# Discovering nucleotide-level and structural variants in cancer genome data from second- and third-generation sequencing technologies

Dissertation

zur Erlangung des Grades eines

D o k t o r s  d e r  N a t u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät Informatik

von

Till Hartmann

Dortmund

2024

Tag der mündlichen Prüfung: 27.08.2024

Dekan: Prof. Dr.-Ing. Gernot A. Fink

Gutachter:

1. Prof. Dr. Johannes Köster
2. Prof. Dr. Jens Teubner

# *Acknowledgements*

I would like to use these pages to thank all the people who have accompanied and helped me during my time as a PhD student. Thanks to Sven Rahmann for the freedom to pursue this opportunity, the support during my studies and his hospitality. I thank Johannes Köster for his unwavering support, optimism and advice and agreeing to serve as the first reviewer for this thesis. Thanks to Jens Teubner for agreeing to serve as the second reviewer for this thesis. I would also like to thank Frank Weichert and Kevin Buchin for completing the examination committee. Further, thanks to Johannes Fischer for serving as my mentor in the PhD programme.

As for all the people I shared an office with or first met online during the pandemic: Thank you very, very much! Bianca, Christo, Daniela, David, Elias, Felix, Hamdiye, Henning, Jan, Jens, Marcel B. and Marcel W. all of you have had a profound impact on so many aspects of my life, be it the academic and research part, the leisure and recreational part, the interpersonal, or something entirely different.

Thank you Bianca and David for your unwavering support, the shared progress sessions, for bringing more structure into my life and especially for being incredibly kind human beings. Thank you Christo for all the times we spent discussing ideas, for the times you helped me out with data, for your contributions to our shared projects, for checking in on me and for having an open ear in general, as well as for enthusing more people to join the group's game nights. Thank you Hamdiye for checking in on me, for all the little chats and your music recommendations, which I enjoy very much and which broadened my horizon. Thank you Henning for being very considerate, tidy, kind and insightful, with strong opinions on visual design, and for showing me how to be a better researcher and a better person.

I thank Alicia, Tatjana and Petra for teaching me about biology, processes in the laboratory and clinics and for solving interdisciplinary communication issues together.

Special thanks go to Bianca, David, Neal and Tristan for proofreading (parts of) this work.

Finally, I thank my partners Neal and Tristan for putting up with my mood swings, for serving as rubber ducks, for all their support, and for giving me advice from perspectives different from my own.

# *Preface*

What this dissertation is, is not, and how to read it: All chapters in this work are a product of the process of finding solutions to problems either I encountered myself or fellow researchers proposed or encountered during their research. That is, the contents of this dissertation tend towards the *practical* side of method development and evaluation.

What all chapters have in common is that they address different kinds of structural variation: While chapter 2 introduces a PairHMM specifically tailored towards homopolymer-rich nanopore sequencing data, it finds application in chapter 3, where circular DNA is to be detected. In chapter 4 regions of DNA that have been duplicated/multiplied or deleted are to be detected, yet another form of structural variation. Finally, chapter 5 is the most software centric one, as it is here that we introduce `vembrane`, a tool for the filtering of variant calls as produced in any of the other chapters, for a commonly used bioinformatic specific file format.

As for the layout of this thesis: I opted for the tufte-book layout, which features a large margin for small figures, remarks, footnotes and citations. The latter part is I think quite convenient, as the short form of the citation is directly accessible without any need for page turning, thus not interrupting text flow. All graphics use colour-blind safe palettes: For qualitative data, it is the *bright* scheme introduced in Paul Tol's Notes[1], while for continuous sequential data it is *viridis*, and for continuous diverging data it is *BrBg*.

[1] https://personal.sron.nl/~pault/

A quick note about using "we" instead of "I", as I will consistently use "we" in the main body of this thesis: It is intended to read as "the reader and I" and at the same time also include all people that may have collaborated with me or contributed to the project in some form or another.
Also, this thesis is written in British English, specifically using the Oxford spelling, which *often* uses the suffix "-ize" instead of "-ise" (e.g. "organize" instead of "organise"), depending on the word's etymology.

"As an AI language model…" — jokes aside, while no large language models (LLMs) were used to produce information or content, I have tried using ChatGPT 3.5 as well as GitHub Copilot for re-phrasing sentences or generating formulations *from existing notes*. I have also used DeepLWrite when I felt that certain sentences did not sound right to get suggestions on how to improve sentence structure and grammar. I personally find that GitHub copilot can be incredibly helpful in generating latex code for figures and suggesting suitable labels and names and other auto-completions (that it can derive from the context), while OpenAI's ChatGPT can be useful in generating sentences from short lists of notes (at least if the prompt is designed accordingly). That being said, it is — at least at the time of writing

this — necessary to revise and edit any LLM generated content. And I reiterate: Content and research have to be original, while "stuffing" the information one wishes to communicate into english sentences is a task these LLMs are quite capable of. In other words, these models can be very useful *tools* for the writing process; and as it is the case with most tools, you need to learn how to wield them properly and conscientiously.

# *Abstract*

The field of bioinformatics is a diverse one, as it — the name giving a quite obvious hint — bridges the fields of biology and computer science. For example, in (human) cancer research, one commonly

1. obtains a blood and/or tissue sample of a patient in a study,
2. sequences or otherwise analyses the sample on a specialized device to obtain relevant information (such as its *genome*, *transcriptome* or *methylome*),
3. determines variation between the sample and some reference sample(s) or determines other aspects that may be of interest,
4. annotates, filters and analyses these for further examination,
5. and subsequently makes use of them to further one's research or study goal.

In this thesis, we will mainly concern ourselves with the last three aspects, though we will also explain the sequencing process for two different technologies. Also, we will focus on the *genome* part of the second item, i.e. the DNA contained within the cells of most organisms. In this context, *determining variation* usually involves comparing many short DNA sequences to a larger reference DNA sequence (such as "the human genome").

The sequencing technology used will become important in chapter 2, where we first introduce a homopolymer-aware PairHMM, which addresses one major issue of nanopore sequencing: due to the design of the technology, so-called homopolymers — runs of identical nucleotides — often have inaccurately called lengths, which impact results negatively. This specialized model allows for more accurate alignments and probability estimates, and can be applied to any nanopore sequencing data.

This model is then applied implicitly in chapter 3, where nanopore sequencing data is used to find a very specific kind of variation: extrachromosomal circular DNA. Extrachromosomal circular DNA is exactly what its name implies: DNA that is both circular and outside the chromosomal structure of the genome. Such extrachromosomal circular DNA plays an important part in cancer research, as a biomarker for certain cancers or as a way to track tumour progression. We developed a graph-based method to detect eccDNA in sequencing samples, and also provide an easy to set up and reproducibility guaranteeing workflow with an interactively explorable report.

In chapter 4, we look at a different kind of variation: copy number variation, where (larger) regions of a sample's genome have been repeated or deleted. Over the years, many approaches making use of different kinds of information have been explored, for example read-depth information: As already hinted at before, determining variation is often done by comparing, *mapping* or *aligning* short DNA sequences to a reference sequence; for each position in the reference sequence, it is then possible to find the number of short sequences that overlap the respective position, which is basically the *read-depth*. This mapping process, however, can be both computationally expensive and require lots of disk-space. To improve on resource usage, we use a kind of pseudo-mapping based on k-mers instead, and show that it is at least as good as relying on classic read-depth information, while at the same time saving disk space.

Because the field of bioinformatics wouldn't be the same without its custom file formats, we address one specific format — the Variant Call Format (VCF) — in chapter 5. VCF is a text-based format describing genomic variation, for example a duplication (copy number variation) or a gene fusion or circle. As the format is widely used and there initially was no binary counterpart, it can be alluring to resort to established text modifying tools such as `awk`, `sed` and `grep`. However, the intricacies of the file format essentially prohibit this, as the results will likely be not what one expects. Even with existing specialized tools such as `bcftools`, certain syntax and semantic combinations are quite unintuitive and potentially lead to incorrect results. We therefore provide `vembrane`, a VCF tool, which does not introduce its own domain specific language but uses Python instead. It also has built-in support for certain custom annotations (such as provided by `SnpEff` or `VEP`) and is the only tool which handles breakend records correctly. For example, extrachromosomal circular DNA variants have to be encoded using such breakend records.

# *Contents*

# 1 Basics

## 1.1 Genome

Every known organism's genetic information is stored in its *genome*, encoding its development, reproduction and general function. The genome consists of a set of Deoxyribonucleic acid (DNA) molecules (RNA virus genomes consist of Ribonucleic acid (RNA) molecules instead), large polymers arranged in a double-stranded helix, with each DNA molecule a chain of *monomers*, in this case *nucleotides*.

Each nucleotide is a combination of a sugar (either deoxyribose in the case of DNA or ribose for RNA), a phosphate and one of four possible nucleobases A denine, C ytosine, G uanine and T hymine (U racil in the case of RNA). Abstracting from the actual molecules, these four nucleotides form the alphabet $\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$ that is used to encode genetic information.

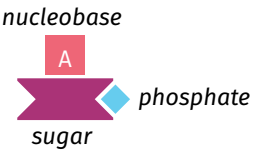



Figure 1.1: Schematic representation of a nucleotide, consisting of a sugar, a phosphate and one of four nucleobases, adenine, cytosine, guanine or thymine.

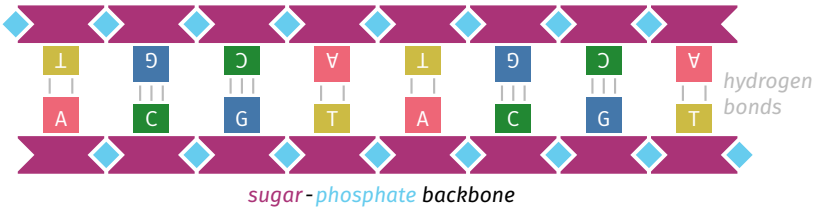



Figure 1.2: Schematic representation of a DNA molecule, consisting of two *complementary* strands, with A always pairing with T and C with G, by means of hydrogen bonds.

In eukaryotes, DNA molecules are arranged into complex three-dimensional structures called *chromosomes*[1]. Each species' genome consists of a specific number of different chromosomes. In each cell, chromosomes have a distinct number of copies, called *ploidy*[2]. For example, humans have 23 distinct chromosome pairs (ploidy 2, the human genome is *diploid*), where the first 22 pairs are exact copies, but the last pair is either a pair of X chromosomes (female) or the pair XY (male)[3].

The DNA molecule itself can be seen as source code whose interpretation (by the chemical machinery of the cell) ultimately leads to the different functions, traits and aspects of the organism. Some sections of the DNA serve as templates for certain products, usually proteins, which take on specific tasks and functions in the organism. We call such protein-coding regions *genes*[4]. However, in order for proteins to be *expressed* (i.e. actually built), one strand of the respective section is translated into messenger-RNA (mRNA) first, serving as blueprints for proteins.
Note that while DNA is static in the sense that it is not intended to change during normal function of the cell[5] but only serves as the source of build-

[1] Chromosomes in prokaryotes are less complex, and typically circular.

[2] For many species, this number does not vary between individuals or cells in general, but it can.

[3] In sexually reproducing species, there are different constellations of sex differentiation: In mammals, XX corresponds to female, XY to male individuals; in birds and reptiles, XX corresponds to male, XY to female individuals; in nematodes, XX corresponds to female, X0 to male individuals — in the latter case, the sperm either carries a single X or no chromosome at all.

[4] There are also non-coding genes, which are not strictly relevant for this thesis.

[5] With some exceptions, such as immunocompetent cells.

instructions, the resulting transcripts in the form of single stranded RNA/mRNA molecules are volatile/short-lived in the sense that they are produced "on-demand", then used and eventually degrade passively.

### 1.1.1 *Reference genomes*

Reference genomes are specific, fixed instances of a species' genome. They facilitate comparison and annotation of individual genomes, and provide a common basis for research. For example, comparing a patient's genomic sequence to a reference genome can help identify the patient's individual genetic diseases, and comparing reference genomes of different species can help identify evolutionary relationships between them.

Reference genomes can both be constructed from a single individual's genome or from a consensus of multiple individuals' genomes.
For Homo sapiens, there has been a consensus reference genome available since 2003[6], as a result of the Human Genome Project[7] which ran from 1990 to 2003[8]. This reference was constructed based on DNA of thirteen individuals. Since 2003, it has been improved using more modern sequencing techniques, resulting in `GRCh38` / `hg38` (last release 2013, latest patch `GRCh38.p14` 2022). This reference (series) has been the gold-standard reference genome for Homo sapiens for the last two decades, and has been studied and annotated extensively. Additionally, the National Institute of Standards and Technology (NIST) provides access to sequencing and benchmark data of human samples via its Genome In A Bottle (GIAB) project[9].

A more recent assembly named `CHM13` has been released in 2022. It is based on a *single* human genome `T2T-CHM13`, a haploid and therefore completely homozygous cell-line. It improves upon `hg38` by resolving difficult to sequence regions, centromeres, telomeres and short arms of acrocentrics chromosomes (13, 14, 15, 21, 22), ribosomal DNA (rDNA) arrays, as well as repetitive regions and satellites.[10]
The assembly of `CHM13` uses a combination of recent sequencing techniques (Illumina PCR-Free, PacBio HiFi, Oxford Nanopore, PacBio CLR, 10X Genomics, BioNano DLS and Arima Genomics HiC)[11], and the data used for assembly is readily available. The fact that `CHM13` is an assembly based on a single homozygous cell-line makes it extremely valuable for benchmarking sequencing techniques with respect to sequencing errors: Any differences between the sequencing data (used for assembly) and the final assembly are due to the error characteristics of the respective sequencing technique, and are not caused by differences between individuals (as there is only one individual).

[6] A working draft was available since 2001.

[7] International Human Genome Sequencing Consortium et al., "Initial sequencing and analysis of the human genome", 2001.

[8] *Fact Sheet: Human Genome Project* (`https://www.genome.gov/about-genomics/educational-resources/fact-sheets/human-genome-project`, visited on 2023-06-19)

[9] *Genome In A Bottle* (`https://www.nist.gov/programs-projects/genome-bottle`, visited on 2023-11-20)

[10] Nurk et al., "The complete sequence of a human genome", 2022.

[11] *CHM13* (`https://github.com/marbl/CHM13/blob/96c0892d70affb42648c277bfe3974e94d4ffebc/README.md`, visited on 2022-10-17)

## 1.2 *Sequencing*

In this context, the term *sequencing* describes the process of determining the type and order of nucleotides in a DNA or RNA molecule. Over the years, several techniques have been developed to that effect, with different advantages and disadvantages. The different techniques are usually categorized into generations[12,13] (see Table 1.1, though in no way complete), but the classification is not always straightforward.

[12] Heather and Chain, "The sequence of sequencers: The history of sequencing DNA", 2016.

[13] Goodwin, McPherson, and McCombie, "Coming of Age: Ten Years of Next-Generation Sequencing Technologies", 2016.

| Gen. | Method | Characteristics |
|---|---|---|
| 1 | Sanger | sequencing-by-synthesis |
| 2 | Illumina MiSeq/HiSeq/etc | Massively parallel, sequencing-by-synthesis |
| | Roche 454 | DNA micro-beads, emPCR |
| | SOLiD | sequencing-by-ligation |
| 3 | Oxford Nanopore Technologies | Long read, single molecule |
| | Pacific Biosciences SMRT | Long read, single molecule |

Table 1.1: Overview of sequencing methods and their characteristics, ordered by generation.

Most of the methods presented in this thesis have been developed with Oxford Nanopore Technologies (ONT) sequencing data in mind, but can in principle also be applied to data produced by other technologies. For example, while the methodology in chapter 3 has been developed specifically for ONT sequencing data, it also applies to Illumina sequencing data, or even a combination of both (see section 3.3). We will therefore give a brief overview of both ONT and Illumina (Illumina) sequencing[14] in the following subsections, highlighting the differences in their characteristics, especially with respect to their error characteristics.

[14] From here on, we will refer to these as *nanopore* and *Illumina* sequencing, respectively.

### 1.2.1 *Nanopore sequencing*

The basic principle of nanopore sequencing is as follows: A chamber containing electrolyte solution is divided by a membrane into two halves; the halves are connected by a *nanopore*, a nanometre scale pore. A molecule to be measured (for example a protein or DNA molecule) is placed into one of the halves of the chamber.

The molecule is then coaxed through the nanopore, by means of applying a voltage bias / potential difference across the membrane. As the molecule travels through the pore, it inadvertently affects certain properties of the system which can be measured/monitored, for example the current along the membrane or the ion current through the pore. This results in a time-series of signal values specific to the part of the molecule in the sensing region at the corresponding time.
For nanopore DNA-sequencing in particular, a motor protein is attached to the end of both strands, facilitating unzipping of the DNA and movement of one strand through the pore. The number of monomers that are in the sensing region of the pore depend on size and design of the pore; usually about 3 to 4 nucleotides/monomers.[15]

In an ideal world, the DNA molecule to be sequenced is unfolded completely, moves through the pore with constant velocity and only moves in one direction. However, because of non-uniform charges of/along the molecule and other interactions between the molecule and the system, this may not hold, potentially leading to sequencing errors.

The result of sequencing is a stream of measurements. To obtain the actual DNA sequence, a *base calling* step is necessary, where the raw signal is translated into a sequence of nucleotides. In this thesis, we will not concern

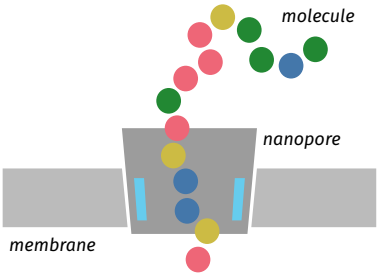



Figure 1.3: A schematic representation of a nanopore. The membrane separates a chamber filled with electrolyte solution into two halves. The nanopore connects the two halves.

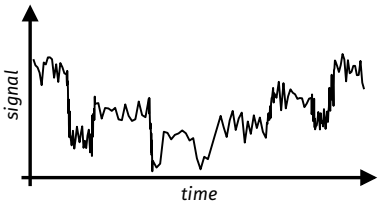



Figure 1.4: Hypothetic measured signal over time of a molecule traversing the nanopore.

[15] Chinappi and Cecconi, "Protein sequencing via nanopore based devices: a nanofluidics perspective", 2018.

ourselves with the base calling step, but rather assume that the base calling has already been performed, and the resulting sequence is available, albeit potentially with sequencing errors.

**Homopolymer errors** Nanopore sequencing is particularly prone to *homopolymer errors*: An error class constituted of cases where a stretch of $n$ identical nucleotides in a molecule is called as a stretch of $m \neq n$ identical bases instead. This can be explained by a combination of the following points: 1. The molecule does not move with constant velocity through the pore, 2. the sensing region of the pore is as large as several nucleotides (i.e. changes in current are due to multiple bases at the same time) and 3. stretches of homopolymers larger than the sensing region result in difficult to segment/base-call signal.
As a reference point, Sarkozy, Jobbágy, and Antal [2018][16] state that "[the nanopore's current] level is representative of a segment of five nucleotides inside the pore", though the exact number is dependent on the actual pore design.

[16] Sarkozy, Jobbágy, and Antal, "Calling Homopolymer Stretches from Raw Nanopore Reads by Analyzing k-mer Dwell Times", 2018.

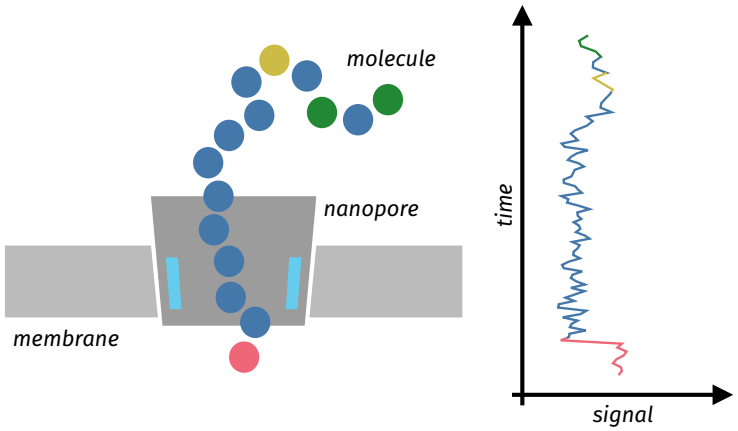



Figure 1.5: A DNA molecule containing a homopolymer run of G s and the corresponding nanopore sequencing signal over time. Because the molecule does not move with constant speed and multiple nucleotides influence the signal readout at the same time, the correct length of the homopolymer run is difficult to discern.

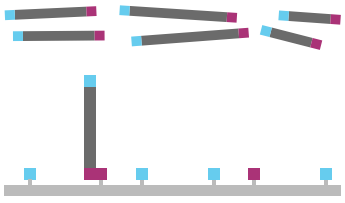
### 1.2.2 *Illumina sequencing*

In contrast to nanopore sequencing, Illumina sequencing[17,18] operates on many small DNA *fragments* in parallel and uses a sequencing-by-synthesis approach as first introduced by Sanger in 1977.[19] The basic approach is as follows:

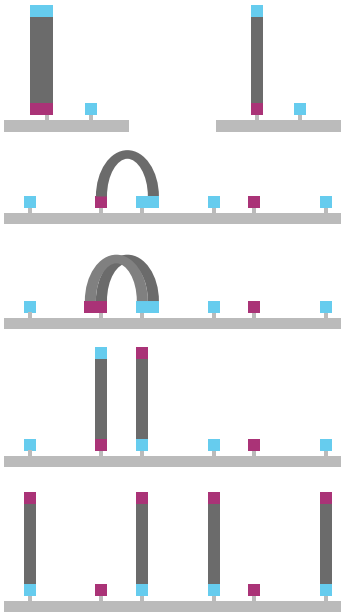
**Sample preparation** During sample preparation, the DNA sample is first fragmented into smaller pieces of DNA. *Adapter* sequences are ligated to each end of these fragments. These serve as binding sites for subsequent steps, providing priming sites for amplification and sequencing. Additionally, they may contain barcode/index sequences, enabling sample multiplexing[20].
The double-stranded fragments are then denatured, resulting in two single-stranded fragments.
These are then introduced to the surface of a so-called *flowcell*, which contains oligonucleotides complementary to the sequences in the fragments' adapters.

[17] Goodwin, McPherson, and McCombie, "Coming of Age: Ten Years of Next-Generation Sequencing Technologies", 2016.
[18] Turcatti et al., "A New Class of Cleavable Fluorescent Nucleotides: Synthesis and Optimization as Reversible Terminators for Dna Sequencing by Synthesis", 2008.
[19] Sanger, Nicklen, and Coulson, "DNA sequencing with chain-terminating inhibitors", 1977.

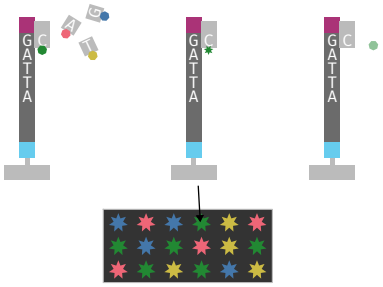
[20] Sequencing multiple samples at the same time.



This allows the DNA fragments to hybridize to the oligonucleotides on the flowcell surface.



■ Amplification Using the bound DNA fragments as templates, complementary sequences to the single-stranded fragments are synthesized. These synthesized strands attach directly to the flowcell surface.
The DNA is denatured again, and the strand not attached to the flowcell is washed away.
The bound DNA strands on the flowcell surface fold over, allowing them to hybridize with a second type of oligonucleotide on the flowcell. This hybridization forms a "bridge" structure where the DNA fragment is connected to the flowcell surface at both ends.
Complementary sequences to the DNA fragments are synthesized again, using the bridge structure as a template.
The DNA strands are denatured once more, but this time both forward and reverse strands remain attached to the flowcell.
The bridge amplification process is repeated several times, resulting in clusters of DNA fragments on the flowcell, each with multiple copies of the original fragment.
Only one strand (either forward or reverse) is kept, the other is washed away.




■ Sequencing A sequencing primer binds to the oligonucleotides of an attached DNA fragment. During each sequencing cycle, a mixture of fluorescently labelled nucleotides is added. These nucleotides are modified in a way that only one can attach to the DNA fragment at a time, based on the complementary base pairing rules.
The fluorescent label attached to the incorporated nucleotide is excited using a laser, and the emitted light is detected to determine the identity of the nucleotide. Because there are *clusters* of identical fragments on the flowcell, the signal from each cluster is amplified, allowing for detection of the fluorescent signal.

The fluorescent label is removed, and the process is repeated for the next nucleotide.

### 1.2.3 *Comparison between nanopore and Illumina sequencing*

Depending on the application and available resources, ONT or Illumina sequencing may be preferable — though there are of course many more sequencing technologies available, which are not discussed here.

The advantages of nanopore sequencing over Illumina sequencing are:

1. No PCR/bridge-amplification is necessary, eliminating bias introduced during the amplification process,
2. in principle, single molecules can be sequenced *completely*[21],
3. read lengths of up to about $0.2$ to $1\text{Mbp}$ can be achieved,
4. real time sequencing (one molecule after another in a single pore) is possible,

5. portability (small portable USB sequencers called *MinION* are available, facilitating sequencing on-the-go with a laptop).

Long read lengths in particular are useful when resolving large structural variation or repetitive regions. However, the raw error rate of ONT sequencing is generally higher (about 5 to 12%[22,23,24]) than that of Illumina sequencing (0.1 to 1%[25]). Additionally, considerations about the cost of sequencing and the time it takes to sequence a sample have to be taken into account.

[21] . . . but not always in practice for example because of pore degradation over time or molecular instabilities.

[22] Jain et al., "Nanopore sequencing and assembly of a human genome with ultra-long reads", 2018.

[23] Amarasinghe et al., "Opportunities and challenges in long-read sequencing data analysis", 2020.

[24] Goodwin, McPherson, and McCombie, "Coming of Age: Ten Years of Next-Generation Sequencing Technologies", 2016.

[25] Ibid., 2016.

## 1.3 *Bioinformatic basics*

In this section, we will introduce basic concepts, terminology and notation used throughout this thesis.

**Definition 1.3.1** (set)**.** A *set* is an unordered collection of unique elements. The *size* (number of distinct elements) of a set $S$ is denoted by $|S|$.

For example, the set $\{1, 2, 3\}$ is equivalent to the set $\{3, 2, 1, 1\}$, as two sets are considered equal if they contain the same unique elements, regardless of their order and frequency.

**Definition 1.3.2** (sequence)**.** A *sequence* is an ordered collection of elements. The *length* of a sequence $S$ is denoted by $|S|$.

For example, the sequences $(1, 2, 3)$ and $(3, 2, 1)$ are not equal, as the order of their elements differs. Similarly, the sequences $(1, 2, 3)$ and $(1, 1, 2, 3)$ are not equal, as the number of elements differs. For convenience, the last element of a sequence $S$ is denoted as $S_{-1}$.

**Definition 1.3.3** (string)**.** A *string* is a finite sequence of characters from an alphabet $\Sigma$. The *length* of a string $s$ is denoted by $|s|$.

For example, CAT is a string of length 3 over the alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$.

**Definition 1.3.4** (substring)**.** For a string $s \in \Sigma^+$, $s_{i..j}$ denotes the substring of $s$ starting and ending at positions $0 \leq i \leq j \leq |s|$ (0-based indexing, end exclusive).

If either $i$ or $j$ is omitted, their values are assumed to be 0 and $|s|$ respectively, i.e. $s_{i..}$ is $s$ with the first $i$ characters removed, or the $|s| - i$ length suffix of $s$. Similarly, $s_{..j}$ is $s$ with the last $|s| - j$ characters removed, or the $j$ length prefix of $s$.

**Definition 1.3.5** (k-mer)**.** A k-mer is a substring of $k$ length of a string. The sequence of k-mers of a string $s$ is

$$K(s, k) \coloneqq (s_{i..i+k} \mid 0 \leq i \leq |s| - k) \ .$$

For example, the sequence of k-mers of length 3 (*3-mers*) of the string ACATT is $K(\mathtt{ACATT}, 3) = (\mathtt{ACA}, \mathtt{CAT}, \mathtt{ATT})$.

**Definition 1.3.6** (reverse complement)**.** The reverse complement of a string $s$ over the alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ is

$$\overleftarrow{s} \coloneqq (\overleftarrow{s_{n-1}}, \overleftarrow{s_{n-2}}, \ldots, \overleftarrow{s_0}) \ ,$$

where $n \coloneqq |s|$ and $\overleftarrow{\mathtt{A}} = \mathtt{T}$, $\overleftarrow{\mathtt{C}} = \mathtt{G}$, $\overleftarrow{\mathtt{T}} = \mathtt{A}$, $\overleftarrow{\mathtt{G}} = \mathtt{C}$.

For example, the reverse complement of the string ACATT is AATGT.

Given a sequencing read obtained from a reference genome, the read may have been sequenced from either strand of the genome. To deal with this ambiguity, canonical k-mers are used.

**Definition 1.3.7** (canonical k-mer)**.** A canonical k-mer is the lexicographical minimum of a k-mer $x$ and its reverse complement $\overleftarrow{x}$:

$$\tilde{x} = \min(x, \overleftarrow{x})$$

For example, the canonical representation of ACATT is AATGT, because AATGT is lexicographically smaller than ACATT.
Note that in the definition above, we use the minimum (or lexicographically smaller) of the k-mer and its reverse complement as the canonical k-mer. This choice is arbitrary, and choosing the maximum (or lexicographically larger) k-mer as the canonical k-mer would be equally valid, as long as the choice of ordering is consistent.

Because k-mers are of constant length $k$ and the DNA alphabet is of size 4, a natural way to encode k-mers is with a two-bit integer encoding: Assign a unique 2-bit pattern to each character of the alphabet, often

$$\texttt{A} = \texttt{00},\ \texttt{C} = \texttt{01},\ \texttt{G} = \texttt{10},\ \texttt{T} = \texttt{11}\ .$$

Then, a k-mer can be represented in $2k$ bits, which allows a k-mer of length $k \leq 32$ to be stored as a 64-bit unsigned integer[26]. This encoding has the added benefits that k-mers can be sorted in $O(n)$ time using radix sort, and that calculating the reverse complement is cheap in terms of CPU instructions (see Figure B.3). If not otherwise specified, this encoding is assumed throughout this thesis. Subsequently, comparing two k-mers takes constant time (instead of $O(k)$ in a string or byte-array representation), and masking bases (e.g. the first or last one) or obtaining pre- or suffixes can also be done in constant time.

[26] More generally, for $k \leq$ word-size$/2$, a single integer is sufficient.

To update values in an array, we use shorthand notation $A_{i..j}\ \mathrel{+}= x$ instead of $A_k \leftarrow A_k + x\ \forall i \leq k < j$. It is assumed that accessing a slice, calculating the reverse complement and comparing two k-mers (or k-mer slices of the same length) can be done in constant time, assuming integer encoding as described above.

A simple bit-shift update rule for generating two-bit encoded k-mers from a given sequence is shown in Figure 1.6. This routine also allows producing canonical k-mers and the position in the sequence from which a k-mer was generated.

Special care has to be taken when a base is not any of $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$ (for example N or other International Union Of Pure And Applied Chemistry (IUPAC) ambiguity codes), since other codes are not part of the two-bit encoding scheme defined above. In that case, skip until a new valid k-mer that only contains bases A, C, G and T can be formed.

```python
def kmers_with_positions(seq: str, k: int, canonical: bool):
    mask = (1 << (2 * k)) - 1
    kmer = encode(seq[0:k])
    if canonical:
        canonize = lambda kmer: min(kmer, revcomp(kmer))
    else:
        canonize = lambda kmer: kmer
    yield canonize(kmer), 0
    for i, base in enumerate(seq[k:]):
        kmer = ((kmer << 2) | twobit(base)) & mask
        yield canonize(kmer), i + 1
```

Figure 1.6: Pseudocode for generating (canonical) k-mers of a given sequence, including the position of the k-mer in the reference.

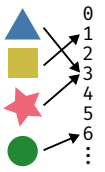
### 1.3.1 *Hashing*

Hashing is a technique to map items of a large input domain into an output domain that is usually smaller than the input domain by orders of magnitude.[27] In the context of this thesis, we will use hashing to map k-mers to integers, which can then be used as indices into an array. That is, we map items from the input domain of $\Sigma^k$ (of cardinality $4^k$) to the output domain of $\{0, \ldots, n-1\}$ (of cardinality $n$), with $n \ll 4^k$.

[27] Konheim, "Perfect Hashing", 2010.

**Definition 1.3.8** (hash function)**.** A hash function $h : X \to V$ is a function that maps items of the input domain $X$ to integers of the codomain $V \subset \mathbb{N}_0$. The result of the application of $h$ to an item $x \in X$ is its *hash value* $v$:

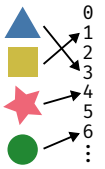
$$h(x) = v \in V \ .$$




**Definition 1.3.9** (perfect hash function)**.** A perfect hash function is a hash function that is injective:

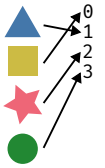
$$x_1 \neq x_2 \implies h(x_1) \neq h(x_2)$$

That is, different items are mapped to different hash values.




Using a perfect hash function allows us to forego collision detection and resolution (i.e. how to deal with the case when two or more k-mers share the same hash value), at the cost of having to know the entire input domain in advance, more complex hash functions, and potentially larger space requirements.

**Definition 1.3.10** (minimal perfect hash function)**.** A minimal perfect hash function is a perfect hash function which maps a set of $n$ items to $n$ consecutive integers ($\{0, \ldots, n-1\}$).




By using minimal perfect hash functions instead, the space requirements are reduced to the exact required amount, again potentially at the cost of more complex hash functions.

### 1.3.2 *Caching*

While caching is mainly relevant for chapter 4, we will still briefly introduce some basic aspects here, as these are generally relevant to any algorithm engineering task. The information and visualizations presented here are based on Drepper [2007][28] and Timm [2021][29].

[28] Drepper, "What every programmer should know about memory", 2007.

[29] Timm, "Analysis and Application of Hash-based Similarity Estimation Techniques for Biological Sequence Analysis", 2021.

The general problem is this: To perform any computation, data has to be loaded from e.g. main memory into the processor's registers. However, the time it takes to load data from main memory is orders of magnitude larger than the time it takes to access data in the processor's registers. Because the size and number of registers are limited, data has to be moved between registers and main memory frequently. That would imply that the processor is idle most of the time, waiting for data to be loaded from main memory. To mitigate this problem, caches are introduced, conceptually between main memory and the processor's registers. Caches are usually tiered, with smaller and faster caches "closer" to the processor, and larger and slower caches further away. Any data that is read for the first time is loaded from main memory into the cache. Subsequent reads of the same data can then be served from the cache, which is much faster than loading it from main memory. When the cache is full, different strategies to evict data from the cache can be employed, e.g. discarding the least recently used data.

When the processor reads data, first its caches are queried. In case a cache contains the requested data, this is called a *cache hit*. If the cache does not contain the requested data, this is called a *cache miss*, and the data has to be loaded from main memory (into the cache).

Caches operate in units of cache lines, for example 64 bytes per cache line. This means that, for instance reading a single byte will actually trigger loading an entire cache line into the cache (see Figure 1.7). Accesses to other bytes of the same cache line will then be served from the cache, while accesses to bytes of other cache lines will result in a cache miss (triggering loading of the respective cache line and potentially discarding of old cache line(s)). This implies the following properties:

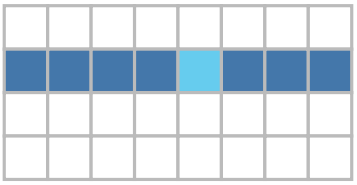


Figure 1.7: Accessing a value in memory will load an entire cache line into the cache.

- Spatial locality: Accessing data from a contiguous memory region increases the likelihood of a cache hit for subsequent accesses.
- Temporal locality: Even if data is far apart in memory, if it was accessed recently and is likely to be accessed again soon, it benefits from caching.
- Predictability: Predictable patterns of memory access can be exploited (by the programmer, compiler or the processor itself) to pre-emptively load data into the cache (*prefetching*) (see Figure 1.8).

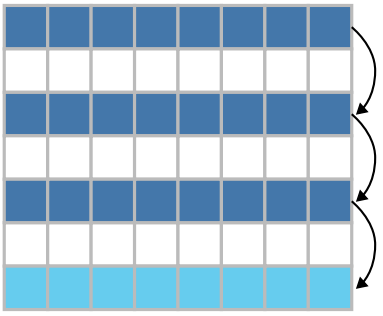


Figure 1.8: After three spaced accesses, a memory-access pattern is detected and the next cache line is pre-emptively loaded into cache, i.e. *prefetched*.

Ensuring that a program's memory access patterns are *cache-friendly*, i.e. leverage spatial and temporal locality and are predictable, can have a huge impact on the program's performance.

### 1.3.3 *Miscellaneous*

In bioinformatics, the PHRED quality score[30] is a logarithmic transformation of probabilities, commonly used as a measure of the quality of the base calls made by a sequencer.

[30] Ewing et al., "Base-calling of automated sequencer traces using phred.", 1998.

**Definition 1.3.11** (PHRED). Let $P$ be the probability of e.g. an alignment being incorrect or a base called *incorrectly* by a sequencer (i.e. $P$ is an error probability). The PHRED score $Q$ is then given as

$$Q = -10 \log_{10}(P) \ .$$

Conversely, to obtain an error probability $P$ from a PHRED score:

$$P = 10^{-Q/10} \ .$$

The lower the error probability, the higher the corresponding PHRED score and vice versa. The minimum PHRED score is $Q = 0$, corresponding to an error probability of $P = 1$. Theoretically, there is no maximum PHRED score, indicating that an error probability of 0 is unattainable. However, in practical applications, the score is often capped e.g. due to domain representation constraints, such as in the FASTQ format, which uses an ASCII encoding for the PHRED score, thus only allowing for integer scores between 0 (ASCII symbol 33: `!`) and 93 (ASCII symbol 126: ~).

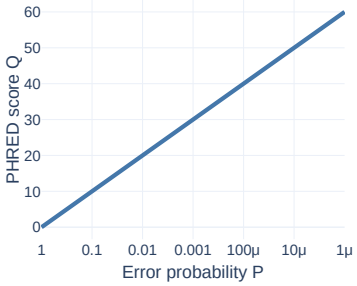



Figure 1.9: Error probability $P$ (x-axis, logarithmic) and PHRED score $Q$ (y-axis).

**Ambiguous and masked bases** Usually, reference sequences are predominately composed of the bases in {A, C, G, T}. However, sometimes it is necessary to represent uncertainty or ambiguity in the reference sequence. To that end, IUPAC ambiguity codes are used to encode an option of multiple different bases (at a single locus) in a single symbol, as shown in Figure 1.10. The symbol `N` is often (ab-)used to *hard-mask* regions in the reference sequence which e.g. could not be assembled or proved problematic in other ways.

Yet another source of uncertainty in reference sequences are repetitive regions. When the actual sequence of a repetitive region is resolved, but the number of repetitions is not, such a region is usually *soft-masked* by using lowercase letters for the corresponding bases in the sequence.
Note that this is only a convention, and the meaning of uppercase and lowercase letters in a sequence is not standardized. In fact, *Ensembl*'s genome browser sometimes uses lowercase letters to denote intronic regions[31].

| Code | Meaning |
|---|---|
| A | A |
| C | C |
| G | G |
| T | T |
| Y | C T |
| R | A G |
| W | A T |
| S | C G |
| K | G T |
| M | A C |
| D | A G T |
| V | A C G |
| H | A C T |
| B | C G T |
| N | A C G T |

Figure 1.10: IUPAC nucleotide ambiguity codes. Additionally, the symbols `-` or `.` are used to denote a gap.

[31] *Ensembl Genome Browser Help Page 155* (`https://www.ensembl.org/Help/View?id=155`, visited on 2023-12-13)

## 1.4 *Alignments and Mappings*

A crucial part of bioinformatic analyses is sequence alignment. Aligning two or more sequences to each other highlights similarities and differences between them. This for example allows studying relatedness between different organisms, as well as evolution of species over time. It also facilitates the study of somatic and hereditary diseases caused by mutations in an organism's genome.

A common application of sequence alignments is the mapping of reads — sequences obtained during sequencing of DNA molecules — to a reference genome. Often, the lengths of reads will be much shorter than the length of the reference genome. For example, the human genome has a length of roughly 3 Gbp, whereas common read lengths are several hundred base-pairs (e.g. Illumina) to a few million base-pairs at maximum (e.g. ONT). In such cases, the shorter sequence (read) is aligned completely against parts of the longer sequence (reference genome), a *semi-global* alignment.

For these tasks, algorithms based on different principles and targeted towards different sequence types (e.g. arbitrary alphabet, DNA, protein) and applications exist. For example, the Smith-Waterman algorithm for local alignments was proposed in 1981,[32] and the Needleman-Wunsch algorithm for global alignments in 1970.[33] Both of these can be formulated in terms of the dynamic-programming principle and employ a scoring scheme, returning those alignments that maximize the respective score. In the case of DNA and protein sequences, there are sophisticated algorithms and scoring schemes tailored towards the corresponding alphabets, as well as heuristics which can improve on the efficiency of these algorithms at the cost of accuracy.
It is also possible to express sequence alignments with probabilistic models such as Hidden Markov Models (HMMs), which can easily be tailored towards specific needs and allow probabilistic interpretation of results, as exemplified in chapter 2.

In this thesis, the term (*read*) *mapping* refers to the process of aligning a set of reads obtained by sequencing DNA samples to a (set of) reference sequence(s). The term (read) *mapper* refers to an algorithm / software tool which performs a mapping.

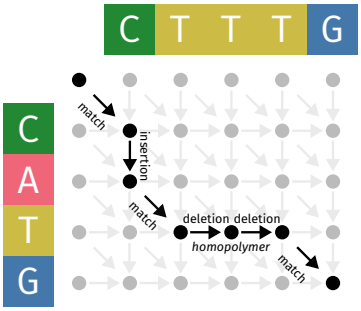



Figure 1.11: Schematic depiction of a dynamic programming table as commonly employed by sequence alignment algorithms. The score of each cell (circles) depends on its 3 predecessors (indicated by the arrows), and the best scoring alignment can be retrieved by backtracking from the bottom right cell (black arrows). Exemplary alignment between `CTTTG` and `CATG`.

[32] Smith, Waterman, et al., "Identification of common molecular subsequences", 1981.

[33] Needleman and Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins", 1970.

■ CIGAR Alignments are often represented either in columnar format (see Figure 1.12) or as Compact Idiosyncratic Gapped Alignment Report (CIGAR) strings, which are a run-length encoding of a specific set of alignment operations, listed in Table 1.2.
As an example, consider the alignment between the two sequences `GATTTTTC` and `GTTTC` in Figure 1.12.

| Operation | Description |
|---|---|
| `M` | match *or* mismatch |
| `I` | insertion |
| `D` | deletion |
| `N` | skipped reference region |
| `S` | soft clip |
| `H` | hard clip |
| `P` | pad |
| `=` | exact match |
| `X` | mismatch |

Table 1.2: CIGAR operations and their description, based on *SAMv1*. URL: https://samtools.github.io/hts-specs/SAMv1.pdf (visited on 2022-10-20)

A corresponding CIGAR representation is `1M1D3M2D1M`:
A match of length 1 (`G, G`), a deletion of length 1 (`A, `), a match of length 3 (`TTT, TTT`), a deletion of length 2 (`TT, `) and a match of length 1 (`C, C`).
We call a CIGAR string *expanded* if all run-lengths are exactly 1. For example, the CIGAR string `1M1D3M2D1M` is expanded to `1M1D1M1M1M1D1D1M` or — because the 1 is superfluous information — simply `MDMMMDDM`. This expanded form will become useful in chapter 2, as they can be interpreted as a sequence of states in a HMM.

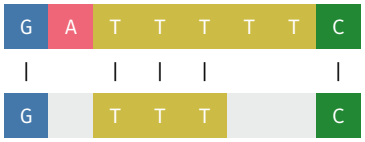



Figure 1.12: An exemplary alignment between sequences `GATTTTTC` and `GTTTC`.

■ SPLIT-READS Sometimes a read is not aligned as one contiguous unit. This can happen for several reasons. For example, the read could originate from a fusion of two different genes (from genomic regions very far apart or even on different chromosomes). Or, an individual's genome may contain a very large deletion compared to a reference genome. In any case, the read sequence will likely[34] be fragmented by the mapper and each fragment is aligned separately/individually. The mapper will however report which fragment alignments belong together, i.e. originate from the same read. We call such reads *split-reads*. See Figure 1.13 for an example.

Figure 1.13: A read (bottom) is mapped to a reference sequence (top). In this example, the read consists of 3 separate parts, each of which maps to a different region in the reference, and is hence labelled as a *split read*.

[34] But not necessarily. Because mappers use heuristics and can often be configured by the user, it can also happen that only part of a read is aligned, and the rest is discarded completely.

■ READ DEPTH Given an alignment or mapping of a set of reads to a reference sequence, the *read depth* at a certain locus is the total number of bases aligned to the locus (across all reads in the set). By this definition, insertions and deletions in reads (relative to the reference sequence) do not count towards the read depth. See Figure 1.14 for an example.

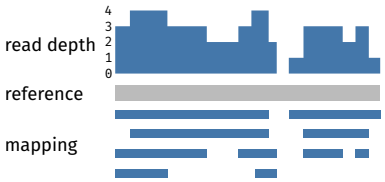



Figure 1.14: The read depth (top) at each locus of a reference sequence (middle) for a given read mapping (bottom).

■ MAPPING QUALITY Read mappers usually report a measure of *mapping quality* (`MAPQ`) for each sequence aligned to the reference. This is an estimate of the probability that the alignment is incorrect, given as a PHRED score. The exact calculation of mapping quality is specific of the mapper. However, in general, the following error sources should be considered, if possible:[35]

- *Contamination*: contamination of the sample with DNA from other sources.
- *Mapping heuristics*: the mapper may use heuristics to speed up the mapping process, which may lead to incorrect alignments.
- *Reference repetitiveness*: if the reference sequence contains repetitive regions, reads may align to multiple locations, leading to ambiguities.

[35] Li, Ruan, and Durbin, "Mapping short DNA sequencing reads and calling variants using mapping quality scores", 2008, (Supplement, Section 2).

## 1.5 *Variants*

Differences between sequences obtained from an individual's genome and a reference genome are called *variants*. Variants can be benign, but they can also be the cause of disease. For example, some variants simply influence the appearance of an individual (e.g. eye colour), while others can cause severe diseases, such as cancer.

There are three main categories of variants: Single Nucleotide Variants (SNVs), Insertion/deletions (indels) and structural variants. As the name suggests, SNVs are concerned with the substitution of a single nucleotide at a specific locus, whereas indels concern variants where multiple bases are added (inserted) and/or removed (deleted), and structural variants are concerned with changes in the structure of the genome. Indels and structural variants and are not clearly distinguished, but in principle arise from different underlying mechanisms. Types of structural variants include inversions, translocations and duplications (see Figure 1.15):

- *Inversions* are structural variants where a segment of the genome is reversed in-place.
- *Translocations* are structural variants where a segment of the genome is moved to a different location.
- *Duplications* are structural variants where a segment of the genome is duplicated.

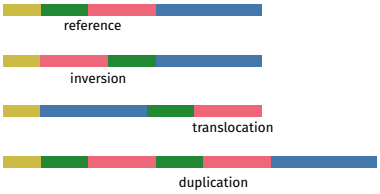



Figure 1.15: Schematic representation of inversions, translocations and duplications. The top row shows the reference genome, the rows below show the variant genome.

Larger deletions and duplications are commonly classified as Copy Number Variation (CNV): a change in the number of copies of a certain genomic region. Such changes in copynumber can have a significant effect on the production of proteins: If a gene is duplicated, the corresponding protein may be produced in higher quantities; if it is deleted, the protein may not be produced at all.

Any variant (in a coding region) can potentially have an effect on the resulting protein. Depending on the change introduced by the variant, its *effect* can be classified as follows:

- *Synonymous* variants do not change the amino acid sequence of the protein.
- *Missense* variants change the amino acid sequence of the protein.
- *Nonsense* variants introduce a premature stop codon, resulting in a truncated protein.
- *Readthrough* variants change a stop codon into a non-stop codon, resulting in the extension of the protein until the next stop codon is encountered.

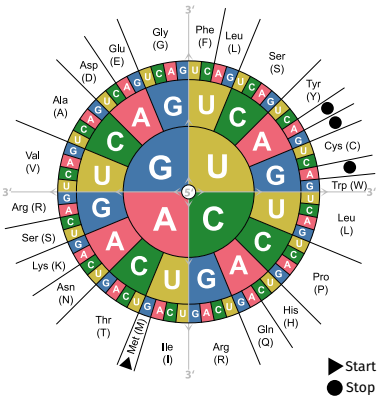



Figure 1.16: The "code sun" shows the mapping between codons and amino acids. It is read from the inside out, i.e. from the 5-prime end to the 3-prime end. Some amino acids are encoded by multiple codons, for example *glutamic acid* (abbreviated `Glu` or `E`) is represented by both G A G and G A A.

For example, a single nucleotide variant which changes the sequence G A G to G A A would be considered a synonymous variant, as both codons encode the same amino acid (glutamic acid). See Figure 1.16 for a mapping between codons and amino acids.

## 1.6 *Variant Calling with* `varlociraptor`

A *variant caller* is a software tool that detects variants in a given dataset, with respect to some kind of reference. There are a wide range of different variant callers available, each of them targeting different types of variants and/or different sequencing technologies, as well as supporting different scenarios, such as tumour-normal pairs or single-sample variant calling. Here, we will focus on `varlociraptor`,[36] a general purpose algebraic variant-caller, which we will extend in chapter 2 and subsequently use the extended version in chapter 3. One important thing to note here is that `varlociraptor` does not itself *discover* variants, but only calls variants, i.e. statistically evaluates the likelihood of any given candidate variant being a true variant. Therefore, a set of candidate variants has to be provided prior to calling variants with `varlociraptor`; this can for example be achieved by using a different *de-novo* variant caller, preferentially one which exposes parameters to control the sensitivity of the calls, for example `freebayes`.[37]

[36] Köster, Dijkstra, et al., "Varlociraptor: enhancing sensitivity and controlling false discovery rate in somatic indel discovery", 2020.

[37] Garrison and Marth, "Haplotype-based variant detection from short-read sequencing", 2012.

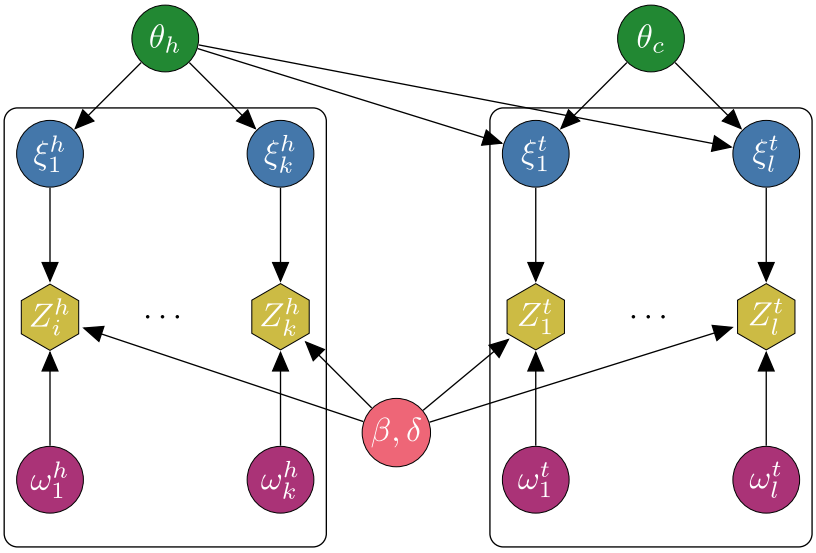



Figure 1.17: A schematic representation of `varlociraptor`'s statistical model for a tumour-normal pair scenario. $h$ corresponds to the healthy sample, $t$ to the tumour sample, and $c$ to the cancer cell population of $t$. $\theta_\bullet$ corresponds to the allele frequency, $\xi^\bullet_\bullet$ to the respective variant allele fragment, $Z^\bullet_\bullet$ to the observations, $\omega^\bullet_\bullet$ to whether the respective fragment is correctly mapped, and $\beta,\ \delta$ to general biases (e.g. strand bias) and artefacts. All variables apart from $Z^\bullet_\bullet$ are latent variables. Arrows from $\theta_h$ to $\xi^t_\bullet$ indicate that the allele frequency of sample $h$ influences the fragments of the other sample (*contamination*).

In its basic form, variant calling compares a set of reads from one or more samples to a reference sequence: Any differences between the mapped reads and the reference are considered candidate variants. How likely it is for a candidate variant to be a true variant then depends on the evidence available which supports the variant. To this end, the statistical (bayesian) model of `varlociraptor` takes into account the *allele frequency*, *contamination* of a sample by other samples, the *sampling bias*, the *mapping uncertainty*, the *typing uncertainty* as well as certain additional *biases* and *artefacts*, such as *strand bias*[38] and *read orientation bias*[39].

A schematic model for a two-sample case where one sample contaminates the second is shown in Figure 1.17[40].
Here, for each variant, $\xi^\bullet_i \sim \text{Bernoulli}(\alpha\theta_c\tau + (1-\alpha)\theta_h\tau)$, where $\theta_h$ and $\theta_c$ denote the (unknown, latent) allele frequencies of the variant in the healthy ($h$) and cancer ($c$) cell populations, $\alpha$ denotes the contamination rate of $c$ by $h$, and $\tau$ denotes the sampling bias. Similarly, $\omega^\bullet_i \sim \text{Bernoulli}(\pi_i)$, where $\pi_i$ denotes the mapping uncertainty (cf. `MAPQ`, section 1.4). Variables

[38] The variant mostly occurs on either strand.

[39] The variant mostly occurs on either forward or reverse reads.

[40] *varlociraptor presentation* (`https://slides.com/johanneskoester/towards-a-unified-theory-of-variant-calling-500a11`, visited on 2023-11-27)

$\beta$ and $\delta$ correspond to general biases and artefacts, respectively. Finally, $Z_{\bullet}^{\bullet}$ correspond to the observations, such as read strand and orientation information or the actual read sequences themselves. Then, the probability of observing a certain observation given the latent allele frequencies and biases is specified as $\mathbb{P}(Z_i^{\bullet} \mid \theta_h, \theta_c, \beta, \delta)$, and allows deriving posterior probabilities for *events* defined in terms of allele frequency combinations (and biases and prior knowledge).

```yaml
samples:
  healthy:
    resolution: 0.1
    universe: "0.0 | 0.5 | 1.0 | ]0.0,0.5[ | ]0.5,1.0["
  tumour:
    resolution: 0.01
    universe: "[0.0,1.0]"
    contamination:
      by: healthy
      fraction: 0.25
events:
  germline:         "healthy:0.5 | healthy:1.0"
  somatic_healthy:  "healthy:]0.0,0.5[ | healthy:]0.5,1.0["
  somatic_tumour:   "healthy:0.0 & tumour:]0.0,1.0]"
```

Figure 1.18: Example varlociraptor scenario definition for a tumour-normal pair.

Take for example the scenario in Figure 1.18[41]. Here, we define two samples, `healthy` and `tumour`, and three events, `germline`, `somatic_healthy` and `somatic_tumour`. The attributes `universe` and `resolution` define the allele frequency domain for each sample, and the resolution with which the domain is discretized. The events are then defined in terms of the allele frequencies; for example, the `somatic_tumour` event requires the (variant) allele frequency of the `healthy` sample to be 0 and the allele frequency of the `tumour` sample to be larger than 0. For each of the events, a posterior probability is calculated.

During the variant calling process, `varlociraptor` will re-align reads to the reference using a pairHMM (see chapter 2) to obtain probabilities of reads stemming from the reference or the variant allele. It is at this point that the modifications to the pairHMM to support ONT sequencing data in chapter 2 will become relevant.

[41] ibid. (https://slides.com/johanneskoester/towards-a-unified-theory-of-variant-calling-500a11, visited on 2023-11-27)

# 2 *HomopolyPairHMM*

In section 1.4 we gave an overview of common algorithms for pairwise sequence alignment. Most of the algorithms presented rely on a scoring scheme to find the best scoring alignment (out of all possible alignments, or at least those within a certain edit distance).

In this chapter we describe HMMs and their extension to pairwise sequence alignment named PHMMs (as introduced in Durbin et al. [1998][1]) and further extend these to explicitly model homopolymer errors (see subsection 1.2.1). This allows for a probabilistic view of the alignment problem, enabling e.g. Bayesian inference using the results.
More specifically, the HMM machinery allows answering the following questions:

1. What is the probability that two sequences are related?

2. What is the most probable alignment for two sequences?

Here we are mainly interested in the answer to the first question, as it is especially useful when trying to determine whether it is more likely for a read to originate straight from the reference sequence or to support a variant introduced into the reference sequence (as exemplified in Figure 2.1).

As described in subsection 1.2.1, ONT sequencing data exhibits systematic errors, most notably homopolymer errors, where the length of a run of identical bases in a base-called sequence differs from its actual length. In standard alignment algorithms, such homopolymer errors are not given any special treatment. Rather, they are viewed as regular gaps instead. However, because the main bulk of errors in ONT sequencing data are homopolymer errors,[2] it is important not to conflate them with indels in general.

The PHMM presented here is designed to work with DNA sequences, not with raw signal data as initially obtained from the sequencer. Translating the raw signal into DNA sequences (*"base calling"*) is a non-trivial task in itself, and is covered by dedicated existing methods such as `guppy`, `bonito`[3], `dorado`[4], `DeepNano`[5] or `Lokatt`.[6] Even though the accuracy of these tools is constantly improving, they still do not resolve homopolymer errors perfectly, which is why it is important to explicitly include them in the model.

Assuming perfect base calling were possible in the future, the model presented here would still be useful for older sequencing data that was base called with less accurate or outdated versions of tools. It is often not feasible to re-do base calling of the data, either because of resource constraints or because the original raw signal is no longer available.

[1] Durbin et al., *Biological sequence analysis: probabilistic models of proteins and nucleic acids*, 1998.

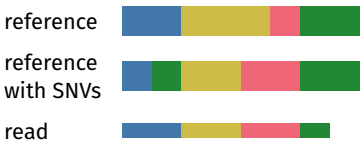



Figure 2.1: A reference sequence (top row), the same reference sequence with two single nucleotide variants (middle row) and a sequencing read (bottom row). A PairHMM allows determining the probabilities of the sequencing read being related to the original reference and the variant reference.

[2] Delahaye and Nicolas, "Sequencing DNA with nanopores: Troubles and biases", 2021.

[3] *bonito* (`https://github.com/nanoporetech/bonito`, visited on 2023-11-09)

[4] *dorado* (`https://github.com/nanoporetech/dorado`, visited on 2023-11-09)

[5] Boža, Brejová, and Vinař, "DeepNano: deep recurrent neural networks for base calling in MinION nanopore reads", 2017.

[6] Xu et al., "Lokatt: A hybrid DNA nanopore basecaller with an explicit duration hidden Markov model and a residual LSTM network", 2022.

■ Related Work The flowgram-to-string alignment algorithm described by M. Martin and Rahmann [2013][7] too addresses homopolymer errors. However, flowgrams differ from base-called DNA sequences in that they consist of pairs of nucleotide and fractional signal intensity values. Potentially, one could collapse DNA sequences into flowgrams with integer intensities and apply the flowgram-to-string alignment algorithm to them. While this would result in an alignment, it would not report the probability of the alignment (but a score, based on heuristics), in contrast to the method presented in this chapter.

[7] M. Martin and Rahmann, "Aligning flowgrams to DNA sequences", 2013.

While `Lokatt`[8] is a base caller, not an aligner, it is still of note here, as it uses an Explicit-Duration HMM (EDHMM) instead of a standard HMM to model homopolymers. In standard HMMs, states with self loops always have geometrically distributed durations, while in EDHMMs, an explicit duration model can be described. In this context, this is useful when the assumption of homopolymer run lengths being geometrically distributed is not valid. The specific distribution of homopolymer run lengths will depend on both the kind of nanopore and chemistry involved,[9] as well as the base caller that is used to translate the raw signal into DNA sequences[10]. However, we find that the geometric distribution is a reasonable assumption for homopolymer run lengths in ONT sequencing data *that has been base-called already* using `guppy (5.0.7)`, as we show in section 2.5.

[8] Xu et al., "Lokatt: A hybrid DNA nanopore basecaller with an explicit duration hidden Markov model and a residual LSTM network", 2022.

[9] Ibid., 2022.

[10] Some base callers only call homopolymer stretches of up to length 5 (Sarkozy, Jobbágy, and Antal, "Calling Homopolymer Stretches from Raw Nanopore Reads by Analyzing k-mer Dwell Times", 2018).

## 2.1 Hidden Markov Models

Before we introduce the HpHMM, we first give a brief overview of HMMs and PHMMs.

We define a HMM as a 5-tuple $(S, A, O, B, \pi)$ where

- $S = \{S_1, \ldots, S_N\}$ is a set of $N$ states,
- $A = (a_{ij}),\ 1 \leq i, j \leq N$ is a transition probability matrix with

  $$a_{ij} = P(s_{t+1} = S_j \mid s_t = S_i) =: p_{i \to j}$$

  being the probability to transition from state $S_i$ to state $S_j$.
  It is subject to $\sum_{j=1}^{N} a_{ij} = 1\ \forall i \in 1, \ldots, N$, i.e. outgoing transition probabilities must sum to 1 for each state,
- $O = \{O_1, \ldots, O_M\}$ is a set of $M$ possible outcomes/observations,
- $B = (b_{ik}),\ 1 \leq i \leq N, 1 \leq k \leq M$ is an emission probability matrix with

  $$b_{ik} = P(o_t = O_k \mid s_t = S_i) =: e_{S_i}(O_k)$$

  being the probability to observe $O_k$ in state $S_i$.
  It is subject to $\sum_{k=1}^{M} b_{ik} = 1\ \forall i \in 1, \ldots, N$, i.e. observation probabilities must sum to 1 in each state,
- $\pi = (\pi_i),\ 1 \leq i \leq N$ is an initial state probability distribution with

  $$\pi_i = P(s_1 = S_i)$$

  being the probability that the model starts in state $S_i$.
  It is subject to $\sum_{i=1}^{N} \pi_i = 1$, i.e. initial state probabilities must sum to 1.

This definition makes the following implicit assumptions:

1. State transitions only depend on the current state and do not depend on the current time step $t$ and
2. the current observation is independent of previous observations.

Therefore, the shorthand forms $p_{x \to y}$ and $e_x(y)$ drop any reference to time step $t$; their subscripts and arguments $x$ and $y$ are either indices into the corresponding set (e.g. $p_{i \to j} = a_{ij} = P(s_{t+1} = S_j \mid s_t = S_i)$) or refer to elements of sets/sequences directly (e.g. $p_{S_1 \to S_2} = a_{12} = P(s_{t+1} = S_2 \mid s_t = S_1)$), depending on context.

### 2.1.1 *Forward algorithm*

Given an instance $\mathcal{M}$ of a HMM and an observed sequence $X = x_1 x_2 \cdots x_L \in O^L$ with length $L$, we can calculate the probability $P(X \mid \mathcal{M})$ that sequence $X$ was generated with the model by summing over all possible state sequences of length $L$[11]:

$$P(X \mid \mathcal{M}) = \sum_{S' \in S^L} P(X \mid S', \mathcal{M}) \cdot P(S' \mid \mathcal{M}) \ .$$

[11] This assumes that no state is silent, i.e. for each state of the state path there is a non-zero probability to observe a symbol.

However, the number of operations needed to calculate this probability scale exponentially with the length of $X$, i.e. are in $O(N^L)$ (because there are $N$ states, the observed sequence has length $L$ and each possible state sequence of length $L$ has to be considered).

Instead, we define

$$\alpha_i(t-1) = P(x_1 x_2 \cdots x_{t-1}, s_{t-1} = S_i \mid \mathcal{M})$$

as the probability of the sequence $x_1 x_2 \cdots x_{t-1}$ ending at state $S_i$ given the model $\mathcal{M}$.

Then the recursion

$$\alpha_j(t) = e_j(x_t) \cdot \sum_{i=1}^{N} \alpha_i(t-1)\, p_{i \to j}$$

with base cases $\alpha_{\mathcal{B}}(0) = 1$ and $\alpha_j(0) = 0, \ \forall j \neq \mathcal{B}$ lends itself to dynamic programming and allows calculating

$$P(X \mid \mathcal{M}) = \sum_{i=1}^{N} \alpha_i(L) p_{i \to \mathcal{E}}$$

in $O(N^2 L)$ time and $O(NL)$ space (where $\mathcal{B}$ and $\mathcal{E}$ correspond to implicit *Begin* and *End* states, respectively).

In order to avoid numerical instabilities (products of probabilities get very small very quickly), calculations are carried out in log-space:

$$\ln(\alpha_j(t)) = \ln(e_j(x_t)) + \operatorname{lnsumexp}_{i=1}^{N} \left(\ln(\alpha_i(t-1)) + \ln(p_{i \to j})\right)$$

where

$$\operatorname{lnsumexp}_{i=1}^{N}(q_i) = q_\tau + \operatorname{ln1p}\left(\sum_{\substack{i=1,\\ i \neq \tau}}^{N} \exp(q_i - q_\tau)\right)$$

with $\tau = \operatorname{argmax}_i(q_i)$ (and $q_\tau = \max_i(q_i)$). The summation of the terms $\exp(q_i - q_\tau)$ itself should also be performed in a numerically stable manner, e.g. by using 2Sum, Kahan summation or similar algorithms.[12]

Performing calculations in logarithmic space has the added benefit of converting computationally expensive products into sums, at the expense of having to exponentiate in lnsumexp. For this exponentiation, reasonably accurate approximations for specific argument domains exist, such as `fastexp`.[13] Depending on CPU architecture, both lnsumexp with `fastexp` and `fastexp` itself can be sped-up (by factors of $2, 4, 8$ or $16$) by means of Single Instruction Multiple Data (SIMD) instructions, since all the operations needed have SIMD counterparts. This can either happen automatically during compilation (depending on the specific compiler's and the target CPU's properties) or manually using `SIMD` intrinsics.

[12] Muller et al., *Handbook of floating-point arithmetic*, 2018, section 5.3.

[13] Kopczynski, "Resource-Constrained Analysis of Ion Mobility Spectrometry Data", 2017, chapter 2.

## 2.2 *PHMM for sequence alignment*

PHMMs are an extension of regular HMMs, where emissions are *pairs* of symbols instead of single symbols, i.e. correspond to a sequence *alignment* between two sequences $X \subset O^{L_1}, Y \subset O^{L_2}$. This requires one additional dimension in the search space, leading to a runtime of $O(N^2 L_1 L_2)$ for the forward algorithm. Because gapped sequence alignments allow insertions and deletions, sequences do not have to be traversed in lock-step. Indels are represented with a special gap character $-$ which is added to the alphabet, augmenting the set of possible observations.

A default formulation of a PHMM (Figure 2.2) for sequence alignment[14] consists of three states (excluding implicit *Begin* and *End* states $\mathcal{B}$ and $\mathcal{E}$), one for a match or mismatch (state $M$) and one for a gap in either of the sequences (states $G^X$ and $G^Y$).

The states and observations are defined as $S = \{M, G^X, G^Y\}$ and $O = (\Sigma \cup \{-\})^2 \setminus \{(-, -)\}$, where $-$ denotes a gap. From state $M$, there is a small[15] probability $\delta$ to transition to either of the gap states or the complementary probability $1 - 2\delta$ to stay in $M$. In the gap states, there is a small[16] probability $\epsilon$ to stay in the corresponding gap state or to transition to the match state with probability $1 - \epsilon$.

[14] Durbin et al., *Biological sequence analysis: probabilistic models of proteins and nucleic acids*, 1998.

[15] In general, alignment matches are more likely than gaps.

[16] Usually $\epsilon < \delta$.

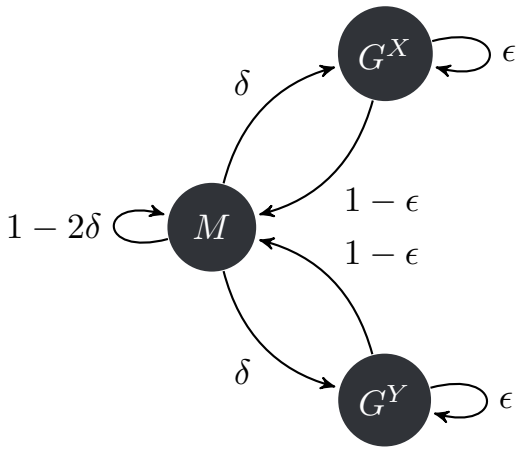



Figure 2.2: Default pair-HMM (without explicit begin and end states), with symmetric transition probabilities with respect to gap states $G^X$ and $G^Y$. Transitions between $G^X$ and $G^Y$ are not allowed by default.

For the PHMM for sequence alignment in Figure 2.2, the recurrence in

the forward algorithm can be stated explicitly as follows:

$$
\begin{aligned}
\alpha_M(i,j) &= e_M(x_i, y_j) \sum \begin{cases} p_{M\to M} & \alpha_M(i-1, j-1) \\ p_{G^X\to M} & \alpha_{G^X}(i-1, j-1) \\ p_{G^Y\to M} & \alpha_{G^Y}(i-1, j-1) \end{cases} \\
\alpha_{G^X}(i,j) &= e_{G^X}(x_i, -) \sum \begin{cases} p_{M\to G^X} & \alpha_M(i-1, j) \\ p_{G^X\to G^X} & \alpha_{G^X}(i-1, j) \end{cases} \\
\alpha_{G^Y}(i,j) &= e_{G^Y}(-, y_j) \sum \begin{cases} p_{M\to G^Y} & \alpha_M(i, j-1) \\ p_{G^Y\to G^Y} & \alpha_{G^X}(i, j-1) \end{cases}
\end{aligned}
$$

The sum notation $\sum \begin{cases} x \\ \vdots \\ y \end{cases}$ is equivalent to $(x + \cdots + y)$, but allows for a more dense/grouped representation of the recurrence.

For reasonable semantics, emission probabilities for a nucleotide paired with a gap have to be zero in state $M$, while in gap states $G^X, G^Y$, emission probabilities for a pair of matched nucleotides have to be zero. All other emission probabilities are specific to the characteristics of actual data, in this context to the species (homo sapiens), sequencing technology (e.g. ONT, Illumina), sequencing device and library preparation as well as whether the whole genome or only specific regions (e.g. exons) were targeted.
Specific details regarding parameter estimation are discussed in section 2.4.

## 2.3 *HpHMM for homopolymer aware sequence alignment*

We now describe the HomopolyPairHMM, a PHMM which explicitly models homopolymer errors as occurring in e.g. ONT or Pacific Biosciences (PacBio) data. In this context, homopolymer errors can be thought of as a special case of gaps, where the inserted or deleted parts of a sequence consist of the repetition of one *unique* base. We will refer to these as *"hops"* (akin to *"gaps"*) and use $\frown$ to symbolize them. In contrast to gaps, hops are never independent of the preceding base — any homopolymer run requires at least one match which it can continue in either $X$ or $Y$. Therefore, the singular match state $M$ from the default model is split into one match state $M_\sigma$ for each base $\sigma \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, and each of these match states is connected to a pair of *base specific* hop states $H^X_\sigma, H^Y_\sigma$. This extension of the default model assumes that elongating a homopolymer by one has constant probability, i.e. homopolymer length counts follow a geometric distribution. We will discuss this assumption in section 2.5, and also refer to Delahaye and Nicolas [2021][17] for a more detailed discussion of homopolymer errors in ONT sequencing data. For an exemplary empirical distribution of homopolymer length counts, see Figure 2.8. Figure 2.3 shows a graph representation of the HpHMM's states and transitions.

[17] Delahaye and Nicolas, "Sequencing DNA with nanopores: Troubles and biases", 2021.

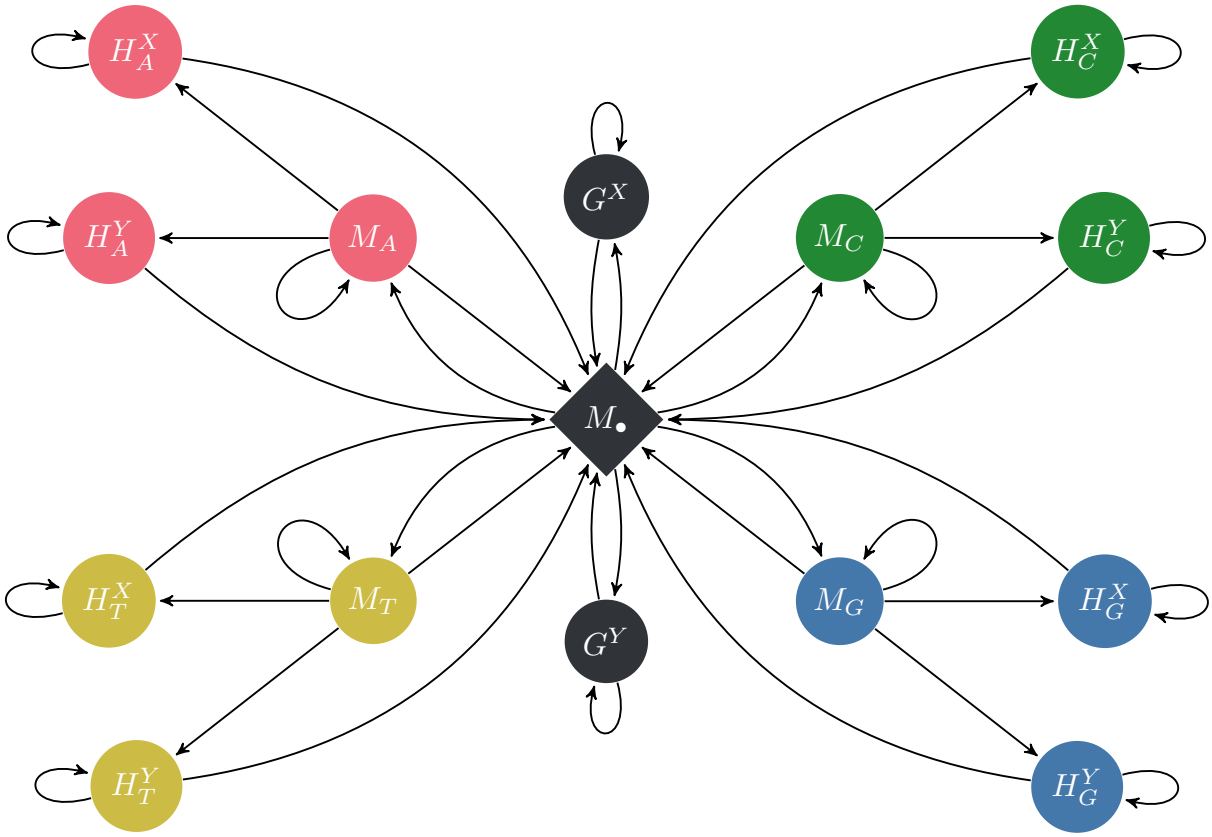



Figure 2.3: The HomopolyPair-HMM for homopolymer-aware sequence alignment. The central node $M_\bullet$ is a pseudo node introduced to reduce visual clutter. It virtually connects match states $M_\sigma$ with each other and with gap states $G^X$ and $G^Y$, as well as each hop state to all match states (excluding the match state the hop is associated with, i.e. $H_\sigma^\bullet \to M_\sigma$), but not to the gap states. Nodes are colour-coded by base, highlighting that hops can only occur after a corresponding match of the same base.

The recurrence for the forward algorithm is then specified by:

$$\alpha_{M_\sigma}(i,j) \quad = e_{M_\sigma}(x_i, y_j) \sum \begin{cases} p_{M_s \to M_\sigma} & \alpha_{M_s}(i-1, j-1)\ \forall s \in \Sigma \\ p_{G^X \to M_\sigma} & \alpha_{G^X}(i-1, j-1) \\ p_{G^Y \to M_\sigma} & \alpha_{G^Y}(i-1, j-1) \\ p_{H^X_s \to M_\sigma} & \alpha_{H^X_s}(i-1, j-1)\ \forall s \in \Sigma \setminus \{\sigma\} \\ p_{H^Y_s \to M_\sigma} & \alpha_{H^Y_s}(i-1, j-1)\ \forall s \in \Sigma \setminus \{\sigma\} \end{cases} \qquad \forall \sigma \in \Sigma$$

$$\alpha_{G^X}(i,j) \quad = e_{G^X}(x_i, -) \sum \begin{cases} p_{M_s \to G^X} & \alpha_{M_s}(i-1, j)\ \forall s \in \Sigma \\ p_{G^X \to G^X} & \alpha_{G^X}(i-1, j) \end{cases}$$

$$\alpha_{G^Y}(i,j) \quad = e_{G^Y}(-, y_j) \sum \begin{cases} p_{M_s \to G^Y} & \alpha_{M_s}(i, j-1)\ \forall s \in \Sigma \\ p_{G^Y \to G^Y} & \alpha_{G^Y}(i, j-1) \end{cases}$$

$$\alpha_{H^X_\sigma}(i,j) \quad = e_{H^X_\sigma}(x_i, \frown) \sum \begin{cases} p_{M_\sigma \to H^X_\sigma} & \alpha_{M_\sigma}(i-1, j) \\ p_{H^X_\sigma \to H^X_\sigma} & \alpha_{H^X_\sigma}(i-1, j) \end{cases} \quad \forall \sigma \in \Sigma$$

$$\alpha_{H^Y_\sigma}(i,j) \quad = e_{H^Y_\sigma}(\frown, y_j) \sum \begin{cases} p_{M_\sigma \to H^Y_\sigma} & \alpha_{M_\sigma}(i, j-1) \\ p_{H^Y_\sigma \to H^Y_\sigma} & \alpha_{H^Y_\sigma}(i, j-1) \end{cases} \quad \forall \sigma \in \Sigma$$

with base cases depending on setting (local, global, semi-global), e.g. $\alpha_\bullet(0,0) = 0$ and $\alpha_{M_\sigma}(0,0) = {}^1/_{|\Sigma|}\ \forall \sigma \in \{A, C, G, T\}$.

## 2.4 *Parameter estimation*

In order to make use of a HMM or PHMM, its parameters have to be tuned to the problem one wishes to solve. This can be achieved in different ways: For simple, human-interpretable models, domain expertise and other prior information can be used to manually assign transition and emission probabilities. For example, in the case of the default pair-HMM (Figure 2.2), there are only two free parameters for configuring transition probabilities. However, for more complex models, manually configuring parameters quickly becomes intractable. In that case, parameter estimation or *training* of the model is required. This training can either be done supervised or unsupervised, where unsupervised training requires only sequences of observations, while supervised training requires information about the states the observations were generated from.

■ Unsupervised In the unsupervised case the Baum-Welch algorithm[18] is employed, an expectation-maximization algorithm for HMMs. Depending on the specific model, the Baum-Welch algorithm may not be feasible due to the number of parameters to be estimated as well as the number of input observation sequences and their lengths[19]. There are also no guarantees on global optimality — the estimation may converge to any local optimum during training. To get a more robust or better estimate, common strategies include repeating the training multiple times with different starting parameters. It is also possible to overfit the model, i.e. unintentionally modelling properties that are specific to the dataset used for training, but that do not correspond to the intended error model (which should be applicable to other datasets as well).

[18] Baum et al., "A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains", 1970.

[19] Khreich et al., "On the memory complexity of the forward–backward algorithm", 2010 describe a $O(N)$ memory and $O(N^2 L)$ time (for one iteration) algorithm, where $N$ is the number of states and $L$ the length of the observation sequence.

**Supervised** In the supervised case, state paths for sequences of observations have to be known. Then it is possible to give maximum-likelihood estimates of transition probabilities as follows:[20]

$$\hat{p}_{i \to j} = \frac{\#[S_i \to S_j]}{\sum_{s \in S} \#[S_i \to s]},$$

where $\#[S_i \to S_j]$ is the number of occurrences of transitions from state $S_i$ to state $S_j$ in the training data. Similarly, emission probabilities can be estimated as:

$$\hat{e}_i(k) = \frac{\#[S_i \uparrow k]}{\sum_{o \in O} \#[S_i \uparrow o]},$$

where $\#[S_i \uparrow k]$ is the number of occurrences of emission $k$ in state $S_i$ in the training data. In other words, the maximum likelihood estimates are the fraction of the number of times a specific event occurs in the training data and the total number of events (relating to the same state) in the training data.

[20] Vidyasagar, *Hidden markov processes: Theory and applications to biology*, 2014, p. 107-108.

## 2.5 Heuristic estimation of HPHMM parameters

Because fast, reliable and established (score based) read mappers are available, and read mapping is a very common part of many bioinformatics pipelines, it is often possible to make use of existing alignments as a starting point for more specialized models. In the case of the HpHMM, we can use alignments from fast mappers to estimate transition probabilities for the HpHMM. However, we need to be careful, as fast mappers are not generally homopolymer-aware and thus do not distinguish between gaps and hops. We therefore present a simple heuristic to identify hops in existing alignments, partitioning gaps into hops and gaps. With these changes, we can then estimate transition probabilities for the HPHMM using the maximum likelihood method described in section 2.4.

### 2.5.1 Gaps and Hops

We introduce the following heuristic to partition gaps into gaps and hops: If

1. the alignment has a deletion of length $l$ starting at position $i$
2. the deletion is enclosed by two matches
3. the reference sequence from $i \ldots i + l$ is a homopolymer
4. the pre- or succeeding base in the reference is the same base as the homopolymer

then the deletion is re-classified as a hop[21]. For insertions, the roles of reference and read are switched. As an example, in Figure 2.4, a homopolymer run of four T s in the reference is reduced to a run of two T s in the read sequence, corresponding to a hop of size 2.

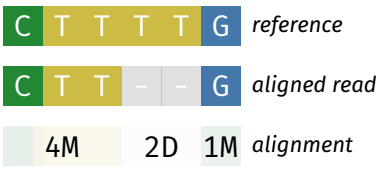



Figure 2.4: Alignment of CTTG to CTTTTG with a gap of size 2 to be reclassified as a hop.

[21] This hop may still be a regular gap, but gaps are expected to be rare.

■ Ambiguous paths Special care is needed for estimating transition parameters that include hop-states, since it is not possible to differentiate between state sequences consisting of matches and hops of the same base ($\sigma$). For example, the state sequences `MHMH` and `MMHH` both correspond to an alignment between $\sigma\sigma$ and $\sigma\sigma\sigma\sigma$. Therefore, assume $p_{M\to H} = p_{H\to H}$, i.e. the probability to start a hop has to be the same as elongating an existing hop.
By considering only homopolymer length differences of 1, which are likely to be due to technical artefacts only, such ambiguous transition sequences (Figure 2.5) can be avoided altogether. Because hop lengths decay exponentially (Figure 2.8), disregarding any hop length differences larger than 1 does not impact the estimation of transition probabilities significantly. Alternatively, it is possible to fit a geometric distribution to the hop length counts and derive the respective transition probability as the fitted distribution's success probability parameter $p$ (cf. Figure 2.9).

*However*, assuming that grouping matches and hops (`MMHH`) is more beneficial to the aligner's score than interspersed matches and hops (`MHMH`), we can assume that most of the time, homopolymer alignments will be of the form `M+H+`. In that case, $p_{M\to H}$ and $p_{H\to H}$ do not necessarily have to be considered equal.

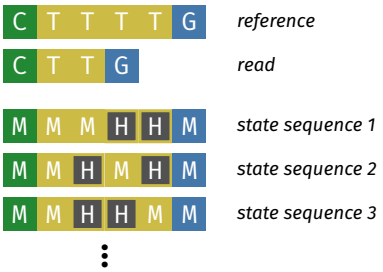



Figure 2.5: Possible state paths including hops for an alignment of `CTTG` to `CTTTTG`.

### 2.5.2 *Training dataset size*

Because different sequencing runs may exhibit different inherent properties, it is sensible to estimate HpHMM parameters individually for each dataset. As long as there are sufficiently many alignments available as labelled data (i.e. expanded CIGAR strings), the parameters can be estimated confidently. However, datasets can be very large (see Appendix A), and considering every single alignment of such datasets can lead to unacceptably long processing times. Subsampling the alignments is a reasonable way to mitigate the problem of excessive processing times.

To determine a lower bound of alignments needed to estimate the parameters, we follow estimation procedures for finite multinomial populations given in Tortora [1978][22]: Let $\alpha \in [0, 1]$ be the desired level of confidence, $k = 74$ the number of different transitions of the HpHMM model (cf graph representation or recurrence representation), $N' \in \mathbb{N}$ be the number of transitions in all alignments of the dataset, $p_i \in [0, 1]$ the probability estimate for transition $i \in \{1, \dots, k\}$, $\beta = \chi^{2^{-1}}(\alpha/k)$ the upper $\alpha/k$ quantile of a $\chi^2$ distribution with 1 degree of freedom and $d_i$ the absolute precision for the estimate of $p_i$ (with $d_i p_i$ then giving the relative precision). The estimated number of *transitions* $\hat{n}'$ needed to calculate transition probabilities is then given by

$$\hat{n}' = \left\lceil \max_i \frac{\beta N' p_i (1 - p_i)}{(d_i p_i)^2 (N' - 1) + \beta p_i (1 - p_i)} \right\rceil .$$

However, we are interested in the number of required *alignments* $\hat{n}$. To that end, we make the following observation: Assuming a constant read length $l \in \mathbb{N}$, there are approximately[23] $l - 1$ transitions in the expanded CIGAR string of each reads' alignment. Now, with $N \in \mathbb{N}$ being the number of *alignments* in the dataset, replace $N'$ with $N(l - 1)$ and subsequently divide

[22] Tortora, "A note on sample size estimation for multinomial populations", 1978.

[23] Soft- and hard-clips may reduce the number of relevant match, insertion and deletion events, see Table 1.2.

by $l - 1$ to obtain

$$\hat{n} = \left\lceil \frac{1}{l-1} \max_i \frac{\beta N(l-1)p_i(1-p_i)}{(d_i p_i)^2(N(l-1)-1) + \beta p_i(1-p_i)} \right\rceil$$

as the number of alignments required to confidently estimate HpHMM transition probabilities.

While read length can be assumed to be constant for Illumina sequencing, it is highly variable for ONT sequencing, violating the assumption of the number of transitions in a single alignment being constant. However, the average ONT sequencing read length is significantly larger than 100 bp and choosing $l = 100$ therefore leads to a conservative estimate of $\hat{n}$.

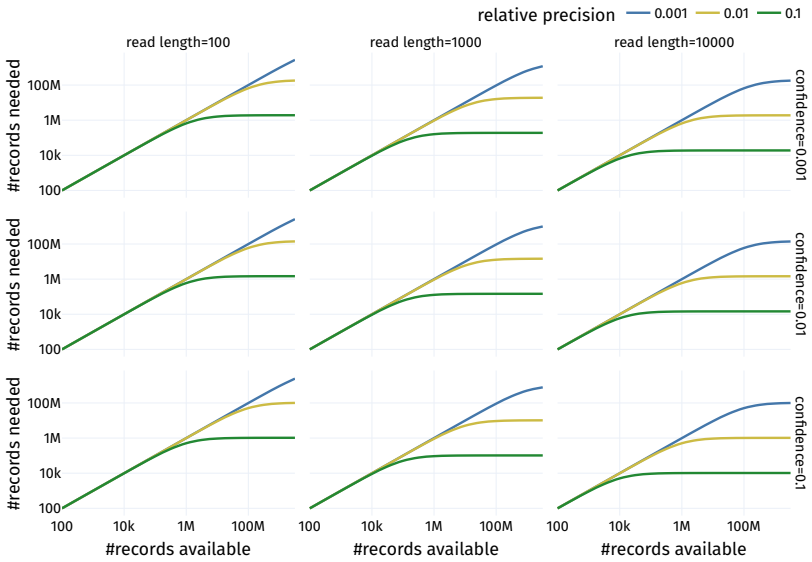



Figure 2.6: Minimum number of records needed to estimate the HpHMM transitions parameters, depending on confidence level (rows), precision (color), read length (i.e. number of transitions per record) (columns) and number of records available (x-axis).

Figure 2.6 shows the minimum number of records needed to estimate the HpHMM transitions parameters, depending on confidence level, precision, read length and number of records available. The estimate also depends on expected transition probabilities — for this application, we expect match to match state transition probabilities to be in the order of $1/|\Sigma_{\text{DNA}}|$ (0.25), while gap and hop extension probabilities tend to be in the range of $1 \times 10^{-1}$ to $1 \times 10^{-5}$ [24]. The expected probabilities have therefore been fixed to $\{0.25, 0.1, 0.01, 0.001, 0.0001, 0.00001\}$ to cover a range of reasonable values. As expected, the number of records needed to estimate the parameters increases with confidence and precision. Furthermore, the number of records needed decreases with increasing read length, since more transitions are observed per record.

[24] Values are from experience.

### 2.5.3 *Homopolymer errors in nanopore sequencing data*

In order to get an impression of real-world transition probabilities, we make use of data provided by the telomere-to-telomere consortium (subsection 1.1.1). The idea is to use nanopore sequencing reads from T2T-CHM13 and to align these against the CHM13 reference assembly in version `2.0` with the Y-chromosome excluded. Then, any differences between the

aligned reads and the reference (that was assembled using these reads) are largely due to technical issues / sequencing errors.

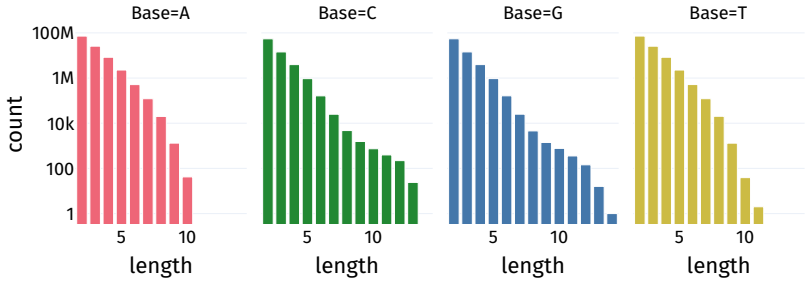
We will first give a short overview of both nanopore sequencing data and CHM13 reference, then derive statistics on the number of matches, gaps and hops from the alignment, and finally use these statistics to estimate transition probabilities for the HpHMM.

**Nanopore reads** Reads are obtained from `https://github.com/marbl/CHM13/blob/master/Sequencing_data.md#rel8-genome-dna` and filtered down to the subset of reads produced at the University of Nottingham. This subset was chosen because it is the smallest subset of reads produced in a single location which is still large enough to be representative of the whole dataset. It contains 1 402 404 reads with an average read length of 11 727 bp. After mapping these reads to the reference sequence using `minimap2`[25] with parameters `-x map-ont`, 1 053 311 alignment records remain (excluding unmapped, duplicate and QC-fail reads), with an average sequence length of 14 403 bp[26].

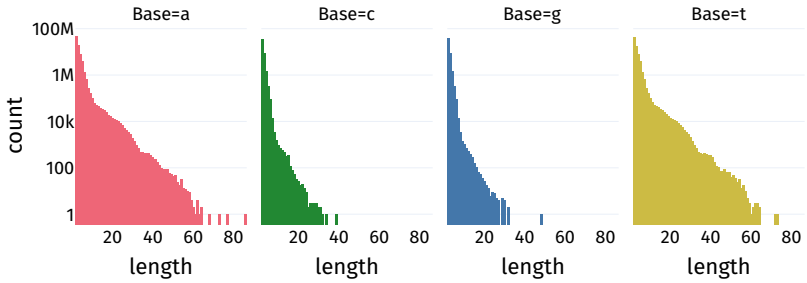
[25] Li, "Minimap2: pairwise alignment for nucleotide sequences", 2018.

[26] Assuming a confidence of $\alpha = 0.01$, a relative precision of $dp = 0.1$ and a read length of $l = 1000$ (which is on the shorter side for nanopore sequencing data), a random sample of 128 097 alignment records is sufficient to estimate the HpHMM's transition probabilities for this dataset.

**Reference sequence** The reference sequence is available at `https://s3-us-west-2.amazonaws.com/human-pangenomics/T2T/CHM13/assemblies/analysis_set/chm13v2.0_noY.fa.gz`.
As a rough orientation, Figure 2.7 shows the distribution of homopolymer lengths in the reference sequence, grouped by base. For non-masked bases, the longest homopolymer runs are of length 14, while for masked bases, the longest homopolymer run is a run of A of length 86. As a direct consequence, deletion hops are limited by the length of the longest homopolymer run (of the respective base) in the reference sequence.




(a) Non-masked bases.




(b) Masked bases.

Figure 2.7: Homopolymer length statistics for the CHM13 reference sequence. The x-axis corresponds to the length of the homopolymer run, the y-axis (logarithmic) to the number of occurrences of the respective length. The different colours correspond to the different bases. The upper figure shows the statistics for non-masked bases, the lower figure for masked bases.

From the alignment of the reads to the reference, we derive the num-

ber of matches, gaps and hops (per base) as described in subsection 2.5.1 (for unmasked bases only). The length differences in homopolymer-runs between reads and reference are then counted and grouped by base. Figure 2.8 gives an overview of homopolymer length difference counts, while Figure 2.9 shows the corresponding empirical probability mass as well as that of a geometric distribution fit.

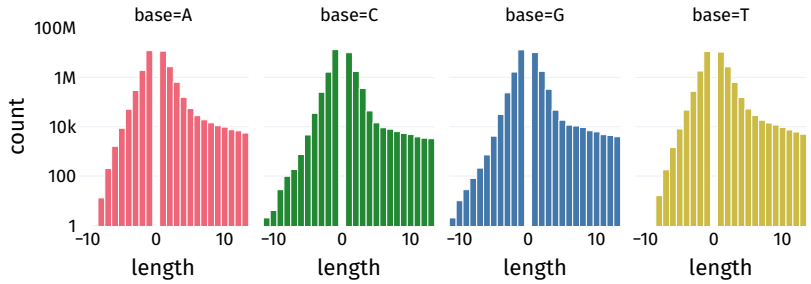



Figure 2.8: Histogram of counts of hops by base, i.e. homopolymer insertions and deletions where the preceding or succeeding base is the same as the homopolymer base. On the x-axis, the length of the insertion ($x > 0$) or deletion ($x < 0$) is given in bp. The y-axis (logarithmic) corresponds to the number of hops of the respective length.

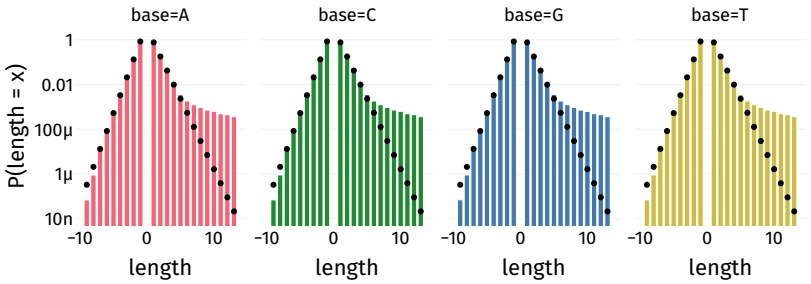



Figure 2.9: Analogous to Figure 2.8, but scaled such that the y-axis corresponds to the probability of a hop of the respective length, and with a geometric distribution fit indicated by black dots. In the case of deletions (length $< 0$), the fit is good; for insertions, the fit is only good for insertion lengths up to about 5.

For deletions (length $< 0$), the counts indeed decrease exponentially with increasing deletion length, while for insertions (length $> 0$) this only holds for insertion lengths up to about 5. Beyond that, it seems that there is a secondary effect that increases the probability of longer insertions. It is unclear whether the secondary effect is due to biological or technological issues or due to limitations during base-calling. However, for this dataset, only about $0.5\%$ of all insertion-hops are longer than $5\,\mathrm{bp}$. We argue that this is a small enough fraction to be ignored for the time being. As an avenue for future work, it would be interesting to investigate whether this secondary effect is also present in other datasets, and if so, whether it can be explained by biological or technological issues. If it is a technological issue, it might be possible to correct for it either during base-calling or at the alignment stage, by providing a more complex model which can account for this effect.

For the combination of read sequences and reference sequence described above, estimated transition probabilities for the HpHMM are shown in Figure 2.10, while the raw frequencies of state transitions are shown in Figure 2.11.

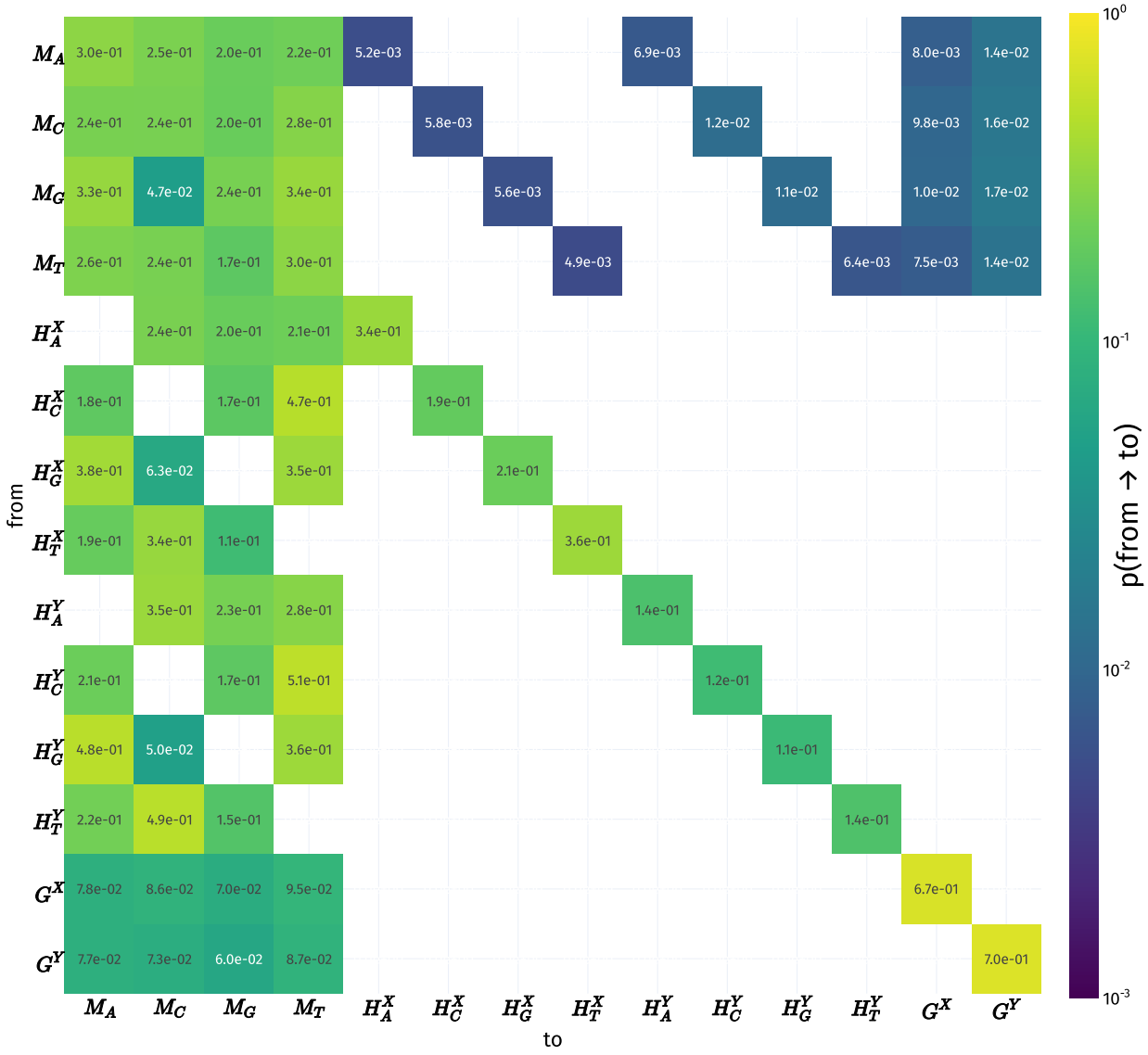



Figure 2.10: Estimated transition probabilities for the HpHMM for the CHM13 dataset. The y-axis corresponds to the state the transition originates from, the x-axis to the state the transition leads to. The colour of each cell corresponds to the estimated transition probability, where dark purple/blue corresponds to low probabilities and yellow to high probabilities. The colour scale is logarithmic. Empty cells correspond to transitions that are not part of the HpHMM model.

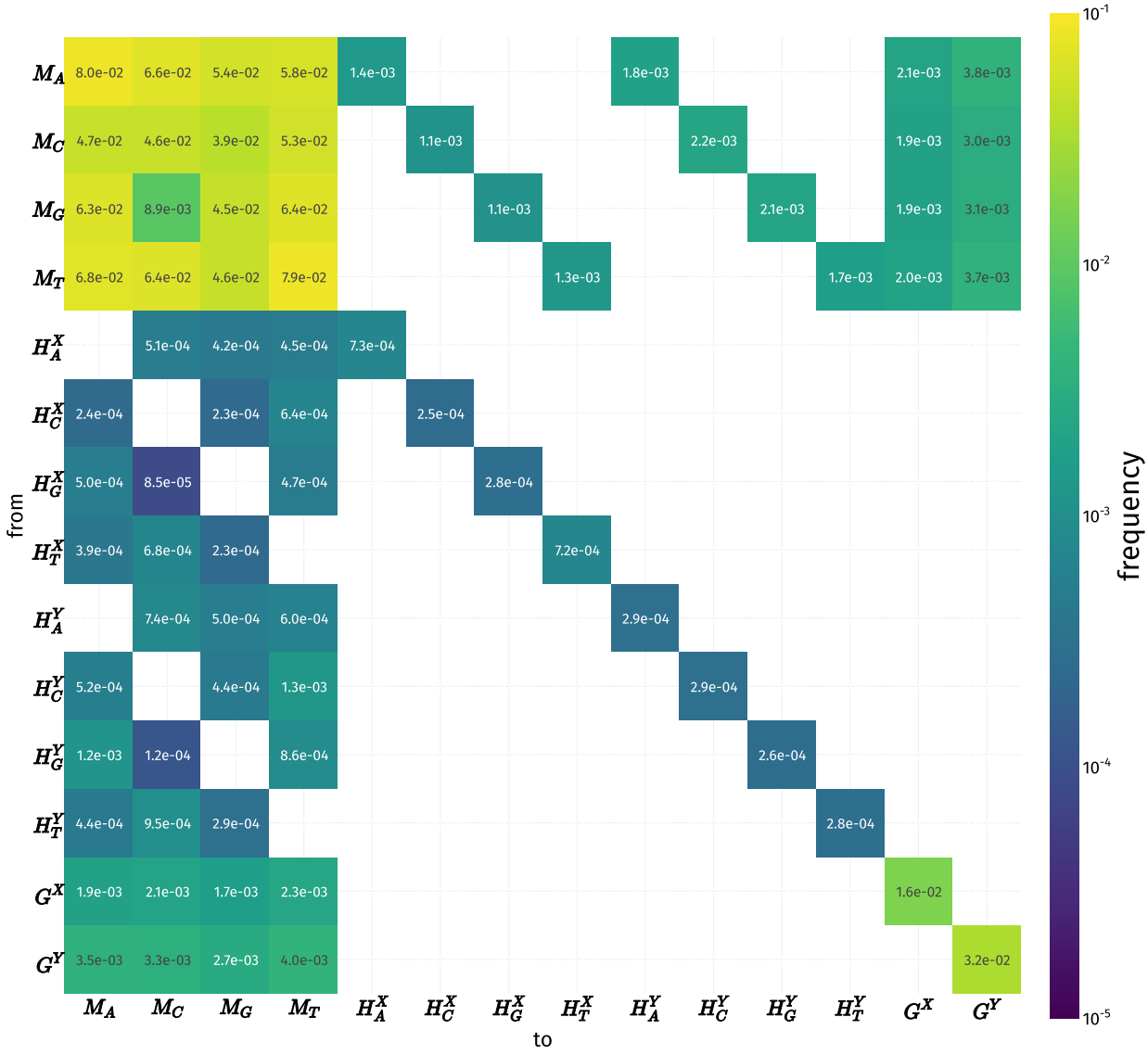



Figure 2.11: Frequencies of *(from, to)* state pairs occurring in the CHM13 dataset. The y-axis corresponds to the *from* state, the x-axis to the *to* state. The colour of each cell corresponds to the frequency of the respective state pair in the dataset, where dark purple/blue corresponds to low frequencies and yellow to high frequencies. The colour scale is logarithmic. Empty cells correspond to state pairs that are not part of the HpHMM model. The most infrequent state pair is $(H_G^X, M_C)$, while the most frequent state pair is $(M_A, M_A)$.

## 2.6 Discussion

In this chapter, we presented the *HomopolyPairHMM*, a pHMM which models homopolymer errors explicitly, instead of conflating them with regular gaps. This allows for more accurate alignments of nanopore data, especially suited for re-aligning existing alignments. It also allows for more accurate alignment probabilities, which can be used for downstream applications such as variant calling with `varlociraptor`.

We also presented a heuristic to identify hops in existing alignments, which allows for a maximum likelihood estimation of the HpHMM's transition probabilities, foregoing the need for unsupervised training. This is especially useful and convenient in existing pipelines, where it is often not feasible to re-train the model for each dataset, and which usually include a (fast) read mapping step anyway (whose results can be used to estimate the transition probabilities as outlined in this chapter).

We did not discuss the estimation of emission probabilities in this chapter, but it is possible to derive emission probabilities from existing alignments as well, analogously to the transition probabilities. Note that each base in the read sequence also has a base quality (assigned during sequencing /base-calling) associated with it, which can either be considered when deriving emission probabilities, or be incorporated into the variant calling process further downstream.

The design of the HpHMM is justified empirically by the observation that homopolymer errors generally follow a geometric distribution. While there also is a secondary effect that increases the probability of longer insertions, we argue that this effect is small enough to be ignored for the time being. In the future, it would be interesting to investigate whether this secondary effect is also present in other datasets, and if so, how it can be explained and addressed (e.g. by a more complex model).

Because existing alignments are used to estimate the HpHMM's parameters, and these alignments can be arbitrarily large, it is sensible to subsample the alignments to avoid excessive processing times. To that end, we derived a formula to estimate the number of alignments needed to estimate the HpHMM's parameters (with a given confidence and precision), based on multinomial population estimation.

As a proof of concept, we applied the HpHMM to nanopore sequencing data from the CHM13 reference genome and derived the transition probabilities for the HpHMM from the alignments of these reads to the reference.

# 3 *Detection of extrachromosomal circular DNA*

*The contents of this chapter have been published in [Tüns & Hartmann et al. 2022].*[1]

All chromosomes of the human genome are linear, i.e. the molecules have two distinct ends. The only exception common among all human cells is the DNA of the mitochondrion, a circular DNA molecule which is about 16kbp long.
Extrachromosomal circular DNA molecules are not part of the set of chromosomes of a cell. They are observed in conjunction with diseases such as cancer, i.e. likely correlated with pathogenicity[2]. They can potentially be used as biomarkers or to monitor disease progression.

However, no matter if the sequenced DNA molecule was (physically) linear or circular, the result of sequencing is still a set of *linear* sequences. Hence, sequences which stem from circular molecules are — at first glance — undistinguishable from those that stem from linear molecules. Naturally, the task here is to detect eccDNA (or any circular DNA molecules) from such a set of linear reads obtained through DNA sequencing.

It is important to note that, depending on sequencing technology, special care has to be taken during library preparation to actually be able to sequence circular molecules. In this chapter, we focus on a combination of ONT sequencing on samples prepared following a protocol for circular DNA enrichment.[3] This protocol ensures enrichment of circular molecules by rolling circle amplification and by digestion of linear molecules. We can therefore expect that *most* reads obtained using this method do stem from circular DNA, but cannot rule out that some linear molecules remain.

[1] Tüns et al., "Detection and validation of circular DNA fragments using nanopore sequencing", 2022.

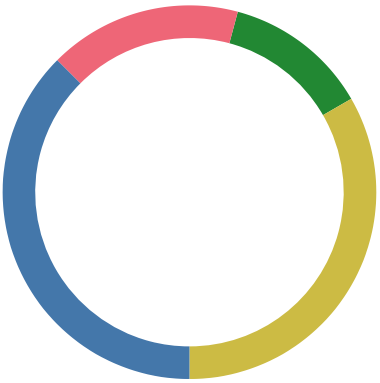


Figure 3.1: Exemplary schematic circular DNA molecule.

[2] Noer et al., "Extrachromosomal circular DNA in cancer: history, current knowledge, and methods", 2022.

[3] Henssen et al., "Purification and sequencing of large circular DNA from human cells", 2019.

## 3.1 *Detecting eccDNA in sequencing data*

Given a set of sequencing reads and a reference genome, the task is to detect circularisations of parts of the reference (or alternatively: detect those reads which stem from genomic eccDNA, and which, when assembled circularly, would result in a circle). To that end, we make use of the following two observations:

1. Almost all reads in the dataset stem from circular molecules, due to the sequencing protocol used.

2. Reads spanning circle junctions will likely be modelled as split reads by conventional read mappers. For such reads, one end maps to one

genomic region, while the rest of the read maps to a *different* genomic location, with a breakpoint separating the two read segments.

Therefore, one strategy is to find regions on the reference where the read depth is greater than some threshold $\theta \geq 0$ and whose start- and endpoints are connected / supported by split reads. Additionally, assumptions on the minimal and maximal region lengths can be imposed, potentially filtering out short, *linear* repetitive regions.
Figure 3.2 gives an overview of the steps performed to detect eccDNA.

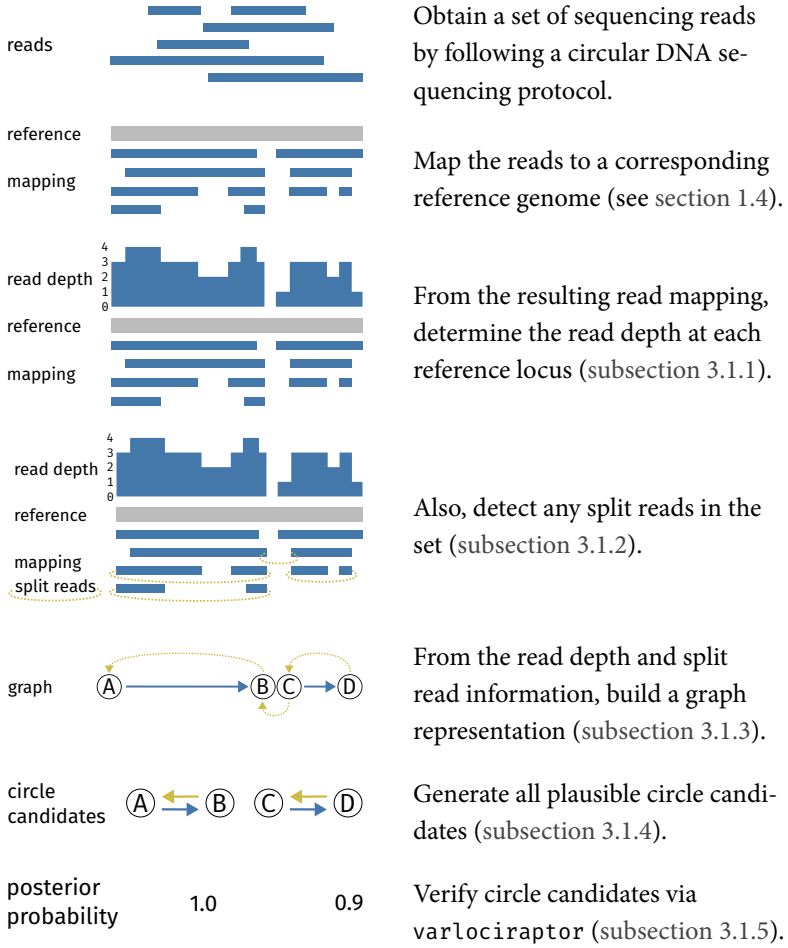



Figure 3.2: Overview of steps needed to identify circular DNA from ONT sequencing reads obtained using a circular DNA sequencing protocol.

### 3.1.1 *Calculating read depth*

To quickly calculate read depth for each position in the reference, we proceed as follows (akin to `mosdepth`[4]): Let $R \subset \Sigma^+$ be a multiset of mapped reads and $T \in \Sigma^+$ be a reference sequence of length $l := |T|$. Initialize an array $D$ of length $l$ with zeroes. For each aligned segment of each read, increment the value at the segment's start and decrement the value at the segment's end. The cumulative sum of $D$ (in-place calculation) then gives the approximate read depth at each locus of the reference $T$. For exact read-depth, insertions and deletions within a read's alignment have to be accounted for as well; this can be achieved by fragmenting a read into smaller "match segments" at each insertion or deletion of the read's alignment.

[4] Pedersen and Quinlan, "Mosdepth: quick coverage calculation for genomes and exomes", 2018.

```
def read_depth(R, l):                # O(|R| + l)
    D = [0] * l                      # O(l)
    for read in R:                   # O(|R|)
        for segment in read:         # o(|read|)
            D[segment.start] += 1    # O(1)
            D[segment.end] -= 1      # O(1)
    cumsum_inplace(D)                # O(l)
```

The overall time needed is linear in the number of reads and the length of the reference: $O(|R| + l)$, if the number of segments per read has a constant maximum limit[5]. Memory is $O(l)$. In practice, for the human genome with length of roughly 3 Gbp and using 8bit integers as counters[6], this amounts to $3 \cdot 10^9 \cdot 8\,\text{bit} \approxeq 2.79\,\text{GB}$ of memory.

[5] This is a reasonable assumption, as the alignment and scoring algorithms used during mapping will not report arbitrarily many segments per read. The number of segments rather depends on the read's length and quality/noisiness: Longer and/or noisier reads will exhibit more micro-segmentation than shorter and/or higher quality reads.

[6] Depending on the expected coverage, the number of bits used for the counters can of course be adjusted accordingly.

### 3.1.2 *Identifying split-reads*

For the identification of split-reads (or *chimeric* reads) from any given read-mapping in Sequence Alignment Map (SAM) format (or its binary counterparts BAM and CRAM), we make use of standard features of the specification: Because each entry in a SAM file corresponds to one alignment (in contrast to one read), reads whose parts map to multiple different loci are subsequently represented as multiple entries. Exactly one alignment entry in a split-read scenario is flagged as a *primary* alignment, while the remaining entries are flagged as *supplementary*.
If the mapper adheres to the SAM format specification including the SAM tag specification,[7] it may also optionally provide information in each entry's `SM` tag, which matches the following pseudo regular expression:

[7] *SAMtag*.

```
SA:Z:(rname,pos,strand,CIGAR,mapQ,NM;)+
```

where `rname` corresponds to the reference name, `pos` to the 1-based starting position of the alignment relative to the reference, `strand` indicates forward (`+`) or reverse (`-`) strand, `CIGAR` string of the respective alignment (see Table 1.2), `mapQ` the mapping quality and `NM` the number of differences (with respect to mismatches and indels) between sequence and reference. Each partial alignment belonging to the same split read contains the same information in its `SM` tag (albeit in potentially arbitrary order), describing the split alignment in its whole.

If a mapper does not make use of the `SM` tag, entries need to be grouped by their `QNAME` (query or read name) instead.

Because sequencing and mapping are error-prone processes, records which are likely erroneous, uninformative or disruptive are discarded according to the following criteria[8]: The read/alignment

[8] See SAM specification, section 1.4.2, bitwise flags.

- is a Polymerase Chain Reaction (PCR) or optical duplicate,
- is unmapped,
- failed (platform or vendor) Quality Control (QC) or
- is a secondary alignment (e.g. when a read maps ambiguously to multiple locations due to homologies or repeats in the reference).

This list of criteria is a common set of filters used in the field of sequencing data analysis and is not specific to the detection of eccDNA.

### 3.1.3 *Building a graph representation*

We define a directed graph $G = (V, E)$ where nodes $V$ represent genomic loci and edges $E \subset V \times V$ between nodes carry additional information such as mean read-depth and number of split-reads between the corresponding nodes. A node is in the graph if and only if there is a sufficient change in read-depth at the corresponding locus[9] or if it is the start or end locus of a split-read. An edge between two nodes exists if and only if the corresponding loci are neighbours on the same reference sequence or are connected by at least one split-read.

[9] For example crossing a threshold $\theta \in \mathbb{R}_0^+$.

It is important to note here that the exact position of a split in a read is a result of the chosen mapping algorithm (and its scoring scheme) and may not be precise, i.e. can be off by a few base-pairs. To account for such imprecisions, nodes (introduced by split reads) which are within a few base pairs of each other are merged, keeping either the node with the highest number of incident split read edges or the closest node induced by a change in coverage.

Additionally, edges corresponding to deletions are also allowed. The reasoning behind this is threefold:

1. The reference sequence used may contain unresolved regions, which will hence have no coverage, and — depending on the corresponding region's length as well as the read length — also have no split reads spanning the gap.

2. Similarly, the reference can be ambiguous, in which case reads map to multiple loci and are disregarded during graph construction.

3. All reads may share the same deletion (i.e. the genome has a deletion variant at that position), in which case, again, the corresponding region has no coverage.

While only the last point directly corresponds to actual deletions, the other two points effectively appear as deletions, even though their root cause is different. For example, Ensembl's GRCh38 build 106 has an unresolved region of length 1000 bp at `chr2:`16 145 118 to 16 146 118, flanked by simple repeats.

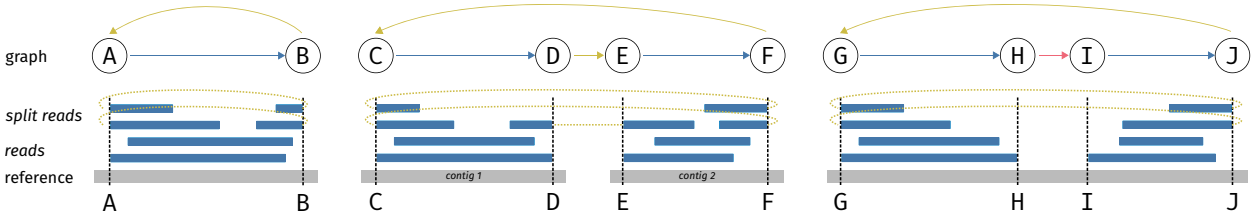



Figure 3.3: Three different exemplary schematics of possible circle scenarios. Left: A simple circle between loci `A` and `B` of a single reference contig. Middle: A circle formed by joining two segments from two different contigs. Right: A circle with a deletion in the middle (red arrow); this can occur either because there is indeed a deletion in the reads with respect to the reference or because the reference is ambiguous (i.e. has homologies) for this region or because this region has not been resolved yet.

Edges are hence classified into three (not mutually exclusive) categories:

1. *Neighbour* edges are edges whose nodes correspond to neighbouring loci in the reference sequence, with no other nodes in-between and where the mean read depth between these loci is larger than $\theta$.
2. *Split* edges are edges induced by split reads.
3. *Deletion* edges are edges between neighbouring loci in the reference sequence where the mean read depth between these loci is 0.

Figure 3.3 shows schematic examples of three different circle scenarios.

### 3.1.4 *Plausible path*

Any cycle in the graph whose edges alternate between *neighbour* and *split* or *deletion* edges is called a *plausible* path. To obtain such plausible paths, we first prune the graph, then partition the graph into its strongly connected components using Tarjan's strongly connected components algorithm.[10]

[10] R. Tarjan, "Depth-First Search and Linear Graph Algorithms", 1972.

Because read noisiness and reference ambiguities potentially also lead to a noisy graph, the graph needs to be pruned. Nodes and edges are removed from the graph if they meet any of the following conditions:

1. Split edges with too few split-reads.
2. Nodes with 1 or fewer edges.
3. Nodes whose incident edges are exclusively of type "split" (cf. Figure 3.4).

The first condition is a free parameter, which can be justified by its dependence on expected properties of the experiment, such as median read length and sequencing depth. The other two conditions remove nodes (together with their incident edges) that cannot possibly be part of a plausible path.

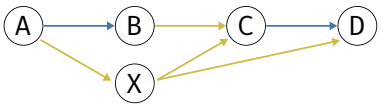



Figure 3.4: Node $X$ is removed, as all of its edges are exclusively of type *split*.

The number of split-reads for the first condition is (by default) arbitrarily set to 5, but can be adjusted by the user. In the future, this parameter could for example be estimated from the data itself.

■ Walking all plausible paths? Even after pruning, a graph's components can still be fairly complex. In other words, there may be numerous equally plausible (non-isomorph) circular paths through each component.

To cut down on processing time in subsequent steps (especially variant calling using `varlociraptor` in subsection 3.1.5), we limit the maximum number of cycles generated and introduce an arbitrary, hand-tuned heuristic score which favours circular paths with large numbers of split reads and mean read depth (as a geometric mean over the length of the cycle):

$$\left[\prod_{\text{edge in cycle}} \left(1 + \log_{10}(\text{read depth}) + \sqrt{\text{number of split reads}}\right)\right]^{1/|\text{cycle}|}$$

The rationale for this lies in the fact that paths (in the same strongly connected component!) exhibiting many supportive split reads and higher coverage inherently have more supporting evidence to correspond to eccDNA molecules, in contrast to paths with fewer split reads and lower coverage. See Figure 3.5 for the histogram of score values for a single sample.

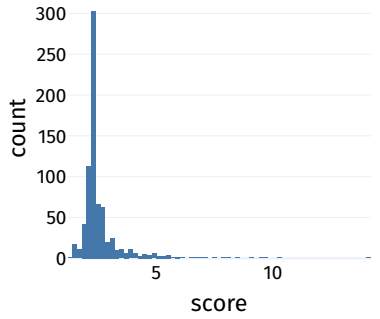



Figure 3.5: Histogram of heuristic score for all plausible paths in all components, for the single sample `kelly` described in section 3.2 and section 3.3.

Furthermore, see Figure 3.6 for a scatterplot of the heuristic score against `varlociraptor`'s probability estimate of the respective circle being present. There is only a very slight trend of higher scores corresponding to higher probabilities. However, very few data points fall within the "lower triangle" of the plot, i.e. there are almost no data points with a high score and a low probability.

In each SCC, starting from each node, perform a Depth First Search (DFS) to find all cycles starting and ending with the respective node. At each level of recursion, the node's children are sorted in descending order by the respective edge weights, such that children whose connecting edge carries a larger number of split reads and larger mean read depth are visited first. Once a cycle is found, it is inserted into a double ended and bounded priority queue, with the priority computed as the heuristic score given above. If the queue is full and the next cycle's score is worse than the lowest score in the queue, cycle generation stops. Otherwise, the lowest scoring cycle in the queue is replaced with the new cycle and cycle generation continues. See Figure 3.7 for pseudocode.

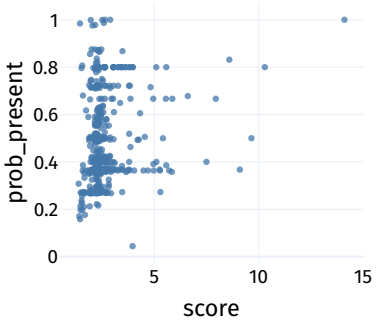



Figure 3.6: Scatterplot of the score (x-axis) against varlociraptor's probability estimate of the respective circle being present (y-axis) (see section 3.2, "VARIANT CALLING") — each marker corresponds to one plausible path / circle.

```
def plausible_paths(scc: Graph, max_cycles: int, score: Callable):
    # initialize a queue whose items are prioritized by `score`
    queue = Queue(priority=score)
    # `valid_cycles` returns cycles in the component `scc`
    # such that children with large edge weights
    # (split reads, read depth)
    # are visited first
    for cycle in valid_cycles(scc):
        # as long as the queue isn't at capacity, add cycles
        if len(queue) < max_cycles:
            queue.push(cycle)
            continue
        # if the queue is at capacity…
        _worst_cycle, worst_score = queue.min()
        # … either replace the worst cycle with a better one …
        if score(cycle) > worst_score:
            queue.pop_min()
            queue.push(cycle)
        # … or stop generating cycles
        else:
            break
    return queue.items()
```

Figure 3.7: Pseudocode for generating plausible paths of any given strongly connected component of the graph. The number of cycles generated is limited by `max_cycles`, and cycles are prioritized by `score`, i.e. the heuristic tries to favour cycles with large numbers of split reads and large mean read depth.

### 3.1.5 *Verification of circle candidates*

Once all eligible plausible paths have been generated, they are translated into VCF using Breakend (BND) notation: Each split or deletion edge corresponds to a breakend variant that has to be encoded. Neighbour edges are not encoded, as they only indicate regions with coverage, but no genetic variation. For example, a split edge connecting `chr2:16659596` with `chr2:15694017` is translated to two VCF records, which reference each other via the `MATEID` tag:

```
#CHROM  POS       ID                 REF  ALT            INFO
2       15694017  graph_1_circle_0_0 C    ]2:16659596]C  SVTYPE=BND;EVENT=graph_1_circle_0;MATEID=graph_1_circle_0_1
2       16659596  graph_1_circle_0_1 C    C[2:15694017[  SVTYPE=BND;EVENT=graph_1_circle_0;MATEID=graph_1_circle_0_0
```

They also share the same `EVENT` name, which allows grouping together multiple BNDs to encode larger circles with multiple junctions.

## 3.2 *An eccDNA calling workflow*

To get from a sample's nanopore reads to a final assessment of circles contained in the sample, more steps than calling circle candidates from the alignment (of the reads to a reference) are necessary. These are bundled in

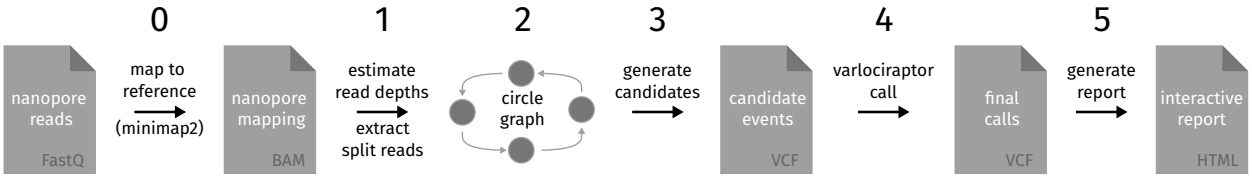



Figure 3.8: Steps of the cyrcular-calling workflow. **0**: Mapping of the reads to the reference sequence using `minimap2`. **1, 2**: Building the graph data structure using read depth and split read information extracted from the mapping. At this point, the graph can optionally be annotated using auxiliary information. **3**: Generating circle candidates by walking plausible paths of all SCCs of the graph. **4**: Circle candidates are statistically assessed using `varlociraptor` and the resulting calls are fdr-controlled. **5**: From the final calls, an interactive HTML report is generated.

the `snakemake`[11,12] workflow `cyrcular-calling`.[13]

[11] Mölder et al., "Sustainable data analysis with Snakemake", 2021.
[12] Köster and Rahmann, "Snakemake—a scalable bioinformatics workflow engine", 2012.
[13] Hartmann and Lähnemann, "snakemake-workflows/cyrcular-calling", 2023.
[14] Li, "Minimap2: pairwise alignment for nucleotide sequences", 2018.

**Read mapping** First, the reads have to be mapped against a reference sequence. For this step, `minimap2`[14] is used because it features presets specifically tailored towards ONT sequencing data and adheres to the SAM specification. More specifically, the preset option `-x map-ont` is used for ONT sequencing data, and `-x sr` for Illumina sequencing data. All other settings are left at their default values.

**Candidate circle calling** From the mapping, circle candidates are called with the method outlined in section 3.1. Because there can potentially be many circle *candidates*, the workflow makes use of snakemake's scatter-gather functionality to split candidate calls such that subsequent jobs can be run in parallel.

**Graph annotation** The resulting graph is annotated with information about genes, regulatory features and known repeats. Integrating such annotation is useful for downstream research, as it allows to quickly search and filter circles depending on the research question. By default, the following annotation sources[15,16] are used:

- Ensembl gene annotation
- Ensembl regulatory annotation
- RepeatMasker database

[15] F. J. Martin et al., "Ensembl 2023", 2023.
[16] Smit, Hubley, and Green, *RepeatMasker Open-4.0. 2013–2015*, 2015.

All of these annotation sources are parsed and inserted into interval trees (one for each contig) to facilitate quick region lookups. Afterwards, the graph is traversed: each neighbour edge is annotated with the annotation

information pertaining to the region between the two nodes[17]. Once all relevant edges have been annotated, the graph is serialized to disk for further processing, such as plot and report generation.

[17] Currently, split edges are not annotated, as this would require lookups in specialized annotation sources, such as gene fusion databases, e.g. chimerDB (Y. E. Jang et al., "ChimerDB 4.0: an updated and expanded database of fusion genes", 2020).

**Variant calling** Each circle candidate is then translated into BCF with breakend notation and statistically evaluated using `varlociraptor`[18] with its generic scenario calling model (see section 1.6). The scenario is defined as follows:

[18] Köster, Dijkstra, et al., "Varlociraptor: enhancing sensitivity and controlling false discovery rate in somatic indel discovery", 2020.

```
samples:
  nanopore:
    universe: "[0.0,1.0]"
    resolution: 0.1

events:
  present: "nanopore:]0.0,1.0]"
```

Basically, for the sample called "`nanopore`", allele frequencies between $0.0$ and $1.0$ are allowed, with a resolution of $0.1$, i.e. $0.0, 0.1, \ldots, 0.9, 1.0$. Then, an event named "`present`" is defined, which here indicates whether a circle is supported by the data or not: if the allele frequency is *larger* than $0.0$, the event is considered present; otherwise, it is considered `absent` by default[19]. [20]

Because of `varlociraptor`'s flexible scenario calling grammar, we can also integrate auxiliary Illumina sequencing reads to improve statistical power. Assuming the sample has been sequenced once with nanopore sequencing and once with Illumina sequencing, the scenario then can be defined as:

[19] Additionally, there is an implicit event called `artefact`, which captures cases potentially arising due to systematic, technical errors and biases.

[20] Also, arbitrary events can be defined. For example, the event "`present`" could also have been divided into two events, "`present_low`" and "`present_high`", with allele frequency ranges e.g. $]0.0, 0.5]$ and $]0.5, 1.0]$, respectively.

```
samples:
  nanopore:
    universe: "[0.0,1.0]"
    resolution: 0.1
  illumina:
    universe: "[0.0,1.0]"
    resolution: 0.1

events:
  present: "nanopore:]0.0,1.0] & illumina:]0.0,1.0]"
```

This can be interpreted as follows: For both `nanopore` and `illumina` samples, the allele frequency has to be larger than $0.0$, i.e. the variant has to be supported in *both* samples. For a less strict scenario, the `present` event could also be defined as "`nanopore:]0.0,1.0] | illumina:]0.0,1.0]`", which would allow for the variant to be supported by at least one of the two samples.

Because `varlociraptor` allows estimating the posterior probability for a candidate being `present`, `absent` or being an `artefact` given the mapping, the calls can be filtered using standard false-discovery-rate (fdr) control methods.[21] Here, the fdr-threshold defaults to $0.1$.

[21] Benjamini and Hochberg, "Controlling the false discovery rate: a practical and powerful approach to multiple testing", 1995.

■ Interactive report After fdr-control, an interactive report is generated which has per sample summary plots (including circle length distribution), a table of circles per sample, and detail views for the segments of each circle. The report is generated using `snakemake`'s `report` functionality in combination with `datavzrd`[22]. The tables also feature links to plots of the graphs, quality control plots for each circle, links to genome browsers and links to primer-blast pre-filled with information needed to design primers for each circle junction / breakpoint. See Figure 3.9a for an exemplary screenshot of the overview table of called circles for the `kelly` cell line and Figure 3.9b for the detail table of the circle encompassing the `MYCN` gene in the `kelly` cell line.

[22] *datavzrd* (`https://github.com/datavzrd/datavzrd`, visited on 2023-11-20)

The table contains the following columns:

1-3 The first column lists the `event_id`, which is a unique identifier for each circle, comprised of the graph component's name (`graph_id`, second column) and the circle's index (`circle_id`, third column) within that component. Columns 2 and 3 link to corresponding visualizations of the respective graph component and circle, see Figure 3.10 and Figure 3.11 for examples.

4, 5 The fourth column lists the `circle_length` as the sum of the lengths of all neighbour edges in the circle, while the fifth column `segment_count` gives the corresponding number of segments/neighbour-edges.

6 The column `regions` lists the genomic ranges spanned by the circle, linking to genome browsers for each range.

7 The column `num_exons` gives the number of exons spanned by the circle (as derived from the annotation sources), while the column `gene_names` lists all genes contained on the circle (in a dropdown menu, with links to detailed information about each gene).

8 The column `regulatory_features` lists all regulatory features contained on the circle, in no particular order.

9 The table also lists any known `repeats` in the column of the same name (if repeat annotation was provided).

10-13 The three columns with the common prefix `prob_` list the posterior probabilities of the circle being `present`, `absent` or an `artefact` of sequencing errors, respectively, as calculated by `varlociraptor`, with the column `af_nanopore` listing the estimated allele frequency.

14 Finally, the last, unnamed, pseudo-column links to detail tables which list each segment/edge of the respective circle, with information about the segment's length, mean read depth, number of split reads, and the annotation information for the segment's region. See Figure 3.9b for an example.

| event_id | graph_id | circle_id | circle_length | segment_count | regions | num_exons | gene_names | regulatory_features | repeats | num_split_reads | prob_present | prob_absent | prob_artifact | af_nanopore | category | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Filter... | 121 211 | 0.0 1.0 | | 1.0 2.0 | Filter... | 2 156 | Filter... | Filter... | Filter... | 10 4,663 | | | 0 | 0.0 1.0 | Filter... | |
| 211-0 | 211 | 0 | 964578 | 2 | 2:15694016-16145118, 2:16146119-16659595 | 156 | | TF_binding_site,open_chromatin_region,promoter,enhancer,CTCF_binding_site | 36 | 1.00 | 0.00 | 0.00 | | 1.00 | coding | circle-details |
| 121-0 | 121 | 0 | 16568 | 1 | MT:0-16568 | 74 | | | 4663 | 1.00 | 0.00 | 0.00 | | 1.00 | coding | circle-details |
| 175-0 | 175 | 0 | 2177 | 1 | 14:50921752-50923929 | 2 | | open_chromatin_region | 10 | 1.00 | 1.11e-3 | 0.00 | | 1.00 | coding | circle-details |

(a) The *overview* table of circles in the interactive report.

| graph_id | circle_id | kind | target_from | from | target_to | to | length | num_exons | gene_names | regulatory_features | repeats | coverage | num_split_reads | breakpoint_sequence |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 211 | 0 | Filter... | 2 | | 2 | | | 70 86 | Filter... | Filter... | Filter... | | 0 36 | Filter... |
| 211 | 0 | coverage | 2 | 15694016 | 2 | 16145118 | 451102 | 70 | | enhancer,CTCF_binding_site,promoter,TF_binding_site,open_chromatin_region | | 36.14 | 0 | ..... |
| 211 | 0 | deletion | 2 | 16145118 | 2 | 16146119 | | | | | | | | ..... |
| 211 | 0 | coverage | 2 | 16146119 | 2 | 16659595 | 513476 | 86 | | enhancer,open_chromatin_region,CTCF_binding_site,TF_binding_site | | 63.26 | 0 | ..... |
| 211 | 0 | split | 2 | 16659595 | 2 | 15694016 | | | | | | | 36 | ...CTTGCTCTTGCTCTTTACTT... |

(b) Detail table for the circle encompassing the `MYCN` gene in the `kelly` cell line. The table lists each segment/edge of the circle, with information about the segment's length, mean read depth, number of split reads, and the annotation information for the segment's region.

Figure 3.9: Screenshots of the overview table of called circles and the detail table of the circle encompassing the `MYCN` gene in the `kelly` cell line.

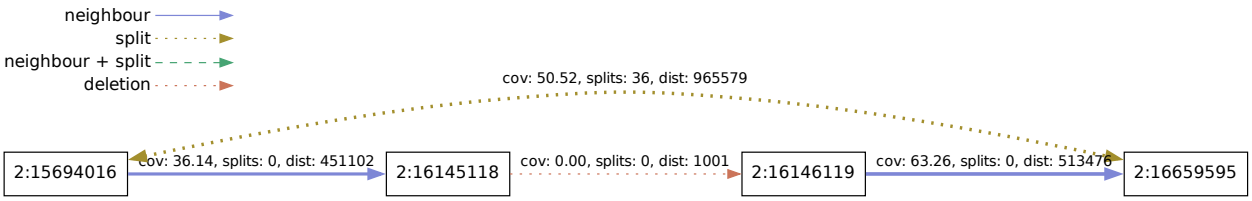



Figure 3.10: Graph visualization of the `kelly` cell line's `MYCN` circle.

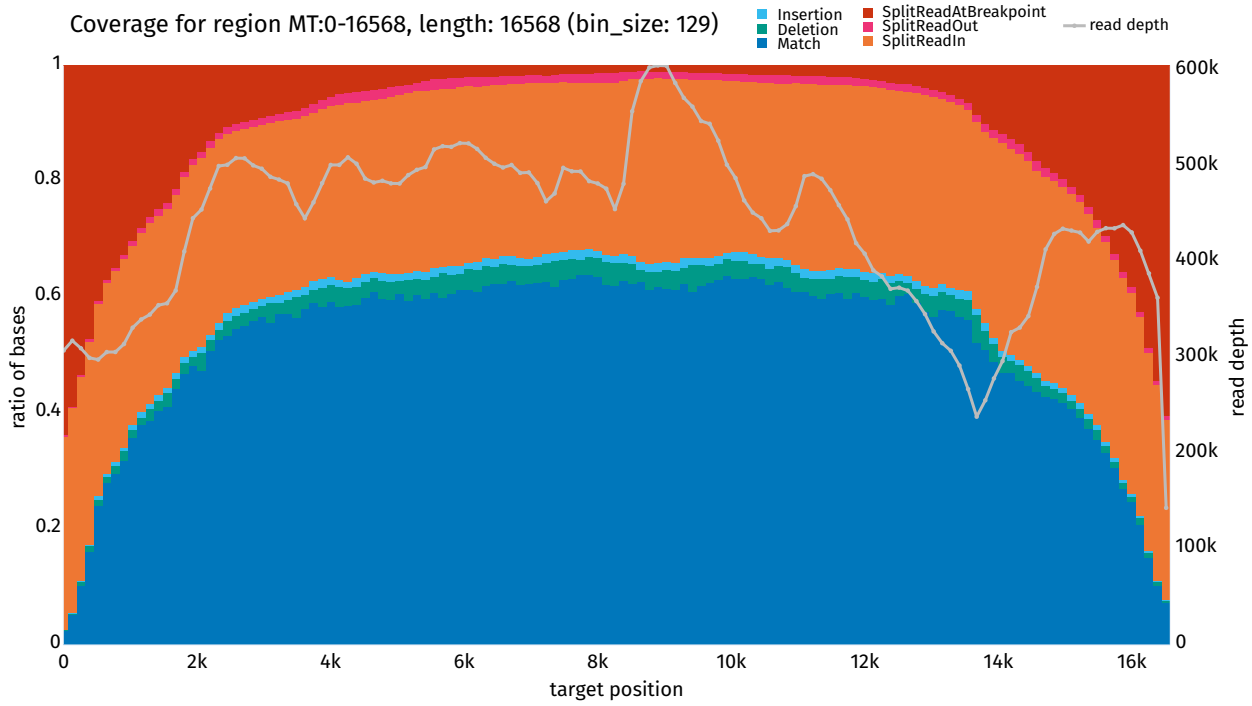



Figure 3.11: Quality control plot for the mitochondrion circle in the `kelly` cell line. For each locus (binned, x-axis), the plot shows the proportions of different features (y-axis). The features are: Insertions (cyan), deletions (green) and matches (blue), as well as split-reads which connect the first locus (in this case 0) to the last locus (in this case $\approx$ 16.5kbp) (red) (*"SplitReadAtBreakpoint"*). The other two categories *"SplitReadOut"* (orange) and *"SplitReadIn"*(magenta) indicate the proportion of split-reads where the mate split-read fragment is not within the circle, or where both split-read fragments are within the circle (but at neither of the circle junction points), respectively. On the secondary y-axis, the read depth is shown (grey).

## 3.3 Validation

The workflow is validated in the following ways:

- Simulated data: Simulated nanopore reads are used to validate the workflow's performance at different target coverages.
- Real data:
  - `kelly` cell line: A cell line which is known to contain a 1 Mbp ecDNA molecule encompassing the `MYCN` gene.
  - Plasmid spike-in: Two plasmids (coding for `MYC` and `ALK`) were spiked into a normal human DNA sample (at different concentrations).

The validation process is encapsulated in the `cyrcular-validation` workflow[23]. All wet-lab and sequencing work was done by Alicia Tüns, using a MinION sequencing device. Because the `kelly` cell line is a human cell line, it also contains the circular mitochondrion sequence; for eukaryotes, this is essentially a quality control target that should be present in all samples.

[23] *cyrcular-validation* (`https://github.com/tedil/cyrcular-validation`, visited on 2023-11-10)

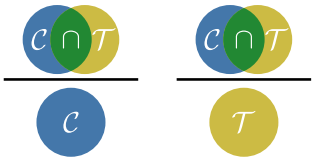
### 3.3.1 Simulation

For a set of 4995 genes (randomly selected, stratified by chromosome length), nanopore reads are simulated using `nanosim`.[24] Reads are simulated as if they were produced by sequencing *circular* DNA fragments. We will call such reads *circular reads*, even though the reads themselves are not circular, but the fragments they stem from are.
Additionally, Whole Genome Sequencing (WGS) reads are simulated at low coverage ($\lesssim 1$) to emulate sequencing reads from linear fragments that failed to be digested during library preparation, acting as noise.

For performance metrics, we look at precision and recall: Let $\mathcal{T}$ be the set of true circles and $\mathcal{C}$ the set of called circles. Then precision $= \frac{|\mathcal{C} \cap \mathcal{T}|}{|\mathcal{C}|}$ and recall $= \frac{|\mathcal{C} \cap \mathcal{T}|}{|\mathcal{T}|}$. In other words *precision* is the fraction of called circles that are true circles (Figure 3.11a), while *recall* is the fraction of true circles that were actually called (Figure 3.11b).

Circular reads are simulated at different levels of coverage[25] to be able to examine the dependence of precision and recall on coverage. Figure 3.13 shows how recall improves with increasing coverage, while precision stays at 1. The precision staying constant indicates that this method does not fabricate circles that are not part of $\mathcal{T}$.

[24] Yang et al., "NanoSim: nanopore sequence read simulator based on statistical characterization", 2017.




(a) *Precision*. (b) *Recall*.

Figure 3.12: Schematic depiction of precision and recall.

[25] In contrast to other read sequencing techniques, nanopore sequencing produces variable length reads, i.e. the coverage in this context is a target *mean* coverage.

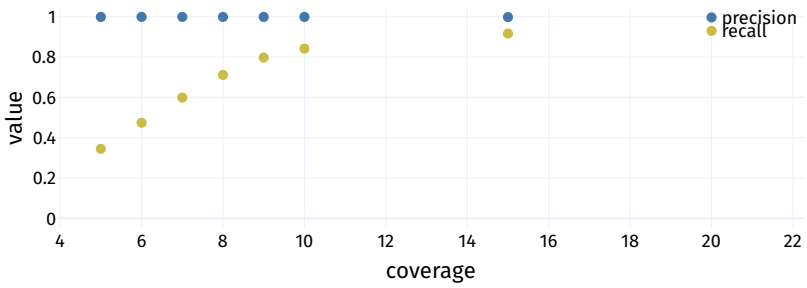



Figure 3.13: Precision (blue) and recall (yellow) as a function of coverage for nanopore reads. While precision is constant at 1 independent of coverage, recall improves with increasing coverage.

**Illumina reads** Above, the performance was evaluated with respect to nanopore sequencing reads, which is the type of data the method is designed for. To test whether the method can also be used with Illumina sequencing data, we performed the same evaluation again, but this time using 100 bp Illumina paired-end reads (simulated with `readSimulator`[26]) as input. Figure 3.14 shows the results. Similar to the nanopore case, precision stays at 1 independent of coverage, while recall improves with increasing coverage. However, recall is consistently lower than in the nanopore case.

[26] https://github.com/wanyuac/readSimulator

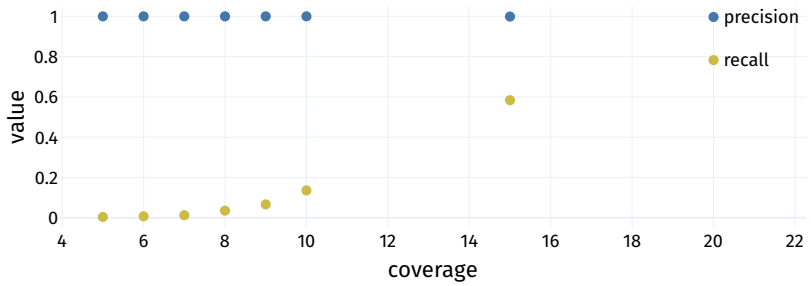



Figure 3.14: Precision (blue) and recall (yellow) as a function of coverage for Illumina reads. Similar to the nanopore case, precision is constant at 1 independent of coverage, while recall improves with increasing coverage. However, the recall is consistently lower than in the nanopore case.

### 3.3.2 *Experimental data*

For the `kelly` cell line, the workflow indeed calls a circle encompassing the `MYCN` gene, as well as the mitochondrion, as can be seen in Figure 3.9a. Additionally, a short circle of length 2177 bp is called on chromosome 14. Notably, this is the only circle where the probability of it being absent is non-zero. Whether this is a circle that is actually present in the sample or an artefact arising during either of the wet-lab, sequencing or calling steps is unclear. The large 1Mbp circle encompassing the MYCN gene, however, was validated by Alicia Tüns in the laboratory using PCR.

The plasmid spike-in experiment was performed with two different plasmids, one coding for `MYC` (`pDONR223_MYC_WT`) and one coding for `ALK` (`ALK_pLenti`). These two circular DNA molecules were spiked into normal human DNA samples. Because our approach is mapping based, and both plasmid sequences do not occur in the human reference genome, these sequences had to be added to the set of reference contigs before mapping. Both plasmid circles were called successfully with high confidence. Additionally, an off-target hit was detected on chromosome 4, which is discarded after fdr-controlling `varlociraptor`'s results.
Due to homology of the MYC plasmid to a region on chromosome 8, one SCC encompassing three different plausible paths through the component is constructed. Only one of these paths corresponds only to the MYC plasmid, while the other two paths also contain parts of chromosome 8. All three paths, however, are considered equally likely under `varlociraptor`'s model.

See Figure 3.15[27] for a schematic depiction of **A** the ALK circle, **B** the off-hit on chromosome 4 and **C** for the SCC encompassing the MYC plasmid and parts of chromosome 8.

[27] Tüns et al., "Detection and validation of circular DNA fragments using nanopore sequencing", 2022, Figure 6.

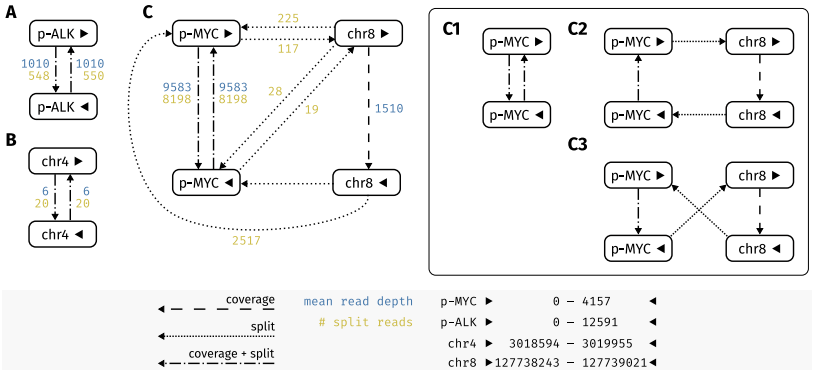



Figure 3.15: The three different SCCs of the graph for the plasmid spike-in experiment. The third SCC, **C**, is further split into three plausible paths **C1**, **C2** and **C3**. Mean read depth is annotated in blue, while the number of split reads are annotated in yellow. The markers ▶ and ◀ indicate the start and end loci of the respective genomic regions, as specified in the figure's legend.

## 3.4 *Discussion*

In this chapter, we presented `cyrcular`[28], a method for calling candidates for extrachromosomal circular DNA in sequencing data which has been enriched for circular molecules and depleted of linear molecules. It was designed with long reads (as e.g. produced by nanopore sequencing) in mind, but can — as we have shown here — in principle also be used with short reads (as e.g. produced by Illumina sequencing). It makes some assumptions about the data, namely that the majority of reads indeed stem from circular molecules and that the reference sequence is available and well resolved. These assumptions may not hold for all use-cases and datasets, especially not for those that have not been produced using a circular DNA sequencing protocol. To accommodate for such cases, adjustments to the method may be necessary, but the general approach of using split-reads to detect circle junctions should still be applicable.

It is also possible to alter the method to be able to detect non-circular fusions. For this use-case however, it is likely more efficient to use a dedicated fusion caller, such as `arriba`[29] or a general structural variant caller such as `gridss`.[30] These two examples, however, are designed for RNA-seq data and Illumina data, respectively, and may not be applicable to nanopore DNA-seq data.

We also presented a snakemake workflow `cyrcular-calling` based on `cyrcular` which uses `varlociraptor` to statistically assess circle candidates. This workflow allows researchers to quickly get from raw sequencing data to a set of *statistically assessed* circle calls. It also includes an interactive report with a structured overview of called circles, including extensive annotation and links to common (web-)resources and tools. Additionally, to facilitate (wet-lab) validation of circles, it includes links to primer-BLAST for designing primers for PCR, with the primer sequences pre-filled.

The workflow was validated on simulated data, as well as on real data from a cell line known to contain a 1 Mbp eccDNA molecule encompassing the `MYCN` gene, as well as on a spike-in experiment with two different plasmids. All of these experiments proved successful. Because the concentrations of the plasmids in the spike-in experiment are known and different, it should in principle be possible to estimate these concentrations from the sequencing data, which is a topic for future work.

[28] *cyrcular* (`https://github.com/tedil/cyrcular`, visited on 2023-11-10)

[29] Uhrig et al., "Accurate and efficient detection of gene fusions from RNA sequencing data", 2021.

[30] Cameron et al., "GRIDSS2: comprehensive characterisation of somatic structural variation using single breakend variants and structural variant phasing", 2021.

Providing a complete workflow including an interactive report cannot always be taken for granted in the field of bioinformatics. We however believe that it should be more common, as it not only facilitates reproducibility, but also reduces the likelihood of human error, as well as the time needed to get started with the analysis.

# 4 *Alignment free CNV calling*

Copynumber Variation is a type of structural variation that is defined as a change in the number of copies of a genomic region. Common examples for CNVs include deletions and duplications of a region, which can range in size from a few basepairs to several megabases[1].

Typically, CNV calling methodologies are based on one or more of the following types of data: read-depth, split-read and read-pair. There are also assembly-based approaches. All of these approaches rely on the alignment of reads (or a de-novo assembly) to a reference sequence.[2]

In this chapter, we introduce a novel approach to CNV analysis that relies on counts of unique reference k-mers instead. This is most similar to the established approach of using read-depth data, which is based on the number of reads that align to a reference locus. However, performing CNV analysis using k-mer counts has several advantages over read-depth based approaches. First, it is faster, since it does not require alignment of reads to a reference sequence. Second, it requires less disk space, since neither reads nor alignments need to be stored. Also, k-mer counts can potentially be used for other applications, such as SNV calling, or estimating the contamination of a sample.

Throughout this chapter, when not stated otherwise, we assume that the reference sequence is the human genome, i.e. is of length $3 \times 10^9$ bp and has ploidy 2.

The basic workflow for this approach involves building a k-mer index of the reference sequence (section 4.1), counting unique reference k-mers in the sample (section 4.2), assigning copynumbers from the sample to each unique k-mer position in the reference (section 4.3), and segmenting the copynumber signal to identify CNVs (section 4.4). Figure 4.1 shows a schematic overview of the steps involved.

Afterwards, we empirically evaluate the time and memory requirements of the method (section 4.5), and compare it to other tools (section 4.6).

[1] While there is no standardized definition, many research papers assume a minimum size of at least 50bp.

[2] Whitford et al., "Evaluation of the performance of copy number variant prediction tools for the detection of deletions from whole genome sequencing data", 2019.

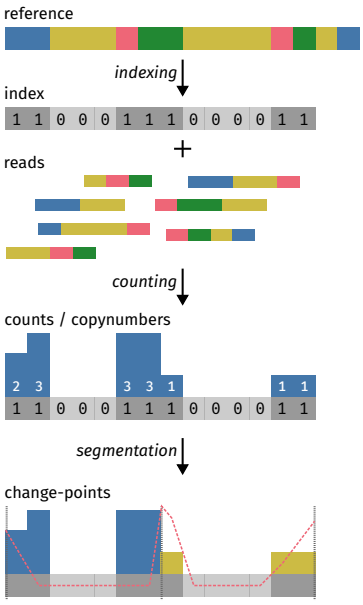



Figure 4.1: Schematic overview of the steps involved in alignment free CNV calling.

## 4.1 *Building an index*

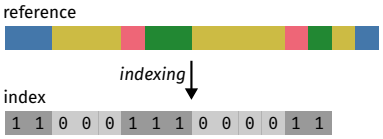



Figure 4.2: Schematic depiction of the indexing step.

As the first step, we build an index of all positions in the reference sequence that correspond to a unique (w.r.t. the reference) k-mer.

Given a reference sequence $\mathcal{R}$ with length $|\mathcal{R}| =: n$ and a k-mer length $k$, define the *unique k-mer index* $\mathcal{I}(\mathcal{R}, k)$ as follows:

$$\mathcal{I}(\mathcal{R}, k) := (b_i)_{i=1}^{n-k+1} \text{ with } b_i = \begin{cases} 1 & \text{if } \mathcal{R}_{i..i+k} \text{ is unique in } K(\mathcal{R}, k) \\ 0 & \text{otherwise} \end{cases}.$$

In other words, the index is a bit sequence of length $n - k + 1$, where each bit corresponds to a position in the reference sequence. If the bit is set, the k-mer starting at that position is unique in the reference sequence.

A trivial way to construct the index is as follows: First, collect all canonical k-mers of the reference sequence along with their positions in the reference. Then, sort the resulting list of k-mers. From the sorted list of k-mers, remove those k-mers that are duplicates. Last, mark each position in the reference where a unique k-mer starts.

```python
def build_index(sequence: str, k: int)
    kmers = kmers_with_positions(sequence)
    radix_sort(kmers)
    drop_non_unique(kmers)
    index = [0] * len(sequence)
    for _kmer, pos in kmers:
        index[pos] = 1
    return index
```

Figure 4.3: Pseudocode for building an index of unique k-mers in an arbitrary sequence.

The procedure outlined in Figure 4.3 runs in $O(n)$ time, since all of its steps can be performed in linear time. It is also linear in memory, since the number of k-mers generated is proportional to the length of the sequence. Both sorting and dropping of non-unique k-mers can be done in-place (see Obeya et al. [2019][3] and Figure B.1).

[3] Obeya et al., "Theoretically-efficient and practical parallel in-place radix sorting", 2019.

However, memory usage in practice can easily exceed available resources. For example: The human genome is about $3 \times 10^9$ bp long, resulting in $3 \times 10^9$ bp (minus $k$bp) k-mers. If a k-mer is encoded as a single 64-bit unsigned integer, generating and storing all k-mers of the human genome requires $3 \times 10^9 \cdot 64\text{bit} \approx 22\text{GiB}$ of space. When naïvely generating (k-mer, locus) tuples instead (as is done in Figure 4.3), this amount *doubles*, as loci generally have to be represented using 64bit integers, as there are genomes larger than $2^{32}\text{bp} \approx 4.3 \times 10^9\text{bp}$, the maximum length that can be represented with 32bit integers.

To avoid doubling the memory needed to build the index, we iterate over the k-mers in the reference two times: In a first pass, the sequence's k-mers are generated without locus information, sorted, and duplicates are removed. In a second pass, the sequence's k-mers are generated *with* locus information, and the corresponding bit in the index is set if the k-mer is unique (Figure 4.4).

The check for uniqueness requires the lookup of a k-mer in the set of unique k-mers. This could be done in $O(1)$ time using a hash set, but would require

```python
def build_index_two_pass(sequence: str, k: int):
    kmers = kmers_without_positions(sequence)
    radix_sort(kmers)
    drop_non_unique(kmers)
    index = [0] * len(sequence)
    for kmer, pos in kmers_with_positions(sequence):
        if kmer in kmers:
            index[pos] = 1
    return index
```

Figure 4.4: To build an index without allocating memory for storing locus information for each k-mer, iterate the k-mers of the reference sequence a second time and mark the respective locus if the k-mer is a unique k-mer.

additional memory. Instead, binary search — without requiring additional memory — can be used, which requires $O(\log m)$ time, where $m$ is the number of unique k-mers. In practice, a *BTree* is used instead, as it is a cache-friendly alternative to standard binary search, which has the same theoretical time complexity but is much faster in practice.

■ Storage Since the index is a simple bit vector, its memory requirements are exactly $n - k + 1\text{bit}$. For the human genome, this results in about $358\text{MiB}$ of memory. The index is often readily compressible, thus requirements for storage space can be further reduced, if needed.

The sole purpose of the index presented here is to indicate whether a k-mer is unique in the reference, i.e. describe the unique k-mer set of the reference in a sufficiently succinct way. It does *not* facilitate quick lookup of k-mers or any associated values. For this purpose, more sophisticated indices are available, such as the indices used in `sshash`[4] or `hackgap`.[5]

[4] Pibiri, "On weighted k-mer dictionaries", 2023.
[5] Zentgraf and Rahmann, "Fast Gapped k-mer Counting with Subdivided Multi-Way Bucketed Cuckoo Hash Tables", 2022.

■ Strongly unique k-mers

**Definition 4.1.1** (Strongly unique k-mer)**.** A unique k-mer that does not have any hamming distance 1 neighbour in its containing set $S$ is called *strongly unique* with respect to $S$.

To make the index more robust, remove all unique k-mers that are not strongly unique with respect to $S \cup \bar{S}$ (where $\bar{S}$ denotes the set of reverse complemented k-mers of $S$). Previously, it was possible for a particular subset of unique k-mers in the index to be converted into a different unique k-mer by simply altering a single nucleotide. However, when counting k-mers from a set of sequencing reads, sequencing errors are inevitable. Since the goal is to count k-mers of a set of sequencing reads, using only strongly unique k-mers removes one class of errors, i.e. those where a single nucleotide variant introduced during sequencing leads to misidentification of the true unique k-mer of the reference. See Figure 4.24 for statistics on the number of (strongly) unique k-mers in a human reference genome.

One $O(kn)$ algorithm to remove all weak k-mers (k-mers with a hamming distance 1 neighbour) from a set of unique k-mers works as follows[6]:

[6] Initial idea suggested by Karl Bringmann and discussed with Jens Zentgraf.

```python
def mark_weak(kmers: List[Kmer], weak: BitVec, offset: int):
    if offset == k: return
    # get start:end positions of blocks
    # starting with A, C, G and T in `kmers`
    blocks = acgt_blocks(kmers, offset)
    for (a0, a1), (b0, b1) in combinations(blocks, 2):
        kmers_a, kmers_b = kmers[a0:a1], kmers[b0:b1]
        weak_a,  weak_b  =  weak[a0:a1],  weak[b0:b1]
        n_i, n_j = len(kmers_a), len(kmers_b)
        i = j = 0
        while i < n_i and j < n_j:
            ka, kb = kmers_a[i], kmers_b[j]
            ka, kb = ka[offset + 1:], kb[offset + 1:]
            if ka == kb:
                weak_a[i] = weak_b[j] = 1
                i += 1; j += 1
            elif ka < kb:
                i += 1
            else:
                j += 1
    # recurse into each block (in parallel)
    for s, e in blocks:
        mark_weak(kmers[s:e], weak[s:e], offset + 1)
```

Figure 4.5: Pseudocode for marking k-mers that are not strongly unique in the lexicographically sorted list of `kmers`. The function `acgt_blocks` returns the start and end indices of blocks of k-mers starting with the same base.

First extend a list of unique k-mers with their reverse complements, then sort the list using radix sort. Determine the start- and end indices of blocks of k-mers starting with the same base via binary search (see Figure B.2). Between these blocks, perform the following (similar to a 4-way merge-sort step): For each combination of two blocks, keep a pointer to the first k-mer of each block. If the two k-mers pointed to are identical disregarding the first base, mark these as weak in the bit-vector — because they only differ in exactly 1 base, namely the leading one (defining the block they are in). Then increment both pointers. Otherwise, increment only the pointer which points to the smaller k-mer. When all pointers have reached the end of their block, conceptually remove the first base of all k-mers in a block (by increasing the `offset`, see Figure 4.6) and recurse into the block. See Figure 4.5 for pseudocode.
Afterwards, discard all k-mers marked as weak as well as all k-mers that are not canonical.
As with most recursion approaches, in practice it is advisable to avoid recursing too deep. We therefore replace the last few levels with a naïve $O(n^2)$ routine for marking k-mer pairs with hamming distance $\leq 1$. The "last few levels" can for example be defined by a (constant) maximum recursion depth or dynamically chosen with respect to the number of k-mers in a block (for example when there are at most 128 k-mers left in a block).

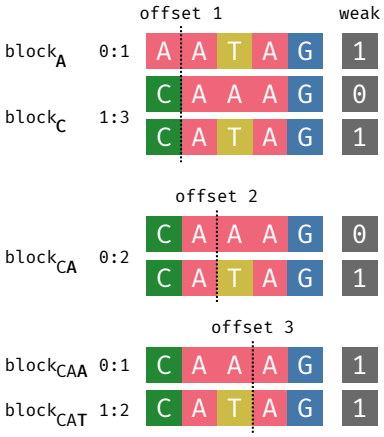



Figure 4.6: Given a 5-mer set $\{s_1, s_2, s_3\} = \{\texttt{AATAG}, \texttt{CAAAG}, \texttt{CATAG}\}$, $s_1$ and $s_3$ are marked as *weak* at offset 1 since their suffices of length 4 are identical, and they only differ in their leading base (A and C). Similarly, at offset 3, $s_2$ and $s_3$ are marked as weak, since their suffices of length 2 are identical, and they only differ in their leading base (A and T).

**Theorem 4.1.1.** *For a sorted set of $n$ unique k-mers $U \subset \Sigma^k$ and $k \in \{1, \ldots, 32\}$, marking weak k-mers in $U$ takes $O(|\Sigma|kn)$ time.*

*Proof.* Let $L = |\Sigma|$ and $n_1, \ldots, n_L$ be the lengths of the blocks starting with the first , . . . , $L$-th character of $\Sigma$ respectively (i.e. $n_1 + \cdots + n_L = n$).

There are $\binom{L}{2}$ pairs of blocks to consider. For an arbitrary pair of blocks with lengths $n_i$ and $n_j$, a maximum of $n_i + n_j$ comparisons are carried out[7]. That is, in total *at most*

$$\sum_{i=1}^{L-1} \sum_{j=i+1}^{L} (n_i+n_j) = (n_1+n_2)+(n_1+n_3)+\cdots+(n_{L-1}+n_L) = (L-1)n$$

comparisons are carried out at each level of recursion.
Additionally, finding the start- and end indices of all blocks never takes more than $n$ operations per level of recursion.
There are exactly $k$ levels of recursion, i.e. in total

$$\underbrace{k(L-1)n}_{\text{comparisons}} + \underbrace{kn}_{\text{block indices}} \in O(|\Sigma|kn)$$

operations are required. □

[7] This happens when the blocks are ordered such that the comparisons alternate between the less and greater cases.

■ Future work A different way to account or allow for small variants in k-mers is to use *gapped* k-mers. In contrast to "regular" k-mers, gapped k-mers do not have to be contiguous substrings of any given sequence but can be arbitrary sub*sequences* of length $k$ of a sequence, not necessarily contiguous. For example, given sequence `ACGTGCA`, `ACG-GCA` is a gapped 6-mer with 1 gap. Each gap in a gapped k-mer can be viewed as a *"don't care"* position, implicitly allowing for variation and sequencing errors. At the same time, gapped k-mers also cover a longer region of their parent sequence than contiguous k-mers of the same length (in the sense of number of bases). With the right choice of k-mer size, number and distribution of gaps for a given problem, better results may be achieved than with contiguous k-mers. There is a large swath of literature on the design of gapped k-mers, with a curated collection of papers aggregated by Laurent Noé[8].

[8] *spaced-seeds* (`https://sites.google.com/view/laurentnoe/spaced-seeds`, visited on 2023-07-06)

## 4.2 Counting k-mers

Counting k-mers is an important part of many bioinformatic analyses, such as sequence assembly, gene expression and transcript quantification or error correction. Here, we are interested in k-mer counts as a proxy for read-depth, which is one common type of data used for copy-number calling. To this end, we are interested in counting the number of occurrences of specific sets of k-mers in a set of reads.

Let $k$ be the k-mer length, $R$ a set of sequencing reads and $S$ a sequence of reference k-mers. We distinguish between the following two tasks:

A Non-indexed counting: Given a set of reads $R$, count all k-mers of $R$.

B Indexed counting: Given a set of reads $R$ and a sequence of reference k-mers $S$, count only those k-mers in $R$ which are contained in $S$.

While these two tasks share many common aspects, they ultimately pose different requirements:
For task A, the number of distinct k-mers encountered during counting can be very high, especially for datasets with high sequencing error rates.

Because the number of distinct k-mers to be counted is not known in advance, the k-mer counter must either be able to dynamically account for new k-mers or an estimate of the expected number of distinct k-mers must be made beforehand.
For task B, the number of distinct k-mers is limited by the size of the set of k-mers one is interested in. The k-mer counter can therefore be designed and tailored specifically towards the k-mers in the index set. It must however be able to identify and handle k-mers not in the index set. Task B may also be seen as a form of pseudo mapping of read k-mers to reference k-mers.

Naturally, the k-mer counts of a task A counter are a superset of the k-mer counts of a task B counter (given the same dataset). Therefore, any task A counter can be used for task B as well, but in general not the other way around.

Most k-mer counters (cf. subsection 4.2.1) address task A, while we are interested in task B. Because of this, we will only briefly discuss k-mer counting in general and then focus on the specific requirements for our task. In general, different k-mer counters make different trade-offs between disk-space, memory and runtime, and scale differently with the number of distinct k-mers to count. Also, the data structures / representation used may or may not be suited for or support removal of k-mers, set operations, parallelism, or streaming. Picking the k-mer counter most suitable for a specific task therefore depends on the specific requirements.

For task A, it is important to note that the number of distinct k-mers to count can be very high, especially for larger $k$ and large datasets. Theoretically, there can be up to $4^k$ different k-mers for random DNA strings. In practice, the number of distinct k-mers is less than but roughly proportional to the length of the genome of the sequenced organism. However, due to sequencing errors, some k-mers that are not part of the reference genome will also be present in the reads, resulting in an even higher number of distinct k-mers, albeit with a lower count. For that reason, many k-mer counters are disk-based / external-memory based, and additionally may include strategies to prune k-mers which appear infrequently and likely appeared due to (sequencing) errors.
As a measure to reduce memory usage, some k-mer counters discard k-mers that appear too infrequently (for example, `kmc3` discards k-mers with counts lower than 2 by default). Here, this is not an option: Low frequency k-mers can be informative in low-coverage copy-number calling scenarios and therefore must not be discarded.

It is also possible to count *approximately*, which can reduce the memory requirements significantly and may also improve performance in general. However, resulting counts are only estimates and may be inaccurate. Here, we will focus on exact counting, and leave the exploration of approximate counting for future work. For more information on approximate counting, refer e.g. to Pandey et al. [2018][9].

In this section, we introduce two k-mer counters for exact indexed counting, one of which uses minimal perfect hashing, and one of which is a naïve concurrent hash-map implementation.

We will now give an overview over existing methods solving either task A

[9] Pandey et al., "Squeakr: an exact and approximate k-mer counting system", 2018.

or task B, followed by a detailed description of our two approaches.

### 4.2.1 *Related Work*

We briefly discuss some k-mer counters and their characteristics, chosen for their range of different approaches (memory-based vs. disk-based, sorting-based vs. hashing-based etc.).

`kmc3`[10] is a disk-based k-mer counter that works as follows: In the first phase, the input file is read, and the reads are partitioned into super-kmers (substrings which share the same minimizer[11] [12]). In the second phase, partitions are read one by one, sorted and subsequently deduplicated and counted.

`hackgap`[13] is a memory-based k-mer counter that tries to reduce the number of bits stored per k-mer, in this case by using a multi-way bucketed (and quotienting) cuckoo hash table. It also allows for counting gapped k-mers.

`jellyfish2`[14] is a memory-based k-mer counter that uses a lock-free concurrent hashmap to store k-mers and their counts. Optionally, instead of re-sizing the hashmap when its capacity is reached, the hashmap can be written to disk and a new, empty hashmap is initialized instead.

`sshash`[15] is a memory-based k-mer counter that uses a compact representation of both k-mers and their weights (counts), based on minimal perfect hashing.

*Further reading*: For a comprehensive benchmark of k-mer counters, see Manekar and Sathe [2018][16] and Shibuya, Belazzougui, and Kucherov [2022][17].

[10] Kokot, Długosz, and Deorowicz, "KMC 3: counting and manipulating k-mer statistics", 2017.

[11] The smallest $m$-mer within any $k$-mer, with $m < k$.

[12] Note that kmc3 uses specialized *signatures* instead of minimizers to ensure that partition sizes are more equally distributed, since certain minimizers are empirically more likely to appear in k-mers derived from DNA sequences.

[13] Zentgraf and Rahmann, "Fast Gapped k-mer Counting with Subdivided Multi-Way Bucketed Cuckoo Hash Tables", 2022.

[14] Marçais and Kingsford, "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers", 2011.

[15] Pibiri, "On weighted k-mer dictionaries", 2023.

[16] Manekar and Sathe, "A benchmark study of k-mer counting methods for high-throughput sequencing", 2018.

[17] Shibuya, Belazzougui, and Kucherov, "Space-efficient representation of genomic k-mer count tables", 2022.

Surprisingly, naïve implementations of task B are just as fast as any of the aforementioned tools – for the use-case of counting unique k-mers in a set of reads, specifically in the case of the human reference genome. It does, however, incur a significant memory overhead, which is why we will discuss it in more detail in the next section.

We present two approaches to counting a fixed set of k-mers, one using a Minimal Perfect Hash Function (mphf) (similar to `sshash`), and one using a concurrent hash-map (similar to `jellyfish`). Both approaches re-use and combine established data structures and algorithms.

### 4.2.2 *An mphf based counter*

For the mphf-based approach, the basic idea is to use a minimal perfect hash function to map k-mers to indices in an array. In other words, construct a function $h : S \to \{0, \ldots, |S| - 1\}$, where $S$ is a set of index k-mers (e.g. unique k-mers of a reference genome), initialize an array $A$ of size $|S|$ with $A[h(s)] = (s, 0)\ \forall s \in S$ and increment the count at $A[h(x)]$ for each k-mer $x$ in a set of reads $R$. To construct the mphf, we use the algorithm introduced by Limasset et al. [2017][18].

However, when a k-mer is encountered that is *not* contained in the index set, the following problems may arise: Hash value calculation can either

[18] Limasset et al., "Fast and scalable minimal perfect hashing for massive key sets", 2017.

fail (producing a hash value out of bounds) or produce a value that collides with the hash value of a k-mer that *is* contained in the index set. When the hash value is out of bounds, the corresponding k-mer clearly cannot belong to $S$ and is therefore discarded. When the hash value collides with that of a k-mer that is contained in $S$, we check whether the k-mer is equal to the k-mer stored at the corresponding index in $A$. If it is not, the k-mer is discarded.

To save memory, the tuple of k-mer $x$ and count $c$ is encoded in a single 64-bit integer $x'$ as follows:

$$x' \coloneqq (x \ll \texttt{COUNT\_BITS}) \mid c$$

where `COUNT_BITS` is the number of bits used to store the count. The maximum number of bits that can be used for the count is $64 - 2k$, where $k$ is the k-mer length. This in turn limits the maximum count that can be stored to $2^{64-2k} - 1$. For example, choosing $k = 27$ leaves $64 - 2 \cdot 27 = 10$ bits for the count, which corresponds to a maximum count of $2^{10} - 1 = 1023$. This is sufficiently large even for high coverage sequencing experiments[19]. The amount of memory needed for storing both k-mers and their counts then is $64 \cdot |S|$ bit.
Alternatively, k-mers and counts can be stored separately in two bit vectors, such that the required memory is $(2k + C) \cdot |S|$ bits, where $C$ is the number of bits used to store the count. This allows for more flexibility in the combination of k-mer length and number of count bits, and can be more memory efficient, depending on the exact combination of $k$ and $C$.

Because calculating hash values using the Minimal Perfect Hash Function (mphf) $h$ is generally more expensive than simple hash functions such as aHash[20] or FxHash[21], we use a quotient filter as a first Approximate Membership Query (AMQ) filter to quickly discard k-mers that are not contained in $S$. AMQ filters are data structures that allow determining whether an element $x$ is part of a set $S$, such that it always returns *true* if $x \in S$, and returns *false* with a probability of $1 - \epsilon$ if $x \notin S$, for some false positive rate $\epsilon \in [0, 1]$. Notably, AMQ filters never generate false negatives. They make a trade-off between space-efficiency and false positive rate, i.e. increasing the space used for storage decreases the false positive rate.

Even though using such a filter adds some runtime and memory overhead, it greatly reduces the number of expensive mphf hash value calculations, resulting (in practice) in a speed-up in total.

While generating k-mers from reads, checking whether a k-mer is contained in the quotient filter and calculation of the mphf values can trivially all be done in parallel, updating counts in $A$ is more difficult to parallelize, because it requires mutating shared state. One solution is to use atomic operations/atomic integer types (if supported by the CPU) to increment the count at the corresponding index in $A$. The solution we use here requires neither atomic operations nor mutexes[22]. Instead, we conceptually split the count array $A$ into multiple disjoint sub-slices $A_i = A[i \cdot l..(i + 1) \cdot l]$ of the same size $l$[23], and each sub-slice $A_i$ has a fixed thread $t_i$ assigned to updating its counts. In other words, thread $t_i$ is responsible for updating counts only for those k-mers whose hash values fall into the range $[i \cdot l, (i + 1) \cdot l)$. Now, after filtering k-mers using the quotient filter and calculating their

[19] As a comparison, the default number of count bits used in `jellyfish` is 7, allowing for a maximum count of $2^7 - 1 = 127$. However, `jellyfish` has a bit-packing/-redistribution scheme to compensate for this.

[20] *aHash* (`https://github.com/tkaitchuck/aHash`, visited on 2023-11-02)

[21] *FxHash* (`https://docs.rs/rustc-hash/latest/rustc_hash/struct.FxHasher.html`, visited on 2023-11-02)

[22] Locks preventing multiple threads accessing and/or mutating state at the same time.

[23] It remains future work to evaluate whether having the sub-slice size be a multiple of the page size is beneficial for performance, or if it hinders performance due to the fact that the last sub-slice is then smaller or larger (depending on whether one is rounding up/down to the nearest multiple of the pagesize), causing the thread to have less/more work than the other threads.

hash values, k-mers are sorted by their hashes (again using radix sort). The sorted k-mers are then partitioned such that each partition $P_i$ contains only k-mers whose hash values fall into the range of a single corresponding subslice $A_i$. Each partition is then handed to its assigned thread $t_i$ for updating the counts in $A_i$.

See Figure 4.7 for an overview of the approach.

To turn this approach into an approximate counting approach while at the same time benefiting from reduced memory requirements, instead of storing the integer encoding of each k-mer in the table, a *fingerprint* (using fewer bits) of the k-mer can be stored instead.

### 4.2.3 *A concurrent hash-map counter*

For the concurrent hash-map based approach, we use the rust library `dashmap`[24] . Its memory requirements are higher than that of the mphf approach, since it stores k-mers and their counts separately. Assuming $64$ bit per k-mer and $C$ bit per count, the memory requirements are at least $(64 + C) \cdot |S|$ bit.

[24] `https://docs.rs/dashmap/latest/dashmap/struct.DashMap.html`

The method then is straightforward: For each k-mer in the index set $S$, insert a count of $0$ into the hash-map. Then, for each k-mer $x$ in the set of k-mers to be counted, increment the count of $x$ in the hash-map if $x$ is already contained in the hash-map.

■ Storing the counts To produce k-mer counts in the order of the k-mers in the reference sequence, simply iterate over the reference sequence and look up the count of each k-mer in the hash-map. Because k-mers that are adjacent in the reference are likely to have a similar count, a run length encoding is a simple and sensible approach for storing the counts.[25]

[25] Using general compression algorithms such as `gzip`, `bzip2` or `xz` is of course also possible, as well as employing more specialized encodings such as Elias-Fano or other small integer encodings.

### 4.2.4 *A small k-mer counting benchmark*

To get an idea of the performance and characteristics of the approaches presented here, as well as the performance of `kmc3`, `jellyfish2`, `hackgap` and `sshash`, we benchmarked them with the following parameters: The reference sequence is chromosome 1 of the human reference genome. The reads are $12\,500\,000$ pairs of 101bp Illumina paired-end sequencing reads. The machine used for benchmarking has the following characteristics:

- CPU: Intel(R) Core(TM) i7-7700HQ (4 cores, 8 threads)
- RAM: 32GB
- Disk: 1TB SSD
- OS/Kernel: 6.4.1-arch1-1

All tools were run with a k-mer length of 27. Tools supporting indexed counting (`jellyfish`, `sshash`, `naïve`) were run in indexed mode, all others (`kmc3`, `hackgap`[26]) in non-indexed mode. For `sshash` and `naïve`, an index for all strongly-unique k-mers in chromosome 1 was built in advance. For `jellyfish`, a FASTA file describing all strongly-unique k-mers in chromosome 1 was derived from the `naïve` index (to restrict the set of k-mers

[26] It should in theory be possible to run `hackgap` with an existing index, but correct CLI usage remains unclear.

1. raw reads

Reads are continuously read from the input file or stream.

2. buffering reads

Reads are inserted into a buffer of fixed size. The buffer size depends on read length, available memory and available parallelism.

3. generating k-mers

From the buffered reads, (two-bit encoded) k-mers are generated in parallel by distributing reads to threads, each of which produces k-mers. To balance work load across threads, threads may steal work from other threads.

4. pre-filtering of k-mers

Using an AMQ filter, k-mers that are definitely not in the index set are discarded. While this usually also involves hashing, the hash functions used during this step can be very simple and therefore faster to compute than the mphf.

5. minimum perfect hashing

For the remaining k-mers, hash values are computed using the mphf $h$.

6. sorting and partitioning k-mers by hash

K-mers are sorted by their hash values and partitioned such that each thread is only responsible for k-mers whose hashes are within a specific range. Even though this involves some overhead due to the additional sorting and partitioning steps, it is beneficial for overall performance, as it obviates the need for synchronization between threads, rendering this approach lock-free.

7. counting k-mers

For each k-mer, its associated count is incremented in the count table if the k-mer stored at the index indicated by the hash value is identical to the one to be counted. After a batch of buffered reads' k-mers has been counted, repeat from step 3.

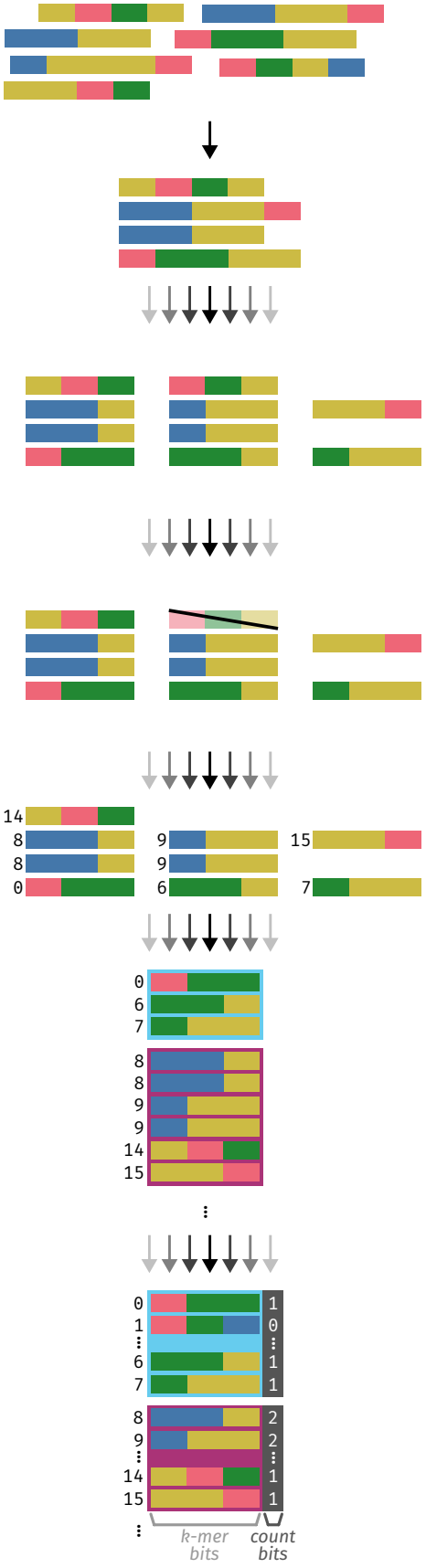



Figure 4.7: Overview over the naïve mphf-based counting approach. Single vertical arrows indicate sequential processing, while multiple arrows indicate parallel processing.

to be counted). Tools that require an estimate of the number of k-mers to be counted were provided with a multiple of the length of chromosome 1 ($\approx$ 250kbp)[27]. Tools that discard low frequency k-mers were configured to keep all k-mers. For `kmc3`, two different configurations were used: One where operations are performed in memory only (`-m24 -r`), and one where operations are performed largely on disk (`-m2`).
Otherwise, default options were used. Performance was measured using `perf` and `chrt` (to set the scheduling priority of the benchmarked processes to the highest possible value): `chrt -f 99 perf stat -r 5 -ddd <invocation>`. Results are shown in Table 4.1.

[27] The multiple was chosen this large to avoid tool specific issues.

| Tool | Total (s) | Count (s) | Mem (GB) | Output (GB) |
|---|---|---|---|---|
| naïve (mphf) | 106 | 71 | 5.44 | 0.036 |
| naïve (mphf + AMQ) | 68 | 44 | 6.17 | 0.036 |
| naïve (dash-map) | 80 | 53 | 6.48 | 0.036 |
| jellyfish | 125 | 82 | 2.60 | 1.852 |
| sshash | 2040 | | 0.42 | ?[a] |
| kmc3 (memory) | 73 | | 13.55 | 6.304 |
| kmc3 (disk) | 47 | | 3.76 | 6.304 |
| hackgap | 154 | 97 | 8.15 | 7.732 |

Table 4.1: Overview of benchmark results. Average of 5 runs. *Total* is the wall-clock time from start to finish of the tool (using 8 threads), including I/O. Where available, the time spent counting is given separately in column *Count*. *Mem* is the peak amount of memory used by the tool during execution. *Output* is the size of the output file. The first 5 entries correspond to indexed counting, the last 3 to non-indexed counting.
[a] `sshash` did not produce any output whatsoever, so the output size is unknown.

It is also important to note that the outputs of the tools are not identical: While our implementations output counts for all k-mers in the index set in the order they appear in the reference sequence, other tools store a dedicated custom index representation. This means that obtaining the counts for all k-mers in the reference sequence in the order of appearance requires an additional step[28,29]. It also means that the tools differ in the disk space requirements for storing the counts, as can be seen in the *Output* column of Table 4.1.

To get an impression of the performance of `kmc3` and our approach on a larger dataset, we also benchmarked them on the complete human reference genome (GRCh38) and a set of Oxford nanopore reads[30]. The results of this mini-benchmark on a machine with 88 cores (Intel® Xeon® Gold 6152 CPU @ 2.10GHz) and 768GB RAM are shown in Table 4.2. While `kmc3` is slightly faster and has lower memory requirements, it requires significantly more disk space, both for temporary auxiliary files and the resulting count file.

[28] For example, for `jellyfish`, a query of the respective k-mers takes an additional 147s for this benchmark.
[29] For kmc, one can use its C-API to perform random queries on the index. For this benchmark, this takes 113s (single threaded).
[30] These were originally obtained from `http://s3.amazonaws.com/nanopore-human-wgs/rel6/rel_6.fastq.gz` with a compressed file size of 137GB. It appears the file has since moved to `https://s3-us-west-2.amazonaws.com/human-pangenomics/T2T/CHM13/nanopore/rel6/rel6.fastq.gz`, but reports a different file size of 351GB.

| Method | Time (s) | Peak memory (GB) | Temp (GB) | Output (GB) |
|---|---|---|---|---|
| kmc3 | 1955 | 12 | 132 | 600 |
| naïve (mphf) | 2033 | 52 | 0 | 2.3 |

Table 4.2: Results of the mini-benchmark for a larger set of nanopore reads. The column *Temp* corresponds to temporary auxiliary files stored on disk.

### 4.2.5 *Future work*

Instead of using the mphf construction introduced in Limasset et al. [2017][31], use a locality-preserving mphf as introduced in Pibiri, Shibuya, and Limasset [2023][32], as its locality preserving nature may improve cache behaviour

[31] Limasset et al., "Fast and scalable minimal perfect hashing for massive key sets", 2017.
[32] Pibiri, Shibuya, and Limasset, "Locality-preserving minimal perfect hashing of k-mers", 2023.

during counting. Additionally, assuming reads are partitioned into super-kmers, it may be possible to exploit the locality preserving nature of the ls-mphf to skip lookup for a subset of k-mers of each super k-mer, blindly incrementing the count for the remaining k-mers. While this may lead to erroneous counts, it may still be acceptable in some cases.

Because the main weakness lies in the amount of memory needed to store index k-mers and their counts, choosing a more compact representation of the fixed set of k-mers (in the order of their hash values) which allows quick lookup by index may be an interesting future improvement.

By mimicking kmc3, adding support for disk-based counting to reduce memory usage is also possible: Instead of constructing one mphf and count table for all k-mers in the index, construct multiple mphfs for sets of k-mers with the same minimizer, resulting in multiple smaller tables. Afterwards, partition reads into super-kmers (by minimizer), batch and sort reads by minimizer, and count each batch of reads with the respective mphf. This way, only a subset of the mphfs and count tables has to be loaded into memory at any given time. Also, instead of re-loading a count table from disk for each batch of reads, it would also be possible to start with an empty table and merge tables after counting each batch of reads.

As there is no dependence on the k-mers to be contiguous, supporting gapped k-mers is also a potential future improvement.

## 4.3 *Estimating reference coverage*

Now that counts for all k-mers in the index set have been obtained, we can estimate the coverage of the reference sequence.

First, to relate these raw counts to copy-number estimates, we associate the ploidy with a property of the sample's count distribution. The mode of a count distribution serves as an obvious choice, given that it represents the most frequent count. This count is expected to correlate with the ploidy, as we anticipate each reference locus will occur about as often as the ploidy implies - assuming the sample lacks copy-number aberrations that could significantly influence the global count distribution. It *is* possible for whole chromosome arms or even chromosomes to be affected, for example in the case of trisomy 21, where there are 3 instead of 2 copies of parts or all of chromosome 21. Even then, the impact on the count distribution of the complete genome will not be significant (because chromosome 21 only accounts for about $1.5\%$ of the human genome).

When the total number of counts for a sample is high and the sample is not particularly special[33], choosing the mode (most frequent count, excluding the count for zero) therefore is a viable strategy.
Especially in low-coverage samples it can be advantageous to assume the counts to follow a Negative-Binomial distribution, and to calculate the estimated distribution's mode instead.

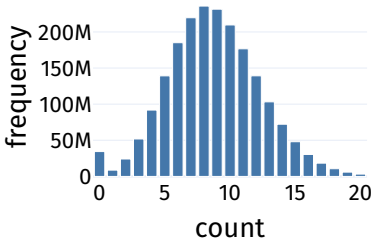



Figure 4.8: Exemplary unique 27-mer count distribution for GIAB's sequencing of sample `NA12878`, downsampled to about $10\times$. The mode is 8, the mean is 8.78 and the median is 9.

[33] i.e. has no large copy-number aberrations

**Definition 4.3.1** (Negative Binomial Distribution)**.** The negative binomial distribution is a discrete probability distribution. It is supported on $k \in \mathbb{N}$ and has parameters $r \in \mathbb{N}, p \in [0, 1]$. Its probability mass function is

defined as

$$\mathbb{P}(X = k \mid r, p) = \binom{r+k-1}{k} p^r (1-p)^k.$$

Its mode, mean and variance are defined as follows:

| | |
|---|---|
| Mode | $\lfloor p^{-1}(r-1)(1-p) \rfloor$ if $r > 1$, otherwise 0 |
| Mean | $\mathbb{E}(X) = p^{-1} r (1-p)$ |
| Variance | $\mathbb{V}(X) = p^{-2} r (1-p)$ |

■ Fitting Negative Binomial distributions Because no closed forms for maximum-likelihood estimators of the parameters $r$ and $p$ exist, usually either an expectation maximization or gradient based approximation method is used to estimate them. Here, we make use of the LBFGS-B[34,35] algorithm to find $r$ and $p$ which maximize the likelihood function given a sample $(k_1, k_2, \ldots, k_N)$ of size $N$. First, the likelihood function for $N$ observations $(k_1, k_2, \ldots, k_N)$ is defined as

[34] Liu and Nocedal, "On the limited memory BFGS method for large scale optimization", 1989.

[35] Byrd et al., "A Limited Memory Algorithm for Bound Constrained Optimization", 1995.

$$L(r, p \mid k_i) = \prod_{i=1}^{N} \binom{r+k_i-1}{k_i} p^r (1-p)^{k_i}.$$

As usual, the likelihood function is transformed into log space:

$$\begin{aligned} l(r, p \mid k_i) &= \sum_{i=1}^{N} \left[\log(\Gamma(k_i + r)) - \log(\Gamma(k_i + 1)\Gamma(r))\right] + \sum_{i=1}^{N} r \log(p) + \sum_{i=1}^{N} k_i \log(1-p) \\ &= \sum_{i=1}^{N} \log(\Gamma(k_i + r)) - \sum_{i=1}^{N} \log(k_i!) - N\Gamma(r) + Nr \log(p) + \log(1-p) \sum_{i=1}^{N} k_i \end{aligned}$$

where $\Gamma$ is the gamma function (with $\Gamma(k+1) = k!$ for $k \in \mathbb{N}$). The partial derivatives with respect to $r$ and $p$ are then given by:

$$\begin{aligned} \partial_r l(r, p \mid k_i) &= \sum_{i=1}^{N} \left(\psi(k_i + r)\right) - N\psi(r) + N \log(p) \\ \partial_p l(r, p \mid k_i) &= Nrp^{-1} - (1-p)^{-1} \sum_{i=1}^{N} k_i \end{aligned}$$

where $\psi$ is the digamma function (with $\psi(x) = \partial_x \log(\Gamma(x)) = {}^{\Gamma'(x)}/_{\Gamma(x)}$). Then the LBFGS-B algorithm can be used to find the maximum likelihood parameters of $r$ and $p$, from which the mode of the distribution can be calculated.

■ Zeros While the assumption of k-mer counts following an NB distribution often is accurate, it glosses over one important thing: Some k-mers (in the index) may never be observed (in the sample), for example due to sequencing errors, small variants (SNPs, small indels), larger deletions or because their true copynumber in the sample is low. Therefore, zero counts will be *inflated*, i.e. occur way more often than expected under the assumption of an NB distribution. There are several ways to mitigate this: 1. Ignore zero counts during NB parameter estimation, 2. use a mixture model of e.g. a geometric distribution and an NB distribution or 3. use special distributions for this case, such as Zero-Inflated Negative Binomial (ZINB)

or Zero-One Inflated Negative Binomial (ZOINB) distributions. Since ignoring zero counts during parameter estimation works reasonably well in practice in the context of this chapter (see section 4.6), we leave evaluating the other options as future work.

■ GC-CONTENT BIAS Depending on sequencing technology and species, systematic errors can be introduced. For example, Illumina sequencing is prone to introducing a GC-content bias, where GC-poor regions are underrepresented in terms of fragments.[36] See Figure 4.9 for an example of the relation between GC-content and k-mer count distribution. This can be addressed by performing any of the above estimation procedures stratified by GC-content. In the case of counting k-mers, there are exactly $k + 1$ discrete GC-content levels, including 0 (all bases in the k-mer are A or T) and $k$ (all bases in the k-mer are G or C).

[36] Browne et al., "GC bias affects genomic and metagenomic reconstructions, underrepresenting GC-poor organisms", 2020.

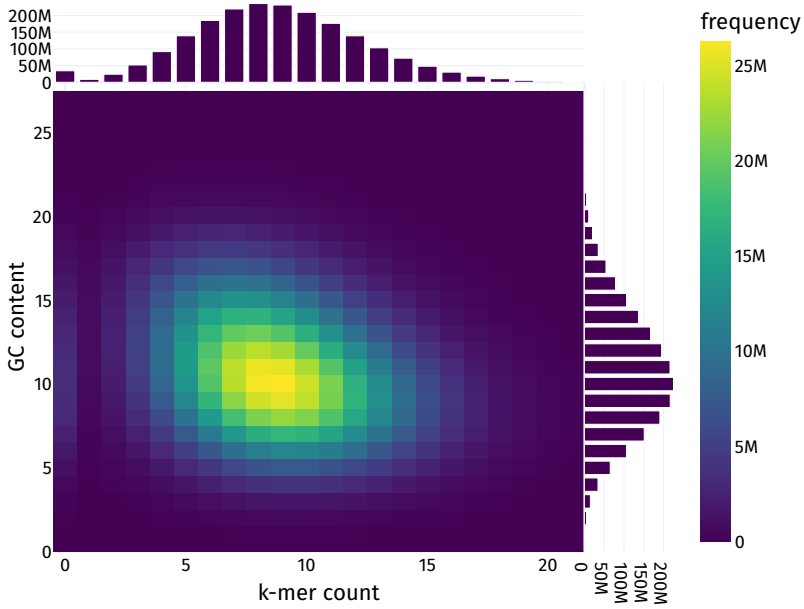



Figure 4.9: 2D histogram of the k-mer count (x-axis) frequency (colour-axis) by GC content (y-axis) for the same dataset as in Figure 4.8. In this example, the most frequently observed combination of GC content and k-mer count is $(9, 10)$. The marginal histogram at the top shows the unique 27-mer distribution of the dataset, while the marginal histogram to the right shows its GC content distribution.

■ FUTURE WORK Instead of obtaining maximum-likelihood estimates with Newton-methods, it would be possible to use an EM algorithm instead. In that case, it may be advantageous to fit a *mixture* of NB (or Poisson[37]) distributions instead, with one mixture component per copynumber level. For each raw count, its copynumber can then be determined as a function of the mixture weights and the respective components.

[37] The NB distribution can be expressed as a (Gamma-) mixture of Poisson distributions.

### 4.3.1 *Estimating copynumbers*

In this step, we will transform raw counts to copy-number estimates, while adjusting for GC-bias. Let $c \in \mathbb{N}$ be a raw count, $p \in \mathbb{N}$ the ploidy for the sample, $g$ the GC-content of the k-mer the count originates from and $C(g)$ the coverage estimate for the given GC-content (as described in section 4.3).

Then the corresponding copy-number estimate is defined as

$$\hat{c} = \left\lfloor \frac{pc}{C(g)} \right\rceil ,$$

where $\lfloor \cdot \rceil$ is stochastic rounding, i.e. when the fractional part is equal[38] to $0.5$, randomly round up *or* down. This is especially important when the estimated coverage $C(g)$ is an even integer, which would induce a bias towards even copy-numbers (assuming a default rounding behaviour of *round even*).

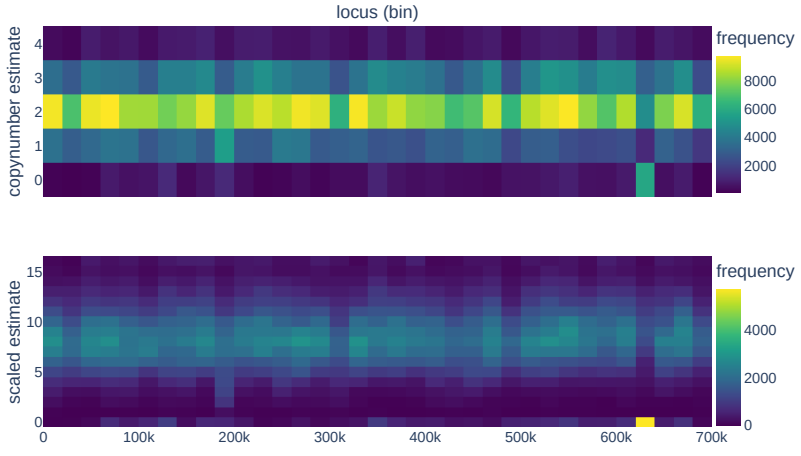
The reasons for rounding here at all are that firstly, changes in copy-number *usually* are integer multiples of the ploidy (e.g. duplication or deletion of a segment of a chromosome), i.e. copy-numbers have discrete levels, and secondly, using (small) integers allows for lower memory usage, as well as fast implementations for Empirical Distribution Function (EDF)-based statistical tests on sliding windows (see subsection 4.4.2).

Since copy-number estimate values are expected to be small (centred around ploidy, with low dispersion), they can be encoded as bytes. This allows storing copy-numbers in a contiguous array of bytes. In the array, each entry corresponds to the copy-number of the k-mer starting at the corresponding index in the reference — if the k-mer is unique. Otherwise, the special value of $255$ indicates that there is no copy-number information for the respective k-mer available at all.[39]

Depending on experimental design, species and sample, it can happen that copy-number levels are not integral, but fractional[40]. In these cases, it is possible to adjust the resolution of copy-number levels, conceptually by de- or increasing the ploidy. To that end, we introduce an additional parameter $r \in \mathbb{R}^+$, which we call *resolution*. Then, the *resolution scaled* copy-number estimate is defined as

$$\hat{c}' = \left\lfloor \frac{rpc}{C(g)} \right\rceil .$$

After copynumber calling, the resolution scaled copy-number estimates have to be re-scaled by $1/r$ (and optionally rounded) to obtain copy-number estimates in the original scale/magnitude.[41] See Figure 4.10 for an example of the effect of resolution scaling on copy-number estimates.




[38] Allowing for a small margin $\epsilon$.

[39] Without using $255$ as a marker for missing values, we would either have to use a sparse representation or skip missing values entirely, potentially resulting in neighbouring counts as coming from k-mers far apart in the reference sequence.

[40] For example in bulk sequencing of cancer cells with clonal and sub-clonal events, groups of cells can exhibit different copy-number variations.

[41] This is very similar to the concept of *binning* counts, but in the y-axis (changing count resolution), not the x-axis (changing locus resolution).

Figure 4.10: Example of the effect of resolution scaling on copy-number estimates. The first row shows the original copy-number estimates, the second row shows the resolution scaled copy-number estimates for $r = 4$. Note that in the second row, no re-scaling has been performed, so the scaled copynumber estimate that corresponds to a copynumber of 2 is 8. Also note that the binning in the x-axis is only for display purposes. Internally, each locus is still represented by a single value.

## 4.4 Segmentation

Now that we have obtained a sequence of copy-number estimates for each locus in the reference, we want to partition this sequence into segments of identical copy-number. Segments with aberrant copy-number (i.e. different from the reference ploidy) are potentially interesting for further analysis, e.g. may be indicative of certain diseases.
The basic concept of 1-D segmentation is to divide a sequence of values into multiple disjoint contiguous subsequences, such that values within each subsequence are similar to each other, while values between neighbouring subsequences are dissimilar. In the context of copy-number estimation, this means that we want to divide the sequence of per-locus copy-number estimates into regions of likely identical copy-number. This is done by performing statistical tests on a sliding window, locating change-points where the tests indicate a significant difference between the left and right halves of the window. The resulting sequence of change-points then implies a partitioning of the sequence into regions of similar copy-number.

Let $C$ be the sequence of (preprocessed) counts, $w \in \mathbb{N}$ be a window size and $k \in \mathbb{N}$ be the k-mer size. Then define a window $W$ of length $2w + k$ over the counts $C$. This window is divided into two halves $W^l$ and $W^r$ of length $w$ with a $k$ sized gap in-between the halves. The gap is introduced to ensure that the counts in the two halves are independent[42]. See Figure 4.11 for an illustration.
Starting at the first position in the reference, a 2-sample statistical test is performed between $W^l$ and $W^r$ (the counts in the left and right half of the window), and the value of the test statistic is recorded. Additionally, an effect between the $W^l$ and $W^r$ is calculated and recorded, such as the difference in medians or means. Then, the window is shifted by one position, and the process repeats until the end of the sequence is reached. To ensure every element of the sequence has an associated test statistic, virtually pad the sequence with the count corresponding to the sample's ploidy. This results in a sequence $T$ of test statistics that has the same length as $C$. Figure 4.12 shows pseudocode which exemplifies this process.

Given a level of significance $\alpha \in \mathbb{R}$, only values of $T$ that cross the chosen test's corresponding critical value $\theta_\alpha \in \mathbb{R}$ are of interest, and are denoted as $T_{>\theta_\alpha}$ Other values are ignored, since in those cases no significant difference between the two halves was detected.

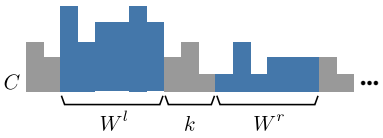



Figure 4.11: Illustration of the window layout. $W^l$ encompasses the counts of $C$ in the left part of the window, $W^r$ analogously for the right part, while $k$ counts in between the window's halves are ignored.

[42] Independent in the sense that the windows do not share k-mers, which would render some counts dependent.

```
def perform_tests(copy_numbers, w, k):
    n = len(copy_numbers)
    test_stats, effects = [], []
    window = Window(copy_numbers, w, k)
    for i in range(1, n + len(window) + 1):
        test_stats.append(window.test())
        effects.append(window.effect())
        window.shift(1)
    return test_stats, effects
```

Figure 4.12: Pseudocode for obtaining test statistic and effect values, for window-size $w$ with gap $k$, given a sequence of copy-numbers.

### 4.4.1 *Statistical tests*

We focus on two non-parametric tests here: The median test[43] and the Kolmogorov-Smirnov test.

[43] Also known as Mood's Median-Test.

The reason for choosing non-parametric tests is that making assumptions regarding a sample's distribution can be overly strict and easily be violated. Often, parametric tests for common distributions (e.g. normal, negative binomial) account for location and scale / variance / shape differences between the samples' distributions. However, in this context, primarily a difference in location is of interest, since a change in scale without a change in location is likely not indicative of a copy-number change (but it may very well be indicative of some other issue).

■ Median test As a non-parametric test, the median test[44] does not make any assumptions on the samples' distribution. It tests whether two (or more) independent samples were drawn from populations with identical medians. It only considers differences between the samples' medians — and nothing else.

[44] Hedderich and Sachs, *Angewandte Statistik*, 2018.

**Definition 4.4.1** (Median)**.** The median $\mathrm{Med}(X)$ of a sequence of $n$ values $X = (x_1, \ldots, x_n)$ is defined as the value $m$ such that at least half of the values in $X$ are less than or equal to $m$ and at least half of the values in $X$ are greater than or equal to $m$:

$$\begin{aligned}&\mathrm{Med}(X) = m\\ &\quad \text{s.t. } |\{x \mid x \in X, x \leq m\}| \geq {}^{n}/_{2} \quad \text{and} \quad |\{x \mid x \in X, x \geq m\}| \geq {}^{n}/_{2}.\end{aligned}$$

For odd $n$, the median is the middle item of the *sorted* sequence $X'$, i.e.

$$\mathrm{Med}(X) = x'_{n+1/2}.$$

For even $n$, we define the median as the arithmetic mean of the two middle items, i.e.

$$\mathrm{Med}(X) = \frac{1}{2}\left(x'_{n/2} + x'_{n/2+1}\right).$$

In the two sample case, the null hypothesis $H_0$ and the alternative hypothesis $H_1$ are defined as

$$H_0 : \mathrm{Med}(S_1) - \mathrm{Med}(S_2) = 0,\ H_1 : \mathrm{Med}(S_1) - \mathrm{Med}(S_2) \neq 0,$$

where $S_1$ and $S_2$ are two independent samples and $\mathrm{Med}(\,\cdot\,)$ is the median of the respective sample.

The test proceeds by determining the median $m \coloneqq \mathrm{Med}(S_1 \cup S_2)$ of the combined sample and counting the number of values that are smaller and greater than $m$ for each of the two samples. The results are tabulated in a *contingency table*, as shown in Figure 4.13.

| | $< m$ | $> m$ |
|---|---|---|
| $S_1$ | $O_{11}$ | $O_{12}$ |
| $S_2$ | $O_{21}$ | $O_{22}$ |

Figure 4.13: Contingency table for the two-sample median test.

Values that are equal to $m$ need special consideration: They can either be consistently attributed to the smaller/greater category, be ignored completely or be distributed evenly.
In the context of this chapter where samples are subsets of $\mathbb{N}$ and the count domain is relatively small in practice (e.g. $\{0, \ldots, 8\}$), it is reasonable to distribute the counts evenly to the smaller/greater categories. If we were

to discard counts equal to $m$ instead, it could lead to division by zero during the calculation of the test statistic (for example in larger regions with constant copynumber).

The $\chi^2$ test statistic with one degree of freedom is computed as follows:

$$\chi^2 = \sum_{i=1}^{2} \sum_{j=1}^{2} \frac{(O_{ij} - E_{ij})^2}{E_{ij}},$$

where $O_{ij}$ is the number of counts less than $m$ ($j = 0$) or greater than $m$ ($j = 1$) in sample $i$ and $E_{ij} = N^{-1}(O_{i1} + O_{i2})(O_{1j} + O_{2j})$ is the expected count for cell in row $i$, column $j$, and $N$ is the total number of observations.

For a given significance level $\alpha \in [0, 1]$, a value of the test statistic that is larger than $\theta_\alpha := \text{CDF}^{-1}(\alpha)$ (where $\text{CDF}^{-1}$ is the inverse cumulative distribution function of the $\chi^2$ distribution with one degree of freedom) leads to the rejection of the null hypothesis $H_0$.

Calculations needed to determine the $\chi^2$ test statistic lend themselves to be expressed using SIMD instructions.

■ Kolmogorov-Smirnov test The Kolmogorov-Smirnov test[45] tests whether two samples are drawn from the same distributions. It does not make assumptions regarding the samples' distributions, but — in contrast to the median test — is susceptible to any difference (e.g. location or shape) between the samples' distributions.

[45] Hedderich and Sachs, *Angewandte Statistik*, 2018.

The null- and alternative hypothesis are defined by

$$H_0 : C(S_1) = C(S_2),\ H_1 : C(S_1) \neq C(S_2)$$

where $C(\cdot)$ denotes the EDF for a given sample. The test statistic is defined as

$$D_{1,2} = \sup_x |C(S_1)(x) - C(S_2)(x)|\,.$$

For a given significance level $\alpha \in [0, 1]$, a value of the test statistic that is larger than

$$\theta_\alpha(n, m) = \sqrt{-\frac{1}{2} \log\left(\frac{\alpha}{2}\right) \cdot \frac{n+m}{nm}}$$

(where $n = |S_1|$ and $m = |S_2|$ are the sample sizes) leads to the rejection of the null hypothesis $H_0$.

### 4.4.2 *A sliding window data structure*

Because the tests are applied to a sliding window over the copynumber estimates, it is beneficial to use a data structure that allows for efficient updates of the test statistics. This is especially important for the median test, as it requires the computation of the median over the combined left and right half of the window for each position. A standard sorting based implementation would require $O(w \log w)$ time per update, where $w$ is the combined window size, while a select-$n$th algorithm[46,47] achieves $O(w)$ average time per update. Both algorithms may additionally need copies of the data or modify the data in place, which is not desirable here: An in-place sorting would render the results of the next update invalid, while a copy would require additional overhead.

[46] Robert W Floyd and Ronald L Rivest, "Expected time bounds for selection", 1975.

[47] Blum et al., "Time bounds for selection", 1973.

Therefore, we first make the following observations: In order to calculate the test statistic for the median test, only the median over the combined left and right half of the window as well as the contingency tables (with respect to the median) for both halves are relevant. Similarly, for the KS test only the EDFs are relevant for computing the KS test statistic. We also note that when shifting the window by one position, exactly one value enters and one value leaves each window half.
The goal then is to efficiently update median, contingency tables and EDFs when the window is shifted by one position.

To that end, define a sliding window data structure $W := (\mathcal{C}, w, i, E, n)$, where

- $\mathcal{C} \subset \mathbb{N} \cup \{\perp\}$ is a sequence of copy number values, with $\perp$ representing a missing value,
- $w \in \mathbb{N}$ is the window size,
- $0 \leq i < |\mathcal{C}|$ is the window's current position (with respect to $\mathcal{C}$),
- $E$ is an integer array of length $\max(\mathcal{C}) + 1 =: |E|$, representing the cumulative frequencies of copynumbers $\mathcal{C}_{i..i+w}$,
- and $n \in \mathbb{N}$ is the number of values in $\mathcal{C}_{i..i+w}$ that are not $\perp$

The array $E$ together with $n$ implicitly encodes the EDF of the copy number values in $\mathcal{C}_{i..i+w}$, with

$$E_j = |\{c \leq j \mid c \in \mathcal{C}_{i..i+w}, c \neq \perp\}| \quad \forall\, 0 \leq j < |E|.$$

To obtain the actual EDF, it is sufficient to divide (each element of) $E$ by $n$.

Because the median can be computed from the EDF, we will start by describing how to update $E$ and $n$ when the window is shifted by one position, and then describe how to update the median and contingency table.

■ Updating cumulative frequencies When a window is shifted by one position to the right, exactly one value leaves and one value enters the window, namely $c_{out} := \mathcal{C}_i$ and $c_{in} := \mathcal{C}_{i+w}$. The update of the frequencies in $E$ can then be formulated in terms of $c_{out}$ and $c_{in}$:
When $c_{out} = c_{in}$, the frequencies do not change and $E$ is not modified.
When $c_{in} \neq c_{out}$, only the $|c_{in} - c_{out}|$ array entries in-between[48] $c_{in}$ and $c_{out}$ have to be modified:

[48] Including the lower bound, excluding the upper bound.

- If $c_{in} < c_{out}$, update $E_{c_{in}..c_{out}} \mathrel{+}= 1$.
- If $c_{in} > c_{out}$, update $E_{c_{out}..c_{in}} \mathrel{-}= 1$.
- If $c_{in} = \perp$, update $n \mathrel{-}= 1$.
- If $c_{out} = \perp$, update $n \mathrel{+}= 1$.

See Figure 4.14 for an example of shifting two separate windows (with a gap of length $k$ in-between) by one position.

In the worst case, the underlying sequence of copynumbers is a sequence of alternating 0s and $h$s (where $h > 0$ is an arbitrary large number), in which case each update requires $h$ operations.

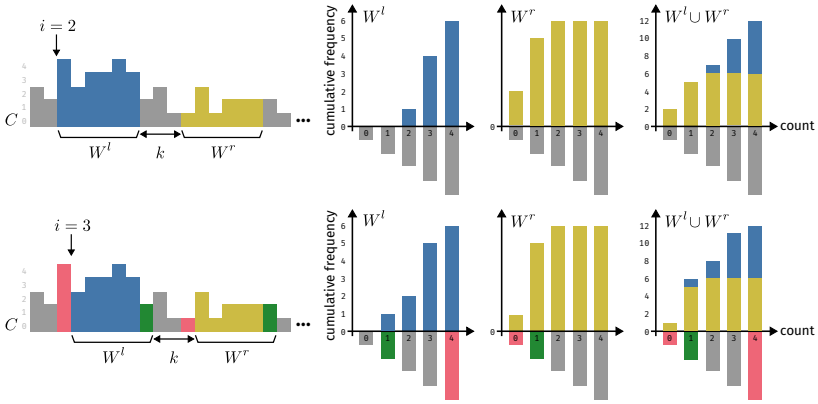



Figure 4.14: Effect of shifting two windows by 1 position to the right, from $i = 2$ (first row) to $i = 3$ (second row). The counts in window $W^l$ are coloured blue, the counts in $W^r$ yellow. When shifting the windows, exactly 1 count leaves (red) and 1 count enters (green) each window. Notice how this only affects the cumulative frequency of count values between those two counts. The common cumulative frequency for $W^l \cup W^r$ is the sum of both parts (notice the change in the y-axis).

However, in this context, the worst case is unlikely to occur in practice for several reasons: First, neighbouring copynumber estimates are likely to be similar, as they refer to neighbouring genomic loci (i.e. are correlated locally). Second, if one assumes copynumber estimates to follow a distribution with a single mode and moderate variance, as we do here with the Negative Binomial distribution, then the probability of observing a large difference between $c_{in}$ and $c_{out}$ is low.

For example, let $X, Y \sim \text{NB}(n, p)$ be random variables following a negative binomial distribution with parameters $n = 14$ (number of successes) and $p = 0.37$ (probability of success)[49]. Then the expected value of the absolute difference between $X$ and $Y$ is $\mathbb{E}[|X - Y|] \approxeq 8.967$. [50] However, since raw counts are scaled to obtain copynumber estimates (section 4.3), the domain is even smaller: Since the mode of $\text{NB}(14, 0.37)$ is 22, raw counts are scaled by $^{ploidy}/_{22}$ (with $ploidy = 2$) and subsequently rounded to obtain copynumber estimates. In that case, the expected value of the absolute difference between $\tilde{X} := \lfloor \frac{2X}{22} \rceil$ and $\tilde{Y} := \lfloor \frac{2Y}{22} \rceil$ is $\approx 0.816$. In other words, each update of $E$ requires on average less than one modification of its elements.

[49] These parameter estimates resemble the NB fit for the counts of 27-mers in the GIAB NA12878 sample reads.

[50] For more information on the difference between two negative binomial random variables, see (Lekshmi and Sebastian, "A skewed generalized discrete Laplace distribution", 2014) and (*stats.stackexchange.com Negative Binomial Difference*).

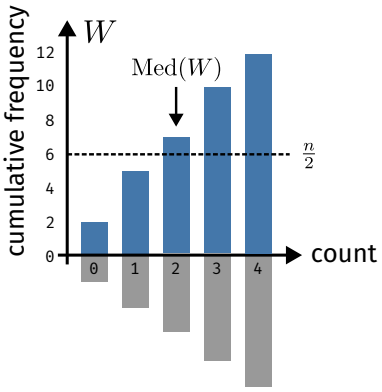



Figure 4.15: From a window $W$'s cumulative frequencies, its median $\text{Med}(W)$ can be derived as the first count for which the cumulative frequency is greater than $^n/_2$ ($n$ odd) or the average of the smallest two counts with cumulative frequency equal to or greater than $^n/_2$ ($n$ even).

■ UPDATING THE MEDIAN A window $W$'s median can be formulated in terms of a window's cumulative frequencies $E$.

For odd $n$, it is equal to the smallest index such that the cumulative frequency at that index is at least $^n/_2$. For even $n$, it is the arithmetic mean of the smallest index such that the cumulative frequency at that index is at least $^n/_2$ and the smallest index such that it is greater than $^n/_2$. These may coincide, in which case the median is an integer and exists in $\mathcal{C}[i..i + w]$, as illustrated in Figure 4.15.

$$\text{Med}(W) = \begin{cases} \min_j E_j \geq {}^n/_2 & n \textit{ odd} \\ \frac{1}{2}\left(\min_j E_j \geq {}^n/_2 + \min_j E_j > {}^n/_2\right) & n \textit{ even} \end{cases} \tag{4.1}$$

Notably, this allows finding the median of $W$ in $O(|E|)$ operations (worst case, when all but the last entry of $E$ are zero).

To update the median when $W$ is shifted by one position to the right, proceed as follows: Let $W = (\mathcal{C}, w, i, E, n)$ be the window before shifting and $W' = (\mathcal{C}, w, i + 1, E', n')$ after shifting, and define $m := \text{Med}(W)$.

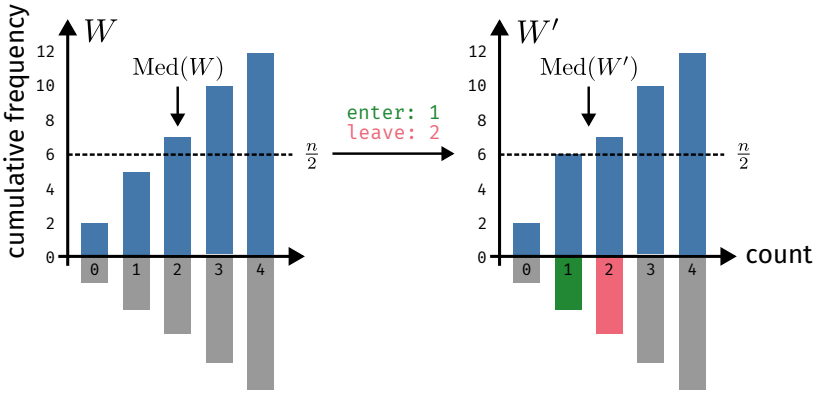



Figure 4.16: After shifting window $W$ to $W'$ (with a 1 entering and a 2 leaving), the median changes from $\text{Med}(W) = 2$ to $\text{Med}(W') = 1.5$.

Then, if the cumulative frequency in the shifted window at the median is less than the cumulative frequency required for it to be the new median, look for the new median in the same manner as in Equation 4.1, but consider only cumulative frequencies to the right of $m$. Similarly, if it is greater than required, only search to the left of $m$.

$$\text{Med}(W') = \begin{cases} (\min_j E'_j > {}^{n'}/_2) \quad , \; m < j < |E'| & \text{if } E'_m < {}^{n'}/_2 \\ (\max_j E'_j < {}^{n'}/_2) + 1, \; 0 \leq j < m & \text{if } E'_m > {}^{n'}/_2 \end{cases}$$

The above formula assumes $n'$ to be odd. For even $n'$ modify analogously to Equation 4.1 and handle the case where $E'_m = {}^{n'}/_2$:

$$\text{Med}(W') = \begin{cases} \frac{1}{2}({\min_j E'_j \geq {}^{n'}/_2} + \min_j E'_j > {}^{n'}/_2), \; m < j < |E'| & \text{if } E'_m < {}^{n'}/_2 \\ \frac{1}{2}\left(\max_j E'_j \leq {}^{n'}/_2 + \max_j E'_j < {}^{n'}/_2\right) + 1, \; 0 \leq j < m & \text{if } E'_m > {}^{n'}/_2 \\ \frac{1}{2}\left([\min_j E'_j \geq {}^{n'}/_2, \; 0 \leq j < m] + [\min_j E'_j > {}^{n'}/_2 \; , \; m < j < |E'|]\right) & \text{if } E'_m = {}^{n'}/_2 \end{cases}$$

■ Merging two windows Later in this chapter, in subsection 4.4.4, we will need to be able to merge two (or more) *directly* neighbouring windows $W^l$ and $W^r$ into a single window $W^m$, i.e. the windows are not separated by a gap. Therefore, we describe this operation here.

Given two gap-less windows $W^l$ and $W^r$ with window sizes $w_l$ and $w_r$ which are direct neighbours (i.e. $i_r = i_l + w_l$), determining $W^m = W^l \cup W^r$ is straightforward: $W^m$'s cumulative frequencies $E_m$ are computed as the element-wise sum $E^l + E^r$, and the starting position is $i_m = i_l$, the window size is $w_m = w_l + w_r$ and the number of values is $n_m = n_l + n_r$.

■ Building a contingency table To build a contingency table needed for the median test, compute $T(E, m) = (l, e, g)$ as the number of counts less than ($l$), equal to ($e$) and greater than ($g$) a given value $m \in \mathbb{R}_0^+$ from $E$ as follows:

$$\begin{aligned} l &\coloneqq |\{c \in W \mid c < m\}| = E_{\lceil m \rceil - 1} \\ e &\coloneqq |\{c \in W \mid c = m\}| = E_{\lfloor m \rfloor} - E_{\lceil m \rceil - 1} \\ g &\coloneqq |\{c \in W \mid c > m\}| = E_{-1} - E_{\lfloor m \rfloor - 1} \end{aligned}$$

This assumes $m > 0$. For $m = 0$, instead define

$$l \coloneqq |\{c \in W \mid c < m\}| = 0$$
$$e \coloneqq |\{c \in W \mid c = m\}| = E_0$$
$$g \coloneqq |\{c \in W \mid c > m\}| = E_{-1} - E_0$$

### 4.4.3 *Defining segments*

Having obtained the test statistic values $\mathcal{T} \subset \mathbb{R}_0^+$, the task now is to find a partition of $\mathcal{T}$ depending on $\theta_\alpha$.

Because the Probability Density Function (PDF) of the $\chi^2$ distribution with one degree of freedom is monotonically decreasing, large test-statistic values correspond to small p-values and vice-versa (Figure 4.17). Therefore, loci with large $\chi^2$ values indicate likely change-points. See Figure 4.18 for an example of $\chi^2$ values for a subset of simulated copynumber estimates.

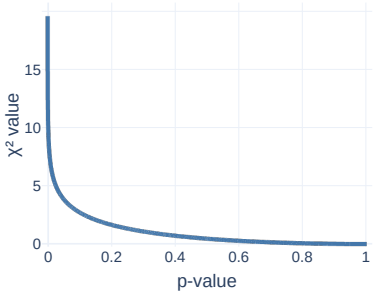



Figure 4.17: The $\chi^2$ value (y-axis) decreases monotonically with increasing p-value (x-axis).

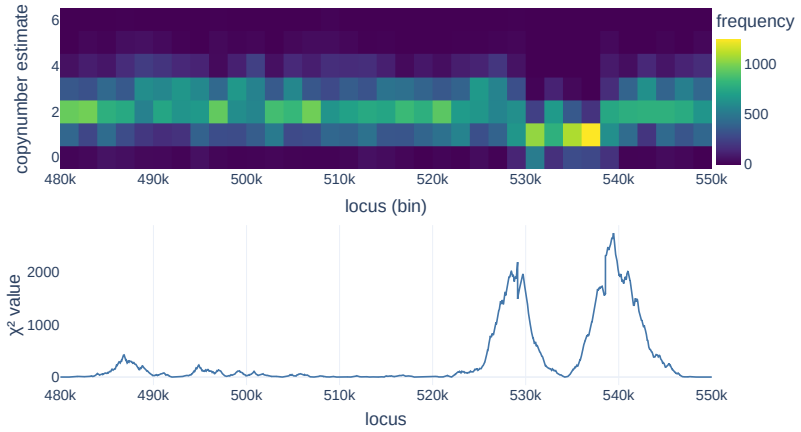



Figure 4.18: Upper row: copynumber estimates (y-axis) for each locus (x-axis), binned; each bin is assigned a colour based on the frequency of the respective copynumber estimate. Lower row: Corresponding $\chi^2$ test statistic values $\mathcal{T}$ for the copynumber estimates in the upper row (with a half-window size of 8001).

Any value less than $\theta$ is ignored, since these correspond to cases where the statistical test between the two window halves cannot discard the respective null-hypothesis. The most likely change-points are hence the positions of local extrema or *peaks* of $\mathcal{T}_{>\theta_\alpha}$ (the $\chi^2$ values larger than $\theta$), as the probability of the contents of the left and right windows being "different" is locally largest at these positions.
To reduce the number of potential change-points considered during peak-calling, they are restricted to those points which *locally* also exhibit a considerable change with respect to an effect, such as the difference of means $\bar{D}$ or medians $\dot{D}$ between the window halves $W^l$ and $W^r$:

$$\bar{D} = \frac{1}{n_r} \sum_{j=1}^{|E^r|} j \cdot (E^r_{j+1} - E^r_j) - \frac{1}{n_l} \sum_{j=1}^{|E^l|} j \cdot (E^l_{j+1} - E^l_j)$$
$$\dot{D} = \mathrm{Med}(W^r) - \mathrm{Med}(W^l)$$

In other words, we choose to keep only those change-points where the effect size[51] crosses some threshold $\theta_e$, for example $\theta_e = 1$ for a median difference or $\theta_e = 0.5$ for a mean difference. See Figure 4.19 for an example of $\chi^2$ values and effect sizes for a subset of simulated copynumber estimates.

[51] Absolute value of the effect, e.g. $|\bar{D}|$.

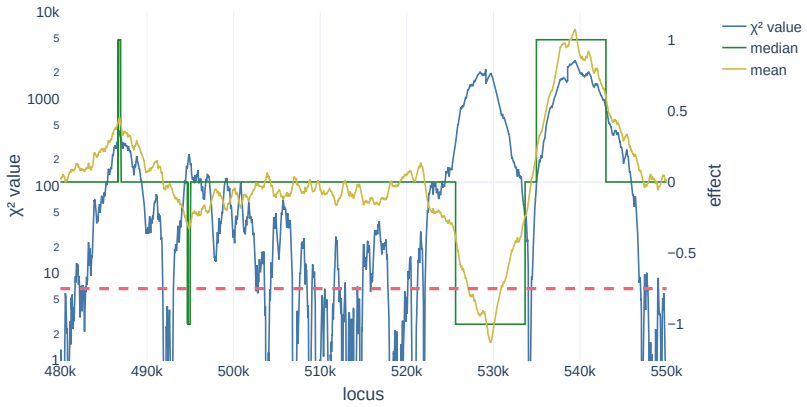



Figure 4.19: $\chi^2$ test statistic values (blue, logarithmic axis) and effect (median: green, mean: yellow), in this case the difference between the medians/means of each window half. The dashed red line marks a significance level of $\alpha = 0.01$ which corresponds to $\chi^2 \approx 6.63$.

For each contiguous range of loci where the $\chi^2$ values are above $\theta_\alpha$ and the effect sizes are at least $\theta_e$, we define the peak as the locus with the largest $\chi^2$ value[52]. The peak is then interpreted as the most likely change-point in the respective range. Any two neighbouring peaks then define a segment, i.e. a contiguous range of loci with the same copynumber level. Additionally, to avoid the effect of overlapping windows (rendering $\chi^2$ values dependent), peaks have to be at least $w/2$ apart. The segments' copynumber levels can again be determined in numerous ways, for example by taking the mean or median of the copynumber estimates in the respective segment.

[52] Using specialized peak detection algorithm remains future work.

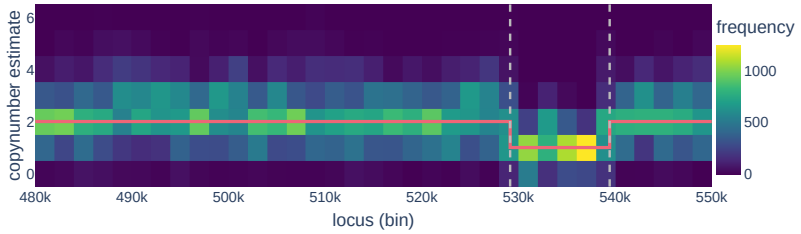



Figure 4.20: The same binned copynumber estimates as in Figure 4.18, this time with change-points marked with dashed grey lines, and segments in red.

Figure 4.20 shows the copynumber estimates and segments for the same subset of simulated copynumber estimates as in Figure 4.18 and Figure 4.19. Peaks were determined using a significance level of $\alpha = 0.01$ and an effect size threshold on the mean of $\theta_e = 0.5$, with peaks at least 8001 loci apart. Segments were determined by taking the median of the copynumber estimates between each pair of peaks.

■ Merging segments Because the detection of changepoints is inherently local, neighbouring segments may end up with identical copynumber level. In that case, all neighbouring segments which were assigned the same copynumber level[53] can be merged.

[53] Up to a certain precision when the mean is used instead of the median.

■ Future work Instead of taking the maximum $\chi^2$ value as the peak, a more sophisticated peak detection algorithm could be used, for example `find_peaks` from the Python package `scipy`.[54]

Another problem to solve is specific to the median test: The median test is not particularly well suited for large sample sizes (i.e. large windows),

[54] Virtanen et al., "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python", 2020.

as its test statistic gets exceedingly large, such that the p-value threshold becomes almost meaningless (though the largest test statistic within any region is still a good indicator of a change-point). A solution to this problem is to use *Cramér's V*, the $\phi$-coefficient or the absolute *MCC*, all of which are identical for a $2 \times 2$ contingency table.[55]

It may also be of interest to use different change-point detection algorithms, such as bayesian online changepoint detection (BOCD)[56].[57] In fact, preliminary experiments using BOCD showed promising results, but need to be evaluated further.

[55] Cramér, *Mathematical methods of statistics*, 1946, chapter 21, page 282; Matthews, "Comparison of the predicted and observed secondary structure of T4 phage lysozyme", 1975.

[56] Adams and MacKay, "Bayesian online changepoint detection", 2007.

[57] Altamirano, Briol, and Knoblauch, *Robust and Scalable Bayesian Online Changepoint Detection*, 2023.

#### 4.4.4 *Alternative segmentation: Recursive segmentation*

Because the method of peak calling and segmentation described above is inherently local (test statistics are for fixed size windows, while segments have variable length), we would also like to evaluate changepoints in a more global manner with variable segment lengths. We therefore make use of ideas initially proposed in Olshen et al. [2004][58]: Calculate the test statistic for all possible segmentations of the data into two segments. Keep the segmentation with the most significant test result, then recurse into each segment. Two ways to do this are the following:

[58] Olshen et al., "Circular binary segmentation for the analysis of array-based DNA copy number data", 2004.

1. Binary segmentation: corresponds to a single split of $\mathcal{C}$ at a position $i$ into segments $\mathcal{C}_{..i}$ and $\mathcal{C}_{i..}$,
2. Circular binary segmentation: corresponds to two splits at positions $i < j$, resulting in two segments $\mathcal{C}_{i..j}$ and $\mathcal{C}_{j..i} = \mathcal{C}_{..i} \cup \mathcal{C}_{j..}$. Note that this also includes binary segmentation (i.e. a single split) when either $i = 1$ or $j = |\mathcal{C}|$.

It has been shown that binary segmentation is prone to miss small segments located in the middle of larger segments,[59] which Circular Binary Segmentation (CBS) is able to detect. We therefore focus on CBS and disregard binary segmentation.

[59] Ennapadam Seshan Venkatraman, *Consistency results in multiple change-point problems*, 1992.

However, CBS was designed for array-based copy number data (i.e. microarray based, such as ArrayCGH), where firstly there are few and sparse probes/loci/markers, in the order of $1 \times 10^4$ to $1 \times 10^5$ and secondly each marker has a sample specific intensity value and a reference intensity. It makes an assumption of the data being normally distributed, either with known variance or without (in that case, the variance can be estimated from the data itself). For non-normal data, a permutation approach to generate a suitable reference distribution is proposed. Since the number of permutations required to calculate the p-value for a small level of significance $\alpha$ is large, an early stop criterion is introduced. Because this is still an expensive procedure, the test is performed on a number of smaller overlapping windows instead, and the test statistic is formulated in terms of these overlapping windows.

Here we have a number of markers (unique k-mers) in the order of the reference length, e.g. $1 \times 10^9$ for the human genome, and each marker has an associated count which can be related to the sample's expected count (coverage). For this number of markers, CBS is not well suited, as its run-time complexity is $O(n^2)$ where $n$ is the number of markers.[60] Even with the

[60] E. S. Venkatraman and Olshen, "A faster circular binary segmentation algorithm for the analysis of array CGH data", 2007.

runtime improvements on the original CBS implementation presented in E. S. Venkatraman and Olshen [2007][61], it remains an infeasible approach for datasets this large.

Therefore, we propose restricting the CBS routine to candidate changepoints: Let $S = (s_1, \dots, s_n)$ be likely changepoints as determined by the sliding window approach given in subsection 4.4.2, and let $W_1, \dots, W_{n-1}$ be *gapless* window data structures as defined in subsection 4.4.2 with $W_i = (\mathcal{C}, s_{i+1} - s_i, s_i, E, n)$.

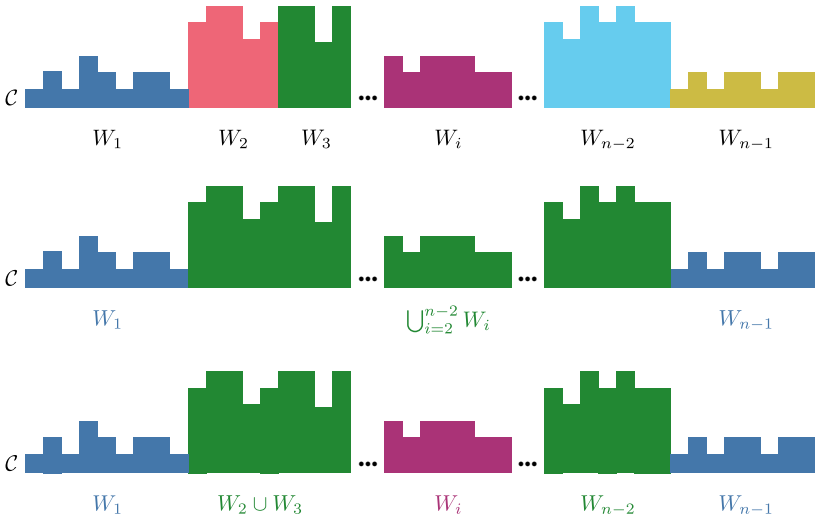



Figure 4.21: Illustration of a CBS like routine for counts $\mathcal{C}$ already segmented with the sliding window approach from subsection 4.4.2. Each window $W_i$ represents the EDF of the counts in the respective segment. In the upper row, each window is the result of the initial segmentation and is indicated by a different colour. In the middle row, a segment consisting of $W_2$ up to $W_{n-2}$ was detected. In the bottom row, the segment corresponding to $W_i$ was detected (within the green segment of the middle row).

Then define

$$W_{l\dots u} = \bigcup_{i=l}^{u} W_i$$

as the union of all neighbouring windows between $l$ and $u$, and the complement (with respect to all windows) as

$$\overline{W_{l\dots u}} = W_{1\dots n} \backslash W_{l\dots u} = \bigcup_{i=1}^{l} W_i \cup \bigcup_{j=u}^{n} W_j$$

for $0 \leq l < u \leq n$.
Let $t : W \times W \to \mathbb{R}$ be a function calculating a test statistic in terms of two windows. Then find the combination of $0 \leq l < u \leq n$ that maximizes the test statistic for the respective window and its complement, i.e. $\hat{l}, \hat{u} = \operatorname{argmax}_{l,u} t(W_{l\dots u}, \overline{W_{l\dots u}})$. If the test statistic is sufficiently large to satisfy the chosen level of significance, recurse into the three (or two if $\hat{l} = 0$ or $\hat{u} = n$) segments $W_{0\dots\hat{l}}$, $W_{\hat{l}\dots\hat{u}}$ and $W_{\hat{u}\dots n}$.

Because set operations on windows can be performed in constant time[62], and there are $\frac{n(n-1)}{2}$ valid combinations of changepoints at each level, this reduced CBS approach takes quadratic time[63] in the number of detected changepoints.

Note that by restricting CBS to (segments defined by) changepoints detected with the sliding window approach in this manner, no new changepoints can be introduced. Rather, only existing changepoints can be discarded, when the respective segments are found to be not significantly different from their enclosing segments.

[61] Ibid., 2007.

[62] The array $|E|$ storing cumulative frequencies of the counts has a maximum size of 256; in practice it is effectively even smaller, since expected copynumbers are usually low.

[63] Assuming the data is split perfectly into three equal segments, the recurrence equation can be stated as $T(n) = 3T(n/3)+n^2$. With the master theorem we find that $T(n) = O(n^2)$.

## 4.5 *Benchmark*

Before we evaluate the performance of the copynumber calling method in section 4.6, we first benchmark the time and memory requirements of the indexing steps and k-mer counting steps.

### 4.5.1 *Indexing*

Time and memory spent for indexing the reference depends on k-mer size $k$, reference length $L$ and whether to discard weak k-mers or not, as can be seen in Figure 4.22 and Figure 4.23. The time spent to determine the

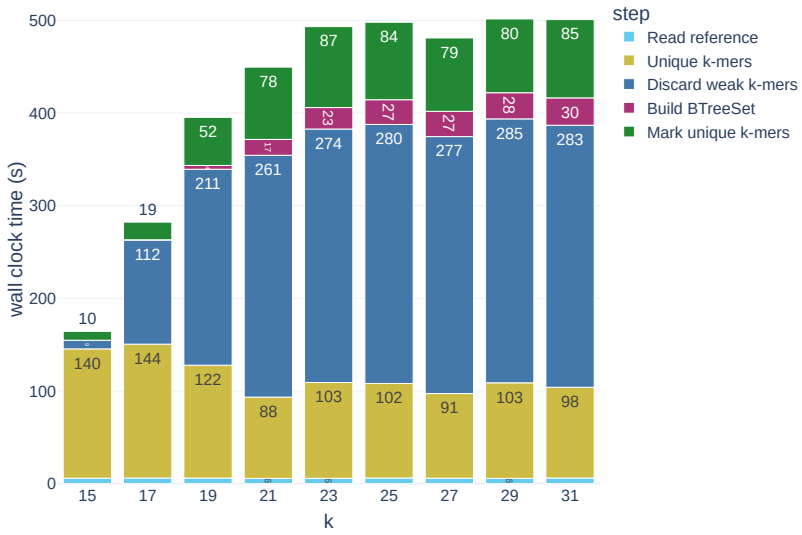



Figure 4.22: Wall clock time spent on building strongly-unique k-mer indices for the human reference genome, for odd $k$ between 15 and 31, on a machine with 88 cores. Timings for indices that were built without discarding weak k-mers are the same except for the Discard weak k-mers step.

number of unique k-mers (yellow bars) does not vary considerably with $k$. This is because the number of k-mers produced, sorted and de-duplicated during the step does not change much for different $k$, as it resembles the length of the input sequence: Assuming the input sequence $s$ only consists of $\{$A, C, G, T$\}$, and no bases are masked (with `N`), there are $|s| - k + 1$ k-mers.

The time spent on discarding weak k-mers however does increase with $k$, or rather with the resulting number of weak k-mers (cf. Figure 4.24 and Figure 4.25).

Time needed for marking unique k-mers in the reference (green bars) is roughly proportional to the number of k-mers in the reference, since this boils down to a set membership test for each k-mer in the reference. This set membership test is done by looking up each reference k-mer in a set backed by a *B-Tree*[64] (a `BTreeSet`[65]) of unique k-mers, constructed from the set of unique k-mers obtained through sorting and de-duplication. Building such a `BTreeSet` of unique k-mers (purple bars) is cheap, as the resulting set of k-mers is already sorted, which the `BTreeSet` can take advantage of. While in theory random access for a `BTreeSet` takes $O(B \log(n))$ time (where $B$ is the node size, and $n$ is the total number of elements), it is fast in practice, due to improved cache efficiency compared to a standard binary search tree.

Note on the choice of $k$: In k-mer applications, $k$ is typically chosen to be as large as possible to minimize ambiguities, and small enough to avoid memory issues. Additionally, $k$ is odd to exclude k-mers whose reverse complements are identical to themselves. Because of the two-bit encoding used in this thesis, $k$ can be at most 32 for 64-bit integers.

[64] Bayer and McCreight, "Organization and maintenance of large ordered indices", 1970.
[65] *BTreeSet* (`https://doc.rust-lang.org/std/collections/struct.BTreeSet.html`, visited on 2023-12-13)

Note that the above timings neither include building the mphf, nor building the quotient filter. The mphf is built using the algorithm described by Limasset et al. [2017][66], which takes about 70s for $k = 27$.
Because this particular quotient filter implementation[67] does not support concurrent construction, the quotient filter is built using a single thread[68], which takes about 500s for $k = 27$.

While building the set of k-mers can be done quickly, its memory requirements are rather large. For example, with the number of unique 31-mers in CHM13 being about $2.43 \times 10^9$, space of at least $2.43 \times 10^9 \cdot 8\text{B} \cdot 1024^3 \approxeq 18\text{GiB}$ is required for storage only. In combination with other overhead, this leads to high memory requirements as shown in Figure 4.23.

[66] Limasset et al., "Fast and scalable minimal perfect hashing for massive key sets", 2017.
[67] *Probabilistic Collections* (`https://docs.rs/probabilistic-collections/latest/probabilistic_collections/`, visited on 2023-08-04)
[68] Parallelizing quotient filter construction can for example be done as described by Maier, Sanders, and Williger ("Concurrent expandable AMQs on the basis of quotient filters").

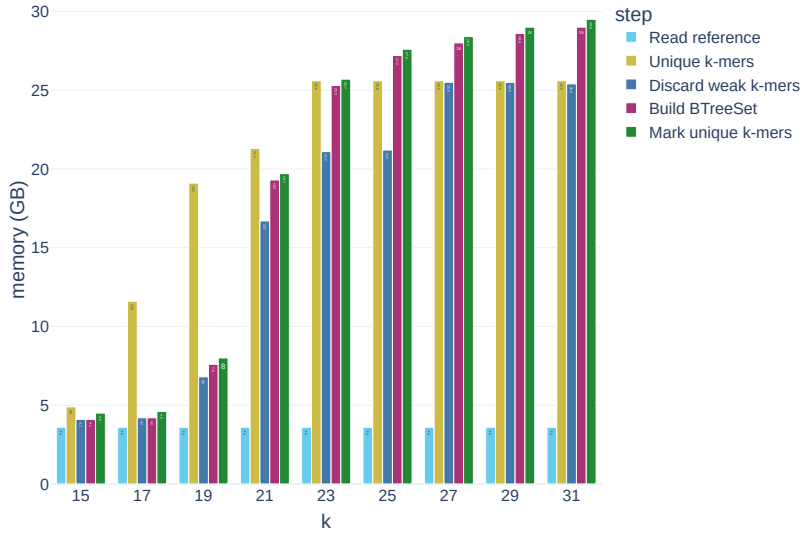



Figure 4.23: Memory allocated during construction of strongly-unique k-mer indices, for odd $k$ between 15 and 31, separated by processing step.

The number of (strongly) unique k-mers in CHM13 for different values of $k$ is shown in Figure 4.24. The gap between strongly unique k-mers

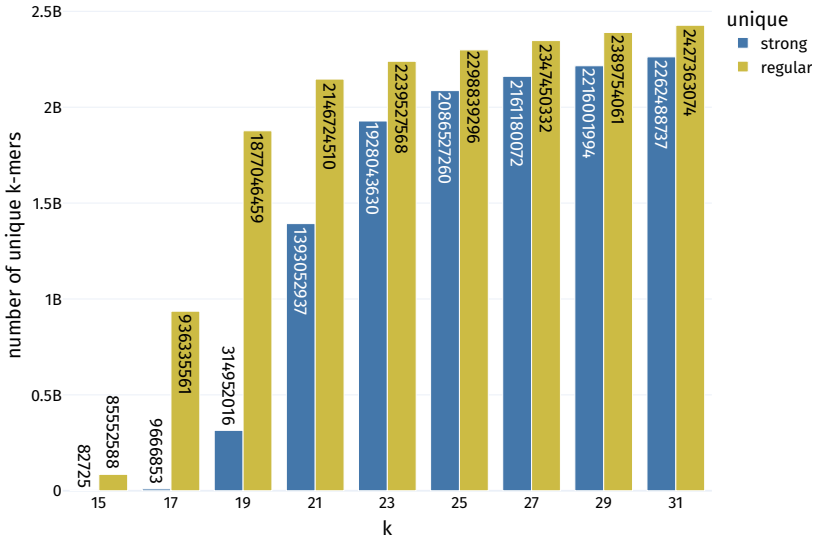



Figure 4.24: Number of (strongly) unique k-mers (y-axis) for CHM13 for different values of $k$ (x-axis).

and unique k-mers closes with increasing $k$. For $k \leq 19$, most of the

unique k-mers are weak k-mers, as can be seen in Figure 4.25. Therefore, we recommend choosing $k \geq 21$ for the human genome.

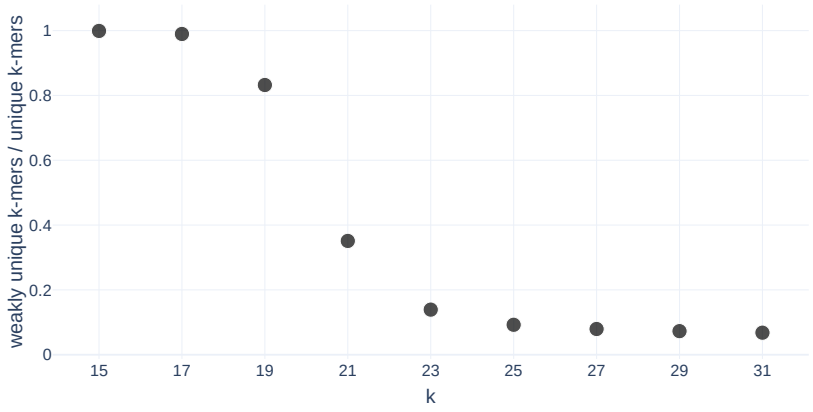



Figure 4.25: Fraction of weak k-mers to unique k-mers (y-axis) for CHM13 for different values of $k$ (x-axis).

### 4.5.2 *Counting*

In addition to the benchmark comparing different k-mer counters in subsection 4.2.4 on a comparably small dataset, we here present a benchmark of the mphf based counter on a larger real-world dataset, using the 27-mer indices for CHM13 built in the previous section[69].

The read dataset consists of Illumina sequencing data for NA12878, with ENA accession `ERR262997`. The 101 bp paired-end reads are stored as 102GiB gzip-compressed FASTQ files, which contain 643 097 275 read-pairs in total.

Using 88 cores, for $k = 27$, our mphf counter takes 2910s wall-clock time.
Because the input data is compressed, some of the time is spent on decompression. Using `pigz`, a parallel gzip decompression utility, with 88-cores, the time spent just on file I/O and decompression is 1483s.
The maximum memory usage is 97GiB

[69] Using $k = 27$ is an arbitrary choice; we would consider any $k$ for which the fraction of weak k-mers is "low enough" (e.g. 23, cf. Figure 4.25) an acceptable choice.

## 4.6 *Evaluation*

To evaluate the performance of our method, we follow the evaluation procedure of Whitford et al. [2019][70] on real data, as well as on simulated data.

[70] Whitford et al., "Evaluation of the performance of copy number variant prediction tools for the detection of deletions from whole genome sequencing data", 2019.

[71] Ibid., 2019.

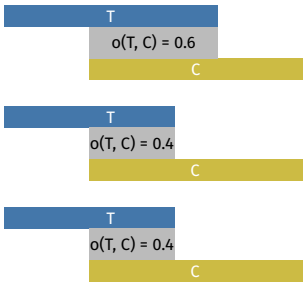
■ Evaluation metrics In order to assess the method's performance, we follow the definition of true positive, false positive and false negative calls, given a ground truth set of segments and their copynumbers from Whitford et al. [2019][71]. Let $\mathcal{T}$ be the set of true segments and $\mathcal{C}$ be the set of called segments (both with associated copynumber). Let $T \in \mathcal{T}$ be a ground truth segment and $C \in \mathcal{C}$ be a called segment. Define the overlap ratio as

$$o'(T, C) = \frac{|T \cap C|}{|T|} .$$

Then define the *reciprocal* overlap ratio as:

$$o(T, C) = \min(o'(T, C), o'(C, T)) = \frac{|T \cap C|}{\max(|T|, |C|)} .$$

The definition of *true positive* (*tp*), *false positive* (*fp*) and *false negative* (*fn*) calls is then as follows:




*tp* A *true positive* is defined as a called segment $C$ that overlaps a true segment $T$ with a reciprocal overlap ratio $o(T, C) \geq 0.5$.

*fp* A *false positive* is defined as a called segment $C$ that does not overlap any true segment $T$ with a reciprocal overlap ratio $o(T, C) \geq 0.5$.

*fn* A *false negative* is defined as a true segment $T$ that does not overlap any called segment $C$ with a reciprocal overlap ratio $o(T, C) \geq 0.5$.

Then define the *true positive rate* as

$$tpr := \frac{tp}{tp + fn}$$

(also called *sensitivity* or *recall*), the *false discovery rate* as

$$fdr := \frac{fp}{tp + fp}$$

and the $F_1$-score as

$$F_1 := \frac{2tp}{2tp + fn + fp},$$

where *tp*, *fp* and *fn* are the number of true positive, false positive and false negative calls, respectively. The $F_1$ score is 1 when *tpr* (or *recall*) and $1 - fdr$ (or *precision*) are 1, and 0 when either *tpr* or $1 - fdr$ are 0.
Note that all of these definitions do not take into account the copynumber of the called segment.

It is also important to note that there are numerous metrics that can be used to evaluate changepoints or segmentations/partitions, which depend on the application[72].[73] Here, we use the above definitions, as the segment-centric view is most prevalent in the literature on copynumber calling.

[72] Aminikhanghahi and Cook, "A survey of methods for time series change point detection", 2017.

[73] Van den Burg and Williams, "An evaluation of change point detection algorithms", 2020.

### 4.6.1 *Simulated data*

To evaluate the accuracy of called copynumbers, data is simulated as follows:
A random reference genome $\mathcal{G}$ of length $3 \times 10^6$ is generated from alphabet $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\} \cup \{\texttt{N}\}$, with corresponding weights $\omega = [0.2826795, 0.2148205, 0.2148205, 0.2826795, 0.005]$[74].
From $\mathcal{G}$, a modified reference genome $\hat{\mathcal{G}}$ is generated by adding/removing copies of (randomly placed) segments of sizes 2000, 5000, 10 000, 50 000 and 100 000[75], such that the resulting copynumbers of these segments are $0, 1, 3$ and $4$. To obtain for example a segment with copynumber 1 in a diploid genome (i.e. ploidy 2), one copy of $\mathcal{G}$ keeps the segment as is, while a second copy of $\mathcal{G}$ drops the segment completely.
From $\hat{\mathcal{G}}$, a set $\hat{\mathcal{R}}$ paired-end Illumina reads of length 100 bp are simulated with `mason`[76] at a target coverage of 6, using artificially high rates of SNVs ($1 \times 10^{-3}$) and small Indels ($1 \times 10^{-4}$).

[74] The weights have been chosen to resemble the GC-content percentage of a human reference genome, with an additional 0.005 fraction of `N`s to increase the number of loci without unambiguous k-mers.

[75] While they resemble common magnitudes of copynumber segment sizes, these sizes are chosen rather arbitrarily, and are multiples of each other at that. In the future, they should rather be drawn randomly from a distribution of segment sizes instead.

[76] Holtgrewe, "Mason–a read simulator for second generation sequencing data", 2010.

We then proceed as described in this chapter:

1. Build a k-mer index of $\mathcal{G}$,
2. count unique k-mers of $\mathcal{G}$ in $\hat{\mathcal{R}}$,
3. estimate GC-stratified coverage using these counts,
4. preprocess the counts,
5. perform statistical tests and
6. segment the signal.

Both simulation and calling steps introduce hyperparameters to examine:

| Parameter | Description |
|---|---|
| Coverage | Target coverage of the simulated reads. |
| Read errors/variants | Rate of SNVs and Indels in the simulated reads. |
| Segment sizes | Sizes of the segments that are added/removed to/from the reference genome. |
| Copynumbers | Copynumbers of the simulated segments. |
| k-mer size ($k$) | Length of the k-mers used to index the reference genome. |
| Coverage estimation method | Method used to estimate coverage from the counts. |
| Window size ($w$) | Size of the sliding window used to gather statistics. |
| Statistical test | Statistical test used to detect changepoints. |
| Level of significance ($\alpha$) | Level of significance used for the statistical test. |
| Effect size | Effect size measured between window halves. |
| Effect size threshold | Threshold used to discard changepoints with low effect. |
| Resolution ($r$) | Scale copynumber estimates by $r$ before rounding. |
| Circular binary segmentation | Whether to use circular binary segmentation on changepoint candidates. |

Table 4.3: Overview and description of potential hyperparameters to be examined in the evaluation.

Thoroughly performing hyperparameter examination/optimization for all possible combinations in Table 4.3 is not feasible[77]. However, preliminary experiments indicate that the choice of window size has the largest impact on the performance of the method, and that choice of test-statistic and level of significance have a smaller impact. We therefore focus on the parameter and value combinations stated in Table 4.4.

[77] For future work: Using bayesian optimization for hyperparameter tuning could be a better alternative.

### Evaluation of copynumber calls for the simulated dataset

Copynumber calls are evaluated on the simulated dataset for the product of the following parameters:

| Parameter | Examined values |
|---|---|
| k-mer size ($k$) | $17, 19, 21, 23, 25, 27, 29, 31$ |
| Coverage estimation method | naïve mode, MLE-NB mode, MLE-NB pseudo mode |
| Window size ($w$) | $\{1000, 1500, \ldots, 9500, 10000\} \cup \{11000, \ldots, 15000\}$ |
| Level of significance ($\alpha$) | $0.01, 0.001$ |
| Statistical test | median, kolmogorov-smirnov |
| Resolution, effect-size, threshold | $(1, \text{mean}, 0.5)$, $(1, \text{mean}, 1)$, $(1, \text{median}, 0.5)$, $(1, \text{median}, 1)$, $(2, \text{mean}, 1)$, $(2, \text{mean}, 1.5)$, $(2, \text{mean}, 2)$, $(2, \text{median}, 1)$, $(2, \text{median}, 1.5)$, $(2, \text{median}, 2)$, $(4, \text{mean}, 2)$, $(4, \text{mean}, 2.5)$, $(4, \text{median}, 2)$, $(4, \text{median}, 2.5)$ |
| Use CBS | False, True |
| Use strongly unique k-mers | False, True |

Table 4.4: Overview of examined hyperparameters for the simulated dataset. For a description of the parameters, see Table 4.3.

In Figure 4.26, we show the distribution of $F_1$ values across all $112896$ parameter combinations. While the mean $F_1$ value is $0.67$ and the median

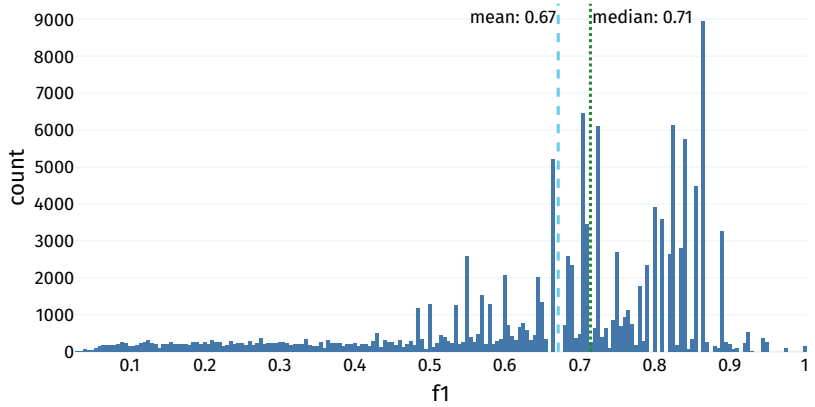



Figure 4.26: Overview of the distribution of $F_1$ values across all parameter combinations.

is $0.73$, the most frequent value is $0.86$. The highest attained $F_1$ value is $1.0$, and the lowest is $0.03$. Because there are exactly $20$ segments with copynumber aberrations, the maximum attainable number of true positives and false negatives is $20$. However, in this evaluation, the minimum number of true positives is $6$, and the maximum number of false negatives is subsequently $14$. The number of false positives ranges from $0$ to $666$, with most of the parameter combinations resulting in $0$ false positives. See Figure 4.27 for a corresponding histogram.

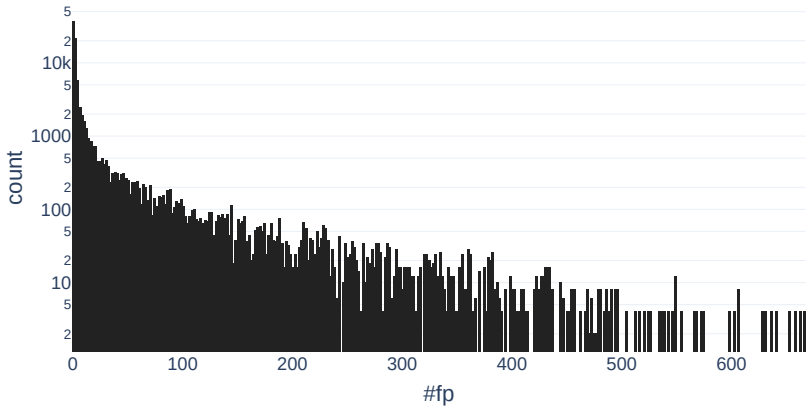



Figure 4.27: Histogram of the number of false positives across all parameter combinations. The y-axis is logarithmic.

As mentioned earlier, preliminary experiments indicated that the choice of window-size has the biggest impact on the performance of the method. Therefore, we stratify the results by window-size in Figure 4.28. For the

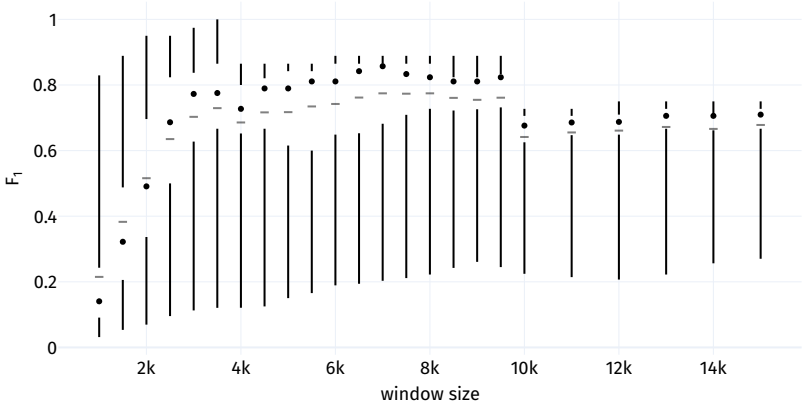



Figure 4.28: Effect of window size on $F_1$ score. Data is aggregated over all possible parameter combinations, grouped by window-size. There is one (tufte) box-plot for each window size, with the middle circle indicating the median value for the respective window-size, and the middle horizontal line indicating the mean. Vertical lines indicate the lower and upper $25\%$-quantiles, i.e. the gap between these lines resembles the central $50\%$ of the data.

simulated data, window sizes between $3000$ and $9500$ tend to result in larger $F_1$ values, with the minimum $F_1$ value increasing with window size (up to a window size of $9500$). Notice, however, that there are perceptual jumps in the $F_1$ value distributions from window-size $3500$ to $4000$ and from $9500$ to $10\,000$. These are likely due to the fact that there are only 5 different segment sizes in the simulated data, and that different window sizes are more or less suitable for detecting different segment sizes. By stratifying calls by segment size as shown in Figure 4.29, we can see that certain window-sizes are not suited for detecting certain segment sizes. This is mainly due to the fact that we enforce peaks to be at least $w/2$ bp apart, to ensure that the test-statistic is not biased by the same k-mers being counted in both window halves.

To get a general idea of the effect of the other parameters, we will now examine the effect of each parameter on the $F_1$ score. To do so, for each parameter, the difference between the $F_1$ scores of the data grouped by the respective parameter and the global mean $F_1$ is calculated. The mean *absolute* difference of the respective values then gives an indication of the impact of the parameter on the $F_1$ score. A corresponding summary plot is

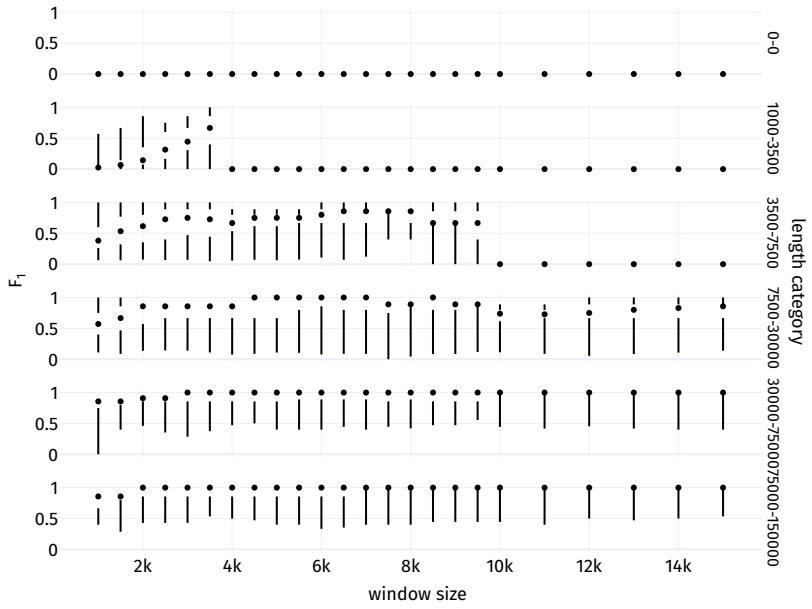



Figure 4.29: Effect of window size on $F_1$ score, stratified by segment length. The category "0-0" refers to segments whose size does not match any of the simulated segment sizes, so naturally the $F_1$ value is 0 in these cases. Note that the $F_1$ values are relative to the respective segment length category, so they are not comparable to the non-stratified values in previous figures.

shown in Figure 4.30. As expected, the window-size is the most impactful parameter, while a combination of effect-size, effect-size threshold and resolution are the next most important. The choice of $k$, coverage estimation method, and whether to use CBS are less important. The choice of statistical test, level of significance and whether to use strongly unique k-mers are irrelevant for the simulated data.

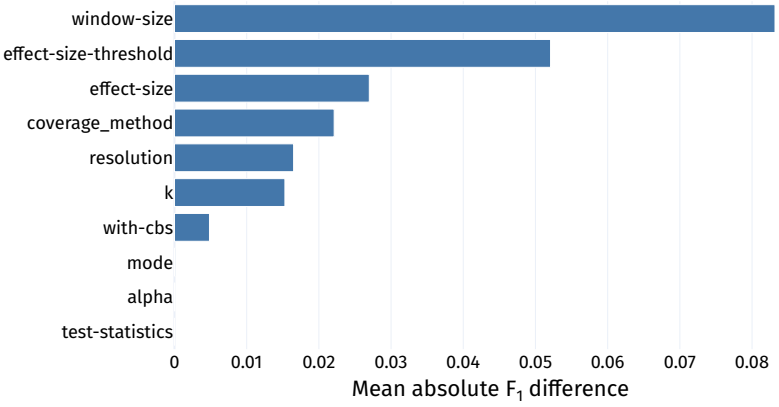



Figure 4.30: Mean absolute difference between global mean $F_1$ and parameter-grouped mean $F_1$.

To get a more detailed picture, Table 4.5 lists each parameter that was examined, and the observations made from the evaluation of the simulated dataset with respect to that parameter. Here, we further stratify by the values of each parameter, and the mean (signed) difference from the global mean $F_1$ is plotted in the column *Plot*.

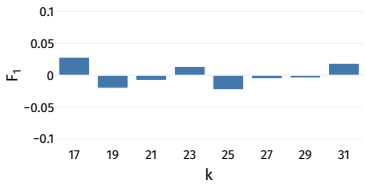
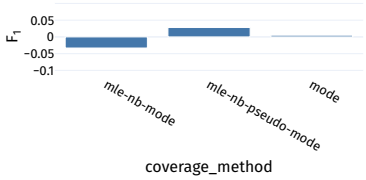
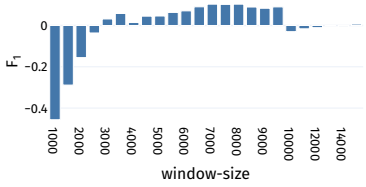
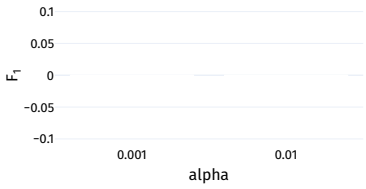
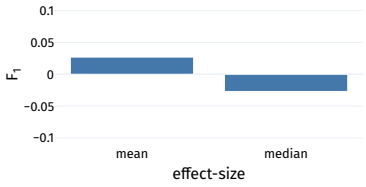
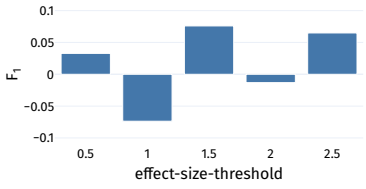
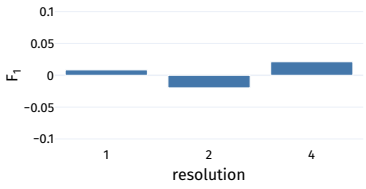
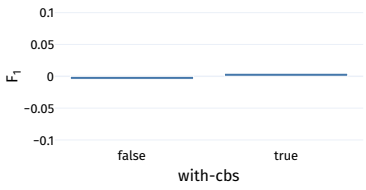
| Parameter | Observations | Plot |
|---|---|---|
| k-mer size ($k$) | The influence of the choice of k-mer length $k$ is low for the simulated dataset. Interestingly, the lowest ($k = 17$) and the highest ($k = 31$) choices show the highest scores. |   |
| Coverage estimation method | Using a fractional pseudo-mode works better than using the theoretically correct mode. This is most likely due to different rounding behaviour during GC-bias correction (see subsection 4.3.1). |   |
| Window size ($w$) | The choice of window-size is the most important parameter. Window sizes less than 3000 are detrimental. |   |
| Statistical test | The choice of test statistic is not relevant at all, likely because sample sizes are so large that test statistic values are often $\gg \theta$. |   |
| Level of significance ($\alpha$) | The choice of level of significance is not relevant for the simulated dataset. |   |
| Effect size | Overall, using the *mean* as effect size scores better than the *median*. |   |
| Effect size threshold | Fractional effect size thresholds are better than integer ones. |   |
| Resolution ($r$) | Changing the resolution potentially allows detecting fractional copynumber changes, but also interacts with the effect-size threshold. For the simulated data with integer copynumbers, a resolution of 2 has a negative impact. |   |
| Circular binary segmentation | Using CBS improves results very slightly (at the cost of runtime). |   |

Table 4.5: Overview of observations made from the evaluation of the simulated dataset, for each parameter that was examined. The column *Plot* contains a plot showing the mean difference of the data stratified by the parameter on the x-axis from the global mean $F_1$ value.

Because the data clearly has interactions between the parameters, we will now examine the effect of some parameter combinations.

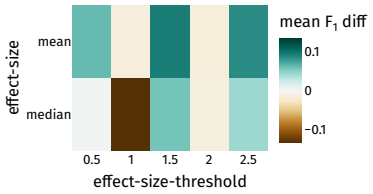
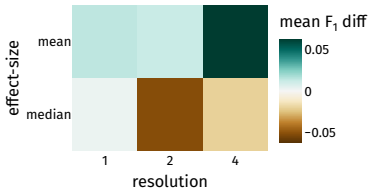
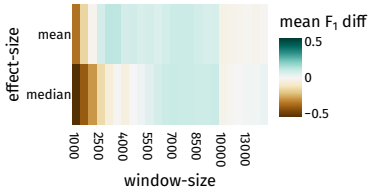
| Parameter combination | Observations | Plot |
|---|---|---|
| Threshold and effect | For both *mean* and *median*, fractional effect size thresholds are better than integer ones. |   |
| Resolution and effect | For the *mean*, a resolution of 4 is best, while for the *median* it is a resolution of 1. |   |
| Window size and effect | Smaller window sizes benefit from using the *mean* as an effect size, while larger windows fare better using the *median*. |   |

Table 4.6: Overview of observations for *combinations* of parameters.

Because the parameters *window-size*, *effect-size*, *effect-size-threshold* and *resolution* interact, Figure 4.31 shows the effect of these parameters on the $F_1$ score.

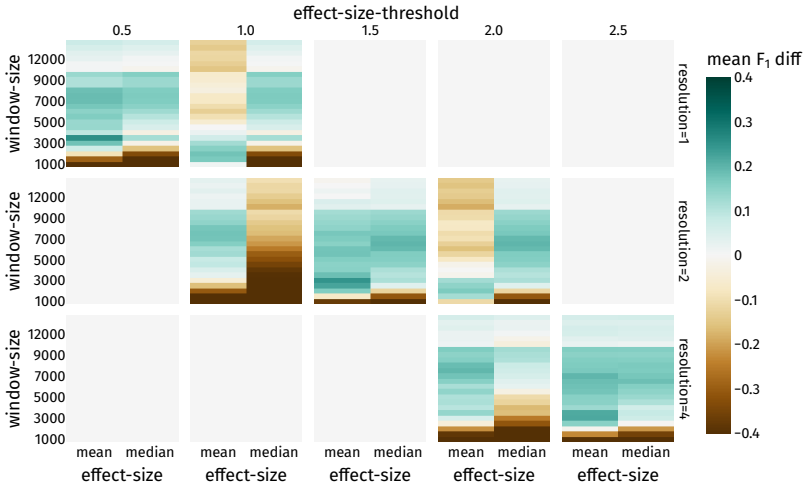



Figure 4.31: Deviation from the global mean $F_1$ (color-axis), stratified by *window-size* (y-axis), *effect-size* (x-axis), *effect-size-threshold* (column facets) and *resolution* (row facets).

For sensible choices of the parameters, the effect size threshold behaves differently between the *mean* and *median* as effect sizes — while the *mean* benefits from lower thresholds, the *median* benefits from higher thresholds. Clearly, the pair of *effect-size-threshold* and *resolution* does not exhibit a linear relationship with the $F_1$ score. For example, the results for threshold 0.5 and resolution 1 are very different from those for threshold 1.0 and

resolution 2 as well as for threshold 2.0 and resolution 4, which becomes especially apparent for the *median* as the chosen *effect-size*.

## 4.7 Comparison with other methods

In this section, we compare our method with results from a benchmark introduced in Whitford et al. [2019][78]. This benchmark compares the performance of different CNV calling methods on WGS data of the NA12878 sample and uses GIAB high confidence deletion calls as ground truth. The CNV callers that are compared differ in the information they use to call CNVs:[79]

- `BreakDancer`[80] uses read pair information,
- `CNVnator`[81] uses read depth information,
- `DELLY`[82] uses both read pair and split-read information,
- `FermiKit`[83] is an assembly based approach, and
- `Pindel`[84] uses split-read information.

[78] Whitford et al., "Evaluation of the performance of copy number variant prediction tools for the detection of deletions from whole genome sequencing data", 2019.

[79] Ibid., 2019, Table 1.

[80] Chen et al., "BreakDancer: an algorithm for high-resolution mapping of genomic structural variation", 2009

[81] Abyzov et al., "CNVnator: an approach to discover, genotype, and characterize typical and atypical CNVs from family and population genome sequencing", 2011

[82] Rausch et al., "DELLY: structural variant discovery by integrated paired-end and split-read analysis", 2012

[83] Li, "FermiKit: assembly-based variant calling for Illumina resequencing data", 2015

[84] Ye et al., "Split-read indel and structural variant calling using PINDEL", 2018

The approach presented in this chapter uses k-mer counts as input, which is most similar to read depth information.

Ground truth calls are filtered to only include deletions with a minimum length of 1000bp, resulting in 612 ground truth deletion calls. A called CNV is considered a *true positive* (*tp*) if its reciprocal overlap with a ground truth CNV is at least 50 percent, otherwise it is considered a *false positive* (*fp*). If a ground truth CNV has no reciprocal overlap with any called CNV, it is considered a *false negative* (*fn*). The benchmark then computes *true positive rate* (*tpr* or *sensitivity*) and *false discovery rate* (*fdr*) for each method.

$$\text{tpr} = \frac{\text{tp}}{\text{tp}+\text{fn}}$$

$$\text{fdr} = \frac{\text{fp}}{\text{tp}+\text{fp}}$$

The results are further stratified by deletion length into the following bins: 1 to 1.5kbp, 1.5 to 2kbp, 2 to 3kbp, 3 to 4kbp, 4 to 5kbp, 5 to 6kbp, 6 to 10kbp, 10+ kbp.
However, it is unclear how exactly the stratification is performed, especially regarding *false positive* calls. We therefore only compare with the overall statistics.[85]

[85] Whitford et al., "Evaluation of the performance of copy number variant prediction tools for the detection of deletions from whole genome sequencing data", 2019, Table 2.

■ Choice of parameters From the evaluation of the method on simulated data, we know that the choice of window size has the biggest impact on the performance of the method, with the choice of effect, effect-size threshold and resolution as the next most important parameters. The best performing instances of the method on the simulated data use a window size of 3500, a resolution of 1, the *mean* as effect, and an effect-size threshold of 1.5. Because the minimum length of ground truth deletions is 1000bp, we choose a slightly smaller window size of 2500 for the benchmark. Also, because the coverage of the NA12878 sequencing data is higher than the simulated data, we choose a resolution of 2.

■ Results The results of applying our method to the GIAB ground truth deletion call data for NA12878 are shown in Table 4.7. This table additionally contains the results of the other CNV calling methods from the original benchmark, extended by the $F_1$-score.

| Method | total | tp | fp | fn | tpr | fdr | $F_1$ |
|---|---|---|---|---|---|---|---|
| This | 851 | 259 | 239 | 353 | 0.42 | 0.48 | 0.47 |
| BreakDancer | 868 | 567 | 300 | 45 | 0.93 | 0.36 | 0.77 |
| CNVnator | 1300 | 404 | 898 | 208 | 0.66 | 0.69 | 0.42 |
| Delly | 1884 | 592 | 1292 | 20 | 0.97 | 0.69 | 0.47 |
| FermiKit | 141 | 97 | 45 | 515 | 0.16 | 0.32 | 0.26 |
| Pindel | 5139 | 423 | 4693 | 189 | 0.69 | 0.91 | 0.15 |

Table 4.7: Results of applying our method to the GIAB ground truth deletion call data for NA12878. The table additionally contains the results of the other CNV calling methods from the original benchmark, extended by the $F_1$-score.

Our method is neither the best nor the worst performing method. The best performing method in terms of $F_1$ score is `BreakDancer`, which uses read pair information. Compared to `CNVnator` which uses read depth information which is the kind of information most closely related to the k-mer count information we use, our method performs slightly better in terms of $F_1$ score.

■ Future work To locate potential targets for improvement, we can look at the *false positive* calls of our method in more detail, to see if they share a pattern. It may also be of interest to look at the number of calls that are shared between our method and the other methods. Additionally, a comparison with the results of the QuicK-mer pipeline[86] would be interesting, as it is the only other method that makes use of k-mer counts (produced by `jellyfish`) as input.

[86] Shen and Kidd, "QuicK-mer: A rapid paralog sensitive CNV detection pipeline", 2015.

## 4.8 *Discussion*

In this chapter, we presented a method to identify copynumber variation in WGS data, along with all necessary steps of indexing strongly unique k-mers in a reference sequence, counting these k-mers in WGS reads, normalizing and correcting counts for GC-bias, detecting likely changepoints based on statistical testing, followed by partitioning of copynumber estimates into segments (induced by changepoints) with equal value. It is a novel approach based on k-mer counts instead of read-depth and/or read-pair information, which obviates the need for explicit read mapping[87]. It is most similar to read-depth approaches, as it makes use of count information available for each locus. While the current method is not the best performing tool for CNV calling, the results are comparably good (especially compared to other read-depth based methods), and there is much room for improvement in every single step, most notably the change-point detection step. Because raw k-mer counts are transformed into absolute copynumber values, the method works in both single-sample and paired-sample (e.g. tumour and normal pairs) scenarios. In the latter case, one can simply use the differences of the counts between the two samples as the input for the subsequent steps.

[87] The k-mer counting step can be viewed as a form of pseudo-mapping, i.e. roughly locating a sequence's k-mers but foregoing the actual alignment of the read sequence against the reference.

# 5 *vembrane*

*The contents of this chapter have been published in Hartmann, Schröder, et al. [2022]*[1].

[1] Hartmann, Schröder, et al., "Insane in the vembrane: filtering and transforming VCF/BCF files", 2022.

Almost all variant callers output their results in the Variant Call Format (VCF). A variant call at the very least carries information about the genomic locus (or loci in case of structural variation) it affects and the modification of bases it makes. Usually it carries additional data, for example the certainty of making this call, meta information about the caller(s), the number of supporting reads having this specific variant, or annotation from databases, effect predictors and other custom sources. Downstream analyses are often concerned only with calls which match very specific requirements, for example due to experimental design or simply in order to keep the list of candidates considered for further analysis small. It is therefore a common task to *filter* variant calls based on user specified requirements.

## 5.1 *The Variant Call Format*

| Field | Description |
|---|---|
| `CHROM` | chromosome / contig of a call |
| `POS` | position in bp in the contig |
| `ID` | an ID or name, e.g. for known variants |
| `REF` | reference allele (usually starting at `POS`) |
| `ALT` | (possibly multiple) alternative alleles |
| `QUAL` | quality of a call |
| `FILTER` | list of flags indicating which filters defined in the header apply |
| `INFO` | a variable length list of key-value pairs as defined in the header |
| `FORMAT` | a definition of key-value pairs that can be queried for each sample described in the file |
| `${SAMPLE}` | For each sample, an additional column can be added with associated values specified according to the value in `FORMAT` |

Table 5.1: Overview of VCF fields. See Figure 5.1 for an exemplary VCF file.

The VCF and its binary companion BCF are file formats commonly used

for storing variant calls obtained from variant calling workflows such as `dna-seq-varlociraptor`[2].

VCF consists of a header and a sequence of variant calls. The header contains meta information, for example information about reference sequences, annotations and additional fields. Additional fields are specified as key-value pairs with type information for the value. Each variant call — in the form of a single line in the text format VCF or *record* in the binary format BCF — has certain mandatory and optional fields as shown in Table 5.1[3].[4] An exemplary VCF file is shown in Figure 5.1.

[2] *dna-seq-varlociraptor* (`https://github.com/snakemake-workflows/dna-seq-varlociraptor`, visited on 2023-03-07)

[3] *VCFv4.4* (`https://samtools.github.io/hts-specs/VCFv4.4.pdf`, visited on 2023-03-07)

[4] Danecek et al., "Twelve years of SAMtools and BCFtools", 2021.

```
##fileformat=VCFv4.3
##reference=artificial_reference
##contig=<ID=chr1,length=1234567>
##INFO=<ID=flt,Number=1,Type=Float,Description="A single precision float">
##INFO=<ID=ANN,Number=.,Type=String,Description="Example annotations. Format: Allele|IMPACT|CLIN_SIG">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Depth">
#CHROM POS ID REF ALT QUAL FILTER INFO                          FORMAT Sample1 Sample2
chr1   7   .  A   C   40   PASS   flt=0.6;ANN=C|HIGH|pathogenic GT:DP  ./.:5   0|1:7
```

Figure 5.1: An exemplary VCF file, with pseudo annotations and genotyping information for two diploid samples named `Sample1` and `Sample2`. The single line after the header lines (lines starting with `#`) encodes a variant on the contig named `chr1` at locus 7, where the reference allele `A` has been substituted with the alternative allele `C`. In its `INFO` column, additional information has been stored: A value of 0.6 for the key `flt`, and the string `C|HIGH|pathogenic` for the key `ANN`. By convention, the `ANN` key is reserved for annotation from external sources and tools, such as `VEP`. The `FORMAT` column describes the layout of information that is sample specific: For each sample, first the genotype (`GT`) is given as a string, then the read or sequencing depth (`DP`) as an integer.

## 5.2 *VCF filtering tools*

As implied above, the task addressed by VCF filtering tools is to either keep or discard records that match certain criteria. To this end, many VCF filtering tools have been developed over the years; an overview is given in Table 5.2. While all of these tools are concerned with the task of filtering variant calls based on their properties and annotations, the general focus and use-cases as well as syntax differ between them.

For example, `filter_vep` mostly concerns itself with Variant Effect Predictor (VEP) annotations, `slivar` pays additional attention to the filtering of group constellations (such as trios with one child and two parents), while `bcftools` aims to be a general VCF toolkit (i.e. not restricted to filtering use-cases).

| tool name | syntax | annotation support |
|---|---|---|
| `bcftools` | custom | `VEP` (via `split-vep` plugin) |
| `bio-vcf` | custom / ruby | none |
| `filter_vep` | custom | `VEP` |
| `slivar` | custom / js | `SnpEff` (partially), `VEP` (partially) |
| `SnpSift` | custom | `SnpEff` |
| `VcfFilterJdk` | Java | `SnpEff` (obsolete `EFF` version), `VEP` |
| `vembrane` | Python | `SnpEff`, `VEP` |

Table 5.2: Overview of VCF filtering tools.

With `vembrane` we provide yet another alternative VCF filtering tool.

However, we strive to keep the command line interface simple, the built-in support for annotations high, the expressive capabilities high and the syntax easy to read, understand and write.

Syntax is of special importance: Most of the tools listed in Table 5.2 define their own custom syntax or domain specific language. From a developer's point of view, this implies writing a grammar with a corresponding parser (or implicitly defining a grammar by means of the parser implementation — this however makes it more likely to get an inconsistent language) and defining reasonable and correct semantics in tandem. From a user's point of view, this means having to learn a new language with possibly unconventional semantics (see section 5.4 for examples) and having to work around unintended shortcomings of the language design.
For these reasons, we think it makes sense to use an established language for evaluating filter expressions. Here we choose Python, so that `vembrane`'s syntax and semantics are implicitly defined by the Python language specification. Figure 5.2 gives an example of a `vembrane filter` expression, showcasing common features such as boolean operators, set operations and generators/comprehensions.

```
CHROM == "chr2" and (QUAL >= 30 or ID in AUX["known"]) and \
  not {"pathogenic", "drug_response"}.isdisjoint(ANN["CLIN_SIG"]) and \
  sum(without_na(FORMAT["DP"][s] for s in SAMPLES if is_hom(s))) > 10
```

Figure 5.2: Example `vembrane filter` expression. It selects only those variants which are located on chromosome 2 and have a quality of at least 30 or their ID is in the set of known IDs, and where at least "pathogenic" or "drug_response" is part of the clinical significance annotations, and where the sum of read depths across all samples with a homozygous genotype is at least 10.

## 5.3 *Implementation*

The basic implementation is straightforward: For reading, parsing and writing VCF files, `pysam`[5] is used. A user specified expression is evaluated with Python's `eval` function for each record in the file. If the expression evaluates to `True`, the record is kept, otherwise it is discarded. The environment a record is evaluated in is constrained to a limited set of variables and functions:

- All fields of the VCF format as listed in Table 5.1,
- the set of samples described in the file,
- functions from Python's `math` and `statistics` modules,
- the regular expression module `re`,
- custom VCF related functions,
- user-provided sets of strings,
- parsers for `SnpEff` and `VEP` annotations.

Additionally, certain Python specific internals[6] are explicitly cleared to make it more difficult to execute potentially malicious code[7].

[5] *pysam* (https://github.com/pysam-developers/pysam, visited on 2023-03-07)

[6] e.g. `globals`, `builtins`, any accesses starting with `.__`.

[7] See https://nedbatchelder.com/blog/201206/eval_really_is_dangerous.html (accessed on 2023-11-08) for more information about why `eval` can be problematic.

Figure 5.3 gives a pseudocode overview of the main program flow.

It is of course not always as straightforward as indicated by the pseudocode (in Figure 5.3); there are at least four cases which need special considerations: Missing values (subsection 5.3.1), multiple alternative alleles (subsection 5.3.2), multiple annotations per allele (subsection 5.3.3) and BNDs (subsection 5.3.4). We will discuss these cases in the following sections.

```
i, o, expression = args.input, args.output, args.expression
with VcfFile(i, "r") as vcf_in, VcfFile(o, "w") as vcf_out:
    # setup the environment VCF records are evaluated in
    env = Environment(expression)
    for record in vcf_in:
        # update the environment with information
        # from the record, such as record.CHROM
        env.update(record)
        # evaluate the expression in the updated environment
        keep_record = env.evaluate()
        if keep_record:
            vcf_out.write(record)
```

Figure 5.3: Pseudocode for the main program flow of `vembrane`.

### 5.3.1 *Missing values*

There are two main reasons for missing values in VCF files:
Firstly, the VCF specification allows missing values by means of the special character `.` (singular dot).
Secondly, some fields may be missing *completely* for some records — for example, the header could include an `INFO` field `forget_me_not` which is only present in some records but missing in others.

To facilitate handling of missing values, we introduce the special value `NA` (not available) which is distinct from `None` (which is Python's representation of "missing" values), with the following semantics:

- Any comparison involving `NA` results in `False`.
- Any attribute access such as `NA.foo` results in `NA` (and a warning).
- For string operations, `NA` is treated as the empty string.
- Only Python's `is` operator can be used to check for `NA` (e.g. `foo is NA`).

Additionally, accesses to the `INFO` and `FORMAT` dictionaries are handled as follows: Any access of the form `INFO['foo']` behaves the same as `INFO.get('foo', NA)`, which does a lookup of the key `'foo'` in the `INFO` dictionary, returning `NA` as a default if the key is not present.
Subsequently, to replace `NA` with a default value `x`, either `INFO.get('foo', x)` or `INFO['foo'] or x` can be used[8].
Because missing values can also appear as part of a list, we also introduce two auxiliary functions `without_na` and `replace_na`, which remove `NA` values and replace `NA` values with a default value respectively.

[8] Because `NA` is treated as `False` in boolean contexts, Python keeps evaluating the expression until it finds a value that is *truthy* or the end of the expression is reached

### 5.3.2 *Multiple alternative alleles*

The VCF specification allows multiple alternative alleles per record. This means that one record can have information on many variants at the same genomic locus. However, such variants are usually neither synonymous nor have the same annotations. There are two reasonable ways to handle these records:

1. Let the user handle the multiplicities by means of language features.
2. Require that records must be normalized prior to processing, splitting one multi-allelic record into multiple single-allelic records.

We argue for (and implemented) option 2: While a single record with more than one alternative allele allows sharing meta information about the locus, potentially saving space, it also describes multiple *semantically different* variants. This can be problematic, as two (or more) different alternative alleles may result in wildly different effects, as well as effect predictions and annotations.
Figure 5.4 gives an example of such a record (`ID` set to `multi`) and also contains the two records obtained by splitting the record into separate singular alternative allele records (`ID`s set to `single1` and `single2` respectively). The first thing that needs consideration is how to handle the `INFO` field named `int`, which specifies one integer per alternative allele in the record. If it were handled as an array or list with typical indexing semantics, then `int[0] == 1` would be a valid filter pseudo-expression for all records in the example, but checking for `int[1] == 2` would fail for the single allelic records, because index 1 would be out of bounds for these.

Almost all filtering tools opted to introduce special handling for this case (and other multiplicities as well) by introducing either or both of the special array indices `*` and `?`. In this example, the indices expand to the disjunction `any(v == 1 for v in int)` and the conjunction `all(v == 1 for v in int)` respectively.

Tools with these special array indices also tend to default to one of the two options when omitting the array access / index completely. For example, in `bcftools` `INFO/int == 1` is equivalent to `INFO/int[*] == 1` (`any`), while for `SnpSift` `int == 1` is equivalent to `int[?] == 1` (`all`).

```
##INFO=<ID=int,Number=A,Type=Integer,Description="An allele specific integer">
##INFO=<ID=ANN,Number=.,Type=String,Description="Example annotations. Format: Allele|IMPACT">
#CHROM POS ID       REF ALT   QUAL FILTER INFO
chr1   7   multi    A   C,T   40   PASS   int=1,2;ANN=C|HIGH,T|LOW,T|MODIFIER
chr1   7   single1  A   C     40   PASS   int=1;ANN=C|HIGH
chr1   7   single2  A   T     40   PASS   int=2;ANN=T|LOW,T|MODIFIER
```

Figure 5.4: Excerpt of a VCF file with one multi-allelic record (with `ID` `multi`), and two single-allelic records as the result of splitting the multi-allelic record (with `ID`s `single1` and `single2`).

### 5.3.3 *Multiple annotations per allele*

Even for records with exactly one alternative allele, there may be several distinct annotations for the variant in its `ANN` field. This occurs, for example, when there are multiple transcripts of a gene, or when a single variant affects both upstream and downstream genes. In these cases, given a filter

expression making use of the annotation field, some annotations may be discarded while others may not. A natural way to handle this is to evaluate the expression separately for each annotation, and keep the record if at least one of these evaluations results in `True`, as shown in Figure 5.5. Here we can choose to keep either all annotations or only those that pass.

```
i, o, expression = args.input, args.output, args.expression
with VcfFile(i, "r") as vcf_in, VcfFile(o, "w") as vcf_out:
    env = Environment(expression)
    for record in vcf_in:
        env.update(record)
        annotations = record.annotations
        kept = [a for a in annotations if env.evaluate(a)]
        if len(annotations) > len(kept):
            record.annotations = kept
        if len(kept) > 0:
            vcf_out.write(record)
```

Figure 5.5: Pseudocode for the main program flow of `vembrane`, adapted for multiple annotations.

### 5.3.4 *Breakends*

BND notation in the VCF allows modelling complex structural variation. Multiple BNDs usually form a single cohesive event, for example describing recombinations of parts from different chromosomes. See Figure 5.6 for an exemplary pseudo VCF file containing multiple groups of breakends. It is therefore crucial to keep the integrity of events intact by making sure they are either kept or discarded *as a whole*. However, the BNDs forming an event may be scattered both across the genome and the VCF file. Thus, the filtering of records belonging to such an event can only ever occur once at least one BND record relating to this event has passed the filter expression. This implies that the output order of VCF records will not be identical to the input order. We implemented two different approaches to handle this:

1. A two-pass approach, where the first pass collects all BND records, and the second pass then considers both regular and BND records in the correct order.
2. A single-pass approach, where the output order is not guaranteed to be identical to the input order.

In case of the two-pass approach, during the first pass, all BND records are memorized, skipping all non-BND records. This allows us to process both regular and BND records in the input order during the second-pass, because we can now group records belonging to the same event together and determine whether to keep or discard them as a whole.
In the case of the single-pass approach, non-BND records are processed as usual, while BND records are buffered as long as it is unclear whether the event they belong to has to be kept (i.e. at least one part of the event passes the filter expression) or discarded (i.e. all parts of the event fail the filter expression). As soon as it is clear that the event is kept, all associated, buffered BND records are emitted, and any further BND records belonging to the same event are emitted immediately upon encountering them.

```
##contig=<ID=fake,length=1234567>
##INFO=<ID=ints,Number=2,Type=Integer,Description="Two integers">
##INFO=<ID=SVTYPE,Number=1,Type=String,Description="Type of structural variant">
##INFO=<ID=MATEID,Number=.,Type=String,Description="ID of mate breakends">
##INFO=<ID=EVENT,Number=1,Type=String,Description="ID of event associated to breakend">
#CHROM POS    ID       REF ALT            INFO
fake   100    bnd_1_0  A   ]fake:150]A    SVTYPE=BND;MATEID=bnd_1_1;ints=1,2
fake   125    .        T   G              ints=0,2
fake   150    bnd_1_1  C   C[fake:100[    SVTYPE=BND;MATEID=bnd_1_0
fake   200    bnd_2_0  A   ]fake:250]A    SVTYPE=BND;EVENT=an_event;ints=1,2
fake   225    .        T   G              ints=0,2
fake   250    bnd_2_1  C   C[fake:200[    SVTYPE=BND;EVENT=an_event
```

Figure 5.6: Excerpt of a VCF file with 2 groups of breakend events. Breakend events that belong together can either be identified by matching of their `INFO["MATEID"]` information (e.g. `bnd_1_0` and `bnd_1_1`) or by grouping by information of the `INFO["EVENT"]` field (if available) (e.g. event `an_event` groups records with IDs `bnd_2_0` and `bnd_2_1`). If neither `MATEID` nor `EVENT` information is available, records are treated as if they were regular, non-BND records, even if they are marked as BND records (with `INFO["SVTYPE"] == "BND"`).

## 5.4 *Quirks*

Defining a custom language has both up- and downsides. On the upside, the language can be tailored towards the problem, restricting syntax and semantic to exactly what is needed. This potentially leads to a more concise language, allowing for constructs which simplify common operations for the specific domain, increasing expressiveness and readability. On the downside, defining a custom language heightens the potential for bugs (e.g. in the parser or the interpreter/compiler) and increases the maintenance burden.
Documentation is another critical aspect — good documentation can help users understand how to use the language correctly, avoid common pitfalls, and troubleshoot problems that may arise. Therefore, it is important to ensure that the documentation is comprehensive, accurate and *up to date* to ensure that users can effectively use the tool.

In this specific, niche case of filtering VCF files, we are of the opinion that the downsides of custom languages outweigh their potential benefits. This becomes especially apparent in the following example, where conventions regarding binary operators `&` and `&&` are invalidated: In most languages, `&` and `&&` denote bitwise and logical conjunction respectively. If both operands are booleans, bitwise and logical conjunction are semantically the same.
In `bcftools`, however, both `&` and `&&` denote logical conjunction — but with different semantics. The former implies that both operands must hold in the same sample for at least one sample, while the latter implies that the operands must hold for any sample independently. For example:

1. `FMT/DP > 0 & FMT/GQ > 10` implies that there must be at least one sample for which both `FMT/DP > 0` and `FMT/GQ > 10` are true,

2. `FMT/DP > 0 && FMT/GQ > 10` implies that there must be at least one sample for which `FMT/DP > 0` is true and at least one sample for which `FMT/GQ > 10` is true.

Hence, a (de-)duplication of `&` leads to different interpretations of an expression. The fact that — by usual programming conventions — both expres-

sions should result in the same filtering behaviour but in practice do not makes it very difficult to interpret such expressions correctly *as a human*.

## 5.5 Benchmark

While not as important as expressiveness, VCF filtering tools should also be performant, as there will often be millions of candidate variants called by variant callers for each sample, and filtering the resulting VCF files is standard procedure in analysis workflows. We find that `vembrane` ranks second-best among the filtering tools compared with respect to the records per second metric, as shown in the remainder of this section, which is a direct quote from the supplement of Hartmann, Schröder, et al. [2022][9], slightly modified to fit the style of this work:
In order to compare different VCF filtering tools with respect to processing times, we ran them on VCF files with different numbers of samples. These were generated by annotating the GIAB[10] VCF files (restricted to chromosome 1) of the samples `HG001`, `HG002`, `HG003` and `HG004` with both `SnpEff` and `VEP`, and creating VCF files with all possible multi-sample combinations of 1, 2, 3 or 4 samples. These were then filtered with tools that adhere to the specifications of VCF in version 4.3 and BCF in version 2.2[11], the versions htslib and bcftools use at the time of writing. This decision rules out `VcfFilterJdk` since htsjdk only supports earlier versions of the BCF specification and `VcfFilterJdk` produces incorrect VCF v4.2 files[12], that cannot be parsed by standard tools such as bcftools, making them incompatible with the rest of the workflow.

We selected a range of different filter expressions, varying in field accesses and general expression complexity, see Table 5.3 for details. We then benchmarked each tool on the annotated files with 10 repeats. Results are shown in Figure 5.7.

To make sure the same records were kept, we computed md5sums on the filtered and sorted VCF files while restricting fields to `CHROM`, `POS`, `REF`, `ALT` and `QUAL`. This restriction is necessary because:

- `bcftools` with `+split-vep` will add `INFO` fields for every field parsed from the `ANN` annotation by default
- `SnpSift` keeps all annotations if at least one of them matches (`vembrane` can mirror this behaviour with `--keep-unmatched`)
- `slivar` adds `impactful`, `genic` and `highest_impact_order` `INFO` fields

To reproduce the results, the snakemake workflow used for benchmarking is available at `github.com/vembrane/vembrane-benchmark` (`10.5281/zenodo.6979842`).

The benchmark (as shown in Figure 5.7) was run on a machine with an Intel(R) Xeon(R) Gold 6152 Processor (88 cores) with 768GiB RAM and 160TB of harddisk space managed in LVM groups. At any given time, a maximum of 4 jobs were run in parallel to limit I/O load.

[9] Hartmann, Schröder, et al., "Insane in the vembrane: filtering and transforming VCF/BCF files", 2022.

[10] *Genome In A Bottle* (`https://www.nist.gov/programs-projects/genome-bottle`, visited on 2023-11-20)

[11] *VCFv4.3* (`https://samtools.github.io/hts-specs/VCFv4.3.pdf`, visited on 2023-12-11)

[12] In contradiction to the VCF specification for v4.1 or newer, the `FORMAT/PS` field is defined as a string instead of a "non-negative 32-bit Integer" and may contain the string `PATMAT`.

| scenario name | tool | expression |
|---|---|---|
| filter_all | vembrane | `False` |
| | SnpSift | `false` |
| | slivar | `--info 'false'` |
| | filter_vep | `0` |
| | bio-vcf | `false` |
| | bcftools | `-e ""` |
| filter_none | vembrane | `True` |
| | SnpSift | `true` |
| | slivar | `--info 'true'` |
| | filter_vep | `not 0` |
| | bio-vcf | `true` |
| | bcftools | `-i ""` |
| at_least_2_platforms | vembrane | `INFO["platforms"] >= 2` |
| | SnpSift | `platforms >= 2` |
| | slivar | `--info 'INFO.platforms >= 2'` |
| | filter_vep | `platforms >= 2` |
| | bio-vcf | `rec.info.platforms >= 2` |
| | bcftools | `-i "INFO/platforms >= 2"` |
| format_dp | vembrane | `any(FORMAT["DP"][s] > 1250 for s in SAMPLES)` |
| | SnpSift | `GEN[*].DP > 1250` |
| | slivar | `--alias resources/empty_alias.txt --pass-only --sample-expr ':sample.DP > 1250'` |
| | filter_vep | cannot access `FORMAT` |
| | bio-vcf | `--sfilter` defaults to conjunctions (“all”), not disjunctions (“any”) |
| | bcftools | `-i "FORMAT/DP > 1250"` |
| impact_high | vembrane | `ANN["Annotation_Impact"] == "HIGH"` |
| | SnpSift | `ANN[*].IMPACT has 'HIGH'` |
| | slivar | `SLIVAR_IMPACTFUL_ORDER=slivar-impactfulness-order.txt slivar expr --info "INFO.impactful"` |
| | filter_vep | ignores SnpEff (or any non-VEP) annotation without raising an error |
| | bio-vcf | No built-in support for annotations |
| | bcftools | No built-in support for `SnpEff` annotations |
| uncertain | vembrane | `"uncertain_significance" in ANN["CLIN_SIG"] or not (ID and ID.startswith("rs"))` |
| | SnpSift | cannot access `VEP` annotations |
| | slivar | No built-in support for annotations apart from “Consequence” |
| | filter_vep | `CLIN_SIG is uncertain_significance or not (ID and ID matches "^rs" )` |
| | bio-vcf | No built-in support for annotations |
| | bcftools | `+split-vep --annotation "ANN" -c CLIN_SIG -i "INFO/CLIN_SIG[*] == 'uncertain_significance' \| (ID \!~ '^rs')"` |

Table 5.3: Expressions used for benchmarking. `impact_high` makes use of `SnpEff` annotations, `uncertain` makes use of VEP annotations, all other expressions only use default VCF fields and/or `INFO` and `FORMAT` fields defined in the header. For some tools it is necessary to specify the commandline options as well, e. g. for `bcftools` the interpretation of the expression changes: `-i` *includes*, `-e` *excludes* variants matching the expression.


Figure 5.7: A benchmark comparing `vembrane`, `bcftools`, `SnpSift`, `filter_vep`, `bio-vcf` and `slivar`. The $y$-axis is in records per seconds, i.e. higher is better. Runs with BCF input are listed separately for tools that support this. Each column corresponds to a different filter expression as described in Table 5.3. Note the logarithmic scale on the $y$-axis.

## 5.6 *Command Line Interface*

`vembrane`'s command line interface consists of the four subcommands `filter`, `table`, `annotate` and `tag`. These correspond to the following four use-cases:

- Filtering VCF files by keeping only records that match the given filter expression (`filter`).
- Tagging VCF records which pass the filter expression(s) (instead of keeping them while discarding those that do not pass the filter expression(s)) (`tag`).
- Formatting VCF files into a tabular format (`table`).
- Annotating VCF records with user supplied genomic range based auxiliary information and annotation (`annotate`).

### 5.6.1 `filter`

The `filter` subcommand is at the heart of `vembrane`. In its simplest form, an invocation of `vembrane filter` only needs two arguments: A filter expression and an input VCF file[13]. For example, the following invocation only keeps VCF records from `input.vcf` for which the quality is at least 30:

```
vembrane filter 'QUAL >= 30' input.vcf > output.vcf
```

Given an exemplary, reduced VCF file `input.vcf`:

```
## … header omitted for brevity …
#CHROM  POS  ID  REF  ALT  QUAL FILTER  INFO
fake    100  .   A    C    40   PASS    .
fake    200  .   T    G    20   PASS    .
```

the invocation above would result in the following VCF file `output.vcf`:

```
## … header omitted for brevity …
#CHROM  POS  ID  REF  ALT  QUAL FILTER  INFO
fake    1000 .   A    C    40   PASS    .
```

[13] The input file may be omitted, as the respective argument defaults to `stdin`, such that `vembrane` can be used (in a streaming fashion) in pipes.

where the second record is discarded because its quality is below 30.

Having the filter expression as the first argument and the input as the second argument allows using vembrane in shell pipes, e.g.:
`samtools sort input.vcf | vembrane filter 'QUAL >= 30' > output.vcf`

### 5.6.2 tag

Instead of keeping only records that pass the filter expression while discarding the rest, tag all records which pass a filter expression with a corresponding tag. This is basically a non-destructive version of `filter`, since no records are ever removed. As information is only ever added, multiple `tag=expression` pairs can be supplied. For example, the invocation

```
vembrane tag -t at_least_quality_30='QUAL >= 30' -t
↪  illumina='"ILLUMINA" in INFO["platformnames"]' input.vcf
↪  > output.vcf
```

adds the tag `at_least_quality_30` to the `FILTER` field of all records where the quality is at least 30 and the tag `illumina` to the `FILTER` field of all records which have the string `ILLUMINA` in their list of platform names. For reproducibility, the expressions used for applying the tags are stored in the VCF header.
Given an exemplary, reduced VCF file `input.vcf`:

```
## … header omitted for brevity …
# …  REF  ALT  QUAL FILTER                       INFO
     A    C    40   PASS                         platformnames=ONT
     T    G    20   PASS                         platformnames=ILLUMINA,ONT
     C    T    30   PASS                         platformnames=ILLUMINA
     G    A    20   PASS                         platformnames=.
```

the invocation above would result in the following VCF file `output.vcf`:

```
## … header omitted for brevity …
# …  REF  ALT  QUAL FILTER                       INFO
     A    C    40   at_least_quality_30          platformnames=ONT
     T    G    20   illumina                     platformnames=ILLUMINA,ONT
     C    T    30   at_least_quality_30,illumina  platformnames=ILLUMINA
     G    A    20   PASS                         platformnames=.
```

where the first record is tagged with the tag `at_least_quality_30` and the third record is tagged with both `at_least_quality_30` and `illumina`.

### 5.6.3 table

The `table` subcommand allows formatting a VCF file into a tabular file format. This is done by specifying a list of expressions which are to be evaluated for each record, and which subsequently are joined together with a delimiter (default: `\t`). For example, the invocation

```
vembrane table 'CHROM, POS, 1 - 10**(-QUAL/10)' input.vcf >
↪  output.tsv
```

generates a table (in Tab Separated Values (TSV) format) with three columns and as many rows as there are variants in the `input.vcf` file. Because this uses the same underlying mechanisms as `vembrane filter`, any valid Python expression can be used. The resulting table's header is automatically generated from the expression, but can be specified explicitly with the `--header` option. For example, we can modify the example above to replace automatically generated column names with custom column names "chromosome", "position" and "quality":

```
vembrane table --header 'chromosome, position, quality'
↪  'CHROM, POS, 1 - 10**(-QUAL/10)' input.vcf > output.tsv
```

The following reduced VCF file `input.vcf`:

```
## … header omitted for brevity …
#CHROM  POS  QUAL
1       100  40
1       200  20
2       300  30
```

then results in the following TSV file `output.tsv`:

```
chromosome  position    quality
1           100         0.9999
1           200         0.99
2           300         0.999
```

■ Handling multi-sample VCF files When accessing sample specific `FORMAT` entries, it is either possible to use the `for_each_sample` helper function (which automatically generates one column per sample) or switch to a long tabular format with the `--long` switch, which then exposes a `SAMPLE` variable.

A usage example of the `for_each_sample` option is as follows:

```
vembrane table 'CHROM, POS, for_each_sample(lambda sample:
↪  FORMAT["AD"][sample])' input.vcf > output_wide.tsv
```

In this case, the number of rows in the table will be equal to the number of records in the `input.vcf`, while there will be two columns named "CHROM" and "POS" plus one additional column (with a systematically generated name) for each sample in the file.
The following reduced VCF file `input.vcf`:

```
## … header omitted for brevity …
#CHROM  POS  FORMAT  Sample1  Sample2
1       100  AD      10       20
1       200  AD      30       40
2       300  AD      50       60
```

then results in the following TSV file `output_wide.tsv`:

```
CHROM                   POS                     FORMAT["AD"]["Sample1"]  FORMAT["AD"]["Sample2"]
1                       100                     10                       20
1                       200                     30                       40
2                       300                     50                       60
```

A usage example of the `--long` switch is as follows:

```
vembrane table --long 'CHROM, POS, FORMAT["AD"][SAMPLE]'
↪  input.vcf > output_long.tsv
```

In this case, the number of rows in the table will be equal to the number of records multiplied by the number of samples in the `input.vcf`. This option always introduces a column named `SAMPLE` as the first column of the output.
For the same example `input.vcf` above, the following TSV file `output_long.tsv` is generated:

```
SAMPLE      CHROM       POS         FORMAT["AD"]
Sample1     1           100         10
Sample2     1           100         20
Sample1     1           200         30
Sample2     1           200         40
Sample1     2           300         50
Sample2     2           300         60
```

### 5.6.4 `annotate`

The `annotate` subcommand enables basic annotation of variants based on genomic ranges. It requires two additional files: One file containing annotations in TSV format and one YAML Ain't Markup Language (YAML) file formally describing the annotations. The annotation file is referenced in the YAML file, so that it does not have to be specified on the command line.

An exemplary invocation is:

```
vembrane annotate example.yaml input.vcf > output.vcf
```

For more information, see `https://github.com/vembrane/vembrane/blob/v1.0.2/README.md#vembrane-annotate`[14].

[14] The functionality of `vembrane annotate` was contributed by Christopher Schröder.

## 5.7 *Future Work*

The field of bioinformatics has many file formats that were crafted ad-hoc, never systematically documented and/or specified and have been added to and modified over the years. This resulted in tools supporting slightly different, incompatible versions of the same file format, making communication between these tools difficult. With the SAM and VCF specifications and their subsequent adaptation by libraries and tools, this has improved over the years. It may in the future prove fruitful to use well established, (potentially schema driven) general purpose file formats instead. This would facilitate easier inter-domain exchange of information, and native library support for many programming languages. One example would be the duo of Apache Arrow/Parquet protocol and format.
In this context, a sensible solution would be to store information from VCF files in a database instead. To that end, work on VCF to SQL conversion functionality in `vembrane` has already started, following the database schemas used by `cutevariant`,[15] a graphical user interface for viewing VCF files as databases.

[15] Schutz et al., "Cutevariant: a standalone GUI-based desktop application to explore genetic variations from an annotated VCF file", 2021.

## 5.8 Discussion

With `vembrane`, we provide a tool to filter, tag, annotate and format VCF files. Unlike other VCF manipulation tools, it does not define its own Domain Specific Language (DSL) but leverages Python instead. Its expressiveness is therefore only limited by the Python language and `pysam`, the library used for VCF/BCF input, interpretation and output.

Using an existing language that many researchers are already familiar with has several advantages:

- The learning curve is much less steep (than for domain-specific languages).
- The ecosystem is well established.
- The syntax is stable.
- The semantics are well-defined.
- The probability of bugs and hence the maintenance burden is lower.

One could argue that designing a DSL for this specific domain allowed for a more concise language, but we think that the advantages of using an established language outweigh this potential benefit, especially since Python is already a rather expressive language.

With regard to performance, `vembrane` is reasonably fast, ranking second best among the tested tools in terms of runtime. Filtering VCF files is usually not the bottleneck in analysis workflows, and most tools are within a similar order of magnitude (with respect to records per second); however, `filter_vep` is an outlier, being considerably slower than the other tools, which can be a resource issue.

Also, `vembrane` is one of the few tools to support both VEP and SnpEff annotations out of the box, which are two commonly used annotation tools/sources.

What sets `vembrane` apart even more is the ability to correctly filter VCF files which contain BND records, which are common in structural variant calling settings. No other tool supports this feature, instead treating BND records as regular records, which can (and will) lead to incorrect filtering results.

# A Read alignment file sizes

To give a very rough overview of file sizes and number of records in alignment files, we show some examples of file-sizes and number of records in Figure A.1, for different technologies (ONT and Illumina) and different types of sequencing experiments (WGS and WES). In Figure A.2 we show the time it takes to iterate once through the file using `samtools view -c`, only counting the records. The files are different in-house datasets, and the timings were measured on different machines. The average time per record is about 3.4µs for the datasets and machines used here.

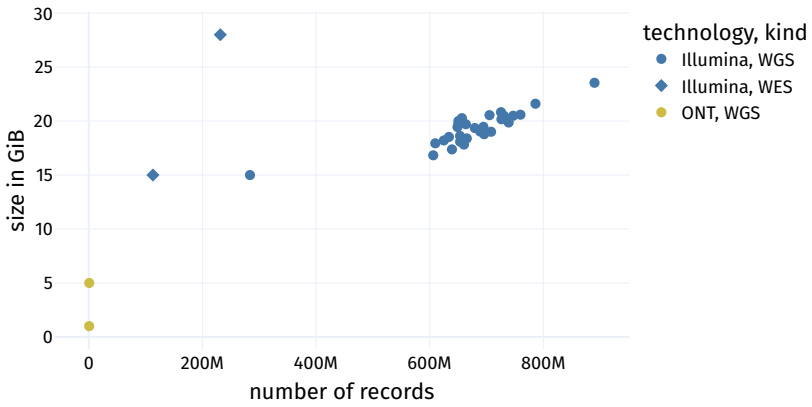



Figure A.1: Overview of relationship between number of alignment records contained in the file and file-size , stratified by technology (Illumina, ONT) and type of sequencing experiment (WGS, WES).

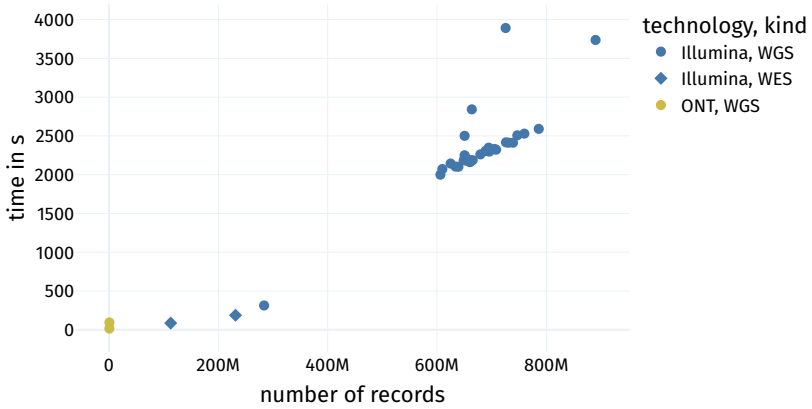



Figure A.2: Overview of relationship between number of alignment records contained in the file and time needed to iterate once through the file (user time), stratified by technology (Illumina, ONT) and type of sequencing experiment (WGS, WES).

# B k-mers

## B.1 Removing duplicates in-place

To drop all non-unique k-mers in-place, sequentially iterate k-mer pairs, marking both k-mers of a pair as duplicates if they are identical, then remove duplicate k-mers from the list by swapping them with the last item of the list, afterwards truncating the list.

```
def drop_non_unique(kmers):
    assert kmers == sorted(kmers)
    # mark duplicates
    remove = BitVector(size=len(kmers))
    for i, (a, b) in enumerate(pairwise(kmers)):
        if a == b:
            remove[i] = remove[i + 1] = 1

    # remove duplicates, starting from the back
    for i, bit in reversed(enumerate(remove)):
        if bit:
            kmers.swap_remove(i)
```

Figure B.1: Pseudo-code for removing non-unique k-mers from the list of k-mers. First mark all neighbouring k-mers that are identical as duplicate, then, starting from the back, remove duplicates by sequentially swapping each duplicate with the last k-mer in the list and truncating the list, from back to front.

## B.2 ACGT block bisection

In Figure B.2, we show pseudocode for finding the start and end indices of all k-mers starting with the base indicated by `offset`. It is a basic binary search / bisection algorithm.

For example, given the *sorted* list of k-mers

```
kmers = ["AACG", "AAGT", "ACTT", "CACT", "CATT"]
```

and the offset 0,

```
acgt_blocks(kmers, offset=0)
```

will yield the index ranges $(0, 3)$, $(3, 5)$, $(5, 5)$ and $(5, 5)$, indicating that the first three k-mers start with A and the next two with C (and no k-mers start with G or T).
Then, within the block of k-mers starting with A, we can find the start and end indices of all k-mers starting with AA, AC, AG and AT by

calling

```
acgt_blocks(kmers[0:3], offset=1)
```

which yields $(0, 2)$, $(2, 3)$, $(3, 3)$ and $(3, 3)$.

```
from bisect import bisect

def acgt_blocks(kmers: List[Kmer], offset: int):
  key = lambda kmer: kmer[offset]
  idx_c = bisect(kmers, C, key=key)
  idx_a = bisect(kmers, A, hi=idx_c, key=key)
  idx_g = bisect(kmers, G, lo=idx_c, key=key)
  return [
      (0, idx_a),
      (idx_a, idx_c),
      (idx_c, idx_g),
      (idx_g, len(kmers)),
  ]
```

Figure B.2: Pseudocode for finding the start and end indices of all k-mers starting with a given base.

## B.3 *A bit twiddling reverse complement function*

In Figure B.3, we show Rust code for calculating the reverse complement of a two-bit encoded k-mer stored in a 64-bit unsigned integer. The basic bit-reversing technique is derived from *Bit Twiddling Hacks*[1] but adapted to work on 64-bit integers instead of 32-bit integers and omitting the first swap step, as each DNA base is encoded as *two* bits, not a single one.

[1] *Bit Twiddling Hacks* (https://graphics.stanford.edu/~seander/bithacks.html#ReverseParallel, visited on 2023-10-27)

```rust
fn reverse_complement(v: u64, k: u8) -> u64 {
    const MASK2: u64 = 0x3333333333333333;
    const MASK4: u64 = 0x0F0F0F0F0F0F0F0F;
    const MASK8: u64 = 0x00FF00FF00FF00FF;
    const MASK16: u64 = 0x0000FFFF0000FFFF;

    assert!(k < 32);
    let shift = 64 - (k as u64 << 1);
    let mut v = v;

    // swap blocks of 2
    v = ((v >> 2) & MASK2) | ((v & MASK2) << 2);

    // swap blocks of 4
    v = ((v >> 4) & MASK4) | ((v & MASK4) << 4);

    // swap blocks of 8
    v = ((v >> 8) & MASK8) | ((v & MASK8) << 8);

    // swap blocks of 16
    v = ((v >> 16) & MASK16) | ((v & MASK16) << 16);

    // swap blocks of 32
    v = (v >> 32) | (v << 32);

    // negate to get complement,
    // shift to significant 2k bits
    (!v) >> shift
}
```

Figure B.3: Rust code for calculating the reverse complement of a two-bit encoded k-mer stored in a 64-bit unsigned integer.

# C *Contributions and Collaborations*

During my work on this dissertation, I published the following co-authored articles that have been expanded upon in this dissertation:
Chapter 3 is based on the article

> Tüns & Hartmann et al., "Detection and validation of circular DNA fragments using nanopore sequencing". *Frontiers in genetics* 13 (2022).

for which I developed the method, implemented the workflow and designed the simulation experiments. Alicia Tüns designed and executed all wet-lab experiments. Sven Rahmann, Alexander Schramm and Johannes Köster supervised method development and assisted in manuscript writing.

Chapter 5 is based on the article

> Till Hartmann, Christopher Schröder, et al. "Insane in the vembrane: filtering and transforming VCF/BCF files". *Bioinformatics* 39.1 (2022-12).

for which I developed the method and performed the benchmark comparing with other methods. Christopher Schröder, David Lähnemann, Elias Kuthe and Johannes Köster helped shape the method and manuscript and contributed code and tests. While not listed as co-authors, I would still like to mention Marcel Bargull, Jan Forster, Felix Mölder and Sven Rahmann for their feedback and advice.

For the remaining chapters, and generally during all stages of this dissertation, both Johannes Köster and Sven Rahmann acted as advisors.

For ad-hoc investigations, simulations and quick research evaluations, the Python package `dinopy`[1] by Henning Timm (to which I contributed as a student assistant) was used, as well as the Rust crate `rust-bio`, to which I contributed as well. For the snakemake workflows, packages from the `bioconda`[2] project (to which I contributed recipes and provided maintenance work) were used.

[1] Timm and Hartmann, *Dinopy — DNA input and output for Python and Cython*, 2020.

[2] Grüning et al., "Bioconda: Sustainable and Comprehensive Software Distribution for the Life Sciences", 2018.

# *Glossary*

| | |
|---|---|
| acrocentric | When the centromere is close to any end of the chromosome (resulting in one short and one long arm), the chromosome is called *acrocentric*. |
| centromere | The centromere is the region where sibling chromatids are linked during cell division. |
| contig | A contiguous stretch of DNA sequence, assembled from shorter fragments/sequences. |
| eukaryote | Organism whose cells contains at least one distinct nucleus. |
| haploid | When a cell has only one set of chromosomes. |
| homozygous | When all alleles at a given locus are identical. |
| PHRED | PHRED quality score, a $\log_{10}$ transformation of error probabilities. |
| prokaryote | Organism whose cells do not have a nucleus. |
| telomere | Repetitive ends of linear chromosomes. |

# *Acronyms*

| | |
|---|---|
| AMQ | Approximate Membership Query. |
| BAM | Binary Alignment Map, compressed binary version of SAM. |
| BCF | Binary version of the Variant Call Format. |
| BND | Breakend. |
| CBS | Circular Binary Segmentation. |
| CIGAR | Compact Idiosyncratic Gapped Alignment Report. |
| CNV | Copy Number Variation. |
| CRAM | Compressed Reference-oriented Alignment Map. |
| DFS | Depth First Search. |
| DNA | Deoxyribonucleic acid. |
| DSL | Domain Specific Language. |
| eccDNA | extrachromosomal circular DNA. |
| EDF | Empirical Distribution Function. |
| fdr | false-discovery-rate. |
| GIAB | Genome In A Bottle. |
| HMM | Hidden Markov Model. |
| HpHMM | Homopoly-pair Hidden Markov Model. |
| Illumina | Illumina. |
| indel | insertion/deletion. |
| IUPAC | International Union Of Pure And Applied Chemistry. |
| mphf | Minimal Perfect Hash Function. |
| mRNA | messenger-RNA. |
| NIST | National Institute of Standards and Technology. |
| ONT | Oxford Nanopore Technologies. |
| PacBio | Pacific Biosciences. |

| | |
|---|---|
| PCR | Polymerase Chain Reaction. |
| PDF | Probability Density Function. |
| pHMM | Pair Hidden Markov Model. |
| QC | Quality Control. |
| rDNA | ribosomal DNA. |
| RNA | Ribonucleic acid. |
| SAM | Sequence Alignment Map. |
| SCC | Strongly Connected Component. |
| SIMD | Single Instruction Multiple Data. |
| SNV | Single Nucleotide Variant. |
| TSV | Tab Separated Values. |
| VCF | Variant Call Format. |
| VEP | Variant Effect Predictor. |
| WGS | Whole Genome Sequencing. |
| YAML | YAML Ain't Markup Language. |
| ZINB | Zero-Inflated Negative Binomial. |
| ZOINB | Zero-One Inflated Negative Binomial. |

# *Bibliography*


Abyzov, Alexej, Alexander E Urban, Michael Snyder, and Mark Gerstein. "CNVnator: an approach to discover, genotype, and characterize typical and atypical CNVs from family and population genome sequencing". *Genome research* 21.6 (2011), pp. 974–984.

Adams, Ryan Prescott and David JC MacKay. "Bayesian online changepoint detection". *arXiv preprint arXiv:0710.3742* (2007).

Altamirano, Matias, François-Xavier Briol, and Jeremias Knoblauch. "Robust and Scalable Bayesian Online Changepoint Detection". 2023. arXiv: 2302.04759 [stat.ML].

Amarasinghe, Shanika L, Shian Su, Xueyi Dong, Luke Zappia, Matthew E Ritchie, and Quentin Gouil. "Opportunities and challenges in long-read sequencing data analysis". *Genome biology* 21.1 (2020), pp. 1–16.

Aminikhanghahi, Samaneh and Diane J Cook. "A survey of methods for time series change point detection". *Knowledge and information systems* 51.2 (2017), pp. 339–367.

Baum, Leonard E, Ted Petrie, George Soules, and Norman Weiss. "A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains". *The annals of mathematical statistics* 41.1 (1970), pp. 164–171.

Bayer, Rudolf and Edward McCreight. "Organization and maintenance of large ordered indices". In: *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. 1970, pp. 107–141.

Benjamini, Yoav and Yosef Hochberg. "Controlling the false discovery rate: a practical and powerful approach to multiple testing". *Journal of the Royal statistical society: series B (Methodological)* 57.1 (1995), pp. 289–300.

Blum, Manuel, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, Robert Endre Tarjan, et al. "Time bounds for selection". *J. Comput. Syst. Sci.* 7.4 (1973), pp. 448–461.

Boža, Vladimír, Broňa Brejová, and Tomáš Vinař. "DeepNano: deep recurrent neural networks for base calling in MinION nanopore reads". *PloS one* 12.6 (2017), e0178751.

Browne, Patrick Denis, Tue Kjærgaard Nielsen, Witold Kot, Anni Aggerholm, M Thomas P Gilbert, Lara Puetz, Morten Rasmussen, Athanasios Zervas, and Lars Hestbjerg Hansen. "GC bias affects genomic and metagenomic reconstructions, underrepresenting GC-poor organisms". *GigaScience* 9.2 (2020-02). giaa008. DOI: 10.1093/gigascience/giaa008.

Byrd, Richard H., Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. "A Limited Memory Algorithm for Bound Constrained Optimization". *SIAM Journal on Scientific Computing* 16.5 (1995), pp. 1190–1208. DOI: 10.1137/0916069.

Cameron, Daniel L, Jonathan Baber, Charles Shale, Jose Espejo Valle-Inclan, Nicolle Besselink, Arne van Hoeck, Roel Janssen, Edwin Cuppen, Peter Priestley, and Anthony T Papenfuss. "GRIDSS2: comprehensive characterisation of somatic structural variation using single breakend variants and structural variant phasing". *Genome biology* 22 (2021), pp. 1–25.

Chen, Ken, John W Wallis, Michael D McLellan, David E Larson, Joelle M Kalicki, Craig S Pohl, Sean D McGrath, Michael C Wendl, Qunyuan Zhang, Devin P Locke, et al. "BreakDancer: an algorithm for high-resolution mapping of genomic structural variation". *Nature methods* 6.9 (2009), pp. 677–681.

Chinappi, Mauro and Fabio Cecconi. "Protein sequencing via nanopore based devices: a nanofluidics perspective". *Journal of Physics: Condensed Matter* 30.20 (2018), p. 204002.

Cramér, Harald. *Mathematical methods of statistics*. Princeton university press, 1946. Chap. 21, p. 282.

Danecek, Petr, James K Bonfield, Jennifer Liddle, John Marshall, Valeriu Ohan, Martin O Pollard, Andrew Whitwham, Thomas Keane, Shane A McCarthy, Robert M Davies, et al. "Twelve years of SAMtools and BCFtools". *Gigascience* 10.2 (2021), giab008.

Delahaye, Clara and Jacques Nicolas. "Sequencing DNA with nanopores: Troubles and biases". *PLOS ONE* 16.10 (2021-10), pp. 1–29. DOI: 10.1371/journal.pone.0257521.

Drepper, Ulrich. "What every programmer should know about memory". *Red Hat, Inc* 11.2007 (2007), p. 2007.

Durbin, Richard, Sean R Eddy, Anders Krogh, and Graeme Mitchison. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.

Ewing, Brent, LaDeana Hillier, Michael C Wendl, and Phil Green. "Base-calling of automated sequencer traces using phred." *Genome research* 8.3 (1998), pp. 175–194.

Floyd, Robert W and Ronald L Rivest. "Expected time bounds for selection". *Communications of the ACM* 18.3 (1975), pp. 165–172.

Garrison, Erik and Gabor Marth. "Haplotype-based variant detection from short-read sequencing". *arXiv preprint arXiv:1207.3907* (2012).

Goodwin, Sara, John D. McPherson, and W. Richard McCombie. "Coming of Age: Ten Years of Next-Generation Sequencing Technologies". *Nature Reviews Genetics* 17.6 (2016), pp. 333–351. DOI: 10.1038/nrg.2016.49.

Grüning, Björn, Ryan Dale, Andreas Sjödin, Brad A. Chapman, Jillian Rowe, Christopher H. Tomkins-Tinch, Renan Valieris, Johannes Köster, and The Bioconda Team. "Bioconda: Sustainable and Comprehensive Software Distribution for the Life Sciences". *Nature Methods* 15.7 (2018). Henning Timm is part of "The Bioconda Team"., pp. 475–476. DOI: 10.1038/s41592-018-0046-7.

Hartmann, Till and David Lähnemann. "snakemake-workflows/cyrcular-calling". *Zenodo* (2023-04). DOI: 10.5281/zenodo.7863210.

Hartmann, Till, Christopher Schröder, Elias Kuthe, David Lähnemann, and Johannes Köster. “Insane in the vembrane: filtering and transforming VCF/BCF files”. *Bioinformatics* 39.1 (2022-12). btac810. DOI: 10.1093/bioinformatics/btac810.

Heather, James M and Benjamin Chain. “The sequence of sequencers: The history of sequencing DNA”. *Genomics* 107.1 (2016), pp. 1–8.

Hedderich, Jürgen and Lothar Sachs. *Angewandte Statistik*. 16th ed. Springer Spektrum Berlin, 2018, pp. 590–591. 1025 pp. DOI: https://doi.org/10.1007/978-3-662-56657-2.

– *Angewandte Statistik*. 16th ed. Springer Spektrum Berlin, 2018, pp. 579–580. 1025 pp. DOI: https://doi.org/10.1007/978-3-662-56657-2.

Henssen, Anton, Ian MacArthur, Richard Koche, and Heathcliff Dorado-García. “Purification and sequencing of large circular DNA from human cells” (2019).

Holtgrewe, Manuel. “Mason–a read simulator for second generation sequencing data”. *Technical Report FU Berlin* (2010).

International Human Genome Sequencing Consortium, US DOE Joint Genome Institute: Hawkins Trevor 4 Branscomb Elbert 4 Predki Paul 4 Richardson Paul 4 Wenning Sarah 4 Slezak Tom 4 Doggett Norman 4 Cheng Jan-Fang 4 Olsen Anne 4 Lucas Susan 4 Elkin Christopher 4 Uberbacher Edward 4 Frazier Marvin 4, RIKEN Genomic Sciences Center: Sakaki Yoshiyuki 9 Fujiyama Asao 9 Hattori Masahira 9 Yada Tetsushi 9 Toyoda Atsushi 9 Itoh Takehiko 9 Kawagoe Chiharu 9 Watanabe Hidemi 9 Totoki Yasushi 9 Taylor Todd 9, Genoscope, CNRS UMR-8030: Weissenbach Jean 10 Heilig Roland 10 Saurin William 10 Artiguenave Francois 10 Brottier Philippe 10 Bruls Thomas 10 Pelletier Eric 10 Robert Catherine 10 Wincker Patrick 10, Institute of Molecular Biotechnology: Rosenthal André 12 Platzer Matthias 12 Nyakatura Gerald 12 Taudien Stefan 12 Rump Andreas 12 Department of Genome Analysis, GTC Sequencing Center: Smith Douglas R. 11 Doucette-Stamm Lynn 11 Rubenfield Marc 11 Weinstock Keith 11 Lee Hong Mei 11 Dubois JoAnn 11, Beijing Genomics Institute/Human Genome Center: Yang Huanming 13 Yu Jun 13 Wang Jian 13 Huang Guyang 14 Gu Jun 15, et al. “Initial sequencing and analysis of the human genome”. *nature* 409.6822 (2001), pp. 860–921.

Jain, Miten, Sergey Koren, Karen H Miga, Josh Quick, Arthur C Rand, Thomas A Sasani, John R Tyson, Andrew D Beggs, Alexander T Dilthey, Ian T Fiddes, et al. “Nanopore sequencing and assembly of a human genome with ultra-long reads”. *Nature biotechnology* 36.4 (2018), pp. 338–345.

Jang, Ye Eun, Insu Jang, Sunkyu Kim, Subin Cho, Daehan Kim, Keonwoo Kim, Jaewon Kim, Jimin Hwang, Sangok Kim, Jaesang Kim, et al. “ChimerDB 4.0: an updated and expanded database of fusion genes”. *Nucleic acids research* 48.D1 (2020), pp. D817–D824.

Khreich, Wael, Eric Granger, Ali Miri, and Robert Sabourin. “On the memory complexity of the forward–backward algorithm”. *Pattern Recognition Letters* 31.2 (2010), pp. 91–99. DOI: https://doi.org/10.1016/j.patrec.2009.09.023.

Kokot, Marek, Maciej Długosz, and Sebastian Deorowicz. "KMC 3: counting and manipulating k-mer statistics". *Bioinformatics* 33.17 (2017), pp. 2759–2761.

Konheim, Alan G. "Perfect Hashing". In: *Hashing in Computer Science: Fifty Years of Slicing and Dicing*. 2010, pp. 164–166. DOI: 10.1002/9780470630617.ch11.

Kopczynski, Dominik. "Resource-Constrained Analysis of Ion Mobility Spectrometry Data". PhD thesis. Technische Universität Dortmund, 2017.

Köster, Johannes, Louis J Dijkstra, Tobias Marschall, and Alexander Schönhuth. "Varlociraptor: enhancing sensitivity and controlling false discovery rate in somatic indel discovery". *Genome biology* 21.1 (2020), pp. 1–25.

Köster, Johannes and Sven Rahmann. "Snakemake—a scalable bioinformatics workflow engine". *Bioinformatics* 28.19 (2012), pp. 2520–2522.

Lekshmi, Vanaja Seetha and Simi Sebastian. "A skewed generalized discrete Laplace distribution". *Int J Math Stat Invent* 2 (2014), pp. 95–102.

Li, Heng. "FermiKit: assembly-based variant calling for Illumina resequencing data". *Bioinformatics* 31.22 (2015), pp. 3694–3696.

– "Minimap2: pairwise alignment for nucleotide sequences". *Bioinformatics* 34.18 (2018), pp. 3094–3100.

Li, Heng, Jue Ruan, and Richard Durbin. "Mapping short DNA sequencing reads and calling variants using mapping quality scores". *Genome research* 18.11 (2008), pp. 1851–1858.

Limasset, Antoine, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. "Fast and scalable minimal perfect hashing for massive key sets". *arXiv preprint arXiv:1702.03154* (2017).

Liu, Dong C and Jorge Nocedal. "On the limited memory BFGS method for large scale optimization". *Mathematical programming* 45.1-3 (1989), pp. 503–528.

Maier, Tobias, Peter Sanders, and Robert Williger. "Concurrent expandable AMQs on the basis of quotient filters". *arXiv preprint arXiv:1911.08374* (2019).

Manekar, Swati C and Shailesh R Sathe. "A benchmark study of k-mer counting methods for high-throughput sequencing". *GigaScience* 7.12 (2018-10). giy125. DOI: 10.1093/gigascience/giy125.

Marçais, Guillaume and Carl Kingsford. "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers". *Bioinformatics* 27.6 (2011), pp. 764–770.

Martin, Fergal J, M Ridwan Amode, Alisha Aneja, Olanrewaju Austine-Orimoloye, Andrey G Azov, If Barnes, Arne Becker, Ruth Bennett, Andrew Berry, Jyothish Bhai, et al. "Ensembl 2023". *Nucleic acids research* 51.D1 (2023), pp. D933–D941.

Martin, Marcel and Sven Rahmann. "Aligning flowgrams to DNA sequences". In: *German Conference on Bioinformatics 2013*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2013.

Matthews, Brian W. "Comparison of the predicted and observed secondary structure of T4 phage lysozyme". *Biochimica et Biophysica Acta (BBA)-Protein Structure* 405.2 (1975), pp. 442–451.

Mölder, Felix, Kim Philipp Jablonski, Brice Letcher, Michael B Hall, Christopher H Tomkins-Tinch, Vanessa Sochat, Jan Forster, Soohyun Lee, Sven O Twardziok, Alexander Kanitz, et al. "Sustainable data analysis with Snakemake". *F1000Research* 10 (2021).

Muller, Jean-Michel, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, Serge Torres, et al. *Handbook of floating-point arithmetic*. Springer, 2018.

Needleman, Saul B and Christian D Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". *Journal of molecular biology* 48.3 (1970), pp. 443–453.

Noer, Julie B, Oskar K Hørsdal, Xi Xiang, Yonglun Luo, and Birgitte Regenberg. "Extrachromosomal circular DNA in cancer: history, current knowledge, and methods". *Trends in Genetics* 38.7 (2022), pp. 766–781.

Nurk, Sergey et al. "The complete sequence of a human genome". *Science* 376.6588 (2022), pp. 44–53. DOI: 10.1126/science.abj6987.

Obeya, Omar, Endrias Kahssay, Edward Fan, and Julian Shun. "Theoretically-efficient and practical parallel in-place radix sorting". In: *The 31st ACM symposium on parallelism in algorithms and architectures*. 2019, pp. 213–224.

Olshen, Adam B., E. S. Venkatraman, Robert Lucito, and Michael Wigler. "Circular binary segmentation for the analysis of array-based DNA copy number data". *Biostatistics* 5.4 (2004-10), pp. 557–572. DOI: 10.1093/biostatistics/kxh008.

Pandey, Prashant, Michael A Bender, Rob Johnson, and Rob Patro. "Squeakr: an exact and approximate k-mer counting system". *Bioinformatics* 34.4 (2018), pp. 568–575.

Pedersen, Brent S and Aaron R Quinlan. "Mosdepth: quick coverage calculation for genomes and exomes". *Bioinformatics* 34.5 (2018), pp. 867–868.

Pibiri, Giulio Ermanno. "On weighted k-mer dictionaries". *Algorithms for Molecular Biology* 18.1 (2023), pp. 1–20.

Pibiri, Giulio Ermanno, Yoshihiro Shibuya, and Antoine Limasset. "Locality-preserving minimal perfect hashing of k-mers". *Bioinformatics* 39.Supplement_1 (2023), pp. i534–i543.

Rausch, Tobias, Thomas Zichner, Andreas Schlattl, Adrian M Stütz, Vladimir Benes, and Jan O Korbel. "DELLY: structural variant discovery by integrated paired-end and split-read analysis". *Bioinformatics* 28.18 (2012), pp. i333–i339.

Sanger, Frederick, Steven Nicklen, and Alan R Coulson. "DNA sequencing with chain-terminating inhibitors". *Proceedings of the national academy of sciences* 74.12 (1977), pp. 5463–5467. DOI: 10.1073/pnas.74.12.5463.

Sarkozy, Peter, Ákos Jobbágy, and Peter Antal. "Calling Homopolymer Stretches from Raw Nanopore Reads by Analyzing k-mer Dwell Times". In: *EMBEC & NBC 2017*. Ed. by Hannu Eskola, Outi Väisänen, Jari Viik, and Jari Hyttinen. Singapore: Springer Singapore, 2018, pp. 241–244. ISBN: 978-981-10-5122-7.

Schutz, Sacha, Charles Monod-Broca, Lucas Bourneuf, Pierre Marijon, and Tristan Montier. "Cutevariant: a standalone GUI-based desktop

application to explore genetic variations from an annotated VCF file". *Bioinformatics Advances* 2.1 (2021-11). vbab028. DOI: 10.1093/bioadv/vbab028.

Shen, Feichen and Jeffrey Kidd. "QuicK-mer: A rapid paralog sensitive CNV detection pipeline". *bioRxiv* (2015), p. 028225.

Shibuya, Yoshihiro, Djamal Belazzougui, and Gregory Kucherov. "Space-efficient representation of genomic k-mer count tables". *Algorithms for Molecular Biology* 17.1 (2022), p. 5.

Smit, AFA, R Hubley, and P Green. "RepeatMasker Open-4.0. 2013–2015". 2015.

Smith, Temple F, Michael S Waterman, et al. "Identification of common molecular subsequences". *Journal of molecular biology* 147.1 (1981), pp. 195–197.

Tarjan, Robert. "Depth-First Search and Linear Graph Algorithms". *SIAM Journal on Computing* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010.

Timm, Henning. "Analysis and Application of Hash-based Similarity Estimation Techniques for Biological Sequence Analysis". PhD thesis. Technische Universität Dortmund, 2021.

Timm, Henning and Till Hartmann. "Dinopy — DNA input and output for Python and Cython". Zenodo. First released on 10.04.2015 on https://bitbucket.org/HenningTimm/dinopy. 2020. DOI: 10.5281/zenodo.4389306.

Tortora, Robert D. "A note on sample size estimation for multinomial populations". *The American Statistician* 32.3 (1978), pp. 100–102.

Tüns, Alicia Isabell, Till Hartmann, Simon Magin, Rocío Chamorro González, Anton George Henssen, Sven Rahmann, Alexander Schramm, and Johannes Köster. "Detection and validation of circular DNA fragments using nanopore sequencing". *Frontiers in genetics* 13 (2022). DOI: 10.3389/fgene.2022.867018.

Turcatti, Gerardo, Anthony Romieu, Milan Fedurco, and Ana-Paula Tairi. "A New Class of Cleavable Fluorescent Nucleotides: Synthesis and Optimization as Reversible Terminators for Dna Sequencing by Synthesis". *Nucleic Acids Research* 36.4 (2008), e25:1–e25:13. DOI: 10.1093/nar/gkn021.

Uhrig, Sebastian, Julia Ellermann, Tatjana Walther, Pauline Burkhardt, Martina Fröhlich, Barbara Hutter, Umut H Toprak, Olaf Neumann, Albrecht Stenzinger, Claudia Scholl, et al. "Accurate and efficient detection of gene fusions from RNA sequencing data". *Genome research* 31.3 (2021), pp. 448–460.

Van den Burg, Gerrit JJ and Christopher KI Williams. "An evaluation of change point detection algorithms". *arXiv preprint arXiv:2003.06222* (2020).

Venkatraman, E. S. and Adam B. Olshen. "A faster circular binary segmentation algorithm for the analysis of array CGH data". *Bioinformatics* 23.6 (2007-01), pp. 657–663. DOI: 10.1093/bioinformatics/btl646.

Venkatraman, Ennapadam Seshan. *Consistency results in multiple change-point problems*. Stanford University, 1992.

Vidyasagar, Mathukumalli. *Hidden markov processes: Theory and applications to biology*. Princeton University Press, 2014.

Virtanen, Pauli, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

Whitford, Whitney, Klaus Lehnert, Russell G. Snell, and Jessie C. Jacobsen. "Evaluation of the performance of copy number variant prediction tools for the detection of deletions from whole genome sequencing data". *Journal of Biomedical Informatics* 94 (2019), p. 103174. DOI: https://doi.org/10.1016/j.jbi.2019.103174.

Xu, Xuechun, Nayanika Bhalla, Patrik Ståhl, and Joakim Jaldén. "Lokatt: A hybrid DNA nanopore basecaller with an explicit duration hidden Markov model and a residual LSTM network". *bioRxiv* (2022), pp. 2022–07.

Yang, Chen, Justin Chu, René L Warren, and Inanç Birol. "NanoSim: nanopore sequence read simulator based on statistical characterization". *GigaScience* 6.4 (2017), gix010.

Ye, Kai, Li Guo, Xiaofei Yang, Eric-Wubbo Lamijer, Keiran Raine, and Zemin Ning. "Split-read indel and structural variant calling using PINDEL". *Copy Number Variants: Methods and Protocols* (2018), pp. 95–105.

Zentgraf, Jens and Sven Rahmann. "Fast Gapped k-mer Counting with Subdivided Multi-Way Bucketed Cuckoo Hash Tables". In: *22nd International Workshop on Algorithms in Bioinformatics (WABI 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022.

# *Web resources*


Anderson, Sean Eron. *Bit Twiddling Hacks*. 2005. URL: https://graphics.stanford.edu/~seander/bithacks.html#ReverseParallel (visited on 2023-10-27).

*BTreeSet*. URL: https://doc.rust-lang.org/std/collections/struct.BTreeSet.html (visited on 2023-12-13).

*dna-seq-varlociraptor*. DOI: 10.5281/zenodo.7690955. URL: https://github.com/snakemake-workflows/dna-seq-varlociraptor (visited on 2023-03-07).

*Ensembl Genome Browser Help Page 155*. URL: https://www.ensembl.org/Help/View?id=155 (visited on 2023-12-13).

*Genome In A Bottle*. URL: https://www.nist.gov/programs-projects/genome-bottle (visited on 2023-11-20).

genome.gov. *Fact Sheet: Human Genome Project*. URL: https://www.genome.gov/about-genomics/educational-resources/fact-sheets/human-genome-project (visited on 2023-06-19).

Hartmann, Till. *cyrcular*. URL: https://github.com/tedil/cyrcular (visited on 2023-11-10).

– *cyrcular-validation*. URL: https://github.com/tedil/cyrcular-validation (visited on 2023-11-10).

Kaitchuk, Tom. *aHash*. URL: https://github.com/tkaitchuck/aHash (visited on 2023-11-02).

Köster, Johannes. *varlociraptor presentation*. 2020. URL: https://slides.com/johanneskoester/towards-a-unified-theory-of-variant-calling-500a11 (visited on 2023-11-27).

Maryland Bioinformatics Labs. *CHM13*. URL: https://github.com/marbl/CHM13/blob/96c0892d70affb42648c277bfe3974e94d4ffebc/README.md (visited on 2022-10-17).

Mozilla. *FxHash*. URL: https://docs.rs/rustc-hash/latest/rustc_hash/struct.FxHasher.html (visited on 2023-11-02).

Noé, Laurent. *spaced-seeds*. URL: https://sites.google.com/view/laurentnoe/spaced-seeds (visited on 2023-07-06).

Oxford Nanopore Technologies. *bonito*. URL: https://github.com/nanoporetech/bonito (visited on 2023-11-09).

– *dorado*. URL: https://github.com/nanoporetech/dorado (visited on 2023-11-09).

*pysam*. URL: https://github.com/pysam-developers/pysam (visited on 2023-03-07).

*SAMtag*. URL: https://samtools.github.io/hts-specs/SAMtags.pdf (visited on 2023-05-15).

*SAMv1*. URL: https://samtools.github.io/hts-specs/SAMv1.pdf (visited on 2022-10-20).

*stats.stackexchange.com Negative Binomial Difference*. URL: https://stats.stackexchange.com/questions/13346/distribution-that-describes-the-difference-between-negative-binomial-distributed (visited on 2023-12-03).

*VCFv4.3*. URL: https://samtools.github.io/hts-specs/VCFv4.3.pdf (visited on 2023-12-11).

*VCFv4.4*. URL: https://samtools.github.io/hts-specs/VCFv4.4.pdf (visited on 2023-03-07).

Wiegand, Felix and Johannes Köster. *datavzrd*. URL: https://github.com/datavzrd/datavzrd (visited on 2023-11-20).

Xiao, Jeffrey. *Probabilistic Collections*. URL: https://docs.rs/probabilistic-collections/latest/probabilistic_collections/ (visited on 2023-08-04).