Annotation Track

We want to succinctly represent annotation tracks and find a natural way to apply operations to a track representation. Python supports higher order functions and has a dynamic type system. Google's MapReduce frame and its underlying ideas were introduced in 2.2.7. Both `map` and `reduce` are conceptually easy to understand, but are still remarkably powerful concepts. The natural hierarchical grouping of track elements into chromosome annotation tracks (3.1.1) mandates an implementation that distributes a chromosome track level operation and collects and combines the results. This makes operations on annotation tracks very suitable for the map and reduce operations. Only two functions are needed to implement a genome wide operation:

1. One function that computes on a chromosome level.
2. One function to reduce (combine) the partial results.

The instances of the `Track` class in listing 4.14 encapsulates annotation tracks and supports operations such as transformations and aggregations. The `mapReduce` method of the `Track` class has one required parameter; the function to be mapped across all chromosome annotation tracks. A combination function is optional, if none is supplied then the result is returned as a new instance of the `Track` class. A second optional argument is the `onValues` flag which is by default false. The combine function is by default supplied a hash table with chromosome names as keys and the chromosome level results as values because it's usually important to know which chromosome each chromosome track belongs to. This relationship is not important for aggregation functions and only the values are supplied to the `combine` function when the `onValues` flag is true.

Listing 4.14: The overall layout of the class representing annotation tracks.

```
class Track(object):

    def __init__(self, chrTracks):
        self.d = chrTracks

    def mapReduce(self,f,r=None,onValues=False):
        d = dict()
        for key in self:
            d[key] = f(self[key])
        if not r:
            return Track(d,name=self.name)
        if onValues:
            return r(d.values())
        else:
            return r(d)

    def __iter__(self):
        return Bed.orderedIterKeys(self.d)


    def __getitem__(self,key):
        return self.d[key]
```

Consider how easily the total number of base pairs is computed using `mapReduce` in figure 4.15. The number of base pairs is implemented completely ad-hoc in this example.

Listing 4.15: Computing the total coverage using only the mapReduce function

```
u = Track.fromPath('<path>')
u = TrackOp.removeOverlaps(u)
u.mapReduce(lambda x: (x.end - x.start).sum(), sum, onValues=True)
```

### MapReduce on Multiple Annotation Tracks

The `mapReduce` method of the `Track` class presented in the previous section only performs operations on a single annotation track. This is useful, but we need to extend this to perform operations on multiple annotation tracks. For this purpose a bit more general mapReduce function has been created. It accepts:

- A map function for chromosome level operations.
- A combination function.

- Two or more Units to operate on.

The `mapReduce` function that operates on multiple annotation tracks is shown in listing 4.16. The function is almost identical to the `mapReduce` method in the `Track` class except that it applies the mapped function to corresponding tuples of chromosome annotation tracks. An example of how the `mapReduce` function is used is given in 4.17.

Listing 4.16: A general mapReduce function that operations on multiple annotation tracks.

```
Line 1  def mapReduce(f,r,*ds):
     -      d = {}
     -      for key in commonKeys(ds):
     -          values = [x[key] for x in ds]
     5          d[key] = f(*values)
     -      return r(d)
```

Listing 4.17: The use of mapReduce exemplified by the genome wide application of the chromosome level intersection function from 4.1.4 on two annotation tracks.

```
Line 1  def intersection(first, second):
     -      return fac.mapReduce(lib.intersection, Track, first, second)
```

## 4.3 Organizing Coupled Annotation Tracks

### 4.3.1 Coupled Annotation Tracks

As described in section 3.2, coupled annotations describe connections between genomes. A coupled track element maps to one or more track elements in other species. A coupled annotation track describes connected track elements on a single species. A mapping between two species is represented by the system as two distinct but related coupled annotation tracks. A mapping between three species is represented by three coupled annotation tracks. Such related coupled annotation tracks are said to be connected. This section will explain how connections between coupled annotation tracks are preserved by the system an how this representation may be utilized for quantitative comparisons or projection between the species.

#### The Relation Model and Junction Tables

A data structure for coupled track elements and their connections suitable for array programming and interoperable with the existing functionality is re-

quired. Section 3.5.5 discussed different alternatives and concluded that the relational model's representation of many-to-many relationships has promising qualities that are applicable for array programming. A junction table is used to describe many-to-many relations in the relational model. The concept of a junction table is directly transferable to connecting coupled annotation tracks using only integer arrays and thus applicable for array programming.

### The Tabular Track Representation

Coupled track elements are stored on disk in a format we've called a Tabular Track Representation (TTR). A TTR is a flat and serial chain of coupled track elements, in contrast to the hierarchical representation used by HTR. Coupled track elements in a table are not ordered by chromosome type. Instead they have the promising property that connected coupled track elements are at corresponding positions in their respective tables. The corresponding aligned rows of tables makes them ideal for quantitative comparisons and projection, two important methods for inter-genome comparisons discussed in 3.5.2 and 3.5.1. The table representation is very useful for inter-genome operations and also provides a compact storage of coupled annotation tracks and their connections.

Listing 4.18: The core functionality provided by a table track

```
Line 1  class TableTrack(object):
     -
     -      def __init__(self, chrom, chromStart, chromEnd, strand=None):
     -          # store args ...
     5
     -      def sortBy(self, x, inplace=False):
     -
     -      @property
     -      def sortedIdx(self):
    10          return np.lexsort((self.start, self.chrom))
     -
     -      @classmethod
     -      def fromDict(cls, d):
     -          # ....
    15          return TableTrack(chrom, start, end, strand)
     -
     -      @classmethod
     -      def fromPath(cls, fpath):
     -          return cls(chrom,start, end,strand)
    20
     -      def toDict(self):
```

```
        # ...
        return dict(zip(chrom, values))
```

### A Reversible Hierarchical Representation

The HTR have track elements grouped by chromosome type and this ordering have shown itself very useful for intra-genome operations. Both HTRs and TTRs are just different representations of the same data, except that the TTR expresses a position based mapping to related TTRs. There's implicitly a gap in the methodology since intra-genome(3.4) operations depends upon the HTR and inter-genome comparisons(3.5) depends upon the TTR.

We'll introduce a new representation called Reversible Hierarchical Track Representation (RHTR) to mend this problem. A RHTR is produced from a TTR and is identical to a regular HTR in every aspect. The RHTR is in addition able to reproduce the TTR. This is as we shall see very useful. A result may be computed relative to coupled track elements using intra-genome operation and compared between genomes by transforming the RHTR results to a TTR representation. This conversion must be unambiguous since inter-genome comparisons using TTRs are position based.

This process is quite simple. The track elements within a TTR appears in no specific order, but the position of each track element within the TTR is crucial. The track elements are ordered ascendingly by chromosome type and location. The intermediate table is divided at the chromosome boundaries to multiple chromosome annotation tracks. The only difference between a HTR and a RHTR is that the latter remembers the reverse ordering between the TTR and the intermediate table, called the *reversible index*. This is exactly what makes the RHTR reversible. All chromosome annotation tracks are combined and then the reversible index is applied to restore the original TTR. This is illustrated in figure 4.3 on the next page.

Listing 4.19: The core functionality provided by a reversible track

```
class RevTrack(Track):

    def __init__(self,chrTracks,revIdx):
        super(RevTrack,self).__init__(chrTracks)
        self.revIdx = revIdx

    def mapReduce(self,f,r=None,onValues=False):
        tmp = super(RevTrack,self).mapReduce(f,r,onValues)
        return RevTrack(tmp,self.revIdx)

    @classmethod
```

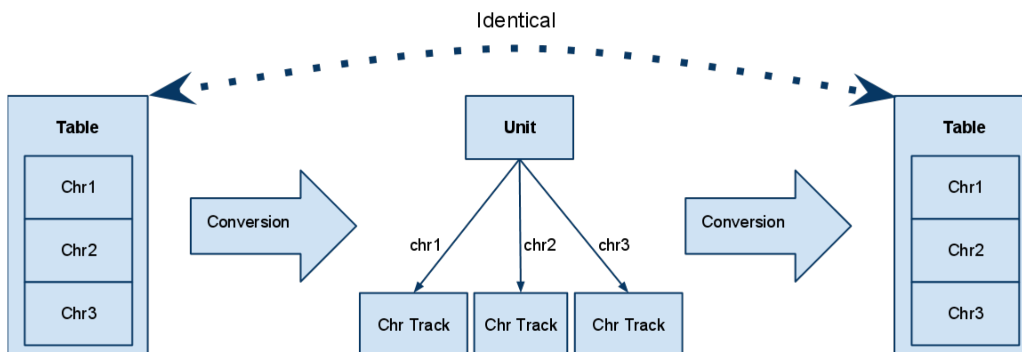



Figure 4.3: An annotation track may be represented by an annotation track table or an annotation track unit. Conversions between the formats are unambiguous..

```
-     def fromTableTrack(cls,regions):
-         sortedIdx = regions.sortedIdx
-         regions = regions.sortBy(sortedIdx)
15        chrTracks = regions.toDict()
-         revIdx = sortedIdx.argsort(kind='mergesort')
-         return cls(chrTracks,revIdx)
-
-     def toTableTrack(self):
20        ga = a.TableTrack.fromDict(self)
-         return ga.sortBy(self.revIdx)
```

## 4.4 Operations on Coupled Annotation Tracks

Operations on track elements from coupled annotation tracks have some restrictions that are irrelevant for normal annotation tracks. Coupled track elements may, unlike regular track elements, overlap. Section 4.3 discussed how a coupled annotation track have multiple representations. A RHTR was almost identical to a normal HTR which is used for normal intra-genome operations, but with the additional ability to convert itself to a TTR which is the preferred representation for inter-genome comparisons. The result of an operation on a RHTR may be converted to a TTR and used for a inter-genome quantitative comparison if the following is true:

1. The computed results must be relative to each coupled track elements. This means that there must be as many partial results as coupled track elements.

2. The operation must produce correct and consistent results even if some coupled track elements overlap.

Most of the previously defined operations in 4.1 break one or both of the requirements above. To see why these requirements are important, it's important to remember the purpose of operations on coupled annotation tracks. The goal is to compute results relative to each coupled track element for later inter-genome comparisons. Set operations such as union or intersection are therefore not sensible since they produce a new annotation track with no direct transformation to the original reference track. They break the second requirement in other words. Other operations such as connecting closest features are more problematic. The current algorithm, presented in 4.1.11, do not allow for overlapping track elements within a single annotation track. The procedure would still be problematic even with an algorithm that supported this. What if two coupled track elements occupied the exact same location which was also closest to a track element from the other annotation track ? If both were seen as closest to the track element, then the final total sum of closest track elements would be greater than the actual amount. This is inconsistent and would break other operations that depend on this operation.

### 4.4.1 Count Overlapping Track Elements

A very useful operation that condones with the above requirements is an overlap count. Given two annotations track, we're interested in how many track elements from the second annotation track that overlaps with each track element from the first annotation track. The number of overlapping track elements is unaffected by internal overlaps. We'll look at the implementation of such an algorithm that must support overlapping track elements.

Section 3.5.4 discussed the abstract methodology and concluded that the computation should be performed in two steps. The first step computes the number of active regular track elements when a reference track element starts. The second step counted the number of regular track elements that started within each reference track element. The actual implementation is available in listing 4.20. The first six lines are initialization code. They extract the positions of all transition start and end events for both chromosome annotation tracks. Line 9-12 count the number of regular track elements active at the start of each reference track element. Line 13-15 computes the number of regular track elements that begins within each reference track element.

Listing 4.20: Find number of overlaps when internal annotation tracks may overlap

```
def hasOverlap(x, y):
    positionArrays = (x.end,y.end,x.start,y.start)
    splitIdx = np.cumsum(map(len,positionArrays[:-1]))
    totallength = sum(map(len,positionArrays))
    orderedIdx = lib._indicesIfSorted(*positionArrays)
    ixe,iye,ixs,iys = np.array_split(orderedIdx,splitIdx)

    flags = np.zeros(totallength,dtype='int64')
    flags[iys] = 1
    flags[iye] = -1
    openAtStart = flags.cumsum()[ixs]
    flags[iye] = 0
    startEndIntervals = lib.intertwine(ixs,ixe)
    openInside = np.add.reduceat(flags,startEndIntervals)[::2]
    ans = openAtStart + openInside
    assert len(ans) == len(x)
    return ans
```

# Chapter 5

# Use Cases

## 5.1 Five-Vertebrate Article

A recent paper[22], published in Science, identifies the genome-wide binding of two transcription factors in the livers of five vertebrates: human, mouse, dog, short tailed opossum and chicken. The two transcription factors were *CCAAT/enhancer-binding protein alpha*(CEBPA) and *hepatocyte nuclear factor 4 alpha*(HNF4A). They were selected because they are strongly conserved and the DNA binding domains of each factor's orthologs are nearly identical. Each transcription factor bound between 16,000 to 30,000 locations in each mammalian genome. For both factors, less than a quarter of bound regions were within 3 kb of known transcription start sites (TSSs). Binding events appeared to be shared between 10% to 22% of the time between mammals from any two of the three placental lineages[1]. This result reveals a rapid rate of evolution in transcriptional regulation among closely related vertebrates. Nevertheless, the number of CEBPA and HNF4A TF binding events shared between any two of the five study species is far greater than could have occurred by chance.

This section will recreate two analyses presented in the paper. Both analysis investigate relations in the distribution of the experimentally determined transcription binding sites published with the article. The authors of the paper have used a 12-way multispecies alignment from ensemble[11] to identify conserved regions among the investigated species.

We have not been able to acquire the same multispecies alignment, instead we have used a 43-way multispecies alignment from UCSC. The methods and

[1] human,mouse,dog

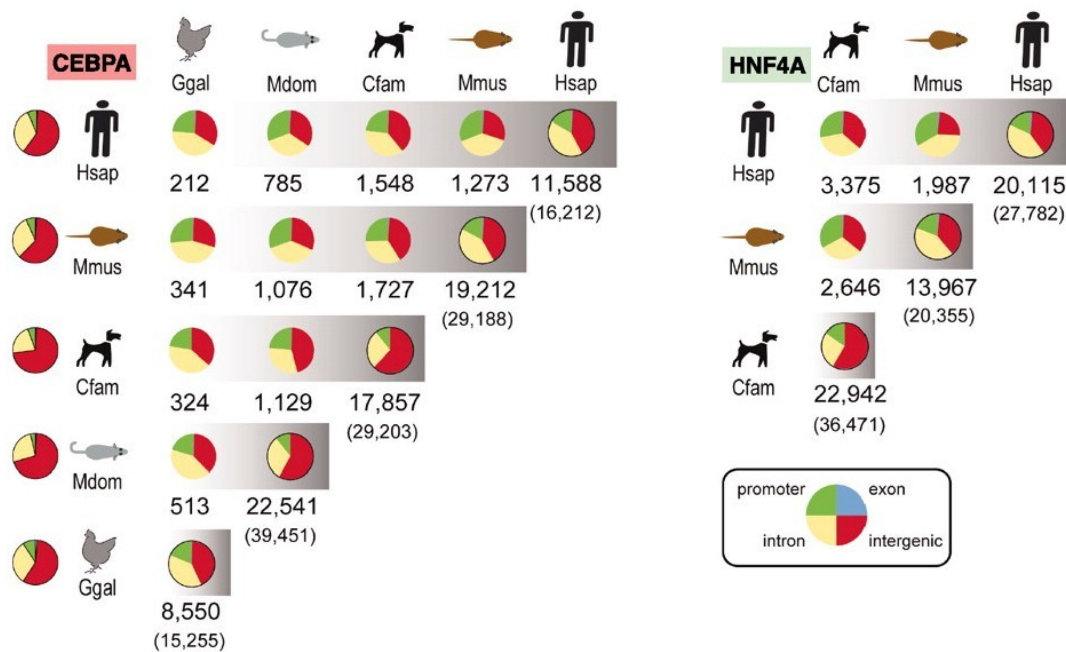



Figure 5.1: Figure from paper[22]. Conservation and divergence of TF binding. The pairwise distribution and numbers of binding events are shown as a pie chart distributed into the following segments: intergenic (red), intronic (yellow), exonic (blue), and promoter (TSS T3 kb) (green) regions. The leftmost column contains the distributions of the bulk genomes. The rightmost pie chart represents all binding events in each species, with the total number of alignable peaks above the total peaks (in parentheses).

procedures used in the original analyses are only generally explained, so we had to decide for ourselves how to resolve much of the ambiguity.

Our recreation of the analyses should ideally be performed using the exact same data sources and procedure as the original. The purpose of recreating the analysis is, however, not to verify the results from the paper. The purpose is to demonstrate how the framework is adaptable to solve relevant and specific problems related to annotation analysis.

### 5.1.1 Conservation and divergence of TF Binding

The results from one of the more interesting analysis presented by the paper is redistributed for reference in figure 5.1. Two identical studies have been made for each of the two transcription factors. The leftmost part of figure 5.1 shows results for the CEBPA TFBSs and the rightmost part shows results for the HNF4A TFBSs. The analysis starts by dividing the genome into four

different segments;

1. intergenic regions
2. exonic regions
3. intronic regions
4. promotor regions

The pie charts in the leftmost column of the table in figure 5.1 depicts the total distribution of segments for every studied genome. The distribution is quite similar in all species and we notice for example that most nucleotides are in intergenic regions. The second column identifies the reference species for each row. The first row has the human genome as reference, the second row has the mouse genome as reference and so on. The rightmost pie chart depicts the distribution of all TFBSs in the reference genome. The intermediate pie charts shows, after our best understanding and interpretation, the number and distribution of TFBSs within genomic regions conserved against the species identified at the top of each column. We will recreate the analysis for the CEBPA TFBS with the human genome as reference. This corresponds to the first row of the CEBPA (the leftmost) table.

### Data Collection

To reconstruct this analysis we'll need a segmentation of the different genomes and the TFBSs published by the authors. It's not documented from where or how the authors divided the genome into different segments. We used the Hyperbrowser to extract what we think is a similar segmentation of the genome, except that we have not defined any promoter regions. The locations of the TFBSs, however, are an important and published finding from the study[2].

### Procedure

Every pie chart is computed from a reference genome and the distribution of TFBSs within genomic regions conserved against a target genome is computed. The analysis can be divided, in our interpretation of the problem, into the following simple operations. For every reference genome and regions conserved against a target genome pair :

- Filter TFBSs within conserved regions.
- Count the segment type for every selected TFBS.

[2] `http://www.ebi.ac.uk/~benoit/cebpa_science2010/`

### Initial Processing

The multispecies alignment used to locate conserved regions is stored on disk as a coupled annotation track. Track elements in a coupled annotation track may, as discussed in section 3.2, overlap. We are in this analysis not interested in the mapping information coupled annotation tracks contains. All the information we need is to know whether or not a given position is conserved against a specific species. It therefore simplifies further processing to transform the coupled annotation track to a normal annotation track without overlaps. We discussed in section 4.1.5 how to remove overlaps from an annotation track and we see the functionality applied in listing 5.1.

Listing 5.1: Removing overlaps and transforming a coupled annotation track to a regular annotation track.

```
Line 1  cons = conserved[species].mapReduce(lib.removeOverlaps)
```

There's a certain ambiguity involved with deciding if a TFBS is inside another track element as the TFBS may span multiple other track elements. This question is valid when deciding if a TFBS is inside a conserved region, but even more valid when deciding what segment a TFBS covers. The resulting distribution would be hard to interpret if a TFBS can be counted as inside multiple segments, but similar problems will arise if decide not to count the ambiguously located TFBS at all. We've decided to simplify the analysis by considering a TFBS to be inside another track element if the midpoint of the TFBS is inside. Listing 5.2 shows the preliminary step of extracting the midpoints from the TFBS track elements.

Listing 5.2: .

```
Line 1  TFBSPoints = TrackOp.asMidPoints(data['tfbs'])
```

### Filter TFBSs within conserved regions

Subsequently, we want to select the subset of those TFBSs that lie within conserved regions. Section 4.1.13 discussed on a chromosome track level how to determine what points are within track elements and introduced the `pointsInsideRegionMask` function. The function accepts a chromosome annotation track and a list of points and returns a boolean array whose values are true when the corresponding point is inside any of the track elements from the chromosome annotation track. Listing 5.3 uses `mapReduce` to apply the `pointsInsideRegionMask` function genome-wide and then uses the returned mask to select and return only those points within track elements.

Listing 5.3: Select subset of points that lie within the boundaries a track element of the regions annotation track.

```
Line 1  def extractPointsWithinRegions(regions,points):
   -        maskTrack = fac.mapReduce(lib.pointsInsideRegionMask,
   -                                  Track.Track,regions,points)
   -        return fac.mapReduce(lambda points,mask: points[mask],
   5                             Track.Track,points,maskTrack)
```

### Count distribution of TFBSs

The three segment types combined covers the whole genome without overlaps. Each segment covers multiple locations and is represented by it's own annotation track. We discussed how to count the number of points within each track element of an annotation track in 4.1.9. Finally, we count the total number of TFBSs per segment by summing the counts per track element. The code is shown in 5.4.

Listing 5.4: Counts the total number of TFBSs inside each segment type.

```
Line 1  for regionName in regions:
   -        perRegion = TrackOp.countPointsPerRegion(regions[regionName], tfbs)
   -        count = TrackOp.sumValues(perRegion)
   -        displayResults(regionName, count)
```

### Results

The results obtained from the described analysis are listed in table 5.1. Our results are significantly higher than the comparable results published by the paper (figure 5.1). This is partly because we have used the published data source they call 'total peaks' in the caption of figure 5.1 and they have used a processed data set they call 'total alignable peaks' which is 2/3 the size. We have done extensive testing and debugging of every aspect of this analysis without discovering any deviations or bugs. We have also compared the results from our operations with the results of external tools where overlapping functionality exists in either the Hyperbrowser or the BEDTools package. Much of the difference in the computed results might be a consequence of the different multiple alignment used as a basis of the analysis. The figure is only briefly explained in the article and we therefore question if we might slightly have misinterpreted the exact analysis. Nevertheless, this attempt at recreating the analysis show how the framework is well-suited for solving real world problems.

| Species | Intronic Regions | Intergenic Regions | Exonic Regions |
|---|---|---|---|
| Total | 7098 | 8519 | 594 |
| Ggal | 279 | 249 | 197 |
| Cfam | 5445 | 6099 | 522 |
| Mmus | 4432 | 4839 | 531 |
| Mdom | 1473 | 1514 | 393 |

Table 5.1: Computed distribution of the CEBPA TFBS in the human genome for regions conserved against the species in the first column.

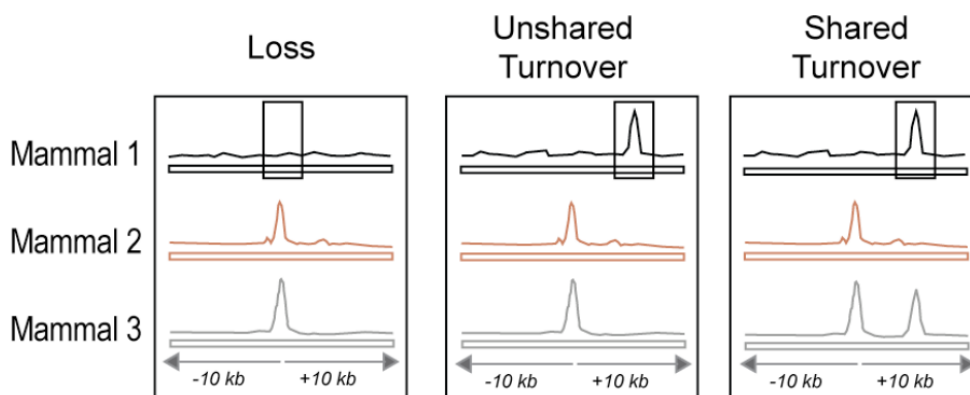
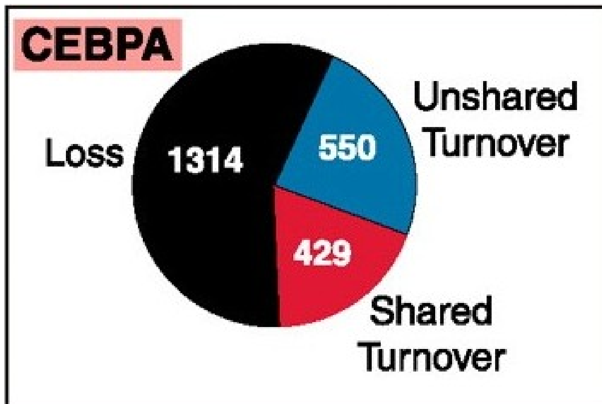



Figure 5.2: Figure S16A from paper[22]. The regions where a binding event was absent in a lineage-specific manner were collected for human, mouse and dog for both CEBPA and HNF4A. Binding events lost in one mammal that were shared in the other two mammals were identified, and the regions 10 kb either side of the peak center were inspected for nearby peaks.

### 5.1.2 Classifying and Counting Turnovers

An interesting analysis presented in the article explored the evolutionary divergence of transcription factor binding sites. Regions conserved between human, mouse and dog were identified. The analysis identify and inspect those conserved regions where one binding event was missing in one lineage, but present in the two complementary lineages. It is considered a *loss* if no supplementary TFBS is found in the vicinity of the conserved region for the species lacking a TFBS. If a supplementary TFBS is found, then it's considered to be a *turnover*. If a supplementary TFBS is also found in the vicinity of the corresponding regions in any of the two other species then the turnover is considered a *shared turnover*. If no additional supplementary TFBS is found, then the turnover is considered to be an *unsheared turnover*. These terms are illustrated in figure 5.2.



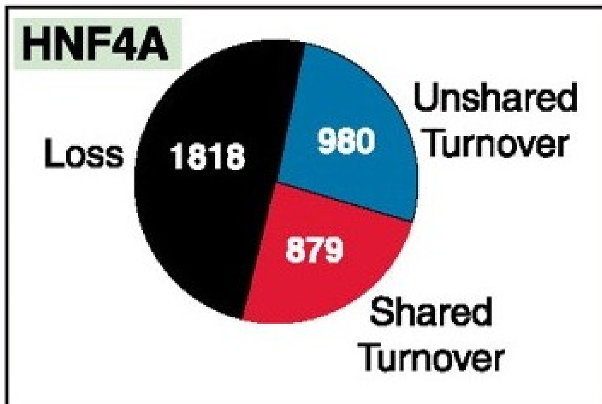




Figure 5.3: Figure from 4B paper[22]. Turnovers occurred near lineage-specific lost binding events approximately half the time; shared turnovers represent cases where a cluster of binding events likely occurred in a common ancestor.

The article[22] made the following conclusions based on the results in figure 5.3 on the previous page:

> Approximately half of lineage-specific losses of TF binding showed evidence of nearby compensatory binding events. A quarter of species-specific losses had a nearby gained binding event that is unique to the same lineage (unshared turnover), and an additional quarter of the losses had a nearby binding event that is shared in one or more other species (shared turnover). The latter case suggests the existence of a cluster of binding events in the common ancestor.

It's reasonable to assume that TFBSs present in conserved regions in all three lineages originates from the common ancestor and has been preserved since speciation. It's also reasonable to assume that no TFBS was present in the common ancestor for conserved regions completely devoid of TFBSs. The fact that two of the conserved regions have an experimentally determined binding site makes it plausible to assume that the binding site was present in a common ancestor of the species and later lost in one lineage. The authors assumed any additional tfbs in the vicinity[3] of the conserved region as a compensatory tfbs. Binding sites are cis-acting. This means they regulate the expression of nearby genes. It's therefore sensible to assume that two adjacent binding sites for the same transcription factor have overlapping functionality. A compensatory tfbs in the vicinity makes the binding site more redundant, so the loss of one of the binding sites would not necessary have an significant impact on expression as the remaining would still be functional. The article assumes that when only one TFBS is missing, then the TFBS was present in a common ancestor and subsequently lost along one lineage.

### Recreating the Analysis

The data used to recreate this analysis is a multiple sequence alignment and the TFBSs published with the article. This analysis is complex, but may be divided into the following concrete steps:

- Compute the location conserved between the three species from pairwise coupled annotation tracks.
- Count within each genome the number of TFBSs that are inside each coupled track element.

[3]within 10kb in either direction from the conserved region

- Compare the counts between the triplets of connect coupled track elements and only further process those triplets where exactly one coupled track element lacks a TFBS.
- Search after a compensatory TFBSs in the vicinity of the coupled track elements missing a TFBS.
  - A coupled track element that neither cover any TFBSs directly nor has one in the vicinity is considered as a *loss*. See figure 5.2 for an illustration of this concept.
  - A coupled track element that did not cover any TFBSs directly, but where a compensatory TFBS was found in the vicinity are considered as turnovers. There are two types of turnovers:
    * It is considered a *shared turnover* if a compensatory TFBS is found in the vicinity of any of the two complementary coupled track elements.
    * It is considered an *unshared turnover* if no compensatory TFBS is found in the vicinity of any of the two complementary coupled track elements.

### Extracting a 3-way Coupled Annotation Tracks

Connected coupled annotation tracks describe mappings between genomes. They are usually in pairs, but this analysis required us to merge two pairs of connected coupled annotation tracks to create a triplet of connected annotation tracks. The procedure for doing so is much like the projection method, but we'll not go into details here.

### Count TFBS for each coupled annotation track

The initial computational step is to count the number of TFBSs inside each coupled track element for all coupled annotation tracks. The function `countTFBSsPerRevTrack` in listing 5.5 performs this operation by calling the function `countPerTrackElement` in listing 5.6 for every Connected Coupled Track Element (CCTE). A little care must be taken since coupled track elements may overlap. We discussed the function `hasOverlap` in section 4.4.1. The function is able to count overlapping elements on a chromosome level, even if coupled track elements overlap. The function `overlapWithOther` used in listing 5.6 is just a convenience library function that scales `hasOverlap` genome-wide. The third line is interesting as it converts the reversible track to a table track. Reversible tracks and table

tracks are just different representations of a coupled annotation track. This is thoroughly explained in section 4.3, but the important thing to remember is that reversible tracks are suitable for intra-genome comparisons and table tracks are suitable for inter-genome comparisons. We convert the counts per coupled annotation track to a table track representation since we later want to compare the counts for CCTEs.

Listing 5.5: Count TFBSs per coupled track element for all connected coupled annotation tracks.

```
Line 1  def countTFBSsPerRevTrack(revTracks, TFBSs):
   -        ans = {}
   -        for key in revTracks:
   -            ans[key] = countPerTrackElement(revTracks[key], TFBSs[key])
   5        return ans
```

Listing 5.6: Count the number of TFBSs within each coupled track element.

```
Line 1  def countPerTrackElement(revTrack,TFBSs):
   -        res = rOp.overlapWithOther(revTrack,TFBSs)
   -        tableTrack = res.toArray()
   -        return tableTrack
```

### Find CCTEs Where Two Out of Three Have a TFBS

We counted the number of TFBSs overlapping each coupled track element in the previous section. We are able to compare the counts between CCTEs since we converted the counts to a table track format. The function `countRegionsWithValues` in listing 5.7 accepts a tuple of table tracks. The second line combines the table tracks into a multi-dimensional array and the integer values are converted to boolean values. This is because we are only interested if the coupled track element contains at least one TFBS. The last step sums the every row of the boolean array. The result is a count of how many of the CCTEs that overlap at least one TFBS.

Listing 5.7: Count the number of CCTEs overlapping a TFBS

```
Line 1  def countRegionsWithValues(countsPerTrack):
   -        boolMatrix = np.vstack(countsPerTrack).bool('bool')
   -        return boolMatrix.sum(axis=0)
```

We use the functionality implemented in 5.7 to extract the indices of those CCTEs where exactly two out of the three coupled track elements overlapped with a TFBS in the function `findWhereScoreIs` in listing 5.8.

Listing 5.8: Create filter for aligned rows with as many true tuples as requested

```
def findWhereScoreIs(countsPerGenomeDict,score=2):
    alignedCounts = countRegionsWithValues(countsPerGenomeDict.values())
    return np.flatnonzero(alignedCounts == score)
```

### Initial Counts

The functionality explained so far in this section all have a specific and understandable task. Listing 5.9 starts gluing these functions together. The locations where exactly two of the three CCTEs contain a TFBS are extracted in line(3-4). Line(5-8) counts the number of TFBSs within and in the vicinity of every coupled track element. Line(9-11) filter those CCTEs counts where exactly two out of three have a TFBS. At the end of listing 5.9 we are left with two types of TFBS counts for all the CCTEs. The first is the count of TFBSs within every CCTE and the second is the count of TFBSs within and in the vicinity of every CCTE.

Listing 5.9: Counting the number of tfbs before and after expansion of the coupled annotation tracks

```
tfbs = readTFBS()
rTracks = readReversibleTracks()
initialCount = countTFBSPerRevTrack(rTracks,tfbs)
idxof2outof3 = findWhereScoreIs(initialCount,2)
expandedRTracks = {}
for species in rTracks:
    expandedRTracks[species] = expandBothDirections(rTracks[species],5000)
expandCount = countTFBSsPerTracElement(expandedRTracks,tfbs)
for key in initialCount:
    initialCount[key] = initialCount[key][idxof2outof3]
    expandCount[key] = expandCount[key][idxof2outof3]
```

### Computing the Final Counts

The previous section computed all the data needed to compute the final results and we will see how to do that here in listing 5.10. The results are computed as the sum of partial results relative to each species. We therefore start looping over the species in line 4.

The variables `before` and `after` contains the TFBS count within that species coupled track element and in the vicinity of that species coupled track elements. Line 9 find those instances where no TFBS was found within

a coupled track element, but at least one supplementary was found in the vicinity. These instances are turnovers, but we need to count what kind of turnover it is. Line(11-15) checks if any of the complementary species have a supplementary TFBS in the vicinity of corresponding the CCTEs. Those instances where a supplementary TFBS is found are 'shared turnovers' and consequently it's an 'unshared turnover' if none are found. The number of 'loss' is computed in line 14 as the number of coupled track elements where no TFBS is located in the vicinity or in within the element.

Listing 5.10: Computing the number of loss shared and unshared turnovers

```
loss = 0
shared = 0
unshared = 0
for key in initialCount:
    others = set(initialCount.keys()) - set([key])
    before = initialCount[key]
    after = expandCount[key]

    local_turnovers = np.flatnonzero(np.logical_and(before == 0,after > 0))
    is_shared = np.zeros(len(local_turnovers),dtype='bool')
    for other in others:
        otherBefore = initialCount[other][local_turnovers]
        otherAfter  = expandCount[other][local_turnovers]
        is_shared_with_other = otherBefore < otherAfter
        is_shared = np.logical_or(is_shared,is_shared_with_other)
    loss += len(np.flatnonzero(after == 0))
    shared += len(np.flatnonzero(is_shared))
    unshared += len(np.flatnonzero(is_shared == False))
print 'loss:',loss
print 'shared turnovers',shared
print 'unshared turnovers',unshared
```

### Results

This analysis contained many steps, but bear in mind that what was calculated is actually quite complex. The computed results are available in table 5.2 and should be compared with the results published with the article in figure 5.3. We notice that the published result counts are more than twice as high as ours. Our result also have proportionally fewer unshared turnovers.

We have done extensive debugging and testing and we fail to find any errors in our implementations. We have based the analysis on a different multiple alignment, as discussed in section 5.1, than what the authors of the

study used. Many details of the procedure are not properly documented, so the methods used may also slightly differ. The results are not too far off and the purpose of recreating this analysis was to prove the framework capable of such an advanced analysis.

| TFBS Type | Loss | Shared Turnover | Unshared Turnover |
|---|---|---|---|
| CEBPA | 655 | 180 | 94 |
| HNF4A | 796 | 346 | 81 |

Table 5.2: The number of loss, shared turnover and unshared turnover. This result is a recreation of the analysis presented in figure 5.3.

# Chapter 6

# Discussion

## 6.1 Inter-Genome Comparisons

One of the original goals of this thesis was to explore the possibilities for inter-genome comparisons and if possible develop proper methodology for usage and discuss the implications. Two methods for inter-genome comparisons were proposed in 3.5. Initially, we started experimenting with different possibilities for measuring distances across genome boundaries. We later became aware of the then recently published article[22], whose findings are attemptedly recreated in 5.1, which performed comparisons between genomes. The authors used what we call 'quantitative comparisons' for the analysis that counted turnover TFBSs, which is discussed in depth in 5.1.2. The idea of the other method, projecting track elements between genomes, is as far as we know novel. Formalizing both methods and exploring the possibilities and limitations inherent is an important step towards more useful analyses in the future.

### 6.1.1 Distance

The distance between two positions on a single chromosome is defined as the number of base pairs between them. The distance between two positions on different chromosomes on the same genome is on the other hand undefined because of the chromosome boundaries. It's not well understood if it's sensible to define a measure of distance between positions on different genomes. If so, it's possible to compute the shortest path between any two positions using a coupled annotation tracks as transition locations between the genomes. The distance between two positions on different genomes is calculated as the shortest total path from the source position to a coupled track element in the source genome and from the corresponding coupled track element in the

target genome to the target position in the target genome. Another consideration is deciding what should constitute as a legal shortest path. Should it be allowed to utilize multiple transitions between the genomes if that leads to a shorter total path? Having a measure of distance could for example allow us to connect closest track elements on different genomes, compare the distance between orthologous features and a plethora of other usages.

### 6.1.2 Projection

Projection is a method for transferring annotation tracks between genomes. Connected coupled annotation tracks defines legal transition locations between the genomes. Section 3.5.1 used tubes as a metaphor to describe the function of connected coupled track elements in relation to projection. Connected coupled annotation tracks describes a series of these imaginary tubes. A regular annotation track is funneled between genomes through coupled track elements. A projection transfers an annotation track between genomes. The resulting projected annotation track may be compared against normal annotation tracks on the target genome using the normal operations on annotation tracks discussed in depth in 3.4.

#### What is Projected?

Track elements within coupled track elements have an unambiguous projection to the target genome. Section 3.5.4 discussed how coupled track elements could overlap. This makes sense biologically as genomic regions may be duplicated and translocated during evolution. Track elements within multiple coupled track elements are therefore projected through all the coupled track elements. We also need to decide how to handle *secluded track elements*. A secluded track element is in this context a track element which is not covered by any coupled track elements. Imagine a track element equally distant from its two closest coupled track elements. As the coupled track elements are used as transition points between the genomes the isolated track element has to either:

- Be projected through both of the coupled track elements. This will create two copies at different location in the target genome seemingly randomly.
- Be projected through one of the coupled track elements. As the coupled track elements are equally distant, one has to be chosen at random or by some other measure.

- Be left out of the projection due to it's location.

### Implications of not Projecting Secluded Elements

The conservative choice is to consider secluded the track element as unprojectable. To only transfer track elements with an unambiguous projections saves us from making predictions we're perhaps not capable of. To leave out secluded track elements from a projection has some consequences. We remember that coupled track elements are on average a lot closer to genes than what an equally sized random sample of track elements would have been. The unambiguously projectable track elements are therefore not a random subset of the original annotation track. The perhaps most evident usage of a projected annotation track is to compare it against a similar annotation track from of target genome. We have to be very careful when concluding on the results of such a comparisons. This is because we're comparing a non-random subset against a complete annotation track. A possible counter measure is to filter the secluded track elements from the annotation track from the target genome. The remaining subsets are consequently compared. This is sensible since both subsets are non-random samples selected by an equal method. We still have to be careful when drawing conclusions on such a comparison since we're not comparing the original annotation tracks, but non-random subsets from them.

### Implications of Projecting Secluded Elements

The inherent problem with the non-random sampling gives a compelling incentive to estimate the projection of isolated track elements. Even a bad prediction of the projection of secluded track elements could result in a projected annotation track with better properties than the alternative. Fundamental edge cases still have to be resolved. Projecting isolated track elements through all roughly equally distant coupled track elements would skew the distribution of the projected annotation track since secluded track elements would often be doubled. This property favors a random selection of only one coupled track element when a secluded track element have multiple equally good alternatives. The actual ramifications of both approaches should be properly tested before an final decision is made. Irrespective of the method for choosing the coupled track element, the projected location in the target genome has to be estimated as it's not unambiguous. A straight forward solution is to compute and use the offset from the secluded track element to the coupled track element. This is probably a good enough estimate as it distributes projected secluded track elements comparably to the distribution

in the original annotation track. Situations where the offset would project a track element outside of the chromosome boundaries of the target chromosome are unlikely and could be solved by moving the projected track element to the boundary of the chromosome instead. The target and host chromosomes could differ greatly in size and it could be possible that relative offset gives better results. Strict or relative offsets are also considerations that must be experimentally tested before we're able to make an educated choice.

#### In Conclusion

Projection of annotation tracks has many promising properties, but many aspects and details must be properly analyzed before conclusions on the results of a projection can be made. The method for only transfer unambiguously projectable track element is more developed as it's done without guesswork. The properties of the projected annotation track are not yet well understood and may be unfeasible as it's a non-random subset of the original annotation track. The method for an estimated projection of a complete annotation track has unresolved issues but the resulting projected annotation track may have more promising properties. The current framework has only implemented support for unambiguous projection. It is trivially to extend the implementation to support predicted projections of secluded track elements when the methodology has been further developed.

### 6.1.3 Quantitative Comparisons

A quantitative comparison uses coupled annotation tracks to connect genomes. This is similar to the projection method discussed above in section 6.1.2. The methods differ in how the coupled annotation tracks are used to perform comparisons between genomes. The projection method views coupled track elements as legal transition locations whereas the quantitative comparison method views coupled track elements as locations for comparisons. Partial results are computed relative to each coupled track element and compared between connected coupled track elements. A very important point and a considerable advantage to the method is that the relative results may be computed from a much wider area than the size of the coupled track element. This fact was a very important part of the analysis in section 5.1.2 where it was searched after TFBSs in the vicinity of coupled track elements.

### Origin

As previously stated, the idea of what we've called quantitative comparisons originates from an analysis in a recent paper[22] that is recreated in section 5.1.2. The authors explains enough of the procedure for the reader to interpret the presented results, but they to not discuss the subject further. We belive that the procedure is useful for many applications and we have therefore made efforts to formalize the procedure as a general method.

### No Loss in Resolution

The first step in a quantitative comparison is to compute a quantitative partial result relative to every coupled track element. Computation of these relative results are performed using normal annotation analysis operations. Every normal operation is allowed on all desirable annotation tracks of the genome. This means that it is possible to perform meaningful and unambiguous comparisons using results based on secluded track elements.

### Coupled Track Elements Appear in Clusters

Coupled annotation tracks are typically based on a multiple sequence alignment. Functional elements, such as genes, are more preserved during evolution and coupled track elements consequently appear in clusters surrounding these preserved areas of the genome. The surrounding environments of coupled track elements will therefore often overlap and passively contribute to the partial results of multiple coupled track elements. Large portions of the genome will furthermore never participate in any quantitative comparisons because no coupled track elements are nearby. Quantitative comparisons are used to measure the relative environmental difference between connected coupled track elements and the clustering of coupled track elements does not change this initial goal. The non-random distribution of coupled track elements might, however, affect the conclusions we may draw from the results.

### Disadvantages Evenly Distributed Between Genomes

The quantitative comparison method has some clear disadvantages. The best advantage of the method is that these disadvantages are evenly distributed between the genomes that are compared. The previous section discussed how coupled track elements appear in clusters and this fact is equally true for all genomes.

**What to Compare?**

The quantitative partial results computed relative to coupled track element are subsequently compared between connected coupled track elements. The partial results must be something uniformly meaningful to compare between genomes. To compare, for example, a count of occurrences of a specific TFBS is not particularly meaningful because the total amount of binding sites may vary greatly between genomes. It might make more sense to compare proportion ratios, but that also has interpretation issues. A large genome with a proportionally large occurrence of a feature does not have more occurrences per base pair than a smaller genome. It would in this case be wrong to compute partial result computed from a fixed size environment relative to coupled track elements as a ratio. Making the the size of the environment a proportion of the genome size would negate this effect. This is not always desirable because many elements, for example cis-acting, does not affect areas proportional to the size of the genome. The analysis in 5.1.2 negated some of this effect by only considering if a feature was present or not.

## 6.2 Array Programming

Array programming is recognized by its ability to express computations at a high-level of abstraction, allowing one to manipulate and query whole sets of data at once. This makes the programming model very suitable for certain kinds of problems and the resulting algorithms are consequently often very clear and concise. Any algorithm that is expressible as a series of operations on whole sets of data is suitable for array programming. Solutions to numerical problems often require very little effort to be solved by array programming.

There are few papers on array programming in general and we found none that discussed usage unrelated to numerical applications. We were curious and optimistic about using array programming to solve problems related to annotation analysis. We wanted to know, if possible, what the limitations of array programming were? What general properties identified problems solvable by array programming? Even a complete, but documented, failure would have been an interesting finding.

### 6.2.1 Guidelines for Interval Problems

A large part of chapter 3 was devoted to explaining a wide range of methods for solving different problems related to intervals, such as for example computing the intersection between two sets of interval and connecting closest

intervals from different sets. We've tried to distill some general guidelines for solving interval problems using array programming. Some concepts are perhaps applicable to other problem domains.

### Combine Arrays for Complex Comparisons

Operations on multiple interval tracks are always based on comparisons between intervals from different tracks. Comparison between two arrays is only possible as a pair-wise operation with array programming. Pair-wise comparisons alone is not enough for complex comparisons. The apparently contradictory step of combining the arrays allows for more complex comparisons. The combined array must subsequently be sorted, but steps must be taken not to loose information about which array each value of the combined array originated from

### Identifying the Origin of Values in Combined Arrays

At least two methods exists for keeping track of the origin of each value. They have partly overlapping functionality, but there are instances where one is more suitable than the other.

1. Compute the position each element of an original array would have in the combined and sorted array.

2. Create an additional array of flags that identifies the originating array for every element in the combined but not yet sorted array. Sort both the combined and the flag array by the order of the combined array.

### Think Differently About the Original Data

It often critical to think and reason differently about a problem to find a solution with array programming. This is often difficult and the issue is thoroughly discussed in the next section (6.2.2). Thinking about interval start and end positions as transition events between states allowed us in section 3.4.1 to compute the intersection and union of interval tracks quite easily by generating and operating on the state sequence. We were priorly unable to develop an algorithm that simply compared the start and end positions.

### Understand the Possibilities of the Array Programming Model

Array programming is recognized by general scalar operations applied to typically large sets of input data. Most available operations performs a uniform

operation to a whole data set, such as computing an aggregate sum or a pairwise addition of two arrays. The definition of array programming is, however, quite broad and allows for operations that operations that have a surprising level of granularity. The `reduceat` function from the Numpy library applies to an universal function. The function accepts an array to operate on and the indices to reduce between as input. It reduces the universal function on the input array between each pair of the supplied indices. There's a few points worth noting about such a function. The function follows the array programming model as it is just an operation on two arrays. The result of the function could easily have been computed by a series of partial reductions in a loop. Such complex functions exists because the array operations are often highly optimized. The `reduceat` function in Numpy is orders of magnitude faster than the perhaps more readable alternative of applying multiple reductions in a loop.

#### Redundant Work is Occasionally Acceptable

The notable computer scientist Donald Knuth has made the following statement on optimization:

> We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

Most programmers accept this fact, but we still too often have a hard time producing sub-optimal results in situations where it's completely acceptable. This is perhaps even more true for array programming, as the operations are highly optimized. A pure array programming solution with clear redundancy is almost without exception fast enough. The restricted set of allowed operations makes it difficult to create solutions with exponential growth in time or space.

#### Split Problems

Some problems are not directly solvable using array programming. Dividing a problem into multiple sub-problems that are solved isolated is sometimes the only option. This point overlaps with the previous point, but we've found it so useful that we'd like to make it explicit. We applied this guideline in section 3.5.4 where we discussed how to count overlapping intervals.

### 6.2.2 Encapsulation and Reuse

Even with experience on developing methods for solving interval problems using array programming and following the guidelines deduced in 6.2.1 using

array programming it's still remarkably difficult to develop methods for new problems. This often holds true even when the new problem is only slightly different from an previously solved problem.

It was therefore natural to attempted to reduce the complexity involved in solving new problems by creating reusable abstractions. Subsequent problems could then be solved by assembling existing encapsulated functionality. Completely modularized and generic code is the holy grail of general programming. The fact that the domain, interval problems, is highly specific makes such an attempt more achievable. This failed completely, but the process made us realize an important concept of array programming. An algorithm for array programming must consist of a series of array operations due to the high-level of abstraction enforced by all operations. This consequently gives little leverage for molding code to fit the model of a problem. A model that conformes with the restrictions of array programming must therefore be deviced specially for every problem. The high-level of abstraction for the array programming operations and the requirement of a perfectly compliant model explains why algorithms for array programming are known to be very concise and succinct. To encapsulate complex operations will never be of much help since the real difficulty resides in creating a problem of the model suitable for array programming.

## 6.3 On Reproducibility of Analyses

Many present papers in bioinformatics cover a long work chain. Data is produced using advanced sequencing technology. Analyses are made on the data and much effort is put on placing the results in a biological context. A recent article[12] sheds light on the problem of reusability. A research group was only able to reproduce the findings from half of a selection of articles from the highly acclaimed Nature Genetics journal. The authors of the galaxy tool discussed in 2.1.1 offers a system that makes sharing analyses and data easier[10]. Researcher may alternatively publish all of the source code that their analyses were created with along with the data. This rarely happens since some steps are often not automated and a full precise textual explanation would perhaps have been too time consuming to create. It's also very possible that many researches feel uncomfortable releasing their in-house and perhaps quite chaotic code publicly. Using a published framework that performs most of the work will make code sharing and thus reproducibility easier. The framework should ideally handle most of the work and the case specific code orchestrates the application of the different framework components on the input data. This is similar to the graphical workflow editor that Galaxy

proposes, but the users additionally have the full flexibility of a programming language at disposal. Both workflows and frameworks lets the researcher precisely describe an algorithm omitting all the unnecessary details. A shared code base also diminishes the chances of bugs within specific operations. It's clearly a lot simpler to publish a small script utilizing a public framework than bundling everything together. This also enhances the readability for the curious reader, since all unimportant details are abstracted away.

## 6.4 Encapsulation and Typing

The implementation described in this paper allows for complex operations and comparisons between annotation tracks within a single genome and between genomes. The system uses integer arrays to represent track elements, mappings between genomes and both partial and final results. An object-oriented design is used to encapsulate and provide proper abstractions for essential concepts in the system. Object-orientation has properties that helps building systems consisting of decoupled independent modules which decrease the overall complexity of the system.

Even when careful considerations have been made to reduce the interdependence between modules, the system still fails to provide a good data abstraction. Using the framework to perform annotation analyses is as a consequence more complex than initially perceived. A characteristic flaw of the system that we were unable to initially predict is how hard it is to detect and restrict unsafe operations. We've identified the omnipresent usage of integer arrays that encodes various structures and relationships as the root cause of these problems. We'll explore the topic extensively in this section.

Section 2.2.5 introduced the concept of type systems and explained the difference between statically and dynamically typed languages. All values have a type, this is common for both dynamic and static type systems. This fact is often overlooked as it has been omnipresent since the advent of high-level programming languages more than fifty years ago. Low-level programming languages provides little or no abstractions from the computers instruction set architecture. A higher burden is put upon the programmer due to the simplicity of the low-level languages. The memory is for example accessed directly and the programmer must at each access decide how to interpret the contents of memory. It's easy to for example store a floating point number at a memory address and subsequently retrieve and interpret the contents as an integer. Floating point and integer values are represented differently by a computer and the value read will therefore be completely different. It's even simpler to make similar mistakes for more

complex data structures. Especially within a larger program where the value may be stored and subsequently retrieved at different locations. Such problems are non-existent in high-level language as all values have an associated enforced type. The lack of a type system is one of the central concepts that differentiate high and low level languages and make in low-level languages much more error prone. All modern programming languages automatically performs one of the following actions when for example a floating point value is added to a integer value:

1. The programming language detects the type error and aborts either the compilation(static typing) or the execution(dynamic typing).

2. The programming language performs an implicit conversion of the floating point value to an integer value and performs the addition.

The latter is option is used by Python and Java, while the first option is typically used for more strictly typed languages such as ML and Haskell.

When we use integers arrays to encode structure we loose the type safety provided by the language. This has serious implications for modularization and how well abstractions encapsulate implementation details. All operations manipulates or queries one or more integer arrays at some level. Operations could try to check if input conforms to the the operations requirements on input. It may do so by checking for situations that never could occur for legal input. This is very difficult to enforce in practice for at least three reasons:

1. Some illegal input may conform to the same restrictions as legal input. Such instances will never be detected no matter how thorough the checks on input are.

2. Tests on input have to describe everything the input cannot be, whereas types describe only what the input must be. The latter is clearly simpler.

3. Thorough testing is often computationally expensive. This will result in a significant degradation of the execution speed. Many operations are used as both standalone operations or as part of another more complex operation. Tests will consequently be repeated throughout the call chain for complex operations on previously checked and confirmed input.

The most prominent problem with the difficulty of asserting correct input for operations is that this directly affects the users of the framework. Usage

of the framework usually consists of writing a small script that applies a series of operations on a set of input data. It's very conceivable that a user misunderstands or forgets some restrictions of an operation. The user will get no warning or indication of the mistake other than an incorrect result. It's often difficult to know what results to expect prior to an analysis and it may very well be that an erroneous result does not deviate sufficiently from what the correct result is or what the expected actual result was. Many operations on annotation tracks require non-overlapping input and it's easy to imagine a user forgetting this requirement or falsely assume that the supplied function input is non-overlapping. The result may be completely different or just slightly different in small fraction of the result. These problems cannot be abstracted and hidden from the user by a better architecture design as it's a fundamental flaw of the system originating from the use of arrays to encode more complex structures such as annotation tracks. It's as we've seen easy to introduce these bugs. They're hard to detect and debug. This makes it consequently harder to trust the results of analyses without a time consuming and thorough examination of every line of code.

## 6.5 Evaluation of Numpy as a Programming API

*Numpy* is an array programming library for Python and has been used to implement the methodology developed in chapter 3. The implementation is described in chapter 4, this section will discuss the applicability of Numpy as an array programming API. Although Numpy strictly speaking is a library for array programming in Python, but it's perhaps more fair to think of Numpy as an environment for scientific applications. Numpy is a part of the Scipy package with the goal of offering scientist an easy to use environment for numerical computations and visualization of results.

### 6.5.1 Best Effort Approach

The Numpy library is most often used interactively or in quite short scripts for numerical computations. It's target audience are computer literate researchers in often other fields than computer science. Numpy is therefore very forgiving and most functions in the library tries to interpret input very liberally. This works especially well for it's intended usage and enables researcher to spend more time solving actual problems instead of arguing with the programming environment. Bugs that occur in the typical very short scripts are often quickly discovered and corrected.

One generally accepted guideline for API development[3] is to fail fast. Fail Fast means aborting further computation as soon as an error is detected. This avoids error propagation and therefore makes debugging easier. There's a clear tradeoff between failing fast and the usability focus in Numpy since erroneous input may falsely be interpreted as legal input by the library.

This 'best effort' approach proved itself as a source of bugs that were especially difficult to detect. Numerous bugs went undetected even after rigorous testing and first emerged during an analysis on large biological datasets. Detecting and tracing these errors were often difficult due to the size of the input data and the fact that it's often difficult to estimate the expected output of operations.

### 6.5.2 Missing Features

The hardest criticism against Numpy given above in 6.5.1 can largely be explained by the fact that Numpy is intended as an interactive environment more than a stable programming API. Still, not all deficiencies of the library may be explained by the mentioned tradeoff between user friendliness and consistency. An illustrating example of this which caused an incredible hard to find bug is an optional keyword argument for the different sorting functions in the library. The default sorting algorithm used by Numpy is the *unstable* quicksort algorithm. Unstable is this context refers to a property of the algorithm. A stable algorithm maintain the relative order of equal values. The stability of a sorting algorithm is not relevant for the majority of usages, but it's essential where needed. The keyword argument is the name of the preferred algorithm. No error is signaled if the algorithm name is misspelled or unknown. Even worse, Numpy defaults to the unstable quicksort algorithm if the supplied keyword is unknown. The bug mentioned was caused by misspelling 'mergesort' as 'mergsort'.

## 6.6 Trusting the Results

Any system should ideally be bug free, but this is in practice exceedingly difficult as was explained in 2.2.9. It is very important for an analysis tool to produce correct results. It's often difficult to know what results to expect from an analysis on annotations and the plenitude of data which is typically processed makes it even more difficult to verify results by hand. A erroneous analysis produces with luck results that are so far off what had been expected that the results induce further investigations. If incorrect results are assumed to be valid, then researchers will waste time trying to under-

stand the implications of the results and attempt to place the results into a biological context. This is directly harmful as it could result in incorrect assumptions that would misdirect further studies and research efforts.

#### Steps Taken to Assess Correctness

Many steps have been made in an attempt to assure correctness. An extensive test suite exists in addition to the source code of the program. All operations are extensively tested with normal input, edge cases and erroneous input. Much time have been spent on verifying and extending the test suite. Major efforts have in addition been put to bring forth a logical, concise and structured architecture. Those operations whose functionality overlaps with counterparts of either the Hyperbrowser(2.1.4) or the BEDTools package(2.1.2) have been verified by comparing the results from operations on real biological data.

#### Why it's Still Hard

Many of the operations on track elements have no comparable counterpart in any known program or tool. It's especially important to have extensive tests for those operations. It is, however, very hard to foresee all possible scenarios up front. Manual inspection and verification of a small fraction of a real biological data set is tedious, but sometimes useful. Experience have shown that bugs that slip past initial testing first emerge on large data sets. This is because the situation that triggers the bug are almost always extreme edge cases which only occur a small fraction of the time in real data.

## 6.7 Architecture and Organization

The hierarchical division of the genome by chromosomes makes it very natural to represent an annotation track as a collection of multiple chromosome level annotation tracks as discussed in 4.2.1. Another consequence of the division of a genome by the chromosome boundaries is that position based comparisons between track elements on different chromosomes are undefined. This fact simplifies the implementation of operations since they only need to compare or operate on track elements from the same chromosome. A chromosome level operation may therefore be applied to all the chromosome annotation tracks of an annotation track and the combined result must also be the correct result for the genome-wide operation.

The alternative is to implement operations that are applied to all the track elements of an annotation track at once and the individual operations

must be aware of the chromosome boundaries and what constitutes as legal comparisons between track elements. This extra bookkeeping must be done for all operations, is error prone and completely unnecessary. We argue that the hierarchical division and the isolated application of operations on chromosome annotation tracks would be an integral part of any good design of the problem domain.

# Chapter 7

# Conclusion

## 7.1 Summary

The rapid advances in sequencing technology in recent years have generated new genomic data that provide the possibility to gain further insight into biological mechanisms if properly analyzed. There already exist solutions for annotation analysis within a genome, such as the Genomic Hyperbrowser and the Galaxy platform. We have explored possibilities for comparative annotation analysis. We propose two distinct methods for comparing annotation tracks between genomes. Both methods depends on a mapping between genomes, called a coupled annotation track.

**Projection** transfers an annotation track between genomes. Track elements within coupled track elements are unambiguously projectable while secluded track elements are not. It is not yet fully understood if it is better to estimate the projected locations of secluded track elements or only project unambiguously transferable track elements. A projected annotation track has the advantage of no loss in resolution for further analysis. It has, once projected, all the properties of a regular annotation track. All the operations developed for normal annotation analysis are consequently available when comparing a projected annotation track against other annotation tracks on the host genome. The usefulness of the method depends on the precision of the projection.

**Quantitative Comparison** is based on partial results computed locally within each genome relative to coupled track elements and subsequently compared between connected coupled track elements. A partial result is computed from the enclosing environment of a coupled track element, which may be as large as desirable for a specific analysis. The environment may therefore unambiguously incorporate the secluded track

elements that were troublesome for the projection method. Partial results are computed using normal annotation track analysis operations. No ambiguity exists when computing and comparing the partial results.

A prototype framework for comparative annotation analysis have been created to test and verify the usability and practicality of ideas. The framework and all of the operations it offers have been implemented using array programming. The array programming model have many promising properties for systems that operates on large quantities of data. We found it especially interesting to evaluate the applicability of array programming to problems that were not strictly numerical as we found no documented prior attempts to do so.

Array programming algorithms for a range of useful operations on annotation tracks were developed as part of this thesis. We were positively surprised by the variety in problems we were able to solve using array programming. We were furthermore able to devise some useful guidelines that eased the process of developing additional algorithms to new problems. We did, however, find a major limitation with array programming. Every algorithm has a model of a problem space and a series of operations that solves a problem according to the model. The amount of possible operations in array programming are restricted. The only possibility for solving a problem is therefore to adjust the model of the problem space until it can be solved by the restricted set array programming operations. A consequence of this is that similar problems often require very different models to be solvable using array programming. The process of modeling a problem suitable for array programming is very hard and usually requires steps that initially seems illogical. This is a trade-off because the resulting algorithm, once the model is properly explained and understood, is succinct and easy to reason over.

The array programming model operates on arrays of simple data types such as integers, floating point numbers, booleans or fixed length strings. Applications other than purely numerical use complex structures that must be expressed in terms of arrays of these basic types. This implies that most functions in a system accepts and returns arrays with a context dependent interpretation. All type information and security is lost as a consequence of this. The burden is put on the programmer to always remember the context to interpret the arrays in and what kind of restrictions different operations have on input. It is easy without any form of type system to introduce subtle bugs that are hard to detect and locate.

## 7.2 Contributions

- Two formal methods for inter-genome comparisons have been proposed.
- An evaluation of the applicability of array programming for solving non-numeric problems with annotation analysis as a use case.
- Algorithms suitable for array programming has been created for both of the inter-genome comparisons methods.
- A collection of algorithms suitable for array programming that solves a range of operations related annotation analysis have been created and documented.
- A fully functional framework for annotation analysis supporting the discussed methods for comparable annotation analysis has been created implementing all of the array programming algorithms.

## 7.3 Future Work

We feel that we were able to properly evaluate many aspects of the array programming model in general and the usefulness of array programming for annotation analysis as a special case. We only scratched the surface of the possibilities for inter-genomic comparisons of annotation tracks and extensive work remains in this area.

- How can we best estimate the projected locations of secluded track elements?
- What conclusions may be drawn from both comparison methods in a biological context?
- What interesting analyses can be developed using one of the two comparison methods.
- Are there other possibilities for inter-genomic comparisons?
- How is it possible to make the framework accessible for the regular biologist?

# Bibliography


[1] D.M. Beazley. Automated scientific software scripting with SWIG. *Future Generation Computer Systems*, 19(5):599–609, 2003.

[2] D. Blankenberg, G. Von Kuster, N. Coraor, G. Ananda, R. Lazarus, M. Mangan, A. Nekrutenko, and J. Taylor. Galaxy: a web-based genome analysis tool for experimentalists. *Curr. Protoc. Mol. Biol*, 10:1–21, 2010.

[3] J. Bloch. How to design a good API and why it matters. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 506–507. ACM, 2006.

[4] P.J.A. Cock, T. Antao, J.T. Chang, B.A. Chapman, C.J. Cox, A. Dalke, I. Friedberg, T. Hamelryck, F. Kauff, B. Wilczynski, et al. Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics*, 25(11):1422, 2009.

[5] E.F. Codd. Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks. *IBM Research Report RJ*, 599, 1969.

[6] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[7] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of symbolic computation*, 9(3):251–280, 1990.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] B. Giardine, C. Riemer, R.C. Hardison, R. Burhans, L. Elnitski, P. Shah, Y. Zhang, D. Blankenberg, I. Albert, J. Taylor, et al. Galaxy: a platform for interactive large-scale genome analysis. *Genome research*, 15(10):1451, 2005.

[10] J. Goecks, A. Nekrutenko, J. Taylor, et al. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome biology*, 11(8):R86, 2010.

[11] T. Hubbard, D. Andrews, M. Caccamo, G. Cameron, Y. Chen, M. Clamp, L. Clarke, G. Coates, T. Cox, F. Cunningham, et al. Ensembl 2005. *Nucleic Acids Research*, 33(suppl 1):D447, 2005.

[12] JP Ioannidis, D.B. Allison, C.A. Ball, I. Coulibaly, X. Cui, A.C. Culhane, M. Falchi, C. Furlanello, L. Game, G. Jurman, et al. Repeatability of published microarray gene expression analyses. *Nat Genet*, 41(2):149–155, 2009.

[13] D. Karolchik, R. Baertsch, M. Diekhans, T.S. Furey, A. Hinrichs, YT Lu, K.M. Roskin, M. Schwartz, C.W. Sugnet, DJ Thomas, et al. The UCSC genome browser database. *Nucleic acids research*, 31(1):51, 2003.

[14] H.P. Langtangen. *Python scripting for computational science.* Springer Verlag, 2008.

[15] J. Loughry, JI van Hemert, and L. Schoofs. Efficiently enumerating the subsets of a set. *applied-math. org/subset. pdf*, 2000.

[16] P. Mougin and S. Ducasse. Oopal: integrating array programming in object-oriented programming. *ACM SIGPLAN Notices*, 38(11):65–77, 2003.

[17] T.E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, pages 10–20, 2007.

[18] P. Peterson. F2PY: a tool for connecting Fortran and Python programs. *International Journal of Computational Science and Engineering*, 4(4):296–305, 2009.

[19] R. Pike and B.W. Kernighan. Program design in the UNIX environment. *AT&T Bell Laboratories Technical Journal*, 63(8/2):1595–1605, 1984.

[20] A.R. Quinlan and I.M. Hall. BEDTools: a flexible suite of utilities for comparing genomic features. *Bioinformatics*, 26(6):841, 2010.

[21] G.K. Sandve, S. Gundersen, H. Rydbeck, I.K. Glad, L. Holden, M. Holden, K. Liestøl, T. Clancy, E. Ferkingstad, M. Johansen, et al. The Genomic HyperBrowser: inferential genomics at the sequence level. *Genome Biology*, 11(12):R121, 2010.

[22] D. Schmidt, M.D. Wilson, B. Ballester, P.C. Schwalie, G.D. Brown, A. Marshall, C. Kutter, S. Watt, C.P. Martinez-Jimenez, S. Mackay, et al. Five-vertebrate ChIP-seq reveals the evolutionary dynamics of transcription factor binding. *Science*, 328(5981):1036, 2010.

[23] J.E. Stajich, D. Block, K. Boulez, S.E. Brenner, S.A. Chervitz, C. Dagdigian, G. Fuellen, J.G.R. Gilbert, I. Korf, H. Lapp, et al. The Bioperl toolkit: Perl modules for the life sciences. *Genome research*, 12(10):1611, 2002.

[24] L. Stein. Genome annotation: from sequence to biology. *Nature reviews genetics*, 2(7):493–503, 2001.