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

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum	Unterschrift

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)

(DRAFT: rewrite. Make more fluent for readers.) (DRAFT: remove "stuff" and (stuff))

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.

(EXTEND: not however so many times!) 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

(TODO: remove ball of wool)

(TODO: Bilder: Image dapted from anstatt Image from ...)

(TODO: Prüfungsamt abklären: Digitale Kopie? -> AJ Müller)

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 Rust.

The main goals of this thesis include testing the integration between Rust (a systems programming language recently gaining in popularity, more information in Section 4.3) and Python (Section 4.3.6), 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 Section 3.1 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.

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 (major chromatin structures are shown in Figure 1). Looking at it from the outside (highest structural layer), DNA looks like 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 2, 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 2 or Figure 3 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.

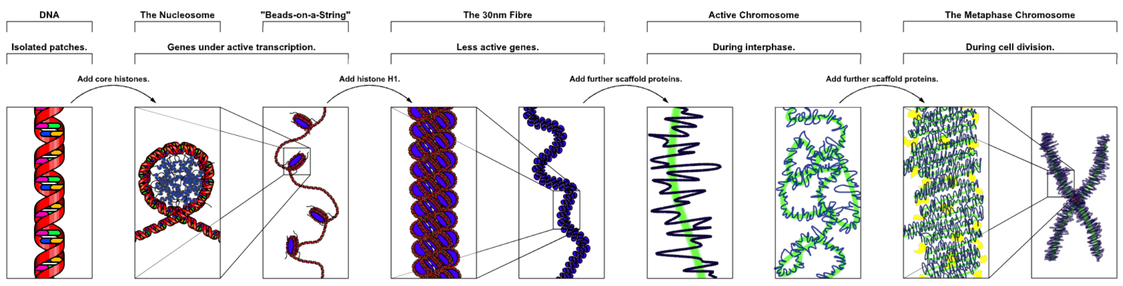


Figure 1: Major Chromatin Structures.
Note that the rightmost structures are only found during cell division, the cell is in interphase during most of the time.

Image 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.

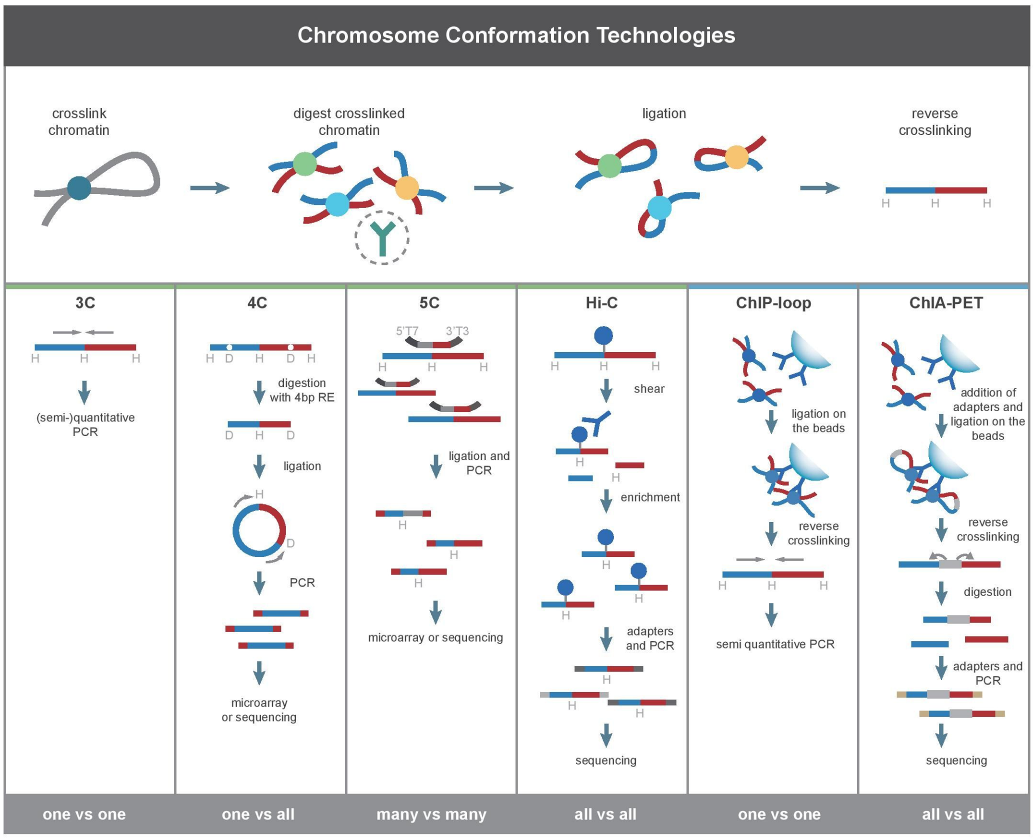


Figure 2: Comparison between 3C and its derived methods.

Image taken from [7].

2.1.2 3-C

In 2002 Dekker et al. [8] 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. [9] [10], 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 2 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. [11], 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 3) 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 [13] 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 4. HiCExplorer can be installed on Linux and MacOS.

2.2.1 Analysis

Corrected Hi-C data can then be further analysed, with `hicFindTADs` one can search for TADs (topologically associated domains) [15], for this a TAD-seperation score is computed and local minima indicative of TAD boundaries are searched for. A visualized result of such a computation can be seen in Figure 4I (created with `hicPlotTADs`).

DNA is compartmentalized [1] in different domains, a model for this can be seen in Figure 5. 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 [13].

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

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

higher value, which immediately catches the eye as a heatmap.

Basic plotting of contact matrices (as seen in Figure 4F) 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 4G).

Computed TADs (as described in Section 2.2.1, see Figure 4I) 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 [13].

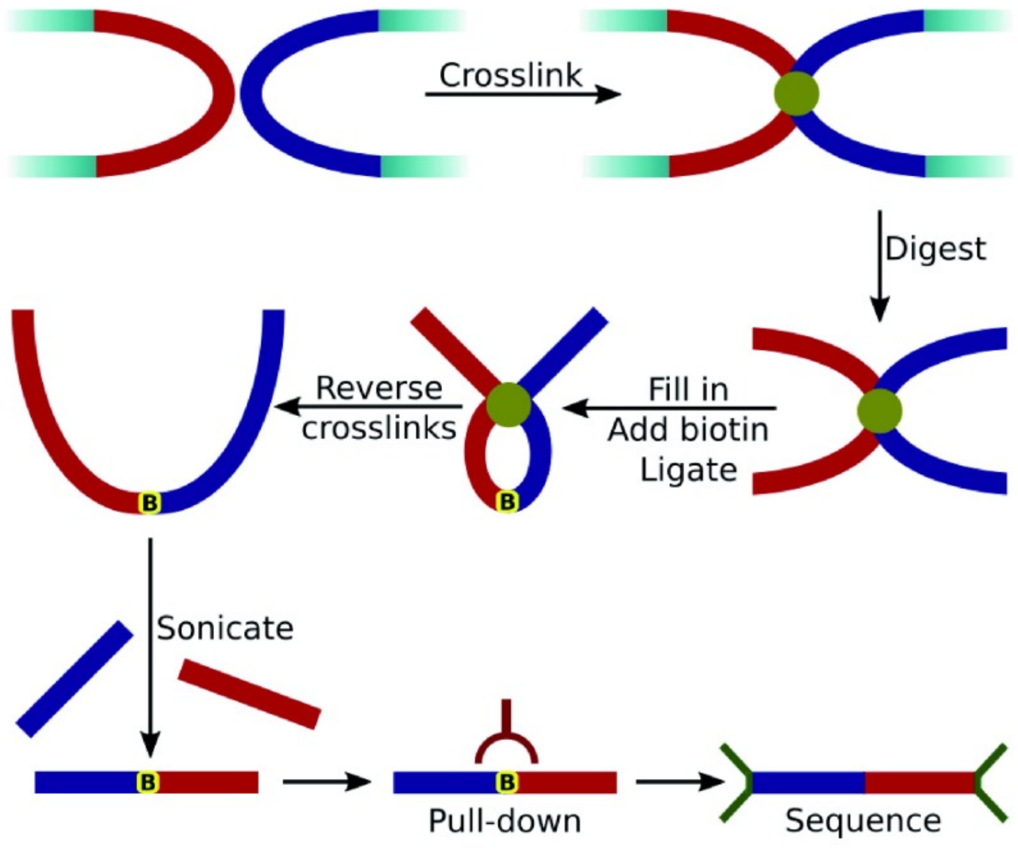


Figure 3: Summarised Hi-C protocol. Biotin is shown by a yellow marker, while the red and blue parts are different parts of cross-linked fragments. The steps are explained in detail in Section 2.1.1 and Section 2.1.5.

Image taken from [12].

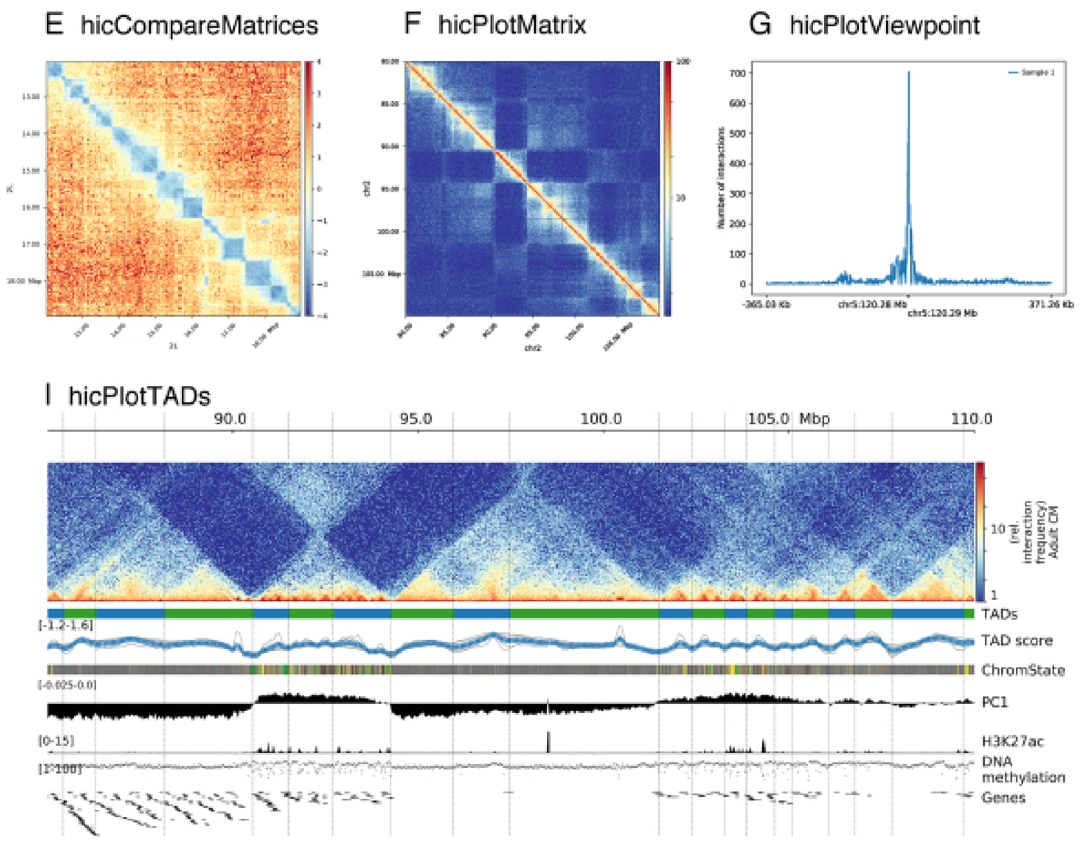


Figure 4: Excerpt of HiCExplorer visualizations E) Pixel difference computed using hicCompareMatrices and visualized using hicPlotMatrix of a Hi-C corrected matrix for wild type condition and knock down. **F)** Plot of a 80 to 105 Mb region contact matrix of chromosome 2 in log scale. **G)** Corrected 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 [13].

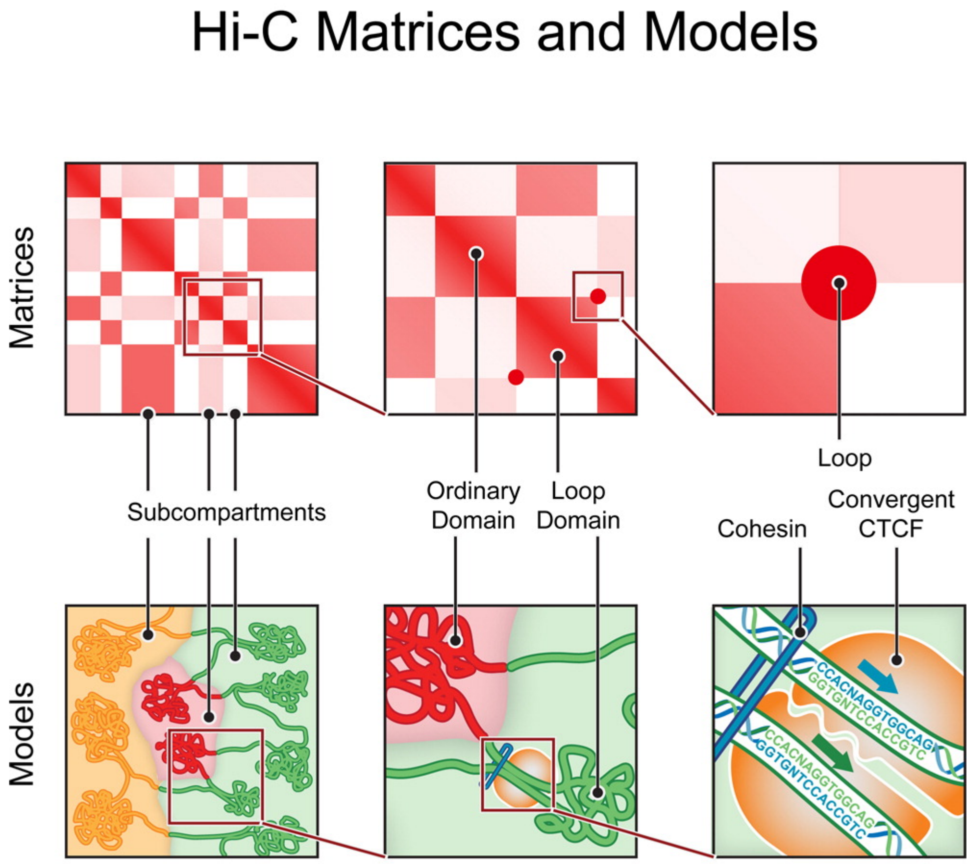


Figure 5: Models relating to HiC-Data.
Image taken from [14].

3 Related Work

(EXTEND: some section here to fill another page)

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

The original implementation was written in Python, since HiCExplorer is 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`.

3.1.1 Advantages

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 lines are not for functionality itself, but for timing, logging and ease of debugging. 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.

Another advantage of Python in general are fast implementation times, which are possible through the concise syntax making the spotting of mistakes easier.

3.1.2 Disadvantages

(DRAFT: Include: Multithreading is locking the execution of the same part of code, processes allow that but require the same memory for writing in it, reading is fine)

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

What follows is a short description to the Algorithm known as Knight-Ruiz from [5]. Essentially, the algorithm is computing the same things, however it is taking advantage of conjugate gradients to solve the iterative steps faster.

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

3.2.1 Implementation

The KR-algorithm was originally implemented in Matlab, here we compare with a version in C++. Calls from Python to C++ can be done over the C-API with considerable help through Python-header files.

3.2.2 Avdantages

A commonly mentioned advantage of C++ is the speed of execution, and fine-grained control over Memory available. However, implementations in C++ can be several orders of magnitude faster than their respective implementation in Python. An advantage of the Algorithm itself is that as long as the matrix itself has total support (meaning that at least one diagonal has only positive nonzero values, this can be artificially done setting zeros to some small positive value), it will converge. Thus, it will converge for way more matrices than the ICE-algorithm.

3.2.3 Disadvantages

Even though the execution is fast, the development process tends to be slow. This is due to the free memory control, which is hard to get right as this requires upholding of implicit assumptions at several places. As these assumptions are implicit only, it is easy to forget them or 'cut corners' when not possible. Those bugs leading to Segmentation-faults (accessing invalid memory) are notoriously hard to find, as they do not follow determinism. Parallelism is even harder to add, since data races (also nondeterministic) and other sources for hard-to-get right problems are added. Additionally, the syntax is considerably complex, making it rather hard to understand. 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

4.1 Problem Description

(DRAFT: Problem: ... what?)

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

As noted in Section 1.1, the main goals include testing the integration of Rust within Python, by implementing a counter-version to the original Python implementation of the iterative part of the ICE algorithm, and then comparing it with the original Python-implementation as well as the recent implementation of the KR-algorithm in C++.

(DRAFT: Possible to improve memory? Testing Parallelization(?) Integration How? Describe Python to C/C++ short)

For this, the iterative correction algorithm will be introduced first. Afterwards, a short introduction to Rust and some of its core concepts is done, before introducing the

4.2 Iterative Correction and Eigenvector decomposition (Algorithm)

In the following the algorithm as defined in the supplementary material to [4] will be cited.

The Algorithm was proposed by Imakaev et al. 2012 [4]

The goal is to obtain the the vector of biases B_i and the 'true' contact map T_{ij} with their relative contact probabilities. This is done by explicitly solving the system of the following two equations:

$$O_{ij} = B_i B_j T_{ij} \tag{1}$$

$$\sum_{i=1,|i-j|>1}^{N} T_{ij} = 1 \tag{2}$$

Equation (1) is stating, that when applying B back again on our corrected matrix T_{ij}, it will be the same as the original matrix O_{ij} again. Equation (2) states, that the sum over the corrected matrix, over arbitrary elements in the upper left triangle, but only one from each column, sums up to one. T_{ij} is doubly stochastic ($\forall_j \sum_{i=1}^{N} T_{ij} = 1$ and $\forall_i \sum_{j=1}^{N} T_{ij} = 1$), **(EXTEND: does it say that it is the same? why is this actually valid?)**

In the algorithm, this is achieved in the following way. First, W_{ij}, a copy of O_{ij} is created. This matrix will converge to T_{ij} during the iterative process. The elements of B are initialized with 1.

$$S_i = \sum_j W_{ij} \tag{3}$$

$$\Delta B_i = S_i / mean(S) \tag{4}$$

Each iteration starts by first calculating the coverage by summing up each row (or column, matrix is symmetric so this does not matter) (Equation (3)) and additional biases based on this by dividing them through their own mean (Equation (4)).

$$W_{ij} = W_{ij}/\Delta B_i \Delta B_j \tag{5}$$

$$B_i = B_i \cdot \Delta B_i \tag{6}$$

Then W_{ij} is iterated by dividing by $\Delta B_i \cdot \Delta B_j$ (Equation (5)), after which B_i is iterated by multiplying with the current biases (Equation (6)). W_{ij} accumulates divisions by ΔB_i, just as B_i accumulates the products of ΔB_i. This is repeated until the variance of ΔB becomes negegible, at which point W_{ij} has converged to T_{ij}.

4.3 Introducing Rust

(DRAFT: where is Rust from, history of Rust)

Rust is classified as a high-level language, even though fine low-level control is possible. This is due the high amount of high-level zero-cost abstractions. Rust has a type system with strong guarantees, promising e.g. that all references (pointers) are valid, or thread safety (memory access from other threads does not result in data races / undeterminism). This is possible through concepts such as ownership and lifetimes. Even though one can program in an object oriented way, Rust is primarily not object-oriented. Additionally it is imperative, procedural, generic and functional.

Syntax The concrete syntax seems similar to C/C++ (curly braces, function signatures), however it is more similar to that of ML or Haskell. A particular example for this case are type classes called “traits” here, similar to C++ templates but inspired from Haskell, supporting polymorphism and generic types. Generic parameters can be constraints, by requiring that generic type to implement a certain Trait.

Memory safety Rust is designed to be memory safe, and does not permit dangling pointers, null pointers, data races in safe code, or usage of uninitialized variables. In case a ‘null’ is needed, the Option-type is provided. Thus, the compiler can guarantee the validity of all references at compile time using its borrow-checker.

Memory management Rust does not have a garbage collector, instead, the resource acquisition is initialization (RAII) convention is used, with optional reference counting. Resource management is deterministic with very little overhead, favoring stack allocation without implicit boxing. References are not run time counted, as their usage is verified at compile time. with this, memory safety can be guaranteed, limiting possible undefined behaviour tremendously.

Ownership In Rust, all values have a unique owner, and the scope of the value is the same as the owners. Immutable references can be passed using `&T`, mutable references by `&mut T`. Pass by value works by passing `T`. Only **one** mutable reference can exist at any point, or any number of immutable ones. This is enforced at compile-time.

Borrowing results directly from the concept of ownership. As mentioned, only one mutable borrow (reference) can happen at a time, however that borrowing variable can further borrow it to other variables or functions. The number of immutable borrows is unlimited, meaning there can be multiple references reading but not modifying part of the memory. This is necessary to guarantee memory safety, as only one mutable reference can write to it at any point in time, wherever that is (in the code).

Lifetimes Lifetimes are the simple concept of keeping track how long each variable and each reference is ‘alive’, this is preventing the simple case of ‘variables going out of scope’ and returning a pointer to it, but can do the same in much more complex environments. Non-Lexical-Lifetimes also work together with borrowing, resulting in variables returning their borrow before the end of the scope, as will be see in the first example (Section 4.3.1).

4.3.1 Examples

Ownership, Borrowing

By executing the following example code, the compiler will throw an error:

```rust
fn main() {
    let mut v = vec![];     // ---| v owns the (empty) vector
    v.push("Hello");        // <--| vector gets first element
                            //    |
    let x = &v[0];          // -| | x borrows the first element from v
    v.push("world");        // <X-| v cannot modify the vector
                            //  | | while x has ownership of it
    println!("{}", x);      // -| | x needed at least until here
}                           // ---| x borrowing v ends, x, v go out of scope
```

(TODO: Add references: Code and Output)

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
 --> src/main.rs:5:5
  |
5 |     let x = &v[0];
  |              - immutable borrow occurs here
6 |     v.push("world");
  |     ^^^^^^^^^^^^^^^ mutable borrow occurs here
7 |
8 |     println!("{}", x);
  |                    - immutable borrow later used here
```

As the compiler is complaining, v needs a *mutable* borrow to modify `v`, however `x` still has an *immutable* borrow! The borrow from `x` cannot be ended yet, because it should be printed later. As the mutable borrow from `v` could modify it in a way such that the reference `x` would be invalid (e.g. delete `v`), this is a potential memory safety problem. However it is fine to print x first, and modify `v` afterwards. The following happens when printing the second element of `v` instead of `x` in the last line, and printing `x` before adding the second element of `v`.

```
Hello
world
```

For reference, here is the code:

```rust
fn main() {
    let mut v = vec![];     // ---| v owns the (empty) vector
    v.push("Hello");        // <--| vector gets first element
                            //    |
    let x = &v[0];          // -| | x borrows the first element from v
    println!("{}", x);      // -| | x needed only until here
                            //    | x returns the borrow here
    v.push("world");        // <--| v can now modify the vector
                            //    | (mutable borrow needed)
    println!("{}", v[1]);   // <--| v can be printed without trouble
}                           // ---| x, v going out of scope
```

Lifetimes

Even though lifetimes exist, the compiler can figure them out itself most of the time. There are cases, however where this is not the case. Take for example the next case:

```rust
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:1:33
  |
1 | fn longest(x: &str, y: &str) -> &str {
  |                                 ^ expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signature does not

error: aborting due to previous error
```

```rust
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

fn main() {
    let string1 = "abcd";
    let string2 = "xyz";

    let result = longest(&string1, &string2);

    println!("The longest string is {}", result);
}
```

Tooling

(DRAFT: introduce awesome tooling)

4.3.2 Advantages of Rust

(DRAFT: Parallelization, tooling, ownership, lifetimes, modularity, stackability, ...)

(DRAFT: writing CSRMatrix bare-metal (thus faster), also the language-features, ...)

4.3.3 Disadvantages of Rust

(DRAFT: includes: longer compilation time and additional time to get to be able to use all the features. Also, frequent updates. 'only' growing userbase)

(DRAFT: *need* to write this bare-metal, no generalized solution, ...)

4.3.4 Comparing Rust and Python

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

4.3.5 Comparing Rust with C/C++

(DRAFT: make table with some numbers, maybe include Python)

4.3.6 Integration of Rust in Python

4.3.7 Using this implementation

4.3.8 Choosing the right API to call Rust from Python

4.4 General Approach

(TODO: biggest points: Python to rust and how it worked, give code examples but not too much) (TODO: presentation: 1/3 für jeden, 1/3 für betreuer + vom fach, 1/3 ich bin experte) (TODO: Link to github repository) (TODO: Tag der abgabe: Github aufräumen und Dokumentieren)

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.)

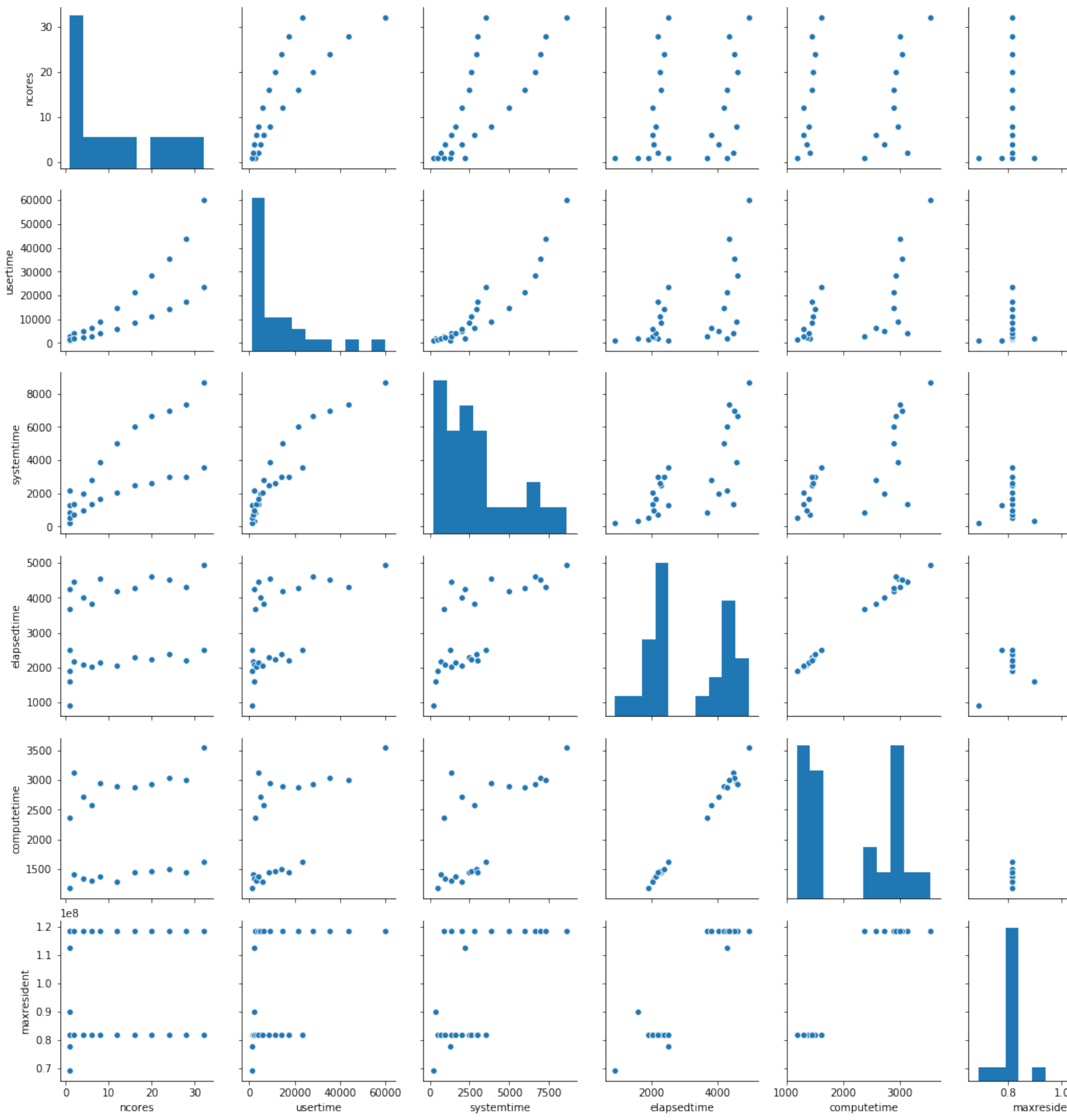
See Figure 6 (more of that will follow).

(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

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



(a) Pairplot over some variables

Figure 6: Pairplot over some of the variables ... moar info here

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 Counters

To Dos: 17;   1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17

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

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

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] R. Wheeler, "Chromatin structures." `https://commons.wikimedia.org/wiki/File:Chromatin_Structures.png`, 2005. Licensed under CC BY-SA 3.0; No changes have been made; accessed 2019-06-26.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.