Bachelor Thesis

Hi-C interaction matrix correction using ICE in Rust

Felix Karg

Examiner: Prof. Dr. Backofen
Advisers: Joachim Wolff, Dr. Mehmet Tekman

Albert-Ludwigs-University Freiburg
Faculty of Engineering
Department of Computer Science
Chair for Bioinformatics

July 10th, 2019

Writing Period

10. 04. 2019 – 10. 07. 2019

Examiner

Prof. Dr. Backofen

Advisers

Joachim Wolff, Dr. Mehmet Tekman

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date	Signature

Abstract

foo bar

Zusammenfassung

German version is only needed for an undergraduate thesis.

(TODO: Schreiben!)

Contents

List of Figures

1 Introduction

(DRAFT: more formal language)

Even though much is known about the one-dimensional structure of our DNA, it also organizes itself in three-dimensional ways. Recently, with the advent of chromosome conformation capture (3C) technologies, it has become possible to gain insight into the three-dimensional structure as well. Gene-regulators have been intensively looked at in the one-dimensional case, but enhancer and promoter also have an influence on their three-dimensional surroundings, making knowledge about the spatial organization relevant.

In this work, data gotten through Hi-C [1] will be used. Hi-C is a method for getting 3D-information of genomes. This is done by 'strapping' together parts of the genome that are close by, cutting apart the genome with restriction enzymes, combining the ends of strapped-together fragments and sequencing them. This will be explained in detail in Section **??**.

However, such technologies tend to suffer from technical (e.g. sequencing, mapping) [2] and biological factors (e.g. distinct chromatin states) [3], making them inherently inaccurate. Biases are unavoidable, in particular, as some regions are more sensible for biotin labeling enrichments (See Section **??** why this is relevant) they will be measured more often when compared to others. PCR artifacts may be one of the reasons [**?**]. Mapping locations may be unclear or not unique, introducing even more sources for possible biases. Sequencing methods have certain biases themselves.

Some of the measured interactions are questionable, it is unclear if these are actual, spatially close points, or if it simply is a technical error or a randomly happened interaction.

However, a basic but strong assumption about the structure of the genome can be made, which is that every location has the same amount of interactions (with other locations) as every other location. The data does not show this due to the several aforementioned inaccuracies. Algorithms such as ICE [4] (Iterative Correction and Eigenvector decomposition, Section 4.2) or KR [5] (Knight-Ruiz, Section 3.2) can be applied to normalize the matrix nonetheless.

1.1 Concrete Task definition

The main goals of this thesis include testing the integration between Rust (a systems programming language recently gaining in popularity, more information in Section **??**) and Python (Section 4.5.1), how easy it is to let both these languages interact, and how it compares to the other two current implementations (ICE in Python and KR in C++, more information can be found in **(DRAFT: section: python ice)** and Section 3.2 respectively). The overall goal however, is to try to implement a more resource efficient version, able to make effective use of parallel computations.

(EXTEND: around half a page to one)

2 Background

Chromatin usually describes different levels of how DNA organizes itself. The well-known double-helix is only the lowest of several structural layers. Looking at it from the outside (highest structural layer), DNA looks similar to a big ball of wool. With the help of chromosome conformation technologies spatial proximity can be visualized

2.1 Chromosome Conformation Technologies

2.1.1 Common steps

As can be seen in Figure 1, The first steps, crosslinking, digestion, ligation and the reversal of crosslinking, are the same for all 3-C-based methods.

Cross-linking DNA The first step is to cross-link DNA strands that are close to each other spatially (see Figure 1 or Figure 2 for reference). This is done by adding formaldehyde, which connects (links) sufficiently close strands together.

A chromatin cross-link is two entirely different parts of the genome held together by a chemical bond with formaldehyde. This process cannot be specifically controlled, so only 'regions near each other' are connected, but not necessarily all regions that are known to be spatially close.

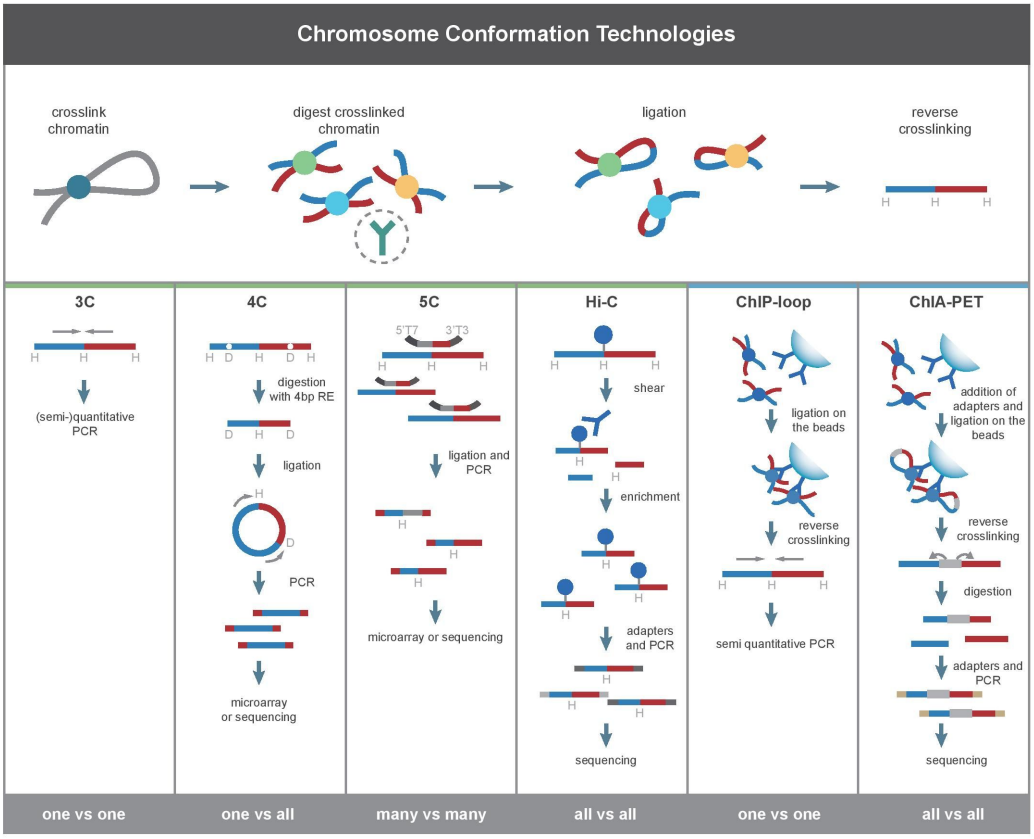


Figure 1: Comparison between 3C and its derived methods.

Image taken from [6].

Digestion The next step is cutting the DNA, currently more similar to a ball of wool, apart in intervals. For this, restriction-enzymes are used (specifically restriction endonuclease). Commonly used enzymes for this are DpnII or HindIII, cutting the genome every 4000 base-pairs [1]. This will result in a lot of cross-linked fragments, as well as not-cross-linked ones.

Ligation After reducing the concentration of fragments, DNA ligase is added, to ligate (weld together) dangling fragment ends. For this a reduction in concentration is done since mostly fragments close together are ligated, it is intend to ligate fragments linked together by formaldehyde.

In HiC, Biotin is added in this step to mark ligated fragments. This allows filtering out most fragments that have not been ligated in a later step.

Reverse Cross-links Adding a high concentration of salt for some time will reverse the cross-linking through formaldehyde, leaving us with our two originally spatially close fragments ligated and with a biotin-marker.

Note that at this point, the fragments are too long to sequence them. They are ligated fragments of around 8000 base-pairs, but most current sequencing methods can only deal with sequence lengths of a few hundred base-pairs at most.

2.1.2 3-C

In 2002 Dekker et al. [7] developed a method test for interactions between a single pair of genomic loci. Candidates for promoter-enhancer interactions can be tested using this method. After the reversal of cross-links (Section 2.1.1), the fragments **(EXTEND: what is (semi-)quantitive PCR?)**

2.1.3 4-C

Chromosome conformation capture-on-chip (4C) was developed in 2006 by Simonis, Zhao et al. [8] [9], a method to test interactions between one genomic loci to all others. This is done by adding a second ligation-step (see Section 2.1.1, Figure 1 for reference) creating loops from the DNA fragments and applying inverse-PCR (method to specifically amplify the unknown parts when beginning and ending parts are known, for this the loops are cut within the known section). Afterwards, this may get sequenced. Due to inverse-PCR knowledge about both interacting chromosomal regions is not needed. Results are highly reproducible for close regions **(DRAFT: find source for this)**.

(EXTEND: What is a microarray?)

2.1.4 5-C

Chromosome conformation capture carbon copy (5C) was developed in 2006 by Dostie et al. [10], this method is able to test a region for interactions with itself, such region being no bigger than a megabase. This is done by adding universal primers to all fragments from such a region. 5-C has relatively low coverage, but is useful to analyse complex interactions of specified loci of interest. Genome-wide interaction measuring would require millions of 5C primers, making this method unsuitable.

2.1.5 Hi-C

Hi-C (as shown by Figure 2) was developed in 2009 by Liebermann-Aiden et al. [1]. After the common steps noted earlier, unique to Hi-C is the following sequence of sonication, pulldown (filtering based on biotin markers) and sequencing.

Sonication Putting the ligated DNA-fragments under the influence of ultrasonic waves is breaking them apart in much shorter fragments (due to long sequences not being able to absorb frequent shocks well), shearing them apart in sequences short enough to enable sequencing.

Filtering and Removal of Biotin 'pulling-down' fragments marked with biotin leaves only those having been marked, and thus ligated, earlier (see Section 2.1.1). Subsequently the marker is removed, as it would hinder further sequencing.

Sequencing Sequencing, short for DNA sequencing, describes processes of measuring a DNA sequence. There are several techniques for doing this, most use PCR (Polymerase Chain Reaction) before or while sequencing, which duplicating fragments several times, allowing them to be sequenced more accurately.

2.1.6 Other methods

Other methods, such as ChIP-loop or ChIA-PET exist, however as the are different from the digestion step onwards, and their subsequent steps are fundamentally different, they will not be covered in depth.

(DRAFT: explain them at least somewhat, reference to some other source)

(EXTEND: placeholder)

2.2 HiCExplorer

HiCExplorer [12] is a collection of tools that are used to process, analyze and visualize Hi-C data. Part of this collection are tools to convert between formats, correcting the data (which this is work is part of), normalizing it, analysing it in various ways, and extensively plotting it. Facilitated is, among othters "the creation of contact matrices, correction of contacts, topologically associating domains (TAD) detection, A/B compartments, merging, reordering or chromosomes, conversion from different formats including cooler and detection of long-range contacts."[1]Those contact matrices may then be visualized, also showing other types of data, including "genes, compartments, ChIP-seq coverage tracks, long range contacts and the visualization of viewpoints"[1]. An excerpt of possible visualizations can be seen in Figure 3.

2.2.1 Analysis

Corrected Hi-C data can then be further analysed, with `hicFindTADs` one can search for TADs (topologically associated domains) [14], for this a TAD-seperation score is computed and local minima indicative of TAD boundaries are searched for. A

[1] `https://github.com/deeptools/HiCExplorer`, accessed 2019-06-26

visualized result of such a computation can be seen in Figure 3I (created with `hicPlotTADs`).

DNA is compartmentalized [1] in different domains, a model for this can be seen in Figure 4. They can be found by computing the Eigenvectors as described in [1] or [4] first by using `hicPCA` and `hicTransform`. With this, a better understanding can be achieved when additionally visualized. Useful metrics include difference, ratio and log2ratio between two matrices. For this, `hicCompareMatrices` can be used. Replications or samples from different conditions can easily be compared when visualized. More information about the analysis capabilities of HiCExplorer can be found in [12].

2.2.2 Visualization

Visualizations are necessary when the goal is to understand complex structures fast or even at all. It is impossible look at rows of numbers and notice that one significantly higher value, which immediately catches the eye as a heatmap.

Basic plotting of contact matrices (as seen in Figure 3F) can be done using `hicPlotMatrix`. Options include region, color and value ranges, as well as plotting A/B compartments or other additional data when added. `hicPlotViewpoint` can visualize the number of interactions around a specific reference region or point in the genome (see Figure 3G).

Computed TADs (as described in Section 2.2.1, see Figure 3I) can be plotted using `hicPlotTADs`. This tool can plot multiple matrices and additional data. Contact matrices are rotated by 45°, and TADs are marked with triangles. The colormap, different ways for visualization, and several configurations to plot coverage tracks can be selected.

More information about plotting with HiCExplorer can be found in [12].

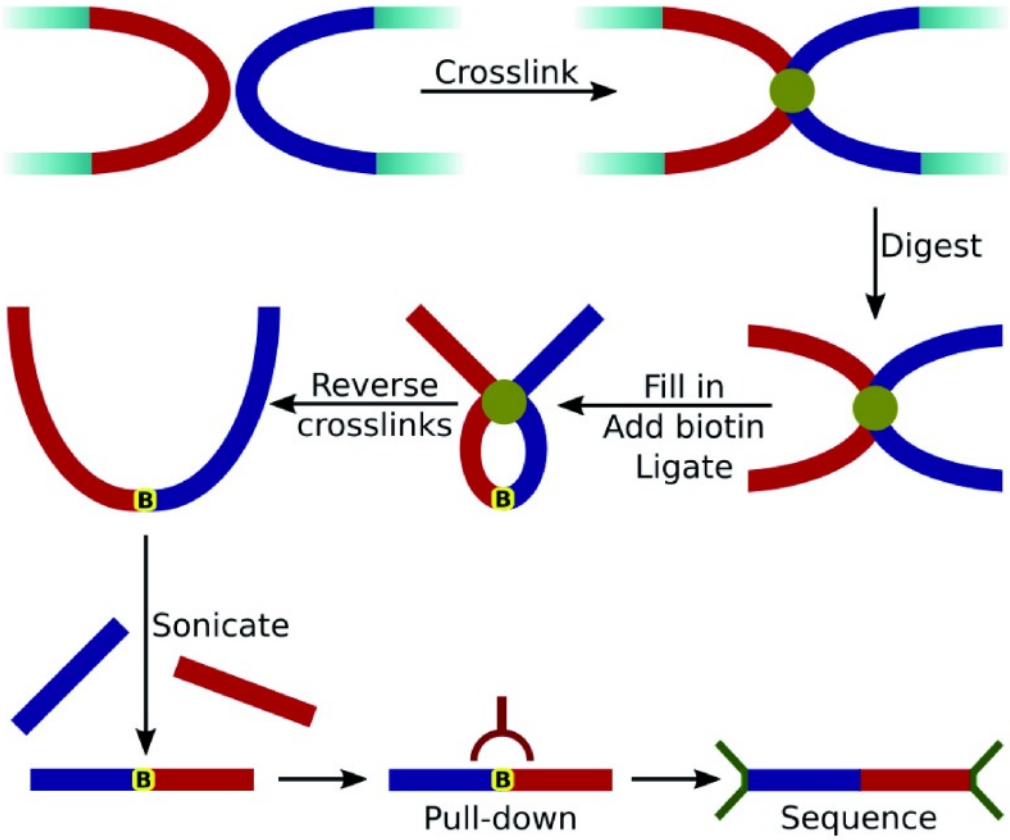


Figure 2: Summarised Hi-C protocol. Biotin is shown by a yellow marker, while the red and blue parts are respectively different parts of cross-linked fragments. The steps are explained in detail in Section 2.1.2.

Image taken from [11].

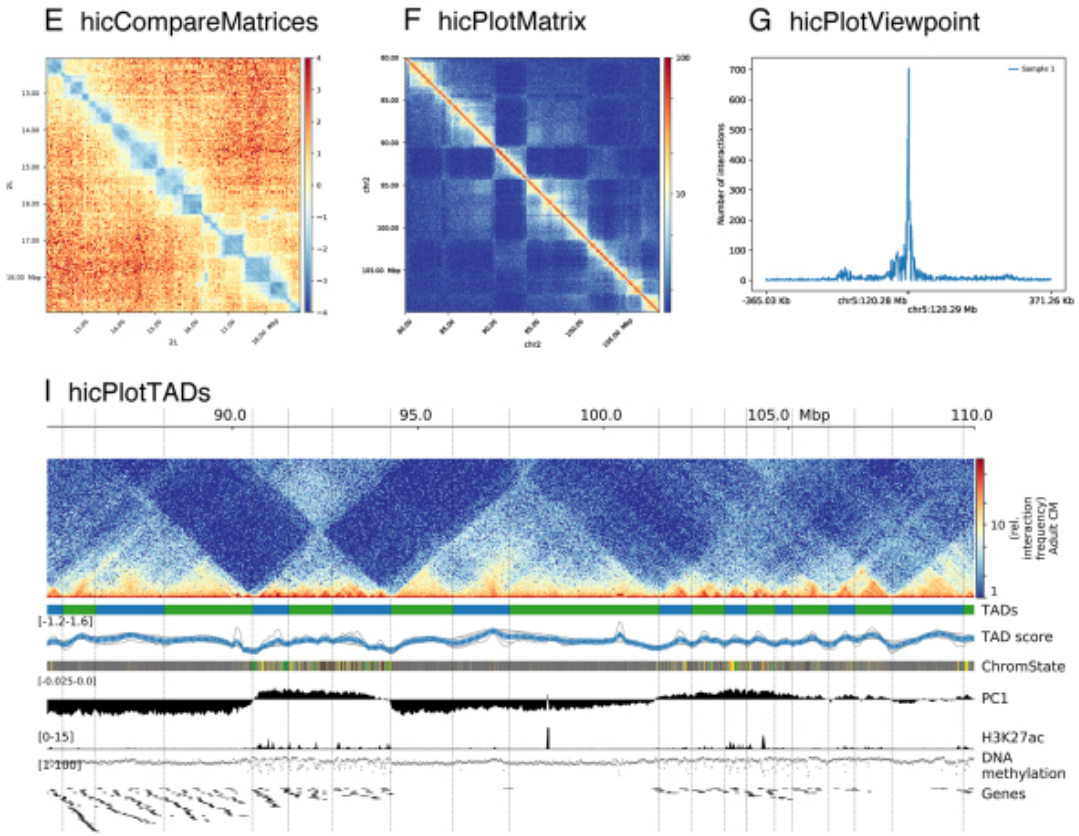


Figure 3: Excerpt of HiCExplorer visualizations E) Pixel difference computed using hicCompareMatrices and visualized using hicPlotMatrix of a Hi-C corrected matrix for wild type condition and knock down. **F)** Plot of a 80 to 105 Mb region contact matrix of chromosome 2 in log scale. **G)** Corrected number of Hi-C contacts shown using hicPlotViewpoint, for a single bin in chromosone 5 (output similar to 4C-seq). **I)** Human chromosome 2 visualization (region 85-110 Mb) using tracks from different tools found in the HiCExplorer toolbox (primarily TAD-related information).

Image taken from [12].

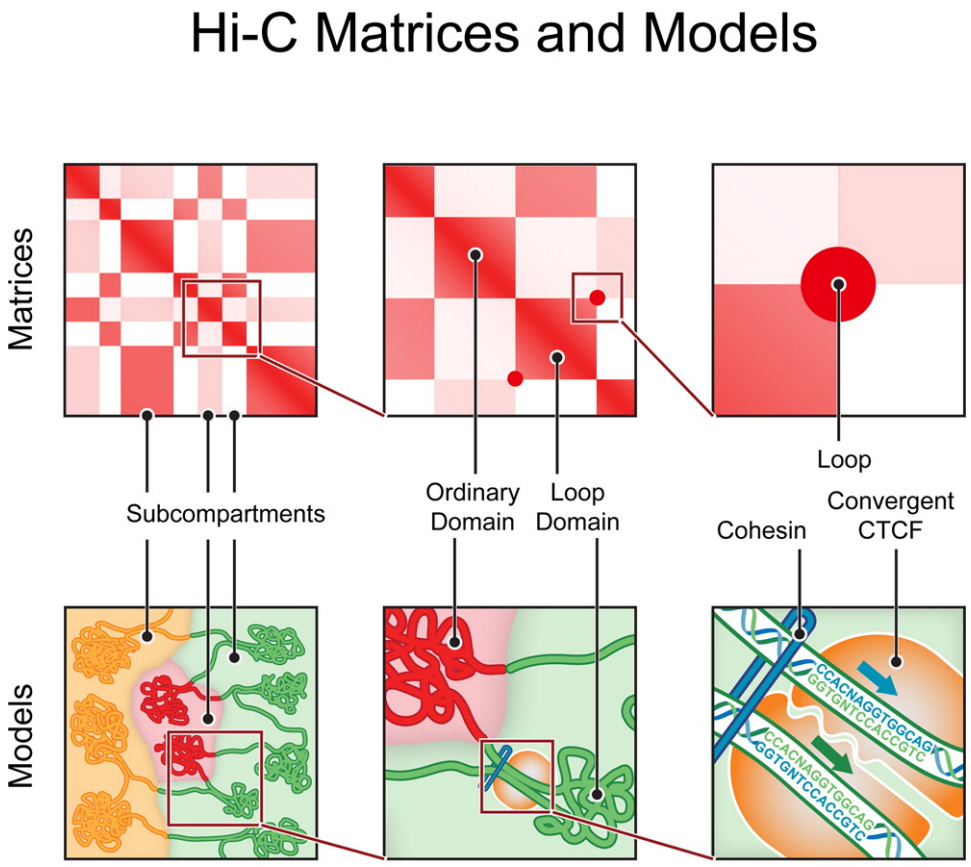


Figure 4: Models relating to HiC-Data.
Image taken from [13].

3 Related Work

Since the main work is the implementation in Rust as well as the testing of an integrating in Python, related work includes the original implementation in Python as well as the recent implementation of the KR-algorithm in C++. Disadvantages and advantages of the respective implementations will be evaluated in the following.

The description of the implemented algorithm can be found in Section 4.2.

3.1 Python implementation

(EXTEND: current state)

(DRAFT: hard to make things actually wrong, fast to implement)

(DRAFT: python having a global interpreter lock for its multiprocessing module, needing to copy the RAM, sometimes numpy is using parallelization, dependent on version)

The original implementation was written in Python, since HiCExplorer is mainly written in Python. This implementation is using common python dependencies extensively, including the Compressed Sparse Row Matrix (CSR Matrix) implementation from `scipy`, as well as the scientific number manipulation library `numpy`.

The implementation itself is considerably short, the file having only 86 Lines, including imports and frequent comments. The advantage of using Python here is showing, as most imports are not for functionality, but rather for timing and logging. With the iteration itself starting no earlier than Line 40, most are High-Level `numpy` / `scipy` commands, some being themselves implemented in C/C++ to be sufficiently fast.

The downsides of this implementation being that datastructures in Python are extensively objectified, meaning they require more working memory, and that even though Python has existing parallelism, a global interpreter lock (python is only interpreted usually, but this still holds for compiling with cpython) prevents multiple threads to use the same parts of memory without duplication. Since Python already has comparatively high memory requirements **(DRAFT: link high memory needs)**, it is not practicable to add the same amount for every further core.

For reference, the Python-implementation can be found here[1].

3.2 KR-algorithm

(DRAFT: describe C++ implementation (the algorithm only superficially), advantages, disadvantages, specific implementation details (C-API, calls from Python to C++, ...))

What follows is a short description to the Algorithm known as Knight-Ruiz from [5], as this algorithm is inherently different from ICE being described later and implemented in both other implementations (the ICE algorithm is described in Section 4.2).

(TODO: add negatives of using C++)

[1] `https://github.com/deeptools/HiCExplorer/blob/master/hicexplorer/iterativeCorrection.py`, accessed 2019-06-26

(EXTEND: total support: at least one nonzero diagonal)

(DRAFT: compare implementations: libraries: eigen + openMP)

(TODO: read paper / algorithm + introduce here)

For reference, the implementation of the KR algorithm can be found here[2].

[2] `https://github.com/deeptools/Knight-Ruiz-Matrix-balancing-algorithm`, accessed 2019-06-26

4 Approach

(TODO: include general setup; HiCExplorer running on Linux/Mac, Python, numpy, scipy, KR in C++, Missing interface from python to C but missing to RUST - done ?)

(DRAFT: Probleme bisher werden jetzt gelöst indem wir Rust verwend)

(TODO: Ausführlich Problemstellung, Rust to Python interface)

4.1 Problem

In the three-dimensional space of a cell the DNA forms a structure that looks close to that of a ball of wool. Obviously, many points of contacts of the DNA wire with itself, called DNA interactions, exist in this "ball of wool" and form structures including DNA loops. However, many of these contacts are random contacts or measurement errors that need to be corrected. A Python implementation exists but is limited for high resolution data due to high memory usage. This [..] project aims to reimplement a more memory efficient method in C++ (which ended up being Rust).

4.2 Iterative Correction and Eigenvector decomposition (Algorithm)

(TODO: put in own words)

The Algorithm as defined in [4] (Supplementary Material):
"We perform iterative correction on the resulting contact maps to obtain biases B_i and 'true' T_{ij} relative contact probabilities by explicitly solving the system of equations:

$$O_{ij} = B_i B_j T_{ij}$$

$$\sum_{i=1,|i-j|>1}^{N} T_{ij} = 1$$

[...]
After the vector of biases is computed, the corrected map of relative contact probabilities is obtained by $T_{ij} = O_{ij}/(B_i B_j)$. Algorithmically, the iterative correction is implemented as follows. We start by creating a working copy of the matrix O_{ij}, denoted W_{ij} as the iterative process gradually changes this matrix to T_{ij}. We initialize the iterative procedure by setting each element of the vector of total biases B to 1. We begin each iteration by calculating the coverage $S_i = \sum_j W_{ij}$. Next, additional biases ΔB_i are calculated by renormalizing S_i to have the unit mean $\Delta B_i = S_i/$ mean (S_i). We then divide W_{ij} by $\Delta B_i \Delta B_j$ for all (i, j) and update the total vector of biases by multiplying by the additional biases. Iterations are repeated until the variance of the additional biases becomes negligible; at this point W_{ij} has converged to T_{ij}."

In between, a couple of other corrections are described. Later, also in the supplementary note from [4], about Eigenvectors:

(TODO: remove part about eigenvector stuff)

(TODO: cite 'requirement of Iterative correction' when introducing Eigenvector stuff)

"**Eigenvector analysis of interchromosomal contact map.**
Eigenvector analysis of a corrected interchromosomal contact map T involves expanding the matrix as a sum of outer products between eigenvectors, E_i^k, weighted by their eigenvalues:

$$T_{ij} = \sum_k \lambda_k E_i^k E_j^k + \langle T \rangle$$

where $\langle T \rangle$ denotes the mean value of the matrix, and the magnitude of the eigenvalue λ_k describes the amount of information captured by the corresponding eigenvector E^k, where k runs from 1 to N. Eigenvectors are then sorted by the absolute value of their eigenvalues, and eigenvectors corresponding to the three largest eigenvalues, E^1, E^2 and E^3, are used for further analysis [..]. Iterative correction is a key prerequisite for eigenvector expansion; performing eigenvector expansion (or principal-component analysis, PCA) on the raw data entangles biases and eigenvectors, making the result nontransparent and bias dependent. Moreover, E^1 is clearly interpretable as the solution to a linear model of chromatin interaction preferences."

4.3 Operation

(TODO: rename: ....)

(TODO: Change Name!!)

4.3.1 Installation

(TODO: for installing using conda add dependencies)

(TODO: Change Name!!) smb can be run on any Unix-based operating system (tested using ubuntu-18.04) with Conda, Python and common development packages installed (e.g. `libopenssl-dev python3-dev build-essential` ...). For the installation itself just enter `conda install -c kargf smb`.

For using, not building, installation of Rust is not needed.

4.3.2 Build

To build the package, assuming you have conda installed, execute the following commands:

```
# first, install rust:
curl https://sh.rustup.rs -sSf | sh -s -- -y

# alternatively install rust with conda:
conda install -c conda-forge rust

# confirm install:
```

```
cargo --version
rustc --version

# download repository and navigate in it
git clone https://github.com/fkarg/HiC-rs
cd HiC-rs

# navigate to the rust code and compile (optional)
cd smb
cargo build
cd ..

# install missing python dependencies
pip install -r requirements.txt

# execute the setup.py (will also compile rust if not done yet)
python setup.py build
```

(TODO: change pip to conda)

(TODO: update packages!!)

4.4 Differences between Rust and Python

Rust and Python are two quite different programming languages, a direct "translation" is not possible. Both implementations are the same semantically, but details differ. Since Rust has a much finer control about memory and the applying of functions to data structures, some operations have been explicitly separated while others have been combined.

The biggest difference, however, is that in Rust certain tasks can easily be parallelized; even after originally writing it for only one core. An example would be:

```
let otherlist = somelist.iter().map(|&v|
                heavy_operation(v)).collect();
```

In this code, some `heavy_operation` is being applied iteratively for every element in `somelist` and later assigned to `otherlist`.

This can easily be parallelized by changing it to the following:

```
use rayon::prelude::*;

let otherlist = somelist.par_iter().map(|&v|
                heavy_operation(v)).collect();
```

The difference here being the imported `rayon::prelude::*` and instead of `iter` now applying `par_iter` to the original list.

4.5 Using Rust

4.5.1 Testing the integration of Rust

(TODO: Answer this question later!!)

One of the main questions for this work was to find out if it is possible to integrate Rust in Python for the HiCExplorer, and evaluate the advantages versus disadvantages.

(TODO: rewrite)

For Rust and Python to interact, there are of course several ways. Those integrating a library written in Rust to allow them to be called from Python will be described in depth in Section 4.6.

(TODO: restructure: Introducing Rust (4.3), Advantages Rust (4.3.1), Disadvantages Rust (4.3.2), Differences Rust Python (4.3.3))

(TODO: move integration of Rust to 4.4)

(TODO: move 'Operation' to 'Using this implementation' in 4.5)

4.5.2 Introducing Rust

(DRAFT: explain more ...)

"Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety" - this was, up until recently, the motto of Rust. It recently got changed into "A language empowering everyone to build reliable and efficient software." At the core of Rust is a compiler applying linear typing, rsulting in the concepts of Ownership and Lifetimes. These concepts, even though notoriously hard to learn, are the main reason the rust compiler can guarantee thread safety, can prevent segfaults, and can run blazingly fast.

(TODO: introduce ownership really)

(TODO: introduce lifetimes really)

(TODO: rewrite on abstract niveau for computer scientists)

Rust does not have a garbage collector, but frees memory the moment it is not needed anymore, which it knows through the Lifetime every variable and reference (pointer) has. Ownership prevents you from modifying data structures in unintended ways, and combined with lifetimes, preventing almost all segfaults. Also, it runs blazingly fast, comparable to C[1], and C++[2].

(DRAFT: introduce awesome tooling)

This is but an introduction to Rust, but the tooling must be mentioned. Rust has a dedicated package manager called `cargo`. Packages are available through `https://crates.io`. Also there are tools like `rustfmt` or `cargo-fix` (a subcommand that can be added later), that format Rust code after predefined guidelines, or fixes most compiler warnings automatically, respectively.

Rust has continuosly claimed StackOverflows position of 'most Loved Language'[3] for the last few years, while both C and C++ rank comparatavely high in the category 'dreaded'.

More information about Rust can be found here[4].

[1] `https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust.html`, accessed 2019-06-26

[2] `https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust-gpp.html`, accessed 2019-06-26

[3] `https://insights.stackoverflow.com/survey/2019#technology-_-most-loved-dreaded-and-wanted-languages`, accessed 2019-06-26

[4] `https://www.rust-lang.org/`, accessed 2019-06-26

4.5.3 Advantages of Rust

The advantage of using Rust over using `numpy` / `scipy` from Python for this work might not be immediately obvious, since the CSRMatrix had to be implemented. We considered using the `numpy` C-API for a while, but its too big to be actually useful for our use-case.

The advantage here is actually much simpler: since we barely need any of the features provided, implementing them ourselves is not much work and gives us way more fine grained control as to what is actually happening.

This includes the parallelization of some parts of the code, which might not have been possible if ther were some other library doing things in its own way (numpy using the C-API would be an example here).

A future advantage is also the modularity of Rust code, meaning in the future additional external libraries (and with them, features) could easily be integrated.

4.5.4 Disadvantages of Rust

The disadvantages follow pretty much directly from looking at the advantages, we do not need much, but we had to implement it ourselves, there is not too much functionality in the case of the CSRMatrix, only the utterly necessary parts. This obviously limits the applicability of this code, no effort has been made to create a generalized solution - several CSRMatrix implementations already exist in Rust, none coming remotely close to the one in `scipy`, but ours is falling short of all the others by a wide margin (at least in most categories).

4.6 Choosing the right API to call Rust from Python

There are three main ways to execute Rust code from Python. In the following, common techniques are investigated.

(DRAFT: versioning is not that relevant)

One common way is rust-cpython. This library requires Rust 1.25 or higher (current versions are 1.33/34/35 for stable/beta/nightly respectively). Rust-cpython grants access to the python gil (global interpreter lock) with which Python code can be evaluated and Python objects modified. The resulting library (directly from compiled rust) can easily be imported into Python (but needs to be renamed). Native Rust code requires some wrapping first, as shown here:

(TODO: make the code ... readable)

```
#[macro_use] extern crate cpython;
use cpython::{PyResult, Python};
// add bindings to the generated python module
// N.B: names: "librust2py" must be the name of
// the `.so` or `.pyd` file
py_module_initializer!(librust2py,
        initlibrust2py, PyInit_librust2py, |py, m| {
    m.add(py, "__doc__",
        "This module is implemented in Rust.")?;
    m.add(py, "sum_as_string",
        py_fn!(py, sum_as_string_py(a: i64, b:i64)))?;
    Ok(())
});
// logic implemented as a normal rust function
```

```
fn sum_as_string(a:i64, b:i64) -> String {
    format!("{}", a + b).to_string()
}
// rust-cpython aware function. All of our python
// interface could be declared in a separate module.
// Note that the py_fn!() macro automatically converts
// the arguments from Python objects to Rust values;
// and the Rust return value back into a Python object.
fn sum_as_string_py(_: Python, a:i64, b:i64)
        -> PyResult<String>
{
    let out = sum_as_string(a, b);
    Ok(out)
}
```

This kind of wrapping, though quite common and based on the Python C-API makes it hard to write idiomatic Code in Rust. Also, since Python is directly affected, the interactions with Python need to be considered while writing Rust-Code. In computer science one does usually not intentionally strive for complexity.

Another common approach is using the pyO3-library, which started off as a fork of rust-cpython, but has since seen quite drastic changes. For example, its using requires at least Rust version '1.30.0-nightly 2018-08-18' (or, in the newest version, '1.34.0-nightly 2019-02-06'). This is due to the usage of unstable features, most of which have recently been able to be promoted to stable. Unstable features are only available in the nightly toolchain. Still missing is Specialisation, which has at the time of writing still a long way to go. The library would also result in an easily importable (needs to be renamed first, still) cdylib (same as rust-cpython). The still

intermingled way of writing the interface (certainly better but not by much compared to rust-cpython) as well as the dependency on unstable nightly rust versions led to the decision of not using it either.

The third way, that is actually been promoted in the official Rust docs, is to generate a dylib and import that in python. No renaming necessary, but the communication between Rust and Python is a bit more low-level. The main wrapper is on the side of Python, transforming Arguments to Pointers and C-Representations, whilst the Rust part needs to conform to C-practices, which includes receiving a list by getting a pointer to the first element and the length of it. Other than that, the Rust code has additional `\#[no_mangle]` and `\#[repr(C)]` (procedural) macros, preventing the compiler to mangle (renaming of functions) and guaranteeing the representation in the memory layout to be as it would be in C. Since like this neither language depends on something only internal (or combinatorial), and both just depend upon the 'common, unchanging' C-interface, this seems to be the preferred way.

(DRAFT: include a nice comparison table)

4.7 General Approach

(TODO: Introduce main approach)

(DRAFT: Steps: read papers, test communication between Python and Rust (API), implementing CSRMatrix, implementing algorithm, fixing bugs (in-memory), making buildable (milksnake, conda), further bug fixing, some testing, building test-suite and a lot of bugfixing, reading papers again, starting to write stuff down.)

4.7.1 Beginning

(TODO: Rewrite to passive voice)

Having read the provided papers ([4], [1] and [11]) I started looking in the Python-implementation. First things first I started testing the feasability of communicating between Rust and Python. The only available way for this is the raw C-API both adhere to.

4.7.2 Feasability Testing

(TODO: Rewrite to passive voice)

Having succeded in calling functions in Rust, and passing the arguments correctly, I started to look in the Python-implementation again. Since in python `numpy` and `scipy` were used quite extensively (especially the Compressed Sparse Row Matrix from `scipy` and available operations through `numpy`) and there was no library available providing functionality similar enough, I implemented the minimal version of a CSRMatrix that would be needed, and tested its functionality.

4.7.3 Implementation of the algorithm

(EXTEND: Answer: could I call numpy from rust)

(TODO: Rewrite to passive voice)

The initial translation from Python to rust happened more or less on a line-by-line basis, as much as this was possible. Seeing the Python-implementation section-wise as a comment I started out with a comparably naive translation from Python to Rust. As I did not have `numpy`/`scipy` available, specific operations had to be done differently, and I needed to care a lot more about the memory (of the variables, also their availability) than the Python-implementation did.

4.7.4 Testing and Bugfixing

(TODO: Rewrite to passive voice)

(DRAFT: give examples for common bugs)

Having succeded at convincing the compiler, I wanted to test my implementation. The compiler in Rust is quite capable, reducing common bugs tremendously. My knowledge about Rust not being on the expert-level, I made the error of not writing back the changes made to the matrix in the matrix (more specifically, the part that should have done that was being handed a immutable matrix). This and some smaller bugs got solved easily, so I set up a small testing environment, even calling from python.

4.7.5 Idiomatic Rust

(TODO: Rewrite to passive voice)

While gradually transforming the naive Python-translation to idiomatic Rust, at some point results ended up being `NaN` pretty fast. A while of debugging later, I reduced it to the following situation:

(DRAFT: exclude mention of debugging)

```
// let list1 = vec![0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0];
// let list2 = vec![0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0];
println!("{}, sum: {}", list1, list1.iter().sum());
println!("{}, sum: {}", list2, list2.iter().sum());
```

Here the output was still “sum: 1” and “sum: 0”. One iteration later however, all the elements have only been multiplied with some factors, their product being `0.16`.

```
// let list1 = vec![0.0, 0.0, 0.0, 0.16000000000003, 0.0, 0.0, 0.0, 0.0];
// let list2 = vec![0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0];
println!("{}, sum: {}", list1, list1.iter().sum());
println!("{}, sum: {}", list2, list2.iter().sum());
```

The expected output here would be “sum: 0.16” and “sum: 0”, or something around that. However, the actual results were “sum: 0.4” and “sum: inf”. As it turns out, the factor they have been subject to was indeed `0.16`, however it was `0.16` with high fraction values. This means that summation of `0.16` and `0.0` (the zero also having high fraction) is being sufficiently inaccurate to not be accurately represented by floating point values. With this happening multiple times, it was unavoidable.

The same happened with the summation of the innocious-looking `0.0`. They had high fractions from the original multiplication by `0.160000000000003`, their continued summation resulting in an overflow. This new number just happens to be one of the representations of `inf`.

4.7.6 Packaging

(TODO: Rewrite to passive voice)

Next was the Packaging of my code. As my work should be used from within the HiCExplorer, My part is supposed to be available as a python-package. The **(DRAFT: only real)** python dependcy (apart from those required for packages) ended up being milksnake, itself a helper for compiling the rust part of my package.

The conda-part was not as easy though, as milksnake was not resolvable there. I ended up porting `milksnake` as a conda package. This turned out to be a nontrivial task, as `conda skeleton pypi milksnake` created a package conda could not build, the issue here being that milksnake was only provided as a `*.zip` file and conda had hardcoded the format `*.tar.gz`.

Additionally I set up a buildserver, adding some tests and fixing smaller bugs.

4.7.7 Parallelizing

(TODO: Rewrite to passive voice)

Nearing the end of my work, I set up ways to test and compare my implementation with the other. One of the last things I did was adding Parallelization.

(TODO: add: compare parallelization in C++ and Python, accessing memory, why it is possible to easily add this, ...)

4.8 Testing

(TODO: Rewrite to passive voice)

I ran a total of 416 different configurations to test for a total of 3 different parameters. The first two are the same for all, the third applies only to this implementation. I tested the original Python-implementation as well as the new KR in C++.

- Size of Matrix (four different ones)
- Number of chromosomes (8 different sizes)
- Number of threads (11 different numbers)

(TODO: add: data is primary data, from: GM12878 something rao2014)

The sizes of the matrices for reference (biggest to smallest):

```
# Matrix information file. Created with HiCExplorer's hicInfo version 3.0
File:   matrix.h5
Size:   309,581
Bin_length:     10000
Sum of matrix:  2416588411.2530212
Chromosomes:    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
                14, 15, 16, 17, 18, 19, 20, 21, 22, X, Y, MT
Non-zero elements:      2,111,867,476
Minimum (non zero):     0.008667398294551536
Maximum:        139544.65657933566
NaN bins:       25948

# Matrix information file. Created with HiCExplorer's hicInfo version 3.0
File:   25kb_raw.h5
Size:   123,841
Bin_length:     25000
Sum of matrix:  2378265786.0
Chromosomes:    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
                14, 15, 16, 17, 18, 19, 20, 21, 22, X, Y, MT
Non-zero elements:      1,530,533,003
Minimum (non zero):     1
Maximum:        320932
NaN bins:       9290
```

```
# Matrix information file. Created with HiCExplorer's hicInfo version 3.0
File:   50kb_raw.h5
Size:   61,928
Bin_length:     50000
Sum of matrix:  2333794628.0
Chromosomes:    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
                14, 15, 16, 17, 18, 19, 20, 21, 22, X, Y, MT
Non-zero elements:      1,053,216,825
Minimum (non zero):     1
Maximum:        320932
NaN bins:       4514

# Matrix information file. Created with HiCExplorer's hicInfo version 3.0
File:   small_test_matrix.h5
Size:   33,754
Bin_length:     5000
Sum of matrix:  35778.0
Chromosomes:    chr2RHet, chr3RHet, chr2LHet, chr4, chrYHet, chr3L, chr2L,
                chrU, chrX, chrXHet, chr2R, chr3R, chrUextra, chrM, chr3LHet
Non-zero elements:      69,213
Minimum (non zero):     1
Maximum:        8
NaN bins:       0
```

(TODO: remove this verbatim part and make table instead)

(EXTEND: building matrix-test-suite and getting results)

(TODO: look for better rust presenter in latex)

(TODO: add the integration of travis more)

5 Experiments

Experiments were run on a Server with the following specs (Specifics here[1]):

Processor	Intel® Xeon® Processor E5 v4 Family
Number	E5-2630V4
Performance	
Number of Cores	10
Number of Threads	20
Base frequency	2.2 GHz
Max Turbo frequency	3.1 GHz
Working Memory (RAM)	120 GByte

(TODO: The time-resource-measures done)

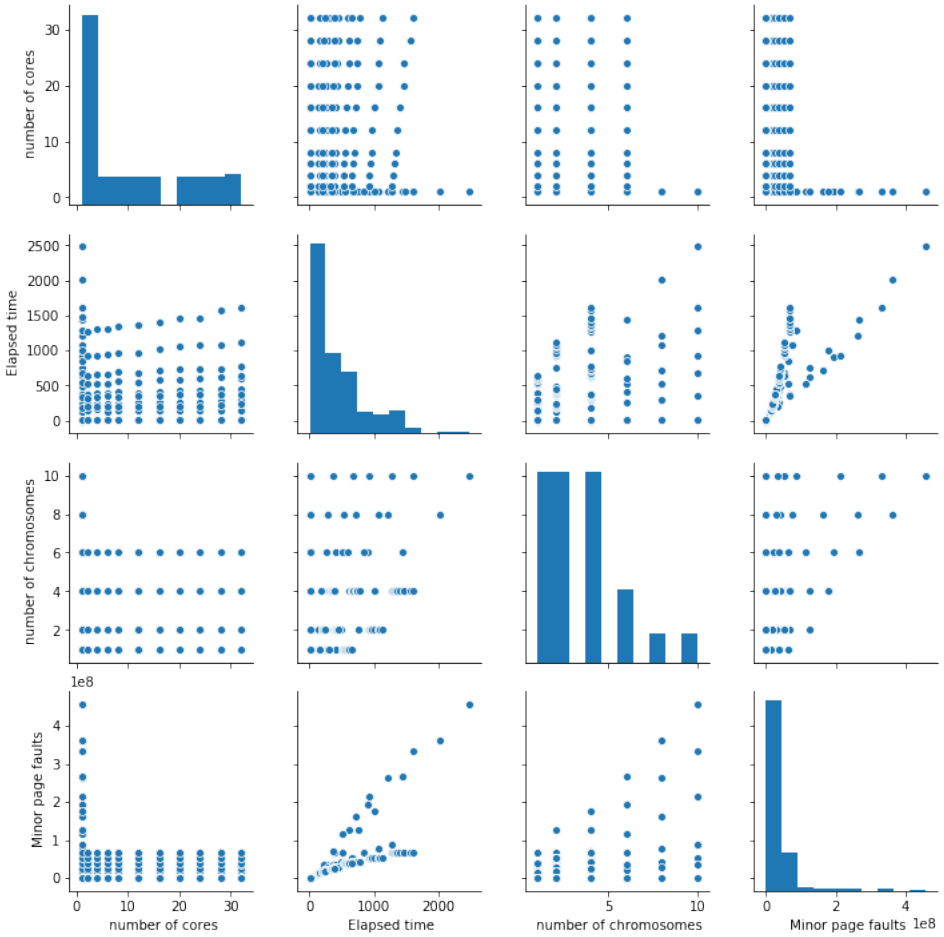
(DRAFT: Something is off with my numbers, they show something entirely different from earlier. I'll check that again.)

See Figure 5 (more of that will follow).

(TODO: add where the matrices / data is from)

(TODO: graphics: runtime vs cores (RUST), RAM vs variants, runtime vs variants)

[1] `https://ark.intel.com/content/www/us/en/ark/products/92981/intel-xeon-processor-e5-2630-v4-25m-cache-2-20-ghz.html`, accessed 2019-06-26



(a) Pairplot over some variables

Figure 5: Pairplot over some of the variables ... moar info here

(TODO: add plot of matrix uncorrected vs corrected ice vs corrected KR vs corrected ice_rust)

6 Conclusion

(TODO: general evaluation; worked better or worse, why, what could be tried / changed, evaluation of rust in this context, ...)

(TODO: clean up repository)

(TODO: remove deepTools entirely)

(TODO: Add Glossary?)

(DRAFT: use more formal language)

(TODO: write poolmanagers for printing)

ToDo Counters

To Dos: 46; 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46

Parts to extend: 8; 1, 2, 3, 4, 5, 6, 7, 8

Draft parts: 20; 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

Bibliography

[1] E. Lieberman-Aiden, N. L. Van Berkum, L. Williams, M. Imakaev, T. Ragoczy, A. Telling, I. Amit, B. R. Lajoie, P. J. Sabo, M. O. Dorschner, *et al.*, "Comprehensive mapping of long-range interactions reveals folding principles of the human genome," *science*, vol. 326, no. 5950, pp. 289–293, 2009.

[2] M. S. Cheung, T. A. Down, I. Latorre, and J. Ahringer, "Systematic bias in high-throughput sequencing data and its correction by BEADS," *Nucleic Acids Res.*, vol. 39, p. e103, Aug 2011. [PubMed Central:PMC3159482] [DOI:10.1093/nar/gkr425] [PubMed:20824077].

[3] L. Teytelman, B. Ozaydin, O. Zill, P. Lefrancois, M. Snyder, J. Rine, and M. B. Eisen, "Impact of chromatin structures on DNA processing for genomic analyses," *PLoS ONE*, vol. 4, p. e6700, Aug 2009. [PubMed Central:PMC2725323] [DOI:10.1371/journal.pone.0006700] [PubMed:12067662].

[4] M. Imakaev, G. Fudenberg, R. P. McCord, N. Naumova, A. Goloborodko, B. R. Lajoie, J. Dekker, and L. A. Mirny, "Iterative correction of hi-c data reveals hallmarks of chromosome organization," *Nature methods*, vol. 9, no. 10, p. 999, 2012.

[5] P. A. Knight and D. Ruiz, "A fast algorithm for matrix balancing," *IMA Journal of Numerical Analysis*, vol. 33, no. 3, pp. 1029–1047, 2013.

[6] G. Li, L. Cai, H. Chang, P. Hong, Q. Zhou, E. V. Kulakova, N. A. Kolchanov, and Y. Ruan, "Chromatin interaction analysis with paired-end tag (chia-pet) sequencing technology and application," *BMC Genomics*, vol. 15, p. S11, Dec 2014.

[7] J. Dekker, K. Rippe, M. Dekker, and N. Kleckner, "Capturing chromosome conformation," *science*, vol. 295, no. 5558, pp. 1306–1311, 2002.

[8] M. Simonis, P. Klous, E. Splinter, Y. Moshkin, R. Willemsen, E. De Wit, B. Van Steensel, and W. De Laat, "Nuclear organization of active and inactive chromatin domains uncovered by chromosome conformation capture–on-chip (4c)," *Nature genetics*, vol. 38, no. 11, p. 1348, 2006.

[9] Z. Zhao, G. Tavoosidana, M. Sjölinder, A. Göndör, P. Mariano, S. Wang, C. Kanduri, M. Lezcano, K. S. Sandhu, U. Singh, *et al.*, "Circular chromosome conformation capture (4c) uncovers extensive networks of epigenetically regulated intra-and interchromosomal interactions," *Nature genetics*, vol. 38, no. 11, p. 1341, 2006.

[10] J. Dostie, T. A. Richmond, R. A. Arnaout, R. R. Selzer, W. L. Lee, T. A. Honan, E. D. Rubio, A. Krumm, J. Lamb, C. Nusbaum, *et al.*, "Chromosome conformation capture carbon copy (5c): a massively parallel solution for mapping interactions between genomic elements," *Genome research*, vol. 16, no. 10, pp. 1299–1309, 2006.

[11] S. Wingett, P. Ewels, M. Furlan-Magaril, T. Nagano, S. Schoenfelder, P. Fraser, and S. Andrews, "Hicup: pipeline for mapping and processing hi-c data," *F1000Research*, vol. 4, 2015.

[12] J. Wolff, V. Bhardwaj, S. Nothjunge, G. Richard, G. Renschler, R. Gilsbach, T. Manke, R. Backofen, F. Ramírez, and B. A. Grüning, "Galaxy hicexplorer: a

web server for reproducible hi-c data analysis, quality control and visualization," *Nucleic acids research*, vol. 46, no. W1, pp. W11–W16, 2018.

[13] S. S. Rao, M. H. Huntley, N. C. Durand, E. K. Stamenova, I. D. Bochkov, J. T. Robinson, A. L. Sanborn, I. Machol, A. D. Omer, E. S. Lander, *et al.*, "A 3d map of the human genome at kilobase resolution reveals principles of chromatin looping," *Cell*, vol. 159, no. 7, pp. 1665–1680, 2014.

[14] F. Ramírez, V. Bhardwaj, L. Arrigoni, K. C. Lam, B. A. Grüning, J. Villaveces, B. Habermann, A. Akhtar, and T. Manke, "High-resolution tads reveal dna sequences underlying genome organization in flies," *Nature communications*, vol. 9, no. 1, p. 189, 2018.