UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Máster Oficial en **Bioinformática y Biología Computacional** (ByBC)

TRABAJO FIN DE MÁSTER

MASV, A MISASSEMBLY DETECTION AND VARIANT CALLING PIPELINE FOR LONG READS DATA.

Autor: Diego Fuentes Palacios

Director: Tyler Scott Alioto
Genome Assembly and Annotation Team
Centro Nacional de Análisis Genómico (CNAG)

Ponente: Ramón Díaz Uriarte
Departamento de Bioquímica
Universidad Autónoma de Madrid

Febrero de 2020

MASV, A MISASSEMBLY DETECTION AND VARIANT CALLING PIPELINE FOR LONG READS DATA.

Autor: Diego Fuentes Palacios

Director: Tyler Scott Alioto
Genome Assembly and Annotation Team
Centro Nacional de Análisis Genómico (CNAG)

Ponente: Ramón Díaz Uriarte
Departamento de Bioquímica
Universidad Autónoma de Madrid

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Febrero de 2020

Abstract

In ***de novo*** **genome assembly** each genome sequenced and assembled presents its own challenges such as sample quantity, DNA integrity, repetitiveness, heterozygosity... but above all, **mis-assemblies** are often the most difficult ones to tackle. Fortunately the longer read size produced by **third generation sequencing** technologies allow a better characterization of complex regions, usually differentiated by its large number of repeats [1],[2]. This masters project aims to develop an automated pipeline for detecting large structural variants (SV) in de novo assemblies produced by **long reads** which may be indicative of errors in the assembly process. By mapping these reads to their assembly we might be able to pinpoint mis-assemblies or sequence blocks with a high discrepancy to the real genomic fragment from which the read derived.

Methodology: In order to do so a *Snakemake* pipeline was developed. It incorporates the most widely used aligners *Minimap2*[3] and *Ngmlr*[4] as well as two SV prediction software *Sniffles*[4] and *Svim*[5]. It also includes custom scripts to measure recall, precision, F1 and precision-recall trade-off for evaluation purposes as well as some custom scripts for VCF (variant call file) formatting and conversion.

Experiments conducted: First the SV predicting power was benchmarked replicating an experiment in the *Svim* paper[5]: using NA12878 nanopore raw reads (obtained from the Nanopore WGS consortium [6]) mapped against the hg19 human genome reference with 2676 high-confidence deletions and 68 high-confidence insertions as the high confidence SV dataset (validated in a previous study using PacBio and Moleculo reads [7]). After successfully replicating the experiment and setting the default parameters, the pipeline functionality was tested with respect to mis-assembly detection. This experiment involved simulating long reads from a reference genome into which we introduced Svs at known positions. The **simulated reads** would then be mapped to the unaltered reference in order to detect the rearrangements (i.e. “mis-assemblies”). In other words, the idea was for the unaltered hg19 reference to resemble a de novo assembly mis-assembled with respect to the reads (rearranged-based simulated reads), provided the reads are the "ground truth", and with the knowledge beforehand of where the real SV were located. The hg19 reference genome (in this case chromosomes 21 and 22 to avoid larger computation times in a more controlled environment) were rearranged introducing simulated **homozygous** SVs. 200 deletions, 100 inversions, 200 tandem duplications and 100 insertions (cut & paste more akin to conservative transposition) were introduced using the R package *RSVsim*[8], providing us a high confidence “truth” SV dataset. The *SimLoRD*[9] python package was required to simulate PacBio reads (x53 coverage) based on the rearranged hg19 reference. Thus two conditions were proposed: rearranged-based reads mapped against the hg19 reference to test prediction in homozygosity; and a merge of simulated reads based on the rearranged reference (x26 coverage) and normal reference simulated reads (x26 coverage) for a total of x52 coverage **heterozygous** reads against the normal reference.

Discussion: The results obtained are quite promising. With the caveat that only simulated data was used instead of an actual assembly, the results seem to indicate that long read SV detection methods can be used as a tool for mis-assembly detection. Though it is not implemented, it would have been interesting to merge the calls from both SV predictors, generating high confidence consensus calls to reduce the impact of spurious calls on complex genome assemblies, such as the ones from the plant kingdom[10]. Future work would focus development on an algorithm or reducing the number of mis-assemblies in draft genomes by rearranging the target assembly according to the calls made by the MASV pipeline.

Key words

-*de novo* assembly, mis-assemblies, third generation sequencing, long reads, structural variants, Snakemake pipeline, simulated reads, homozygous, heterozygous.

Agradecimientos

Me gustaría comenzar agradeciendo este trabajo al equipo de *Genome Assembly and Annotation*. En primer lugar a mi director, Tyler, por su comprensión así como por la confianza depositada en mi. También a Fernando y a Jessica, por estar siempre ahí pendientes y por su paciencia enseñándome. También a Oscar, Yago y Emanuele, por esas conversaciones tan *Revertianas* a la hora de la comida. Y, por supuesto, al resto de compañeros del CNAG por su calidez y cercanía de trato, que facilitaron mucho el proceso de adaptación y la estancia.

A mi familia, por el apoyo y confianza desmesurado que siempre me han dado. A mis padres Ángel y Marian, por permitirme aprovechar esta oportunidad y hacer todo lo posible aunque fuera a la distancia, a David porque es un ejemplo de autosuperación y por los manjares que me preparaba al volver a casa, a Rodrigo por todas esas conversaciones que siempre sacaban lo peor y lo mejor de mí, por Consuelo por su atención y preocupación perpetuas y por último para Hermelinda, porque me hubiera encantado que pudieras estar ahí para verme finalizar este capítulo en mi vida. Os lo debo todo.

Para mis amigos, por todas las conversaciones intrascendentes que hemos mantenido y por estar pendientes de mi desde cualquier sitio. A los que estáis cerca, por esos buenos ratos de desconexión. A los que estáis lejos, por esas vueltas a casa legendarias.

Por último, que no menos importante, a Ainoa por apoyarme y aguantarme transcendiendo todo tipo de barreras y distancias. Mucho de lo que he conseguido se lo debo a su comprensión, su confianza y sus ánimos

Muchas gracias a todos.

Per aspera ad astra

List of Figures

List of Codes

List of Tables

1 Introduction

1.1 Third generation sequencing: its impact on *de novo* assemblies and structural variant detection

In *de novo* genome assembly each genome sequenced and assembled presents its own challenges: for example, sample quantity, DNA integrity, polyploidy, heterozygosity and/or repetitiveness. Moreover, no genome assembly is perfect: mis-assemblies are often made. Previous studies[2],[11] have observed than most assembly breaks are caused by genomic repeats that are equal to or longer in length than the sequencing reads. This is because unresolved repeats confound how the assembled contigs should be linked together, causing the assembler to end the contig at that break point (*Figure 1.1).* Consequently, when using reads shorter than the common repeats in the sequenced genome, repeats can cause assembly errors where distant regions of the genome are incorrectly assembled together[12] or a contig truncated by an unresolved repeat region is stitched back together with an incorrect genomic region reference. Although some progress has been made developing tools to identify mis-assembly and structural variants in assemblies from short reads, they are still quite unreliable as they have been reported to suffer from low sensitivity [13],[14] (30–70%) and up to 85% false discovery rate as it has been reported by Sedlazeck in his review[1].

In the recent years, the development and use of third generation sequencing (TGS) technologies have increased rapidly in the genomics field; being able to yield reads of 10kb length as opposed to the few hundred bp produced by second generation sequencing technologies (NGS). The most widely used third generation single-molecule sequencing technologies are Pacific Biosciencies (PacbBio[15]) and Oxford Nanopore Technologies(ONT[16])

Pacbio's Single Molecule, Real-Time (SMRT) technology performs sequencing-by-synthesis (in essence akin to Illumina technologies), but it does so by capturing a single DNA molecule. The PacBio platform uses a circular DNA template by ligating hairpin adaptors to both ends of target double-stranded DNA[15]. Thus the DNA template is sequenced multiple times to generate a continuous long read. By removing the adapter

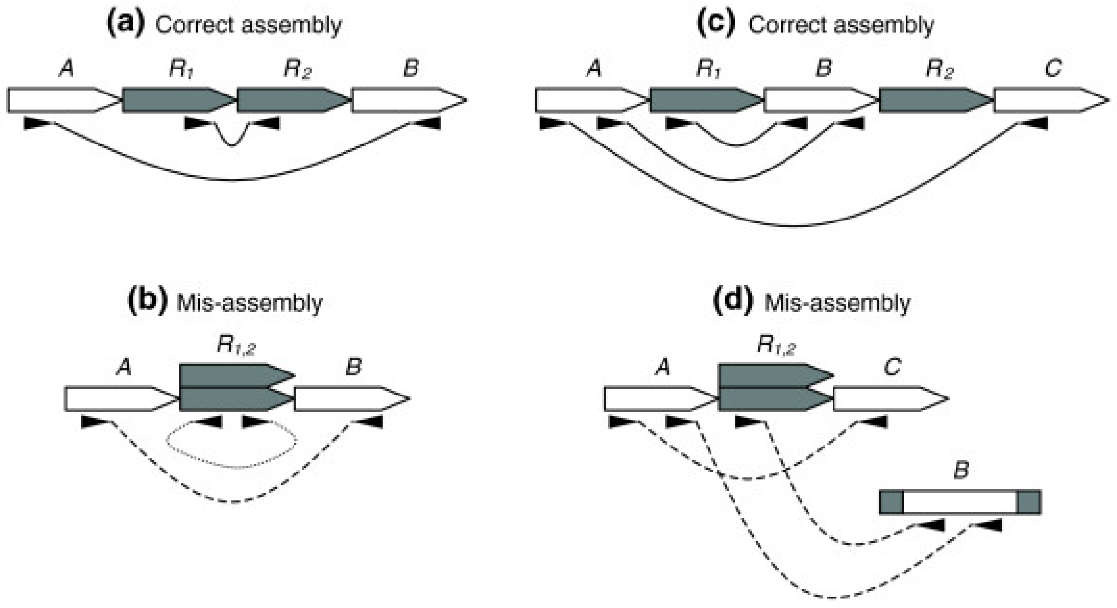

Figure 1.1: **Pair-End signatures for mis-assemblies**. Images (a, c) represent the correct assembly of a complex region caused by genomic repeats, while images (b, d) represent the mis-assembly due to the collapse and mis-linking of mate-pair: (a) Represents a two-copy tandem repeat shown with oriented mate-pairs in a correct fashion. b) Represents a collapsed tandem repeat shown with confounded mate-pairs (c) Represents a unique sequence B within a two-copy repeat, shown with oriented mate-pairs in a correct fashion. (d) Represents a collapsed repeat shown with compressed and mis-linked mate-pairs. Source of the image: [12]

sequences of the continuous long read, multiple subreads can be generated. This step is key to generate the circular consesus sequence reads (CCS) with higher reported accuracy, often $>$ 99%[17] as shown in the *Figure 1.2*.

On the other hand ONT technologies sequence directly a native single-stranded DNA molecule by measuring characteristic current changes as the bases are threaded through the nanopore by a molecular motor protein[16]. Similarlly to SMRT, ONT technologies (MinION and PromethION) originally used a hairpin adaptor to bound the DNA template to its complement. The raw read that passed through the nanopore followed a template-adaptor-complement structure. Thus it could be split into two 1D reads just by filtering out the hairpin adaptor and a higher accuracy 2D read could be produced by generating the consensus sequence of the two 1D reads[18] (*Figure 1.3*) However 1D tech with no hairpin is the standard library prep right now. 2D with hairpins was abandoned due to some technical limitations: the evaluation of the DNA quantity at different stages of the protocol suggested that the hairpin tagmentation step and the bead purification step resulted in a huge loss of DNA for the 2D analyses[19].

Both currently exhibit a similar error rate (around 10-20%[20]) albeit for different reasons: ONT struggles to detect the transition between two identical k-mers (hence complicating the identification of homopolymers longer than the k-mer length [21]) while SMRT error rate mostly originates from lower signal-to-noise ratio from single DNA molecules[17]. Consequently, this error rate poses a tangible challenge for further use of these technologies. However, correction methods exist and the advantages of using long reads far outweigh the disadvantages for de novo assembly.

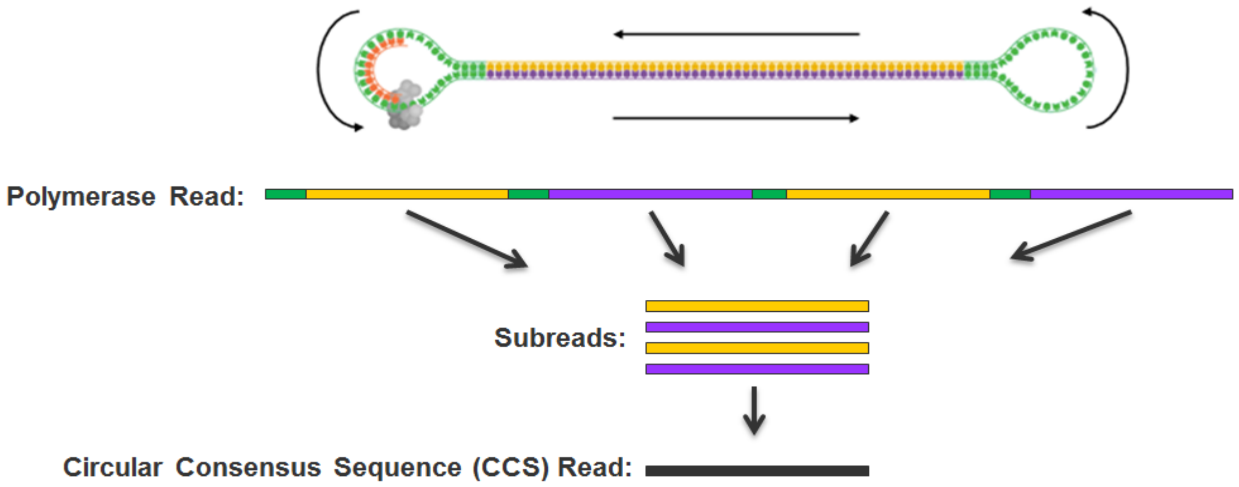

Figure 1.2: **Schematic representation of the circular consensus sequence generation through PacBio's SMRT® .** The circular DNA template has ligating hairpin adaptors to both ends of target double-stranded DNA. The **polymerase** reads are a sequence of nucleotides incorporated by the DNA polymerase while reading the circular template and are most useful for quality control of the instrument run. The **subreads** are generated after filtering the adaptor sequences. **CCS** is an example of a special case where at least two full subreads are collected for an insert (the highest quality single sequence for an insert, regardless of the number of passes). Image and information obtained through PacBio's SMRT® Portal Help v2.2.0 documentation.

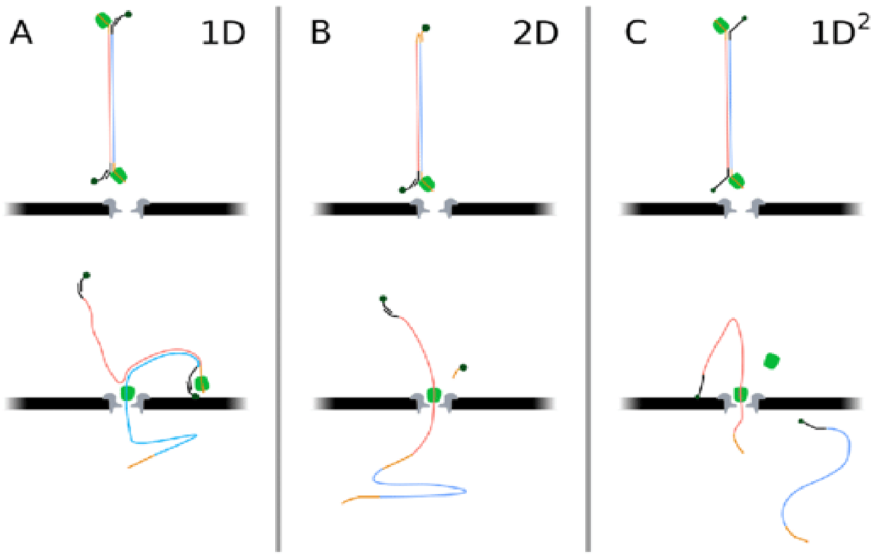

Figure 1.3: **Schematic representation of the 1D, 2D and 1D^2 sequencing approaches developed by Oxford Nanopore Technologies. A)** In the 1D approach just the template strand (in blue) is threaded by the motor protein (in green). The complement strand (red) is discarded and sequenced.**B)** In the 2D approach both the template and complement are sequenced provided they are both linked through a hairpin (in orange).**C)** For the 1D^2 approach both strands are sequenced too. However the complement strand is tethered to the membrane while the template is sequenced. Subsequently the complement strand is drawn in and the tether is pulled loose, as opposed the 2D approach. Image and information obtained from [22]

One of such advantages is structural variant detection. Using these third generation sequencing several studies have reported around 20,000 SVs per human genome, most of which could not be detected using short-read sequencing[23]].

Structural variants (SVs) are typically defined as genomic variants larger than 50bps (e.g. deletions, duplications, inversions, translocations ...). They are often studied due to their impact on genes malfunction and regulatory regions disruption. However SVs can be hard to correctly identify (*Figure 1.4*).

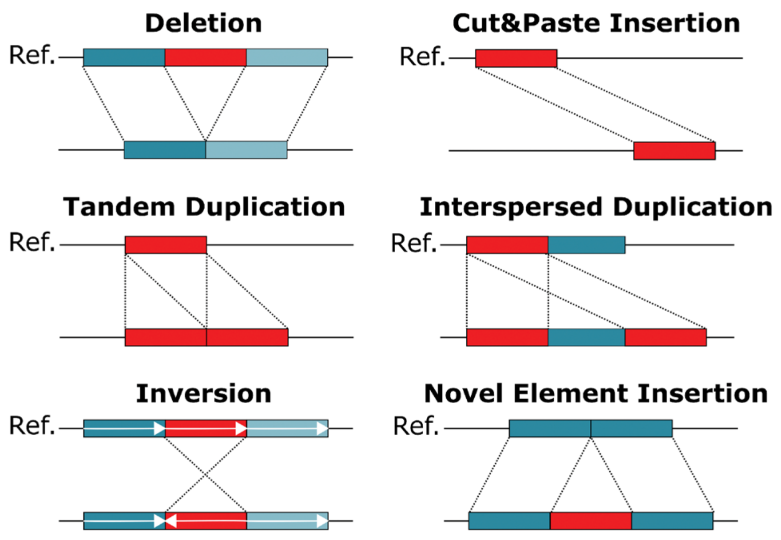

Figure 1.4: **The SV calling problem:** Some SV types can be simple to describe conceptually but they are quite difficult to correctly identify via current variant calling methods. Some callers might classify a call a Interspersed Duplication a Translocation while other might just simply call it Insertion. And both would have been close to the truth but not quite there yet. It is important to know the limits of the variant calling software in order to correctly interpret the results. Image obtained from [5]

Two main approaches have been used for SV discovery: a mapping-based approach or a *de novo* assembly-based approach[1]. In the first approach the SVs are detected through the means of direct mapping of the reads to a reference genome. The second approach utilizes *de novo* assembly followed by whole-genome alignment between the samples or the reference genome.

Both approaches have their fair share of advantages and disadvantages .. The main strengths of a mapping approach are that it requires the least amount of coverage (a minimum of only 15), is able to identify heterozygous SVs and is more robust to genomic amplifications which tend to assemble poorly[4]. The main strength of the *de novo* assembly approach is to provide sample-specific variations that might be hard to resolve with the mapping approach".

The *de novo* approach however faces some challenges which have been brilliantly put into words by Sedlazeck in his review: " The *de novo* approach has a higher sequence coverage cost (about 50x), it is more demanding computationally and it has difficulties resolving large insertions and novel sequences in the sample will not map well or at all in the reference. Additionally it depends on the quality of the assembly, the quality of the reference and the quality of the reads: a major challenge are posed by the repetitive sequences which can mask SVs, although longer contigs can be more robustly aligned than short contigs. Moreover, detection of heterozygous variants and the analysis of polyploid regions remain challenging as heterozygous variants will often be left out of an assembly or represented only as alternative contigs"[1].

1.2 Aim of the study

Most of the mis-assembly detection software are applied on second generation sequencing reads. We however intend to to develop a mis-assembly detection pipeline using the long read data advantages to resolve this complex repetitive regions. In order to do so, it is proposed the **mapping approach** to detect possible mis-assemblies in *de novo* sequencing of new species through an automated pipeline able to be run in a cluster or local drive.

In a *de novo* assembly process our closest "ground truth" so to speak are the genomic reads (after taking into account the error rate implicit by the technical limitations). Hence by mapping the raw reads against our assembly we would be able to pinpoint the rearrangements (i.e. "mis-assemblies") or sequence blocks with a high discrepancy with the real genomic fragment derived from which the read derived.

However, this is easier said than done. SV detection methods require fine-tuning metrics (such as precision, recall and F1 score) on benchmark datasets. As benchmarks usually require a control dataset for which we already know beforehand the SVs against which the prediction software can be evaluated. Thus, in this work we generate a simulated environment in which we artificially introduce SVs into an assembly, simulate long reads from it and then map the simulated reads to the unaltered assembly.

1.3 Main objectives and work plan

In order to fullfill this project's aims, several specific objectives are proposed each with specific milestones.

1. **Benchmarking analysis:** Compare mapping and calling algorithms for structural variants.
   - Milestone 1.1: Establish a benchmark dataset and evaluation criteria.
   - Milestone 1.2: Compare different preprocessing strategies.
   - Milestone 1.3: Compare different aligners.
   - Milestone 1.5: Compare different SV callers.
2. **Develop a long read SV calling pipeline for TGS data:** Clear and concise rules through `Snakemake` taking advantage of `conda` environments.
   - Milestone 2.1: Develop and debug the pipeline using Snakemake.
   - Milestone 2.2: Implement the final workflow.
   - Milestone 2.3: Fine tune and test it on a simulated environment.
3. **Upload all the code to a Github repository.**

It must be said that most that not all the objectives nor all the milestones had the same work focus. It was prioritized the second objective above all as it was inherently related to the other two; developing the snakemake pipeline allowed easier benchmarking and all the code used was being uploaded constantly to a Github repository.

1.4 Limitations and challenges

Unfortunately, this project has suffered a fair amount of logistical challenges which have hampered its ability to develop some of the proposed milestones, specifically the ones requiring testing on preprocessing approaches.

It was intended to benchmark the pipeline on whether the raw reads were preprocessed or not, as some evidence [24] indicated that preprocessing the reads yielded better alignments and thus better SV detection calls.

```
#!/bin/bash
canu [-correct | -trim | -assemble | -trim-assemble] \
  [-s <assembly-specifications-file>] \
   -p <assembly-prefix> \
   -d <assembly-directory> \
   genomeSize=<number>[g|m|k] \
   [other-options] \
   [-pacbio-raw | -pacbio-corrected | -nanopore-raw | -nanopore-corrected] *fastq
```

Code 1.1: Canu command -help.

The proposed approach to generate preprocessed reads was using `CANU/1.8.0.`, the cutting edge version of *Canu* at the moment. *Canu* is a a successor of Celera Assembler that is specifically designed for noisy single-molecule sequences(such as the PacBio RSII or Oxford Nanopore MinION)[25]. The `canu` command is the ‘executive’ (or ’overseer’) program that runs all modules of the assembler. It oversees each of the three principal tasks (correction, trimming, unitig construction), each of which performs multiple and varied tasks (*Code 1.1*). *Canu* ensures that input files for each step exist, that each step successfully finished, and that the output for each step exists, which is critical for assembling provided it requires huge computational resources. Additionally it is capable of performing minor bits of processing (such as reformatting files) but its main function is executing other programs. More information can be found at https://canu.readthedocs.io/en/latest/index.html.

Canu allows the user to run one task at a time by using the -correct, -trim or -assemble options, although the default is to perform all three tasks. Each of the three tasks (read correction, read trimming and unitig construction) follow the same pattern in the *Canu* pipeline (information extracted from https://canu.readthedocs.io/en/latest/tutorial.htmlcanu-the-pipeline):

- Load reads into the read database, gkpStore.
- Compute k-mer counts in preparation for the overlap computation.
- Compute overlaps.
- Load overlaps into the overlap database, ovlStore.
- Process the reads and the overlaps:
  - The read correction task will replace the original noisy read sequences with consensus sequences computed from overlapping reads.
  - The read trimming task will use overlapping reads to decide what regions of each read are high-quality sequence, and what regions should be trimmed. After trimming, the single largest high-quality chunk of sequence is retained.

  - The unitig construction task finds sets of overlaps that are consistent, and uses those to place reads into a multialignment layout. The layout is then used to generate a consensus sequence for the unitig.

It was intended to use the correction and the trimming tasks consecutively in order to preprocess our reads. The intention was to benchmark whether error correction and trimming improved the SV prediction calls. Unfortunately it was proven to be a technological bottleneck hard and unfeasible to maintain at a time when the cluster management team were prioritizing resources.

I *Canu* is a computationally intensive software. The MHAP (Adaptive MinHash k-mer weighting as an overlapping strategy) is able to produce up to a thousand batch jobs for large repetitive genomes. Although you can set the number of threads and the degree of parallelism; the sheer volume of data processed limits it for longer runs in addition to overwhelm the job priority system in our cluster.

II *Canu* is storage intensive software. By design, *Canu* will only remove intermediate files after each step is 100% finished. In large repetitive genomes (such as humans), the intermediate files size reached over 20 Terabytes. With most of `\scratch` quota completely and after two storage quota extensions, it was unfeasible to keep it up.

III *Canu* is quite vulnerable to data corruption. Ideally, each job would start and finish without any kind of issues. With several cluster restarts, some of the submitted jobs were corrupted, though initially they weren't reported as such. Thus after starting the next step that it would be found out that some of the jobs were indeed corrupted. Thus it was required to return back to the last successful step and resubmit manually most of those jobs, which proved to be really time consuming.

For all the above mentioned reasons, *Canu* preprocessing of the reads was scrapped, but only after some taxing and time consuming efforts were already invested, which affected some of the time originally intended to polish the benchmark objective.

2

Snakemake. State of the Art

2.1 Workflow and basic rule structure

The Snakemake workflow management system is a tool to create reproducible and scalable data analyses[26]. Workflows are defined through a language close to Python syntax. Moreover, it can be seamlessly scaled to server, cluster, grid and cloud environments, without the need to modify the workflow definition; making it one of the best candidates to develop an automated pipeline and becoming one of the most popular workflow builder tools to date, with an impressive rate of 3 citations per week according to its documentation. The online documentation can be found at https://snakemake.readthedocs.io/en/stable/index.html.

A workflow consists of a set of user-defined rules that denote how to produce an output file from an input file through the means of a shell command or Python code (*Code 2.1*). The workflow is implied based on the dependencies that arise between the rules: for example, from rule B needing the output of rule A as input. In addition to that, Snakemake is capable of handling multiple named wildcards whose values are inferred automatically from the files. This functionality is extended to how the workflow handles the input and output files.

A rule is defined by 3 basic components:

I The rule's name.

II The input files required and the output files produced.

III The shell command or Python code used to create the output from the input.

It must be noted however that with each new Snakemake release, more utils have been added to the rule structure. Some of the most relevant are:

- Conda environments: can be specified per rule in order to use the dependencies of a fully functional conda environment using the directive `conda`. In this way, conflicting software versions (e.g. combine Python 2 with Python 3) can be used together in the same workflow. Additionally this is extremely beneficial in order to create user-friendly reproducible and scalable data analsyses as all the dependencies located within the conda environment which can be cloned or duplicated.

- Benchmarking: With the `benchmark` directive, Snakemake can be instructed to measure the wall clock time of a job. Similar to output files, the path can contain wildcards (although it must be the same wildcards as in the output files)
- Error logs: With the `log` directive, Snakemake can be instructed to output the error log of a job. Similar to output files, the path can contain wildcards (it must be the same wildcards as in the output files)
- Non-file parameters: Allows the user to define certain parameters separately from the rule body using the directive `params`. It is quite handy as it is able to handle wildcards as well as config file parameters.
- Threads and resources: Snakemake can be instructed to use a user-defined number of `threads` in a specific rule. Likewise, you can specify the cluster/grid `resources` for every rule.
- Tool wrappers: A wrapper is a short script that wraps (typically) a command line application and makes it directly addressable from within Snakemake. They can be accessed from the Snakemake Wrappers repository.

```
rule index_bam:
    input:
        "wdir/pan_troglodytes.bam"

    output:
        "wdir/pan_troglodytes.bam.bai"

    log:
        "logdir/pan_troglodytes.index_bam.log"

    params:
        mode = "index"

    benchmark:
        "benchmarkdir/pan_troglodytes.index_bam.txt

    threads: 8

    conda: "snakemake_alignment.yml"

    shell:
        "samtools {params.mode} -@ {threads} {input} {output} 2> {log}"
```

Code 2.1: Example of a basic Snakemake rule.

Snakemake has control structures and checkpoints too, e.g. When two rules can produce the same output file, Snakemake cannot decide per default which one to use. In those cases an `AmbiguousRuleException` is thrown. In order to avoid this kind on situations, it is recommended to provide a `ruleorder` for example:

`ruleorder`: rule1 > rule2).

In addition to that, there are other ways to bypass the `AmbiguousRuleException` such as conditional control structure for different rules (e.g. using `If` conditionals). To sum up, all this features complement Snakemake brilliantly as a workflow tool.

2.2 Modularization and complex features

It is worth mentioning that some additional features have been added release after release to enhance the workflow directives. Most notably:

- Rule modules: Through the use of the directive `include`: Snakemake can be instructed to include another Snakefile into the current one. This is useful to re-use entiere set of rules or simply to structure large workflows in a clearer way.
- Configuration file: With the `config` directive, Snakemake allows the user to provide configuration files for making the workflow more flexible, e.g. Providing parameters to the defined rules. Moreover, it can be used effectively for abstracting away direct dependencies to a fixed HPC cluster. The config file must be provided as JSON or YAML(.yml) extensions.
- Functions as input: Snakemake can also make use of functions that return single or lists of input files. It can be used lambda expressions instead of full function definitions. Using this, rules can have entirely different input files (both in form and number) depending on the inferred wildcards enhancing the workflow's versatility.

Wildcards are quite handy in Snakemake. Being able to define a wildcard and use through all the rules dependencies speeds up a lot of coding time. In addition to that, they can be defined through some basic functions implemented into Snakemake such as `expand` (*Code 2.2*).

```
# CONFIG AND MODULES #

include: "get_reference.smk"
config: "config.json"

ids = config["fastq_ids"] #List with all the fastq ids.

# RULE #
rule minimap2:
    input:
        fastq = expand("wdir/{files}.{format}"; files= ids.split(’,’), format=["fastq",
            "fastq.gz"]),
        reference = rules.get_reference.output

    output:
        protected(outdir + "/{files}.bam")

    log:
        "logdir/{files}.mapping.log"

    params:
        technology = "ont"
        outdir = config["outdir"]
    benchmark:
        "benchmarkdir/{files}.mapping.txt

    threads: 8

    conda: "snakemake_alignment.yml"

    shell:
        "mkdir -p {params.outdir}; minimap2 -MD -t {threads} -ax {params.technology} {
            input.reference} {input.fastq} | samtools sort -@ {threads} -O BAM -o {
            output} 2> {log}"
```

Code 2.2: Example of a more complex Snakemake rule.

Let's briefly describe the above rule: First it must be noted that a config file "config.json" was provided. In addition to that, other rule module was included, specifically the rule "get_reference" which will probably obtain the reference genome from a database. Then the **rule minimap2** is defined.

If we take a look on the input, we notice a more complex syntax. The reference input is provided directly from the output of the rule module "get_reference" (explicitly stating a dependency between the two rules). The `fastq` input is the result of a Snakemake function `expand`. This function uses the python itertools function product that yields all combinations of the provided wildcard values. In this case file is each file id from the `ids` list defined above and format allows any combination of the defined formats either fastq or fastq.gz. Looking at the output, log and benchmark, they all share the same wildcard file. Interestingly the output has the "protected" flag which protects it against accidental deletion or overwriting. As opposed with the input, this wildcard is actually inferred from the output and both log and benchmark require to use the same wildcard as the output.

Taking a look into the params, it must be noted that some of them have been explicitly provided from the config file. Finally you can add quite complex shell syntax: creating a directory, then using minimap2 whose output will be piped through samtools sort. Snakemake has proven itself as a really handy tool; easy to use but hard to master.

3 MASV pipeline implementation

In order to get a fully reproducible data analysis, it is not sufficient to be able to execute each step and document all used parameters. The used software tools and libraries have to be documented as well. Consequently it was decided from the very start that the MASV pipeline would incorporate three elements to achieve this goal: A config file as well as a custom script to build it, a highly modular workflow defined by the specific task per rule set and conda enviornments, allowing a user-friendly installation and to keep track of all the software dependencies used to create this pipeline (Appendix A)

3.1 Snakefile MASV_pipeline.smk

The workflow base directory is where the main *MASV_pipeline.smk* is located. Fortunately there is no need to use the workflow base directory as the working directory of the pipeline as it can be set up directly from the config file. The *MASV_pipeline.smk* is the main Snakefile of the pipeline as it handles all the other rule modules as well as providing a base config and conda environment (*Code 3.1*). In the main Snakefile seven target rules are defined:

- **Rule all**: It is the main target rule and it is defined as the first rule of the pipeline. It takes as an input the output of all the rules in the pipeline.
- **Rule mapping_only**: This target rule is set to perform only the mapping on the provided input reads.
- **Rule sniffles** or **rule svim**: These target two rules produce the final SV prediction VCF (after filtering) for either *Sniffles* or *Svim*, individually.
- **Rule eval_sniffles** or **rule eval_svim**: These two target rules are used to obtain the evaluation metrics of either *Sniffles* or *Svim.* These evaluation metrics and its script is properly explained in the custom scripts section.
- **Rule sanity_check**: This target rule is set to produce a set of plots and stats based on the coverage, the depth and mapping quality of the alignment.

```
###############
# CONFIG FILE #
###############

configfile: os.path.join(workflow.basedir, "lib/config/config.json")

##############
# PARAMETERS #
##############

""" A set of global parameters was set for the pipeline """

################
# RULES STEPUP #
################

include: "lib/rules/alignment.smk"
include: "lib/rules/QC.smk"
include: "lib/rules/calling.smk"
include: "lib/rules/bedtools_eval.smk"

##############
# MAIN RULES #
##############

rule all:
    input:
        """input for target rule all"""

rule mapping_only:
    input:
        """input for target rule mapping_only"""

rule sniffles:
    input:
        """input for target rule sniffles"""

rule svim:
    input:
        """input for target rule svim"""

rule eval_sniffles:
    input:
        """input for target rule eval_sniffles"""

rule eval_svim:
    input:
        """input for target rule eval_svim"""

rule sanity_check:
    input:
        """input for target rule sanity_check"""
```

Code 3.1: Schematic representation of the pipeline.smk .

Each of these target rules can be directly called using the `sanakemake` module. In addition to these 7 target rules, I've defined 4 additional Snakefiles. They were named after their functionality. Hence they were named *alignment.smk, QC.smk, calling.smk* and *evaluation.smk.*

We can use the `sanakemake` command option `--dag` on any rule (for example the **rule eval_svim**). This yields a directed acyclic graph (DAG) of jobs where the edges represent dependencies and the nodes each rule (*Figure 3.1*). For rules with wildcards, the value of the wildcard for the particular job is displayed in the job node.

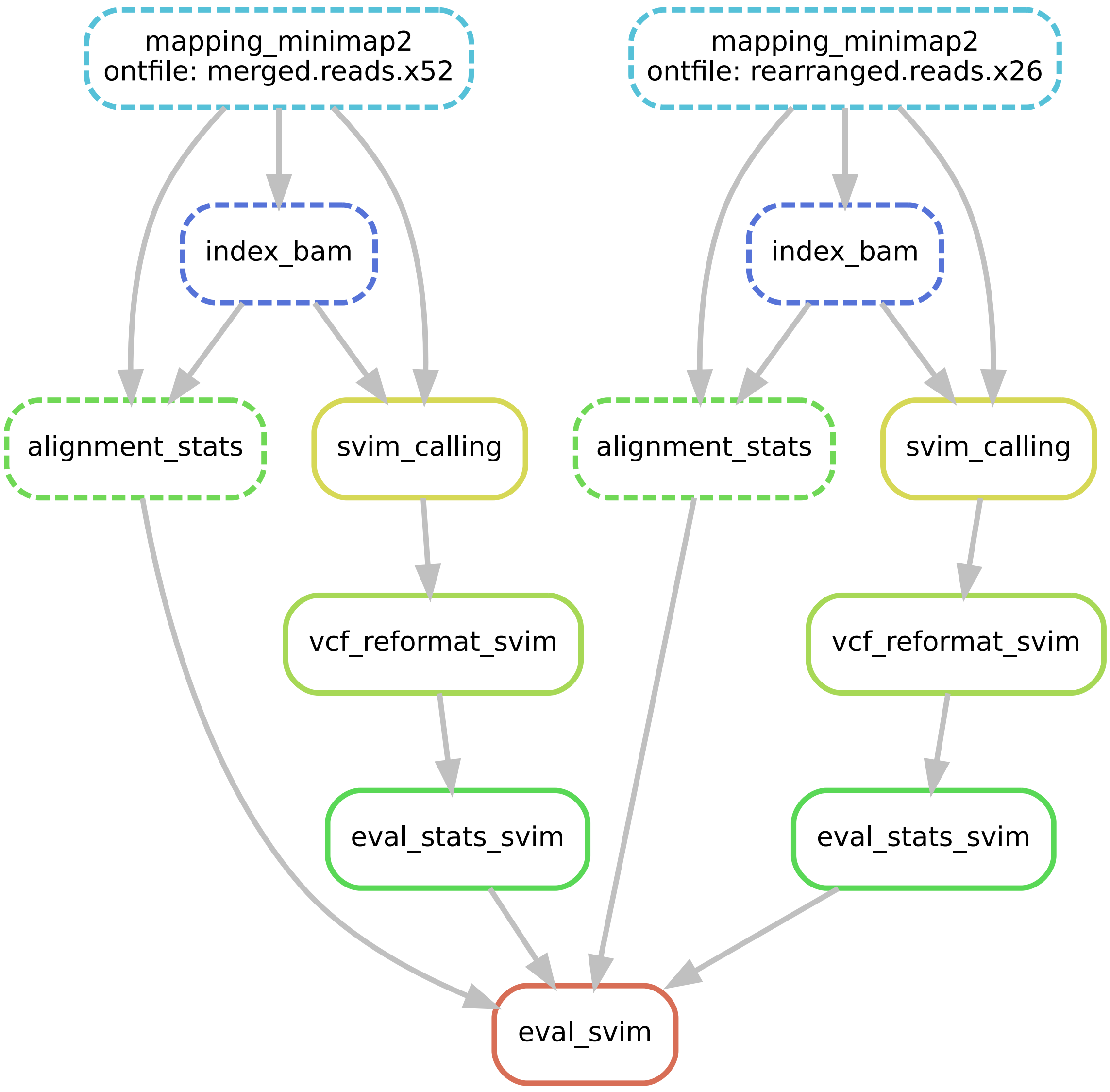

Figure 3.1: **DAG for the rule eval_svim**: All the inferred dependencies for the target rule are present in the image as dashed boxes while the rules explicitly called by the rule eval_svim are present as normal boxes. Please note that in this particular example two fastq files went through the target rule (being the "ontfile" the infered wildcard).

Now we are going to take a look at each of the other four beforementioned Snakefiles and how they are implemented along MASV.

3.1.1 MASV implementation: Snakefile *alignment.smk*

The *alignment.smk* Snakefile contains four rules, two of which are dependant on the user-defined aligner software (which is located in the *config.json*). Depending on the selected aligner either **rule_minimap2** or **rule_ngmlr** are going to be called. This first set of rules are the ones mapping the raw reads to a reference. After doing so, **rule index_bam** will generate the index necessary for the SV calling step as both *Sniffles* and *Svim* require the input BAM to be indexed. At the same time, the **rule alignment_stats** will produce some basic alignment stats such as the proportion of reads that were mapped to the reference and the ones which were discarded.

The alignment step is the most time consuming of the MASV pipeline. *Ngmlr* is often used as an accurate aligner (and it has been reported enhancing *Sniffles* results[4]) but it is computationally intensive. For large repetitive genomes and between 16 and 24 threads, it can take a few days for computation time. *Minimap2* on the other hand is one of the fastests aligner[3]. With half the threads it would take less than 24 hours to have the job done. As it is case specific, it is left to the user discretion which of the two aligners should be used.

3.1.2 MASV implementation: Snakefile *QC.smk*

The *QC.smk* Snakefile contains three rules. All 3 rules require the output of the mapping rule in the *alignment.smk*. It performs some basic quality control plots using *Mosdepth* and *Nanoplot*.

Mosdepth was selected because it is about 2x as fast `samtools depth`[26]. Through some scripts we are able to plot the distribution of proportion of bases covered at or above a given threshold for each chromosome/scaffold and genome-wide.

Nanoplot is a plotting tool for long read sequencing data and alignments[27]. It has been a quite popular choice for quality control for long read data (and more specifically ONT data), so it was incorporated into the workflow. It can produce up to 12 different plots: Histogram of read length (including log transformed), bivariate plot of length against base call quality, heatmap of reads per channel, cumulative yield plot, Violin plot of read length over time... etc.

3.1.3 MASV implementation: Snakefile *calling.smk*

The *calling.smk* Snakefile contains three rules. The first two are called **rule sniffles_calling** and **rule svim_calling**. These two rules require the output of the mapping rule and the output of the index rule from the *alignment.smk* Snakefile as input. Depending on the target rule called from the workflow Snakefile, either rule or both will be called. The final rule is a **rule svim_filtering,** which will apply the Q-score filtering threshold provided in the config file.

Sniffles is a structural variation caller software that was jointly developed with *Ngmlr*[4]. It requires third generation sequencing data and it detects SVs using evidence from split-read alignments, high-mismatch regions, and coverage analysis. *Sniffles* report deletions (DEL), duplications (DUP), insertions (INS), inversion (INV) and translocations (TRA) as the standard SV types. Furthermore, it is able to report some complex events such as inverted duplications (INVDUP) and other rare cases where it is not certain what type the SVs is e.g. DEL/INV [4].

On the other hand, *Svim* is a novel structural variant caller for long reads[5]. It is able to detect, classify and genotype five different classes of structural variants. Unlike existing methods, *Svim* integrates information from across the genome to precisely distinguish similar events, such as tandem and interspersed duplications and insertions. To do so first it gathers the SV signatures from the split-read-alignments. Then the detected signatures are clustered using a graph-based clustering approach and a novel distance metric for SV signatures[5]. Lastly multiple SV events are merged and classified into higher-order events (i.e. events involving multiple regions in the genome). *Svim* reports the following SVTYPEs: DEL, INV, DUP:TANDEM, DUP:INT, and INS. It must be noted however that *Svim* makes the following distinctions in order to capture and classify these higher-order events[5]:

- DUP signature clusters are called as interspersed duplications unless the genomic origin overlaps a deletion call, in which the duplication is marked as potential cut & paste insertion in the INFO field.
- INS signature clusters that are overlapping or close to matching breakpoints (BRK) are called as interspersed duplications. However, if the genomic origin (as defined by the BRK) overlaps a deletion call, the interspersed duplication is marked as potential cut & paste insertion in the INFO field. The remaining INS signature clusters are called as novel element insertions.

3.1.4 MASV implementation: Snakefile *bedtools_eval.smk*

The *bedtools_eval.smk* Snakefile contains 4 rules. The first two are called **rule vcf_reformat_sniffles** and **rule vcf_reformat_sniffles** and their purpose is to use a custom script to reformat the output vcf of the calling rules to fix the insertion calls. This script is key to correctly determine the correct END coordinates for cut & paste INS. It is necessary because for most of the SV callers the begin and end coordinates are the same or +1.

The other two rules are **rule eval_stats_sniffles** and **rule eval_stats_svim**. Both use as an input the output of the previous two rules. They are the rules in charge of obtaining the precision, the recall and the F1 based on `bedtools intersect` overlaps. In order to do so it is required to provide as an input a high confidence "truth" dataset in order to generate the intersects. The current iteration is tuned for a reciprocal 50% overlap between callset and truth dataset; but it can be manually modified if required.

3.2 Custom scripts implemented

3.2.1 Config file builder custom script: MASV_get_config.py

The config file builder script is named *MASV_get_config.py.* It defines a **CreateConfigurationFile** object which stores all the parameters the user can determine for the Snakemake pipeline (*Code 3.2*).

The **CreateConfigurationFile** class object can be divided in 4 elements:

- The **constructor** function of the object.
- The r**egister methods** whose objective will be to parse the arguments provided by the user.
- The **check_parameters method** whose objective will be to check if the provided relative paths are correct and process the provided reads directory to store their ids into a list in order to be used as wildcards for the Snakemake workflow..
- The **storeParameters** methods whose objective will be to store the user provided parameters in a JSON like format.

```
#!/usr/bin/env python3
import json
import argparse
import re
import os
#######################
###CONFIG FILE CLASS###
#######################
class CreateConfigurationFile(object):
    """Class object which manages Configuration file Manager"""

#####
#1.Create object class Configuration File
configManager = CreateConfigurationFile()

#2.Create object for argument parsinng
parser = argparse.ArgumentParser(prog="create_configuration_file",
                description="Create a configuration json file for the MASV pipeline.")

#2.1 Updates arguments and parsing
configManager.register_parameter(parser)

args = parser.parse_args()

#2.2 Check Parameters
configManager.check_parameters(args)

#3. store arguments to super map structure
configManager.storeGeneralParameters(args)
configManager.storeInputParameters(args)
configManager.storeOutputParameters(args)
configManager.storeWildcardParameters(args)
configManager.storeMinimap2Parameters(args)
configManager.storeNgmlrParameters(args)
configManager.storeSnifflesParameters(args)
configManager.storeSvimParameters(args)

#4. Store JSON file
with open(args.configFile, 'w') as of:
    json.dump(configManager.allParameters, of, indent=2)
```

Code 3.2: Schematic representation of get_config.py.

3.2.2 Config file builder custom script: insertion_fix.py

The purpose of this script is to reformat a VCF file accounting for the correct end coordinates of insertions. Insertions are often presented with START coordinates = END coordinates or END coordinates = START + 1. As I am going to use `bedtools intersect` to detect overlap between the calls and the high confidence rearrangements, it is key for the insertions coordinates to be reliably represented.

In order to do so, this script defines two class objects: **SV_Info** and **VCFile**. **SV_Info** class object is going to be used to process the **VariantFile** objects (through *pysam* dependencies) selecting a set of relevant features including the start and end coordinates. The **VCFile** class object is going to be used to process the VCF, reformat it with the fix and rewrite it keeping the same header (*Code 3.3*).

```
#!/usr/bin/env python3
import re
import pybedtools
import logging
import os
import sys
```

```
from argparse import ArgumentParser, RawDescriptionHelpFormatter
from pysam import VariantFile
from pyfaidx import Faidx
######################
# Define the Objects #
######################
##First create the SV_Info object##
class SV_Info:
    def __init__(self, vcf_provided, caller):
        self.id = vcf_provided.id
        self.type = vcf_provided.info.get('SVTYPE', None)
        self.chr1 = vcf_provided.chrom
        self.pos1 = vcf_provided.pos
        self.pos2 = vcf_provided.stop
        if caller == "sniffles":
            self.length = abs(vcf_provided.info.get('SVLEN', None))
        else:
            self.length = vcf_provided.info.get('SVLEN', None)
        self.vcf_record = vcf_provided
        if len(vcf_provided.samples.keys()) != 1:
            raise RuntimeError(
                "Currently only a single sample per file is supported")

    def __repr__(self):
        return "{} {} {} {} {} {} ".format(self.id, self.chr1,
                                           self.pos1, self.pos2,
                                           self.type, self.length)
##Then create the VCFile object##
class VCFile:
    def __init__(self, vcf_path, genome, fix_param=True, caller="svim"):
        self._variants, self.header = self.read_vcf(vcf_path, genome, fixing=fix_param,
            caller="svim")
        """  Here SV_TYPE_NAMES and SV_BASE_TYPE are defined too """

    def read_vcf(self, vcf_path, genome, fixing=False, caller="svim"):
        variants = []
        contig_lengths = {}
        vcf = VariantFile(vcf_path, "r")
        fa = Faidx(genome)
        for item in fa.index:
            contig_lengths[item] = int(fa.index[item].rlen)
        for item in vcf.fetch():
            sv_info = SV_Info(item, caller)
            if fixing and sv_info.type in ['INS', 'DUP_INT', 'DUP/INS']:
                if int(sv_info.pos2 - sv_info.pos1) <= 1:
                    logging.debug("Changing {}/{} to {}/{}".format(sv_info.pos1,
                                                                  sv_info.pos2,
                                                                  sv_info.pos1,
                                                                  (sv_info.pos1 + int(
                                                                      sv_info.length)))
                    for contig in contig_lengths:
                        if contig == sv_info.chr1:
                            max_len = contig_lengths[contig]
                            final = int(sv_info.pos1 + int(sv_info.length))
                            real_stop = sv_info.vcf_record.start + int(sv_info.length)
                            if final > max_len:
                                final = max_len
                            if real_stop > max_len:
                                real_stop = max_len
                            sv_info.pos2 = final
                            sv_info.vcf_record.stop = real_stop
                else:
                    pass
            variants.append(sv_info)
        return variants, vcf.header

    def write_vcf(self, vcfpath):
        vcf = VariantFile(vcfpath, 'w', header=self.header)
        for variant in self._variants:
            vcf.write(variant.vcf_record)
        vcf.close()
```

Code 3.3: Snippet of insertion_fix.py. Only the key code was kept.

The most important function is the **read_vcf method** (**VCFile** object). It takes into account the length of the scaffolds/chromosomes of the provided assembly to make sure the insertions do not exceed that maximum length size if the SV is located in a specific scaffold/chromosome. For each record called by vcf.fetch(), it is extracted our features of interest defined in the SV_Info object. Then it performs the fix only for INS, DUP_INT (this is how the DUP:INT are correctly defined by *Svim*) and DUP/INS that meet the condition END - START <= 1. The new END position is defined and stored back into the SV_Info object.

3.2.3 Evaluation custom script: bedtools_eval.py

This script purpose is to perform the SV calling evaluation with the knowledge beforehand of the SV present in the analysis. This script have two evaluation modes:

- Default mode: The VCF is only filtered by SV_TYPE. The evaluation is performed and the results are stored in a .txt file
- Iterative mode: The VCF is filtered first by SV_TYPE and the evaluation is performed in a iterative loop filtering for the Read Support score for *Sniffles* or Q score for *Svim*. Wether the results are plotted or not; an average of each of the metrics as well as their standard deviation are stored in a .txt file.

As we require some kind of measure of quality/trust on a variant call, Read Support (number of reads supporting a specific variant call) and Q score were selected for *Sniffles* and *Svim* respectively. As the Q score is mainly based on the number of supporting reads, it was deemed a feature that would enable a fair comparison between the two.

The following metrics were selected for our analysis:

- **Precision**: As the name suggests precision means how precise is the model in predicting a class. It is derived from the following formula:

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive}$$

- **Recall**: Can be defined as out of all the actual Positives how many times our model was able to correctly predict it as Positive. It is derived from the following formula:

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative}$$

- **F1 score**: It is the harmonic mean of precision and recall. F1 score is used to find the best or say optimal of both precision and recall. It must be noted that finding optimal of both does not mean to take the average. It accounts for extreme and absurd situations, i.e if precision is 1 and recall is 0 then F-1 score will be 0 as precision and recall are being multiplied. It is derived from the following formula:

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

To correctly estimate the above mentioned metrics, bedtools intersect was used as represented by the code below (*Code 3.4*).

```python
#!/usr/bin/env python3
import sys
import os
import pandas as pd
import numpy as np

def recall_precision_stats(truth, callset):
        """Returns a table with the Recall, Precision and F1 scores when intersecting
            the high confidence dataset and the call dataset for a single SVTYPE.
        Params:
        - truth: Str. Provided hq dataset path.
        - callset: Str. Provided callset path.
        The measures above mentioned where calculated using the following formulas:
        - Recall: TP/(TP+FN) . TP were any called sv that overlaps once with any sv of
            the truth dataset and FN were any of the truth dataset SVs left to be called
            .
        - Precision: TP/(TP+FP) . TP were any called sv that overlaps once with any sv
            of the truth dataset and being FP any of the other called svs that did not
            overlap not even once
        - F1 score: 2*((Recall*Precision)/(Recall+Precision)). It is the harmonic mean
            of precision and recall
        Just to simplify, the TP+FN would be total number of variants for the same
            SVTYPE in the high confidence dataset and the TP+FP would be the total
            number of variants for the same SVTYPE in the callset."""

        #Total number of variants in the high confidence (hq) dataset: TP+FN
        command_hq_1 =  'cat ' + os.path.abspath(truth) + ' | awk \'OFS="\\t" {{ if($1
            !~ /^#/) {{print $0}} }}\' | wc -l'
        number_variants_hq = os.popen(command_hq_1).read()

        #Total number of variants called in the call dataset: TP+FP
        command_call_1 = 'cat ' + os.path.join(callset) + ' | awk \'OFS="\\t" {{ if($1
            !~ /^#/) {{print $0}} }}\' | wc -l'
        number_variants_callset = os.popen(command_call_1).read()


        #Number of hits by the intersect -a truth -b callset
        command_out_1 = 'bedtools intersect -u -a '+os.path.abspath(truth)+' -b '+os.
            path.join(callset)+' -r -f 0.5 | wc -l'
        truth_vs_call = os.popen(command_out_1).read()

        #Number of hits by the intersect -a callset -b truth
        command_out_2 = 'bedtools intersect -u -a '+os.path.join(callset)+' -b '+os.path
            .abspath(truth)+' -r -f 0.5 | wc -l'
        call_vs_truth = os.popen(command_out_2).read()

        #Evaluation Metrics:
        recall = (int(truth_vs_call)-1)/int(number_variants_hq)
        precision = (int(call_vs_truth)-1)/int(number_variants_callset)
        f1 = 2*((recall*precision)/(recall+precision))

        df = pd.DataFrame([[recall],
                    [precision], [f1]],
                     columns = ['Results in proportion'])
        df.insert(loc=0, column='Eval Metrics', value=['Recall','Precision', 'F1'])
        return df
```

Code 3.4: Bedtools intersect implementation to calculate the precision, recall and f1 score.

There are two relevant options in the `bedtools intersect` command line: `-u` and `-r -f 0.5`. `-u` is the command for unique, as we are only interested on whether the variant called is located in the other dataset once (correcting for multiple hits for the same variant). The `-r -f 0.5` option represent a reciprocal overlap between the two variants' coordinates of 50%. By doing so, we are able to be more stringent regarding the overlap region and rejecting spurious hits that are not significant, i.e. rejecting a variant call of 100 bp that overlaps with a real variant of 10 kb.

It is important to note that the iterative loop that is going to be generated for a range of filtering values depend on the **min_read_support** parameter selected for *Sniffles*. As this parameter allows an initial prefiltering, it is only fair to take into account that factor or else we would have the same value for the first iterations where the iterator =< **min_read_support**.

4 Materials, methods and benchmarking

4.1 Materials used for this project

Before conducting the main experiments it was required to perform some benchmarking on the SV caller software. The benchmarking was peformed on raw nanopore data in which several set of parameters were tested for both aligners and sv callers; ultimately selecting a default for both.

The idea behind the main experiments conducted was for the unaltered hg19 reference to resemble a de novo assembly mis-assembled with respect to the reads which are going to be simulated from the rearranged hg19 reference. In order to do so, it was required first to rearrange the reference and then to simulate reads from it. Two main experiments were conducted:

- Evaluate the precision, recall and F1 score in homozygosis.
- Evaluate the precision, recall and F1 score in heterozygosis

4.1.1 Raw data and truth callset for the Benchmarking

Finding a high-quality benchmark structural variant calls for human reference genomes is difficult. High-quality benchmark small variant calls for "platinum quality" reference NA12878 have been developed by the Genome in a Bottle Consortium (GiaB)[28], but so far (and to our current knowledge) only a set of 2676 high-confidence deletions and 68 high-confidence insertions have been provided for benchmarking SV callers[7].

In order to perform the benchmarking using that high quality SV set, it was required to download raw long read data from NA12878. Hence the entire release 6 (rel6) dataset from the Nanopore WGS Consortium was downloaded because since earlier releases are considered deprecated and not representative of the current state-of-the-art nanopore sequencing.

The total rel6 dataset is composed of:

- 53 flowcells
- 132,931,102,331 bases
- 15,666,888 reads

The Nanopore WGS Consortium developed a ultra-long protocol able to yield a N50 above 50 kb or more. They incorporated the x5 ultra-long reads to the x30 of nanopore long read data that was originally sequenced for a total of x35 coverage NA12878 reads[6].

4.1.2 Rearranging the hg19 reference genome

The final milestone of the MASV pipeline implementation required testing the Snakemake workflow with a simulated genome in order to detect rearrangements within the assembly. This experiment involved simulating long reads from a reference genome into which we introduced SVs at known positions. With that objective in mind, the hg19 reference genome was selected as the target for SV simulation. To reduce computation size, only a subset of the hg19 reference genome was modified: the chr21 and chr22 due to their lower size. Using *RSVsim* we were able to simulate and introduce SVs into our chr21-22 hg19 reference. *RSVsim* is a tool for simulating deletions, insertions, inversions, tandem duplications and translocations in any genome available as FASTA-file or BSgenome data package[8]. Structural variations are placed within the given genome in a random, non-overlapping manner. Additionally *RSVsim* is able to introduce biases towards SV formation mechanisms and repeat regions by setting the parameter repeatBias=TRUE: *RSVSim* will then simulates a bias of breakpoint positioning towards certain kinds of repeat regions and regions of high homology.

The bias is calculated taking into account the following two steps:

I Weighting SV formation mechanisms (*RSVsim* provides by default: NAHR, NHR, VNTR, TEI, Other) for each SV type.

II Weighting each SV formation mechanism for each kind of repeat (supported by default: LINE/L1, LINE/L2, SINE/Alu, SINE/MIR, segmental duplications (SD), tandem repeats (TR), Random).

The default weights were chosen by the authors from (*RSVsim* from studies with SVs >1.000bp such as [29],[30] and [31]. This bias feature requires the coordinates of repeat regions for hg19 which can be provided from the UCSC Browser's RepeatMasker track[8].

Moreover, the user can specify the size vector for each of the SV that are going to be simulated. Fortunately *RSVsim* provides another function **estimateSVSizes** that draws random values for SV sizes from a beta distribution. The default shape parameters for deletions, insertions, inversions and tandem duplications were estimated from sequencing studies in the Database of Genomic Variants release 2012-03-29 to estimate the shape of the beta distribution[8]. All of these perks were considered as it was intended to generate human SV as realistic as possible.

With that in mind, the following homozygous SV were simulated :

- 200 deletions.

- 100 cut & paste insertions (more akin to conservative transposition) .
- 100 inversions.
- 200 tandem duplications.

The output of the script is a genomic FASTA as well as a BED file for each of the SV types introduced into it. The code used to generate the rearranged reference is provided below (Code 4.1). The R session info can be located at the supplementary material (Code B.1).

```r
#R 3.4.4
######### LIBRARY #########
#if (!requireNamespace("BiocInstaller", quietly = TRUE))
#  install.packages("BiocInstaller")
#BiocInstaller::biocLite("BSgenome.Hsapiens.UCSC.hg19")
library("BSgenome.Hsapiens.UCSC.hg19")

#BiocInstaller::biocLite("BSgenome.Hsapiens.UCSC.hg19.masked")
library("BSgenome.Hsapiens.UCSC.hg19.masked")

#BiocInstaller::biocLite("RSVSim")
library("RSVSim")

##### SETUP PARAMS  #####
#As the original data ranges from 500 bp to 10kb, we are going to use the default beta
    distribution as the sv vector size and then adjust it with minSize of 50 and maxSize
     of 10000.

del.default <- RSVSim::estimateSVSizes(200, default = "deletions", hist = TRUE )
del.vect <- RSVSim::estimateSVSizes(200, del.default, minSize = 50, maxSize = 10000,
    hist = TRUE )

ins.default <- RSVSim::estimateSVSizes(100, default = "insertions", hist = TRUE )
ins.vect <- RSVSim::estimateSVSizes(100, ins.default, minSize = 50, maxSize = 10000,
    hist = TRUE )

inv.default <- RSVSim::estimateSVSizes(100, default = "inversions", hist = TRUE )
inv.vect <- RSVSim::estimateSVSizes(100, inv.default, minSize = 50, maxSize = 10000,
    hist = TRUE )

dup.default <- RSVSim::estimateSVSizes(200, default = "tandemDuplications", hist = TRUE
    )
dup.vect <- RSVSim::estimateSVSizes(200, dup.default, minSize = 50, maxSize = 10000,
    hist = TRUE )

max.dups <- 10 #Maximum number of repeats for tandem duplications

#Provide the weights that we are going to apply to the biases of our generated sequence:
data(weightsMechanisms, package="RSVSim")
data(weightsRepeats, package="RSVSim")

#For chr21-22
RSVSim::simulateSV(output = "/scratch/devel/dfuentes/hg19_rearranged/chr21-22.rearranged
    /", chrs = c("chr21","chr22"),
                    dels = 200, ins = 100, inv = 100, dups = 200, sizeDels = del.vect,
                        sizeIns = ins.vect, sizeInvs = inv.vect, sizeDups = dup.vect,
                        maxDups = max.dups,
                    weightsMechanisms = weightsMechanisms, weightsRepeats =
                        weightsRepeats, percCopiedIns = 0, repeatBias = TRUE)
```

Code 4.1: R script used to simulate SV into the chr21-22 hg19 reference genome.

4.1.3 Simulating long reads from the rearranged hg19 genome

In order to simulate long reads from the rearranged chr21-22 hg19 genome, the Python package *SimLoRD*. *SimLoRD* is a read simulator for third generation sequencing reads currently focused on the Pacific Biosciences SMRT error model[9]. The baseline error probabilities per subreads can be specified individually for substitutions (`-ps`), insertions (`-pi`) and deletions (`-pd`). Error probabilities for subreads are by default 1, 12 and 2% for substitutions, insertions and deletions, respectively, on average (15% total error probability)

For the proposed experiments that are going to be discussed in detail the Results section, the following simulations were required:

I Simulation of long reads based on the rearranged chr21-22 hg19 genome selecting a specific coverage of x52. Hence we were able to successfully simulate homozygous reads.

II Simulation of long reads based on the unaltered hg19 genome selecting a specific coverage of x26. Through downsampling the rearranged reads (obtained through the first simulation) to x26 coverage and merging them to the unaltered simulated reads we were able to generate a x52 heterozygous simulated reads.

4.2 Benchmarking

While simulated datasets enable the comprehensive comparison of SV caller tools in a controlled and precise manner, they might be unable to reflect the complexity in real sequenced data. The NA12878 dataset is more realistic than the simulated datasets but impose the limitation that there exists no complete gold standard set of SVs that we are currently aware of.

It was selected a high confidence SV dataset that consisted of only 2614 deletions and 68 insertions[7] hence we were expecting a underfitting issue, provided that a recent study by the Human Genome Structural Variation Consortium (HGSVC) found an average of 12,680 deletions and 18,919 insertions per individual[32]. This is precisely true for insertions due to the small size ($n = 68$) of the high confidence dataset, and as such their results should be take with caution. Consequently, the only measure taken into account for the benchmark analysis was recall because precision could not be accurately measured in this scenario.

The raw reads were first mapped against the hg19 reference using *Minimap2* and *Ngmlr* (with default parameters).

After the mapping step, a quality control of the output BAMs was performed. Using the *QC.smk* rules there was plotted the average depth and N50 of NA12878. The total depth was slightly below the expected 35x[6] for both *Ngmlr* and *Minimap2* (*Figure 4.1*). *Ngmlr* performed a little bit better, as it yielded 34.2 instead of the 34.0 provided by *Minimap2*.

N50 statistic defines assembly quality in terms of contiguity. Given a set of contigs, the N50 is defined as the sequence length of the shortest contig at 50% of the total genome length. It can also be described as a weighted median statistic such that 50% of the entire assembly is contained in contigs or scaffolds equal to or larger than this value. In our

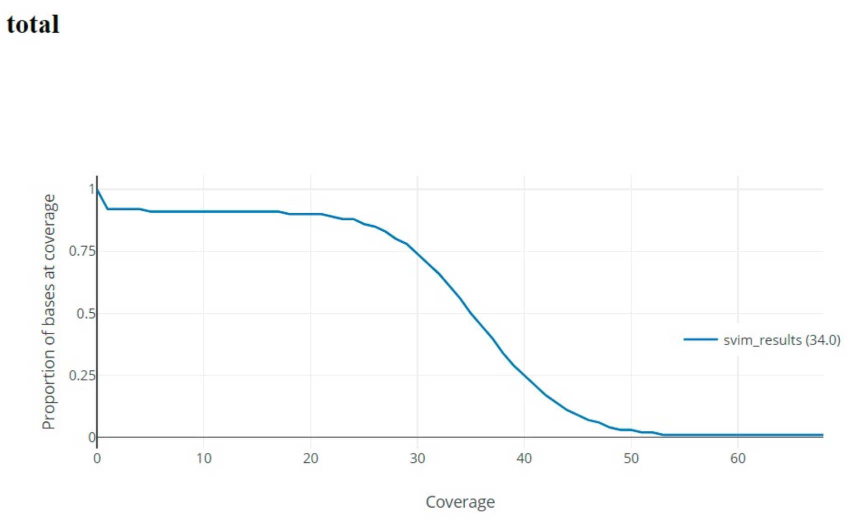

Figure 4.1: **Mosdepth total coverage for the NA12878 dataset using minimap2**. Please note that the svim_results was the directory flag. The total depth is roughly x34 coverage

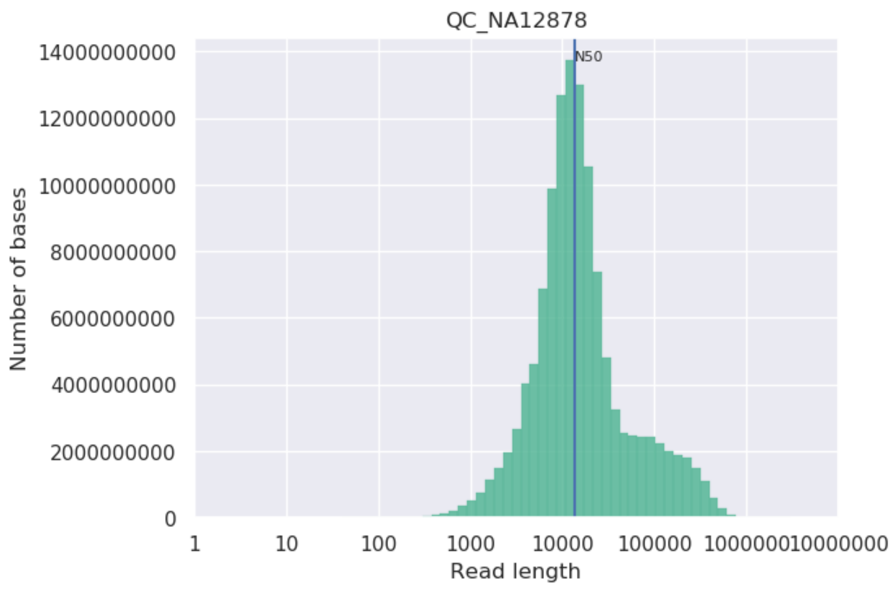

Figure 4.2: **Histogram of read length log transformed for the NA12878 dataset using minimap2**. This raw data incorporates 30x normal long reads with 5x of ultra-long read reads. The read length histogram has shifted slightly towards a N50 of 20k due to a positive skew produced by the impact of the ultra-long reads

case we can clearly detect a N50 20 kb for both *Minimap2* and *Ngmlr*. This result was expected as the effect of the ultra-long reads have shifted slightly the distribution towards larger read lengths *(Figure 4.2)*. The skewness can be observed for those reads whose length is >= 100 kb.

However *Minimap2* and *Ngmlr* are clearly noticeable when comparing their computation times. *Minimap2* was able to finish in less than a third of the time spent by *Ngmlr* (**Table 4.1**).

Aligner	Threads	s	h:m:s
Minimap2	16	84106.4999	23:21:46
Ngmlr	32	331998.7294	92:13:18

Table 4.1: **Benchmark for Minimap2 and Ngmlr**. Although Snakemake provides additional benchmarking parameters, only time in s (seconds) and h:m:s (hours:minutes:seconds) was kept. The number of threads provided was introduced manually.

4.2.1 Comparison between *Sniffles* and *Svim*

For the first comparison only default parameters was selected. Here we present two figures with 4 subplots each. The high confidence used the one for deletions, and the evaluation was performed on the DEL SVTYPE.

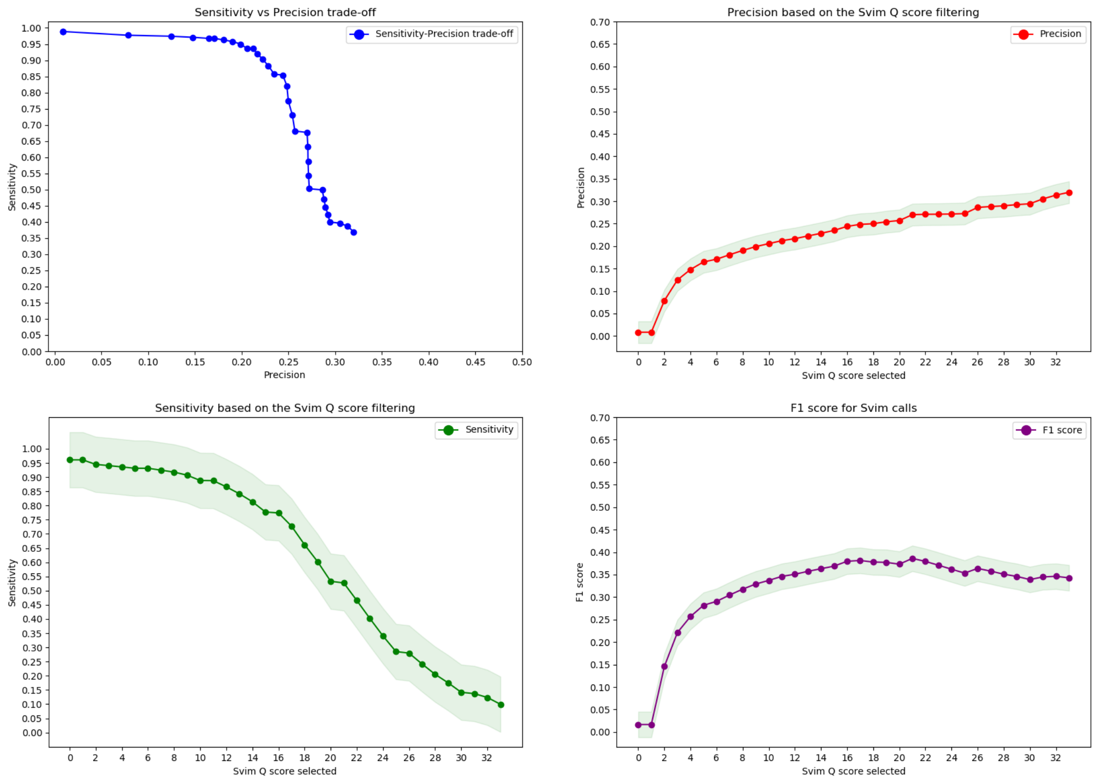

Figure 4.3: **Benchmarking: Svim DEL evaluation with default settings using minimap2**. The 95% confidence interval was plotted around the data points (in blue). The trade-off between sensitivity (recall) and precision is quite clear. Please note this figures were plotted with the benchmarking script, and as such a few differences in format are present with the final MASV plotting script.

For this comparison, we plotted the results for both *Svim* and *Sniffles* for 34 iterations of filtering on both Q score and Read support respectively. *Svim* results clearly show the trade-off between recall (sensitivity) and precision. It must be noted however that we are expecting a large number of false "False Positives" as we are using a limited high confidence dataset. However we can clearly see the that filtering allows to balance out the recall vs precision trade off, with an optimum Q score filtering value between 2 and

4 in which we would maintain a high recall 93% for an increase of 15% in accuracy.

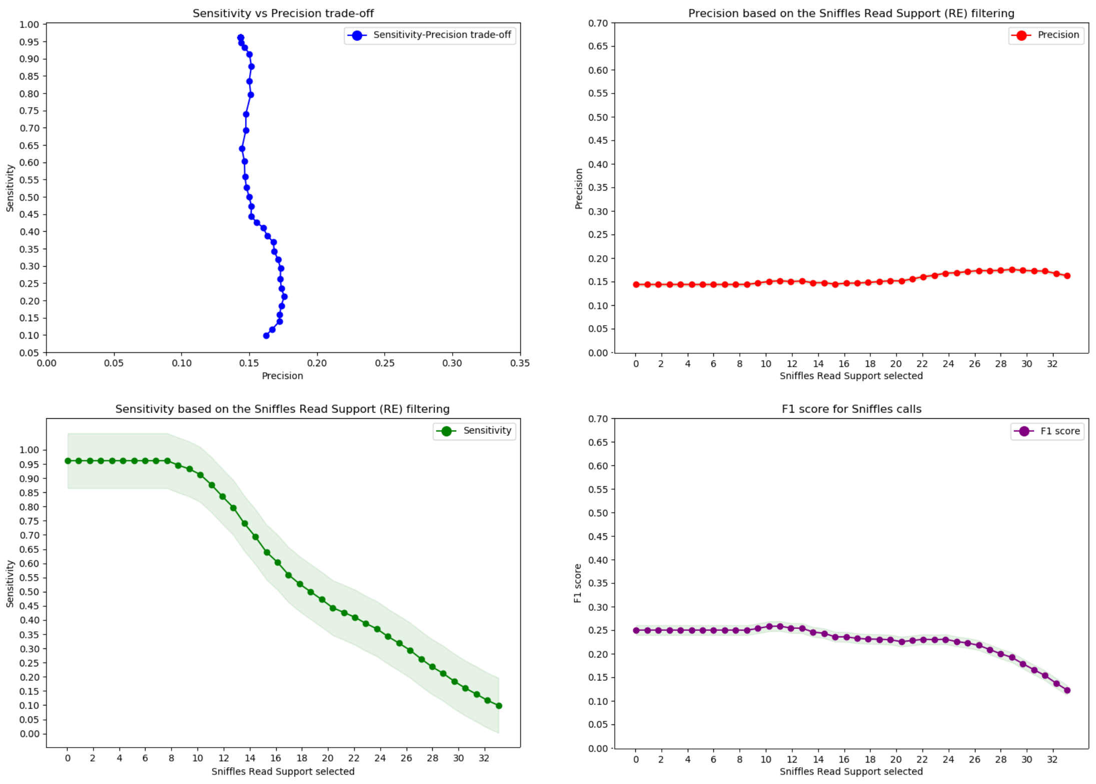

Figure 4.4: **Benchmarking: Sniffles DEL evaluation with default settings using minimap2**. The 95% confidence interval was plotted around the data points (in blue). Recall and precision is kept for the first 10 iterations because Sniffles performs a prefiltering step based on minimum read support set to 10 reads.

This however cannot be said for *Sniffles*. As the default settings allow a prefiltering SV with a read support < 10, the real trade-off starts above 10. However the gain in precision is minimal at best, while the cost for recall is really high.

For insertions, the results were significantly lower. Considering a really low effective size of 68 high confidence insertions, precision can be almost rejected for the benchmarking. It must be noted however two current limitations of the evaluation script.

I The average result and its standard deviation depend on the number of iterations used for the evaluation. Hence, a lower number of iterations (in other words, a less stringent filtering) would yield higher averages and lower standard deviations.

II It doesn't provide the filtering score for which the value is optimum. Right now it requires to manually check the visualization plots.

Below it is provided as a table the benchmarking on both callers with default parameters using *Minimap2* as the aligner (**Table 4.2**):

SVTYPECaller	Metric	Results at 0 filt	Average results	Std values	
DEL	Svim	Recall	98.91%	71.41%	22.84%
		Precision	0.0801 %	22.63 %	7.77%
		F1 score	1.66 %	31.93 %	8.81%
	Sniffles	Recall	96.30 %	60.22%	30.69%
		Precision	13.34%	15.96%	1.47%
		F1 score	23.43 %	21.99%	2.88%
INS	Svim	Recall	72.05%	32.41%	23.24%
		Precision	0.011%	0.35%	0.13%
		F1 score	0.036%	0.68%	0.24%
	Sniffles	Recall	57.35%	24.34%	29.87%
		Precision	0.029%	0.087%	0.007%
		F1 score	0.058%	0.15%	0.1%

Table 4.2: **Benchmark performed with default parameter for *Minimap2* alignments**. Recall is significantly lower than deletions, probably due to the small effective size of the high confidence dataset. Moreover, precision for INS is extremely low, provided the number of false positives (both real and "false") is huge: 68 real TP against an expected 18,919.

The results on default parameters when using *Ngmlr* were quite similar albeit significantly better for *Sniffles*. *Svim* results are mostly the same, with an small improvement on the insertion detection. *Sniffles* yielded better results for both INS and DEL as opposed to using *Minimap2*.

SVTYPE	Caller	Metric	Results at 0 filt	Average results	Std values
DEL	Svim	Recall	98.87%	69.69%	23.81%
		Precision	0.083 %	24.42 %	7.02%
		F1 score	1.84%	33.66 %	8.96%
	Sniffles	Recall	99.01 %	71 .12%	21.65%
		Precision	16.37%	25.49%	5.51%
		F1 score	27.01 %	31.71%	3.09%
INS	Svim	Recall	76.24%	34.06%	22.39%
		Precision	0.061 %	0.420 %	0.19%
		F1 score	0.094%	0.284 %	0.185%
	Sniffles	Recall	65.35%	31.34%	27.62%
		Precision	0.047%	0.205%	0.057%
		F1 score	0.068%	0.310%	0.140%

Table 4.3: **Benchmark for INS with default parameters using *Ngmlr***. *Sniffles*' performance has improved, specially for INS calls. SVIM performance is kept mostly the same with a slight improvement in INS detection.

For a proper benchmarking, more situations were considered. However, it must be said that the time spent on benchmarking was hampered as it has been discussed in the section **1.4 Limits and challenges** (pages 6-7).

As I invested more than a month trying to obtain the preprocessed reads through *Canu*, the number of additional benchmarking tests were quite limited. However, as expected several SV calling parameters had a great impact on the evaluation metrics such as **min_support_reads** (which performs a prefiltering on the calls based on Support Reads score) and **minimum_sv_length** for *Sniffles* and **minimum_sv_length** as well for *Svim*.

Selecting a predefined **min_support_reads** is equivalent to select a specific datapoint in the plot. Consequently, stringent values will yield lower recall and higher precision than relaxed values.

Modifying the **minimum_sv_length** will affect the recall and precision of the evaluation, although I wouldn't recommend it as the bias impact would largely impact the outcome of the evaluation. Consequently, the default parameters were kept for the experiments conducted, with the exception of **minimum_sv_length** which was adapted to 50bp to keep it up with the SV size convention.

5 Experiments conducted and results

In order to test MASV pipeline functionality for mis-assembly detection, two experiments were proposed:

I Evaluate the precision, recall and F1 with homozygous reads.

II Evaluate the precision, recall and F1 with heterozygous reads.

All the following experiments were performed using the final version of the MASV pipeline. Provided the time to spend to end this project was limited, the slow computation time far outweighs the improved results by using *Ngmlr*. Hence *Minimap2* was selected as the default aligner and all the reads were mapped against the unaltered hg19 reference.

The experiments were conducted with the following structure in mind:

I Run the *MASV_pipeline.smk* selecting as the target the rule all. 16 threads were provided and it was allowed a maximum parallelization of up to 8 threads.

II After the first run was performed, modify config adding a new high confidence dataset for the evaluation script and manually edit the *bedtools_ eval.smk* file, specifically the feature type to be evaluated.

III Repeat step II for each feature type that is going to be tested.

IV Combine all the metric results in a large table file.

5.1 Evaluating homozygous reads

Hereby we present the results for the homozygous reads evaluation (Table 5.1).

Both *Sniffles* and *Svim* are quite precise at detecting the four SV types with the exception of cut & paste insertions, which could not be detected by *Sniffles*. For both, the recall is quite good, although *Sniffles* outperforms *Svim* in tandem duplications and

SVTYPE	Caller	Metric	Results at 0 filt	Average results	Std values
DEL	Svim	Recall	99.50%	98.70%	0.5783%
		Precision	48.38%	64.00%	5.22%
		F1 score	65.10%	77.50%	4.14%
	Sniffles	Recall	99.50%	99.50%	1.11e-16%
		Precision	68.58%	68.47%	2.39e-03%
		F1 score	81.19%	81.12%	1.68e-03%
INS	Svim	Recall	95.00%	73.70%	13.80%
		Precision	97.93%	98.51%	0.326%
		F1 score	96.44%	83.97%	9.37%
	Sniffles	Recall	Null	Null	Null
		Precision	Null	Null	Null
		F1 score	Null	Null	Null
INV	Svim	Recall	95.00%	94.00%	0.447%
		Precision	97.95%	98.89%	0.215%
		F1 score	96.45%	96.38%	0.204%
	Sniffles	Recall	94.00%	94.00%	1.11e-16%
		Precision	94.94%	95.99%	1.56e-02%
		F1 score	94.47%	94.97%	7.57e-03%
DUP:TAND	Svim	Recall	83.50%	73.70%	4.87%
		Precision	98.33%	99.19%	0.3103%
		F1 score	90.31%	84.47%	0.309%
	Sniffles	Recall	99.00%	98.87%	0.216%
		Precision	98.93%	98.89%	0.042%
		F1 score	98.96%	98.88%	0.127%

Table 5.1: **Evaluation on the homozygous x53 simulated reads during 20 iterations**. With the exception of DEL, both callers are quite precise independently of the SVTYPE. It must be noted that *Sniffles* is not able to capture cut & paste insertions. However this limitation is directly linked with how *Sniffles* do not allow an insertion region (key for our overlaps approach)around the breakpoints, which *Svim* does.

deletions. The precision is quite similar between the two although *Svim* tends to be more precise than *Sniffles* with the exception of deletions.

Due to how *Svim* classifies cut & paste insertions, they could only be detected for the feature DUP_INT (Interspersed Duplication). Hence, it is key to understand the limits of each variant calling software in order to adjust the analysis before interpreting the results.

It seems that *Svim* is more dependent on Q score filtering than *Sniffles* regarding Read Support (RE) filtering. It must be said however that as default parameters were for the most part kept, *Sniffles* had already undergone a prefiltering step. It might already affected some of the *Sniffles* results, nonetheless we can observe that the impact of RE filtering isn't as high as for Q score.

Below we are going to provide as an example the plot comparison between *Svim* (*Figure 5.3*) and Sniffles (*Figure 5.4*) for deletions. The rest of the plots can be found in the Annex B.

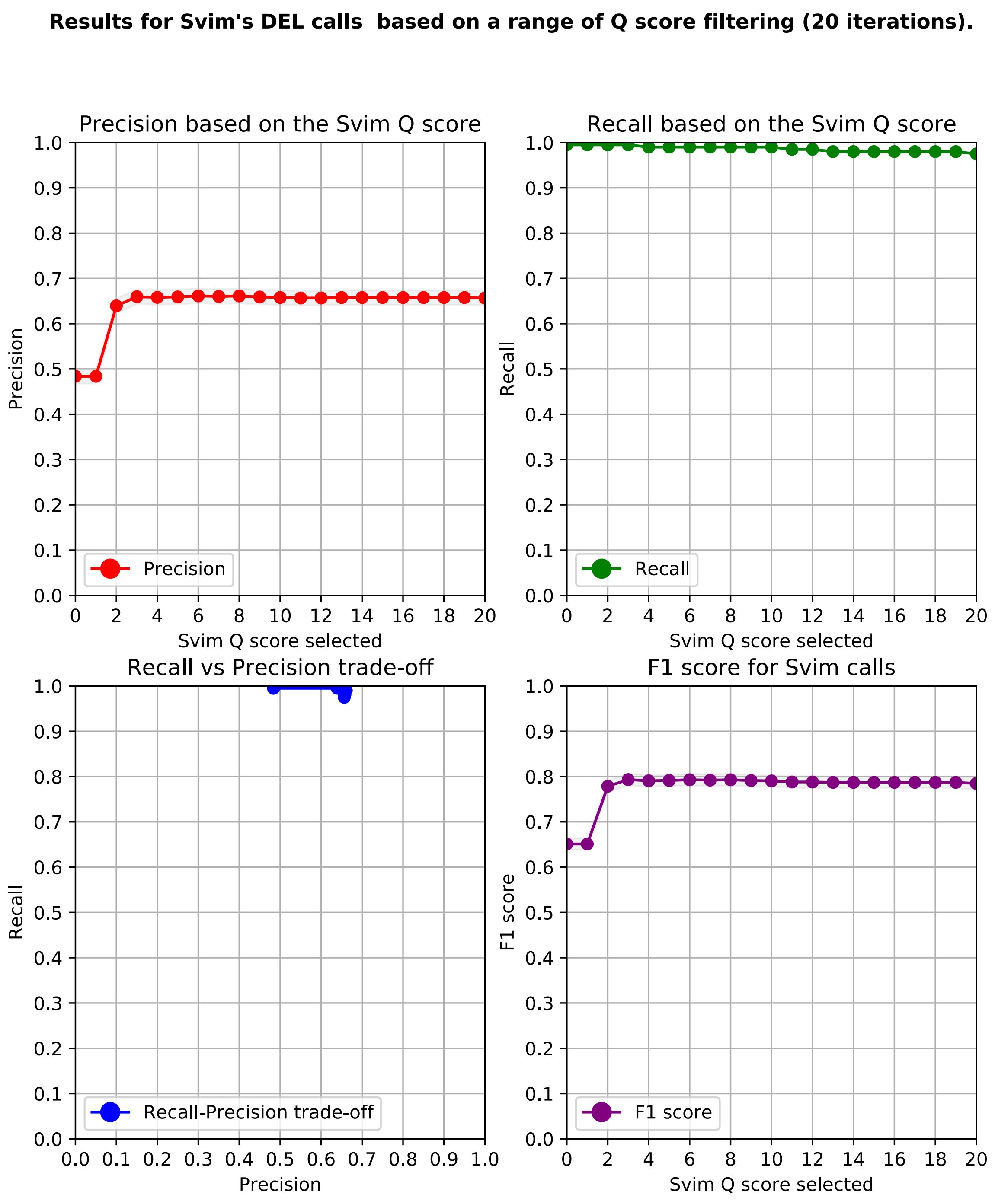

Figure 5.1: ***Svim* evaluation of homozygous x52 simulated read with default settings for DEL**. There is a huge tradeoff between recall and precision where for an almost neglectable loss of recall there is a gain of up to 15% accuracy gain when using a filtering score >= 2. However with a Q score filtering of 4, we are able to reach the optimum filtering to maximize both precision and recall as it is supported by the F1 plot.

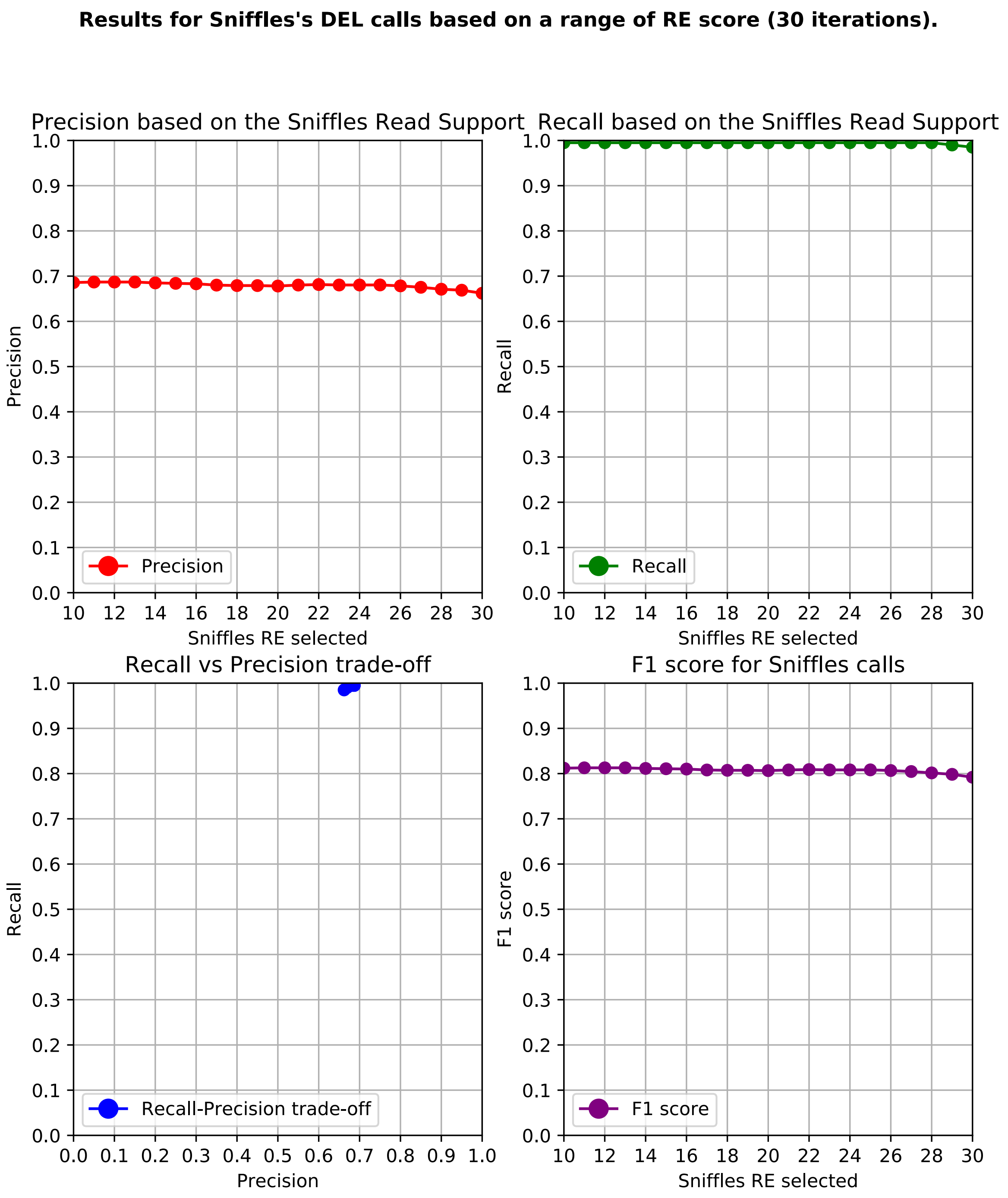

Figure 5.2: ***Sniffles* DEL evaluation of homozygous x52 simulated read with default settings for DEL**. We can clearly observe a slight decrease in recall. At the same time, a decrease albeit a bit more pronounced can be observed for precision too. I would not recommend a more stringent filtering, as we are able to keep the optimum filtering with just the default parameter for *Sniffles* prefiltering. Please note that although the number of iterations is 30, as the default prefiltering minimum RE is 10, the first 10 are skipped so in essence is just 20 iterations (starting from 10).

It must be noted however that in order to determine the best filtering cutoff, we need to take into account the full picture, in other words all four SV evaluation.

For *Svim* the final optimum filtering score would have been between 2 and 4. INS detection optimum value is at 2(*Figure B.6*), while for the other ones is closer to 4(*Figures 5.3, B.7* and *B.8*).

Although the base prefiltering score of 10 yields quite good results for *Sniffles*, it would be recommended a filtering score of 16 provided it is the optimum value for INV (*Figure B.9*). Recall and precision are quite robust for DEL and DUP:TANDEM calls (*Figures 5.4* and *B.8*) hence there isn't any major trade-off for selecting a more stringent filtering score.

5.2 Evaluating heterozygous reads

However most of the time we are not sure whether the organism we have sequenced and assembled is heterozygous or not. It is key then to evaluate how both SV callers perform for heterozygous data as we could be unable to detect some mis-assemblies or detect some of them incorrectly. In order to test this precise scenario, we perform an analysis on the simulated heterozygous reads and these are the results obtained:

SVTYPE	Caller	Metric	Results at 0 filt	Average results	Std values
DEL	Svim	Recall	99.00%	98.20%	0.865%
		Precision	35.60%	61.66%	9.69%
		F1 score	52.36%	75.16%	8.35%
	Sniffles	Recall	99.50%	85.33%	27.06%
		Precision	69.60%	69.76%	2.57%
		F1 score	81.90%	73.89%	18.70%
INS	Svim	Recall	91.00%	70.85%	16.02%
		Precision	98.91%	98.53%	0.357%
		F1 score	94.79%	81.40%	11.30%
	Sniffles	Recall	Null	Null	Null
		Precision	Null	Null	Null
		F1 score	Null	Null	Null
INV	Svim	Recall	94.00%	93.09%	1.06%
		Precision	97.93%	98.88%	0.213%
		F1 score	95.92%	95.90%	0.564%
	Sniffles	Recall	94.00%	84.80%	12.43%
		Precision	98.94%	98.80%	0.217%
		F1 score	96.41%	90.74%	8.01%
DUP:TAND	Svim	Recall	83.50%	74.50%	5.12%
		Precision	97.17%	98.78%	0.76%
		F1 score	89.82%	84.82%	0.303%
	Sniffles	Recall	98.00%	92.03%	11.82%
		Precision	98.85%	98.79%	0.182%
		F1 score	98.42%	94.83%	7.30%

Table 5.2: **Evaluation on the heterozygous x52 simulated reads during 20 iterations**. Both performed slightly worse than during the evaluation of homozygous reads., however the performance drop is quite small. Hence both *Sniffles* and *Svim* are quite robust to heterozygosity.

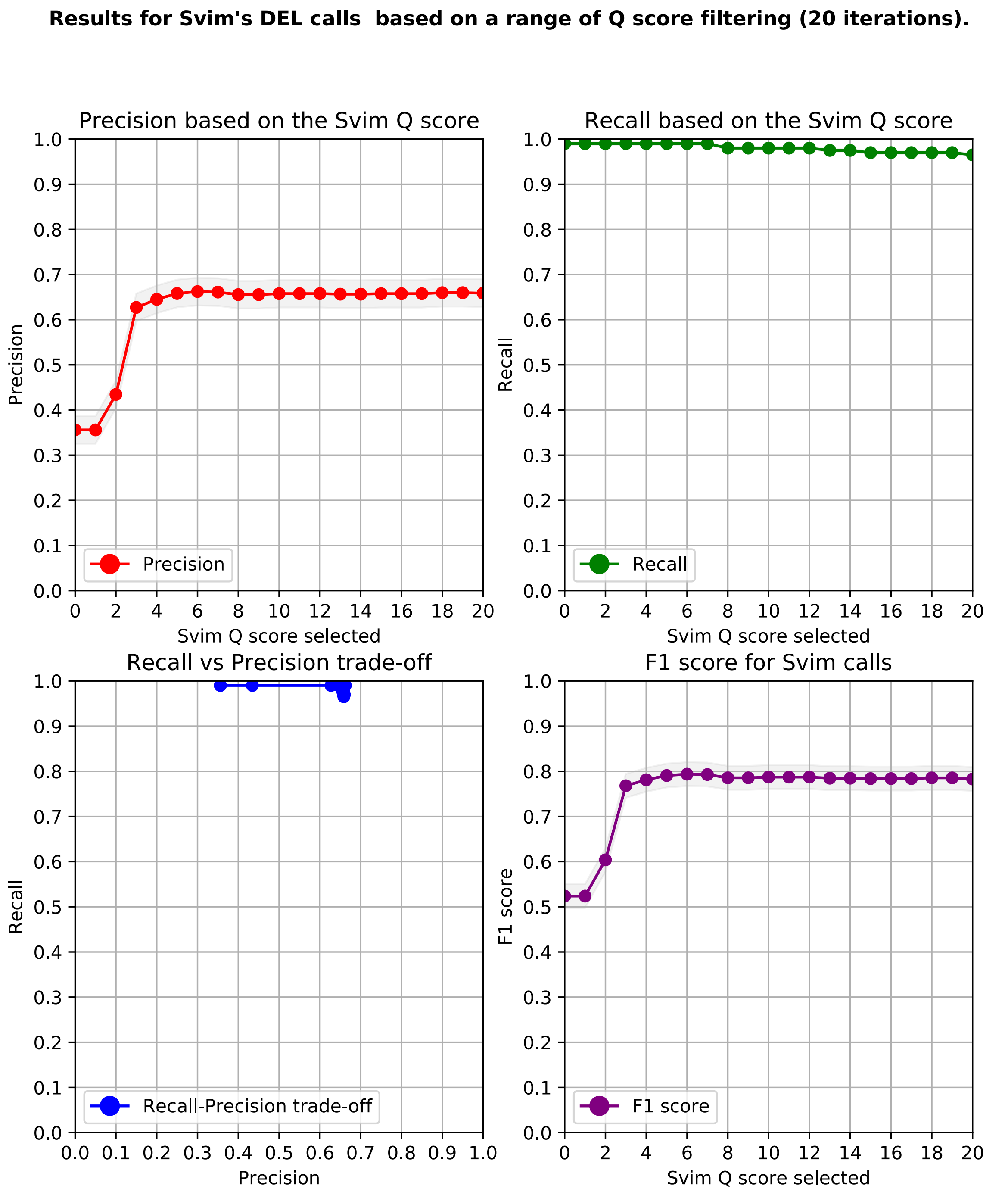

Figure 5.3: **Svim evaluation of heterozygous x52 simulated reads with default settings for DEL**. We are able to observe again a huge trade-off between recall and precision with a boost over 15% accuracy gain when using a filtering score >= 2. There isn't any major difference with the results of homozygous DEL.

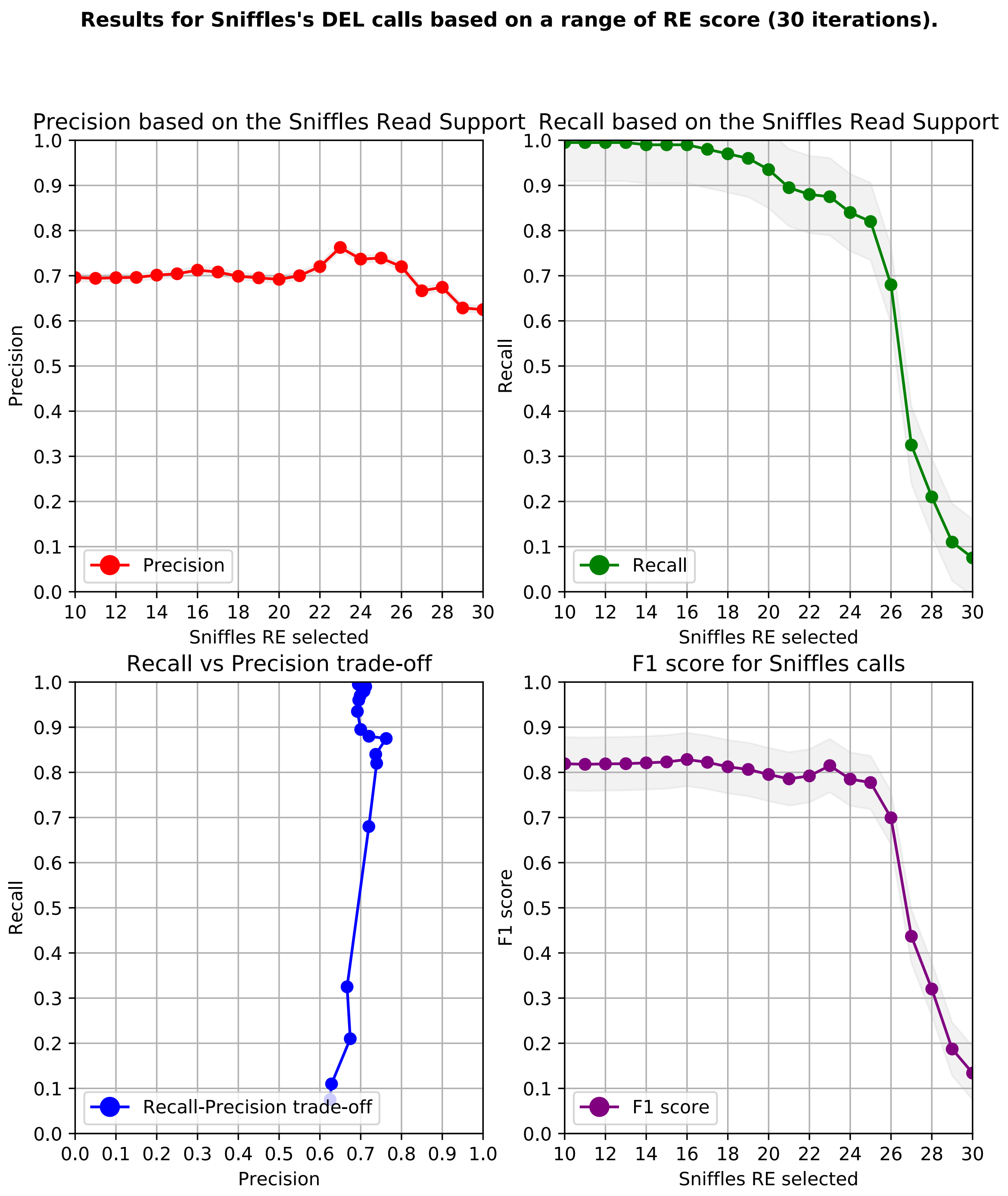

Figure 5.4: **Sniffles evaluation of heterozygous x52 simulated reads with default settings using for DEL**. As well as with the Svim evaluation before, there isn't any major difference with the results of homozygous DEL.

The weird behaviour in the *Sniffles* plots can be explained by the composition of the dataset. The heterozygous dataset was created by merging a 26x rearranged reads with 26x normal reads. Hence we would expect to have maximum read support around 26 reads.

This might have exclusively affected the *Sniffles* results, as the iterative loop reached a filtering value of 30. The results for filtering values above 26 will have reduced the averages of the metrics tested, as well as affecting the standard deviation. However, as we can clearly see from the plots, the tendency is well kept for the most part.

In conclusion: Overall both are quite robust. Albeit performing slightly worse than for homozygous reads, both were able to accurately pinpoint most of the time the correct variants after some filtering. Interestingly the filtering thresholds set for homozygous reads would have sufficed for both *Svim* and *Sniffles*.

6 Discussion and conclusions

6.1 Discussion and future work

During the benchmarking process I was wondering which of the two metrics should be prioritized. If set on maximizing recall, the algorithm should be able to detect most if not all of the possible mis-assemblies, but not accurately. Whereas maximizing precision would detect few mis-assemblies, but all of the ones detected would have been real errors of the assembly process. I would recommend however to try to maximize precision above recall (but keeping recall at a reasonable level, high enough to fully exploit this pipeline). For future implementations such as a script able to re-assemble and correct the mis-assemblies based on the information gathered by MASV pipeline, it would be far more critical to correct the real mis-assemblies that are detected that be able to detect all real mis-assemblies but have the uncertainty of introducing new rearrangements based on false positives.

Both *Svim* and *Sniffles* reported similar results for both heterozygous and homozygous reads. *Sniffles* has proven to be the most robust caller out of the two although the maximum values were more often found in *Svim*. It must be noted however that *Sniffles* currently is not able to detect cut and paste insertions. At the very best it is able to provide break points' coordinates. Unfortunately, the way the evaluation was designed relying on overlaps, it was not possible to take individual breakpoints into account. Whereas *Svim* was able to identify these complex events and incorporate into the calling. Thus, it would be really interesting to merge the calls from both SV predictors. Hereby I propose two approaches:

I Merging and keeping only the overlaps between *Sniffles* and *Svim*. By doing that we would generate a **high confidence consensus call** (HCCC). Though complex, with a merging algorithm that would account for a merged Q score/Read Support to even further weigh the likeliness of that HCCC to be a real mis-arrangement.

II Merging both VCF files, without any special weighting on the overlaps between the two. Using this approach, we would be able to capture SV that wouldn't be captured by using only one or the other SV predictor.

This feature alone might prove to be invaluable to reduce the impact of spurious calls on complex genome assemblies, such as the ones from the plant kingdom[10] or cancer genomes.

In addition, future work would focus development on an algorithm for reducing the number of mis-assemblies in draft genomes by rearranging the target assembly according to the calls made by the MASV pipeline. If the SV can effectively represent mis-assemblies, then the logical step would be to incorporate the information from the SV calls to correct the mis-assemblies. Perhaps it could draw inspiration from the **GATK Alternate Reference Maker algorithm** or something similar. However, it is easier said than done and it probably warrants a project in itself.

The conducted experiments perhaps required further exploration and additional ideas. As such, some of this project's caveats were:

- The experiments were conducted on a simulated genome, which may be unable to capture the complexities of real genomes. It would have been interesting to test it on a real assemble from which we knew beforehand where were the mis-assemblies located.
- A small size genome (only chr21 and chr22 from the hg19 reference genome) was rearranged. It would have been desirable to test them on a full rearranged genome.
- It was used a fairly good coverage for the experiments (x52). It would have been interesting using lower coverage simulations as it would have provided us a better understanding of the coverage dependant limitations of these SV caller.

Finally, we were not able to test everything. A few ideas were left out due to responsible time and focus allocation. Trying to work out the preprocessing step for the benchmarking consumed an important amount of time and resources. Perhaps in the future this idea could be revisited.

6.2 Conclusions

De novo genome assembly has been forever changed with the addition of third generation sequencing technologies. Longer reads facilitate the complex task of assembling repetitive regions which are prone to errors. New and precise long read structural variant callers have been developed as well. We originally hypothesised that SV calling might help to detect real mis-assemblies. To test this idea, we developed the MASV pipeline.

The MASV pipeline was designed through Snakemake. Snakemake is a flexible and intuitive tool, easy to use but hard to master. It has however plenty of advantageous perks making it one of the most popular choices for workflows design. MASV pipeline has proven its utility, being able to accommodate several modular steps into a single easy-to-follow workflow. Some of the conscious decision regarding the pipeline design have improved reproducibility and scalability exceptionally well, specially the usage of conda environments. In addition to just adding software dependencies to the pipeline workflow, it has been further enhanced through the use of custom scripts to provide additional utility (a script to generate on-demand config files) and functionality (reformat VCFs and evaluation metrics).

After benchmarking the pipeline with a fair share of difficulties and challenges, two experiments were proposed: test the SV detection capabilities for homozygous and heterozygous calls. The high precision and recall obtained through these two experiments suggest that MASV could be used effectively for mis-assembly detection as well as for SV detection. Further work is required but the future of MASV is promising.

Glossary of Terms and Acronyms

- **TGS**: Third Generation Sequencing
- **SV**: Structural Variant
- **VCF**: Variant Call File
- **CNAG**: Centro Nacional de Análisis Genómico
- **NGS**: Next Generation Sequencing
- **SMRT**: Single Molecule, Real-Time
- **ONT**: Oxford Nanopore Technologies
- **CCS**: Circular Consensus Sequence
- **BAM**: Binary Alignment Map
- **JSON**: JavaScript Object Notation
- **YAML**: YAML Ain't Markup Language
- **DEL**: Deletions
- **INS**: Insertions
- **INV**: Inversions
- **DUP:TANDEM**: Tandem Duplications
- **DUP:INT**: Interspersed Duplications
- **NAHR**: Non-Allelic Homologous Recombination
- **NHR**: Non Homologous Recombination
- **VNTR**: Variable Number of Tandem Repeats
- **LINE**: Long Interspersed Nuclear Element
- **SINE**: Short Interspersed Nuclear Element
- **UCSC**: University of California Santa Cruz
- **BED**: Browser Extensible Data
- **HGSVC**: Human Genome Structural Variation Consortium
- **GIAB**: Genome In A Bottle consortium

Bibliography

[1] Fritz J. Sedlazeck, Hayan Lee, Charlotte A. Darby, and Michael C. Schatz. Piercing the dark matter: bioinformatics of long-range sequencing and mapping. *Nature Reviews Genetics*, 19(6):329–346, 2018.

[2] Luis Acuña-Amador, Aline Primot, Edouard Cadieu, Alain Roulet, and Frédérique Barloy-Hubler. Genomic repeats, misassembly and reannotation: a case study with long-read resequencing of Porphyromonas gingivalis reference strains. *BMC Genomics*, 19(1), 2018.

[3] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, Oct 2018.

[4] Fritz J. Sedlazeck, Philipp Rescheneder, Moritz Smolka, Han Fang, Maria Nattestad, Arndt Von Haeseler, and Michael C. Schatz. Accurate detection of complex structural variations using single-molecule sequencing. *Nature Methods*, 15(6):461–468, 2018.

[5] David Heller and Martin Vingron. SVIM: structural variant identification using mapped long reads. *Bioinformatics*, 35(17):2907–2915, 2019.

[6] Miten Jain, 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, and et al. Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nature Biotechnology*, 36(4):338–345, 2018.

[7] Hemang Parikh, Marghoob Mohiyuddin, Hugo Y. K. Lam, Hariharan Iyer, Desu Chen, Mark Pratt, Gabor Bartha, Noah Spies, Wolfgang Losert, Justin M. Zook, and et al. SVclassify: a method to establish benchmark structural variant calls. *BMC Genomics*, 17(1), 2016.

[8] C. Bartenhagen and M. Dugas. RSVSim: an R/Bioconductor package for the simulation of structural variations. *Bioinformatics*, 29(13):1679–1681, 2013.

[9] Bianca K. Stöcker, Johannes Köster, and Sven Rahmann. SimLoRD: Simulation of Long Read Data. *Bioinformatics*, 32(17):2704–2706, Oct 2016.

[10] Manuel Gonzalo Claros, Rocío Bautista, Darío Guerrero-Fernández, Hicham Benzerki, Pedro Seoane, and Noé Fernández-Pozo. Why assembling plant genome sequences is so challenging. *Biology*, 1(2):439–459, 2012.

[11] Konstantinos Mavromatis, Miriam L. Land, Thomas S. Brettin, Daniel J. Quest, Alex Copeland, Alicia Clum, Lynne Goodwin, Tanja Woyke, Alla Lapidus, Hans Peter Klenk, and et al. The fast changing landscape of sequencing technologies and their impact on microbial genome assemblies and annotation. *PLoS ONE*, 7(12), Dec 2012.

[12] Adam M Phillippy, Michael C Schatz, and Mihai Pop. Genome assembly forensics: finding the elusive mis-assembly. *Genome Biology*, 9(3), 2008.

[13] Matthew Pendleton, Robert Sebra, Andy Wing Chun Pang, Ajay Ummat, Oscar Franzen, Tobias Rausch, Adrian M Stütz, William Stedman, Thomas Anantharaman, Alex Hastie, and et al. Assembly and diploid architecture of an individual human genome via single-molecule technologies. *Nature Methods*, 12(8):780–786, 2015.

[14] John Huddleston, Mark J.p. Chaisson, Karyn Meltz Steinberg, Wes Warren, Kendra Hoekzema, David Gordon, Tina A. Graves-Lindsay, Katherine M. Munson, Zev N. Kronenberg, Laura Vives, and et al. Discovery and genotyping of structural variation from long-read haploid genome sequence data. *Genome Research*, 27(5):677–685, 2016.

[15] Jonas Korlach, Keith P. Bjornson, Bidhan P. Chaudhuri, Ronald L. Cicero, Benjamin A. Flusberg, Jeremy J. Gray, David Holden, Ravi Saxena, Jeffrey Wegener, Stephen W. Turner, and et al. Real-Time DNA Sequencing from Single Polymerase Molecules. *Methods in Enzymology Single Molecule Tools: Fluorescence Based Approaches, Part A*, page 431–455, 2010.

[16] Miten Jain, Ian T Fiddes, Karen H Miga, Hugh E Olsen, Benedict Paten, and Mark Akeson. Improved data analysis for the MinION nanopore sequencer. *Nature Methods*, 12(4):351–356, 2015.

[17] Aaron M. Wenger, Paul Peluso, William J. Rowell, Pi-Chuan Chang, Richard J. Hall, Gregory T. Concepcion, Jana Ebler, Arkarachai Fungtammasan, Alexey Kolesnikov, Nathan D. Olson, and et al. Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome. *Nature Biotechnology*, 37(10):1155–1162, Dec 2019.

[18] Carlos Lannoy, Dick Ridder, and Judith Risse. The long reads ahead: De novo genome assembly using the MinION. *F1000Research*, 6:1083, 12 2017.

[19] Andrea D. Tyler, Laura Mataseje, Chantel J. Urfano, Lisa Schmidt, Kym S. Antonation, Michael R. Mulvey, and Cindi R. Corbett. Evaluation of Oxford Nanopore's MinION Sequencing Device for Microbial Whole Genome Sequencing Applications. *Scientific Reports*, 8(1), 2018.

[20] Shuhua Fu, Anqi Wang, and Kin Fai Au. A comparative evaluation of hybrid error correction methods for error-prone long reads. *Genome Biology*, 20(1), Apr 2019.

[21] T. Laver, J. Harrison, P.a. O'Neill, K. Moore, A. Farbos, K. Paszkiewicz, and D.j. Studholme. Assessing the performance of the Oxford Nanopore Technologies MinION. *Biomolecular Detection and Quantification*, 3:1–8, 2015.

[22] Senne Cornelis. *Forensic Lab-on-a-Chip DNA analysis*. PhD thesis, 01 2019.

[23] Mark J. P. Chaisson, John Huddleston, Megan Y. Dennis, Peter H. Sudmant, Maika Malig, Fereydoun Hormozdiari, Francesca Antonacci, Urvashi Surti, Richard Sandstrom, Matthew Boitano, and et al. Resolving the complexity of the human genome using single-molecule sequencing. *Nature*, 517(7536):608–611, Oct 2014.

[24] Haowen Zhang, Chirag Jain, and Srinivas Aluru. A comprehensive evaluation of long read error correction methods. 2019.

[25] Sergey Koren, Brian P. Walenz, Konstantin Berlin, Jason R. Miller, Nicholas H. Bergman, and Adam M. Phillippy. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome Research*, 27(5):722–736, 2017.

[26] J. Koster and S. Rahmann. Snakemake: a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, 2012.

[27] Brent S Pedersen and Aaron R Quinlan. Mosdepth: quick coverage calculation for genomes and exomes. *Bioinformatics*, 34(5):867–868, 2017.

[28] Justin M. Zook, Jennifer Mcdaniel, Nathan D. Olson, Justin Wagner, Hemang Parikh, Haynes Heaton, Sean A. Irvine, Len Trigg, Rebecca Truty, Cory Y. Mclean, and et al. An open resource for accurately benchmarking small variant and reference calls. *Nature Biotechnology*, 37(5):561–566, Jan 2019.

[29] Ryan E. Mills, others, and 1000 Genomes Project. Mapping copy number variation by population-scale genome sequencing. *Nature*, 470(7332):59–65, 2011.

[30] Andy Wing Chun Pang, Ohsuke Migita, Jeffrey R. Macdonald, Lars Feuk, and Stephen W. Scherer. Mechanisms of Formation of Structural Variation in a Fully Sequenced Human Genome. *Human Mutation*, 34(2):345–354, 2012.

[31] Z. Ou, P. Stankiewicz, Z. Xia, A. M. Breman, B. Dawson, J. Wiszniewska, P. Szafranski, M. L. Cooper, M. Rao, L. Shao, and et al. Observation and prediction of recurrent human translocations mediated by NAHR between nonhomologous chromosomes. *Genome Research*, 21(1):33–46, Jan 2011.

[32] Mark J.P. et al. Chaisson. Multi-platform discovery of haplotype-resolved structural variation in human genomes. *Nature Communications*, 2019.

A
User manual

The MASV pipeline is located at https://github.com/Dfupa/MASV-pipeline. All the available code has been uploaded in that repository. Below we provide a quickstart

A.0.1 Getting Started

First it is required to pre-install conda with either anaconda or miniconda3.

anaconda - please follow these instructions: https://docs.anaconda.com/anaconda/install/

Alternatively, you can install:

miniconda3 - please follow these instructions: https://conda.io/projects/conda/en/latest/user-guide/install/index.html

A.0.2 Pipeline Installation

After installing conda, download and install the pipeline

```
# First clone the MASV repository
$ git clone https://github.com/Dfupa/MASV-pipeline.git
# Change to MASV directory
$ cd MASV-pipeline/
# Set the conda environment with all the necessary dependencies
$ conda env create -f MASV_pipeline.yml
# Activate the conda environment
$ conda activate MASV_pipeline
```

A.0.3 Input

This pipline uses as an input a config file located in the MASV_pipeline/lib/config/ directory. The user can find in that directoy a script called MASV_get_config.py which will build a config file with the parameters provided by the user.

More information is provided by using the help parameter of MASV_get_config.py

```
# Always assuming the MASV_pipeline environment is active
$ python3 lib/config/MASV_get_config.py -h
```

A.0.4 Target rules

The target rules currently available to use are:

- rule **all**: Outputs both *Svim*(https://github.com/eldariont/svim) and *Sniffles*(https://github.com/fritzsedlazeck/Sniffles) filtered calls, as well alignment QC and evaluation output (only if a truth dataset has been provided to the config file).
- rule **mapping_only**: Outputs only the BAM alignment between the reads and the reference.
- rule **sniffles**: Outputs *Sniffles* filtered calls.
- rule **svim**: Outputs *Svim* filtered calls.
- rule **eval_sniffles**: Outputs evaluation output for *Sniffles* only.
- rule **eval_svim**: Outputs evaluation output for *Svim* only.
- rule **sanity_check**: Outputs alignment QC based on *mosdepth*(https://github.com/brentp/mosdepth) and *Nanoplot*(https://github.com/wdecoster/NanoPlot).

A.0.5 How to run Snakemake

In order to run the MASV pipeline just use the following command

```
# Always assuming the MASV_pipeline environment is active
$ snakemake -s MASV_pipeline.smk -r all -j 16 -n
```

The -n option is for a dry run, avoding the execution of any job yet. -j is the number of threads provided to the pipeline. In this specific example, we selected the rule all using the parameter -r. More information regarding Snakemake and its commands can be found through Snakemake documentation: https://snakemake.readthedocs.io/en/stable/index.html.

A.0.6 Important considerations

- Right now the pipeline is able to handle .fastq and .fastq.gz. Please keep this in mind when providing your own reads.
- Right now the pipeline uses either *Minimap2*(https://github.com/lh3/minimap2) or *Ngmlr*(https://github.com/philres/ngmlr) aligner, but not both.
- The evaluation script requires one specific SV feature (default is DEL) and a truth dataset for tha feature types. The user must manually edit the *bedtools_eval.smk* file in lib/rules/ and account for the feature change.
- The pipeline was developed and test in Unix like environments. As such, functionality is not assured when used from Windows or Mac.

B
Supplementary material

```
> sessionInfo()
R version 3.4.4 (2018-03-15)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 18.04.2 LTS

Matrix products: default
BLAS: /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.7.1
LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.7.1

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C               LC_TIME=es_ES.UTF-8        LC
     _COLLATE=en_US.UTF-8     LC_MONETARY=es_ES.UTF-8
 [6] LC_MESSAGES=en_US.UTF-8    LC_PAPER=es_ES.UTF-8       LC_NAME=C                  LC
     _ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=es_ES.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats4    parallel  stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
 [1] RSVSim_1.18.0                          BSgenome.Hsapiens.UCSC.hg19.masked_1.3.99
      BSgenome.Hsapiens.UCSC.hg19_1.4.0
 [4] BSgenome_1.46.0                        rtracklayer_1.38.3
                           Biostrings_2.46.0
 [7] XVector_0.18.0                         GenomicRanges_1.30.3
                         GenomeInfoDb_1.14.0
[10] IRanges_2.12.0                         S4Vectors_0.16.0
                            BiocGenerics_0.24.0

loaded via a namespace (and not attached):
 [1] zlibbioc_1.24.0            GenomicAlignments_1.14.2   BiocParallel_1.12.0
     lattice_0.20-35            hwriter_1.3.2
 [6] tools_3.4.4                SummarizedExperiment_1.8.1 grid_3.4.4
     Biobase_2.38.0             latticeExtra_0.6-28
[11] matrixStats_0.55.0         Matrix_1.2-12              GenomeInfoDbData_1.0.0
    RColorBrewer_1.1-2        bitops_1.0-6
[16] RCurl_1.95-4.12            DelayedArray_0.4.1         compiler_3.4.4
    Rsamtools_1.30.0          ShortRead_1.36.1
[21] XML_3.98-1.20
```

Code B.1: R session info for the RSVsim usage.

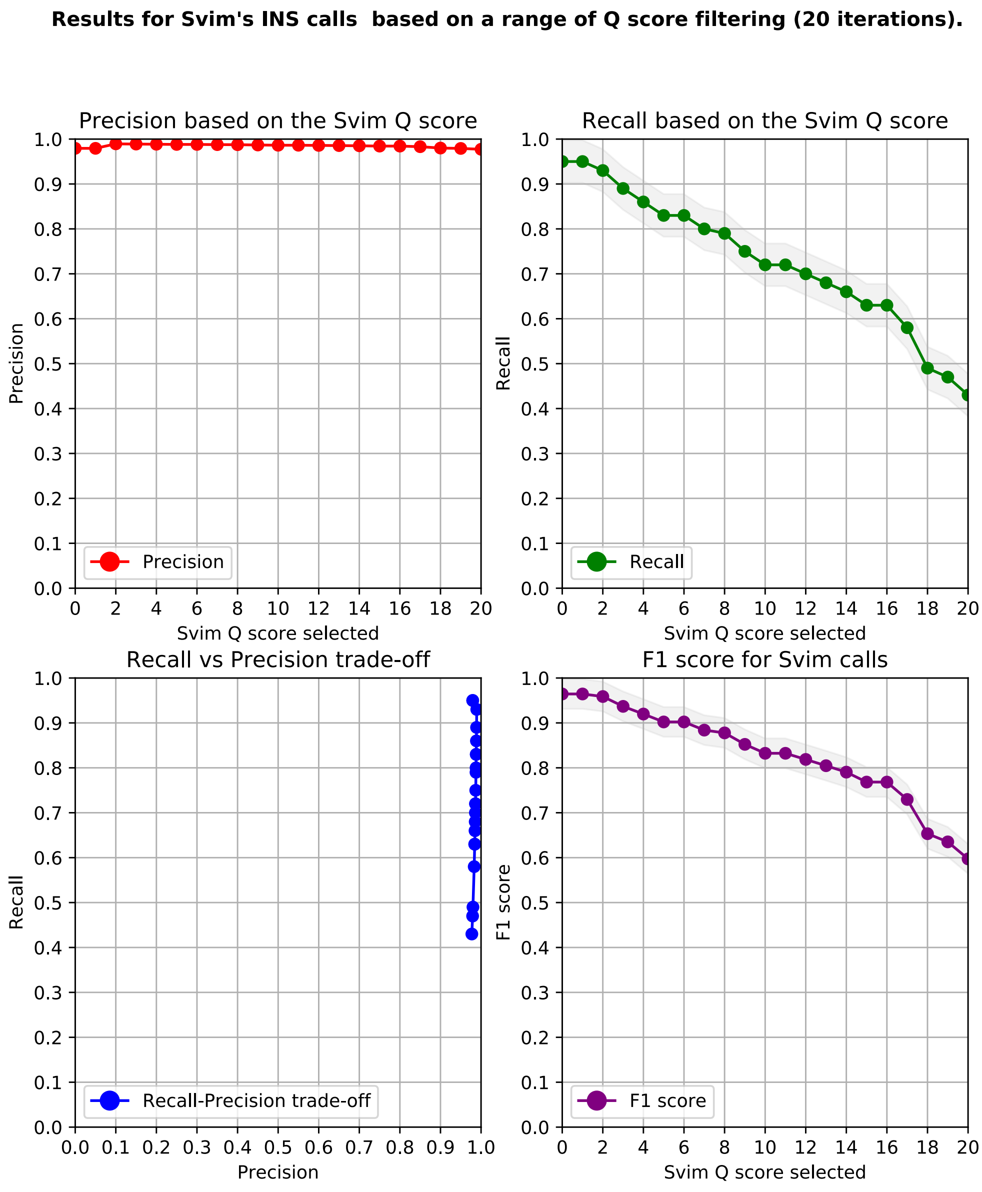

Figure B.1: **Supplementary plot: *Svim* evaluation of homozygous x52 simulated reads with default settings for INS**. The optimum Q score filtering value seems to be located at 3, where the precision is at its maximum and recall starts to significantly drop.

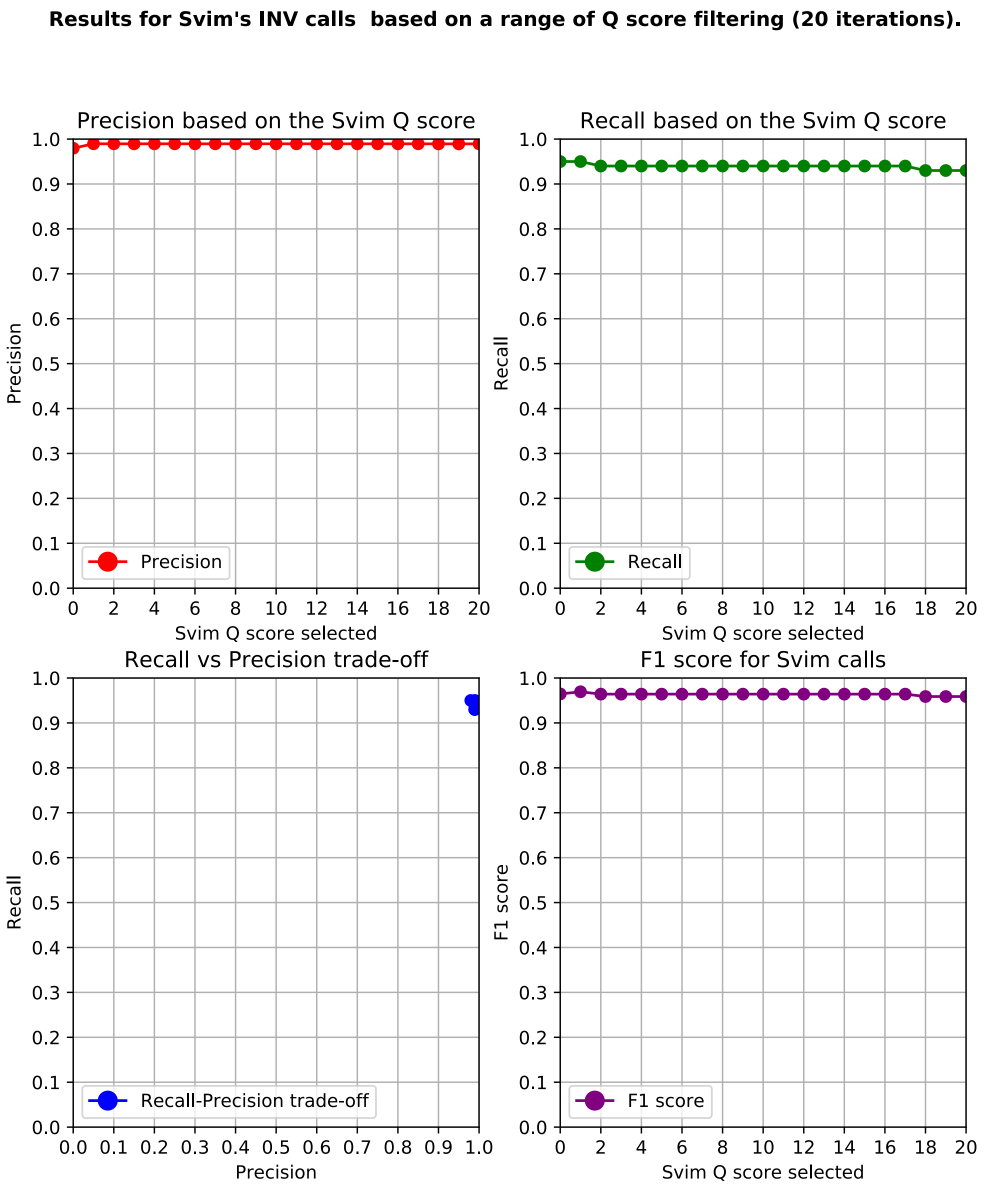

Figure B.2: **Supplementary plot: *Svim* evaluation of homozygous x52 simulated reads with default settings for INV.** The optimum Q score filtering value is clearly located at 2, although the performance is quite robust for all the filtering values.

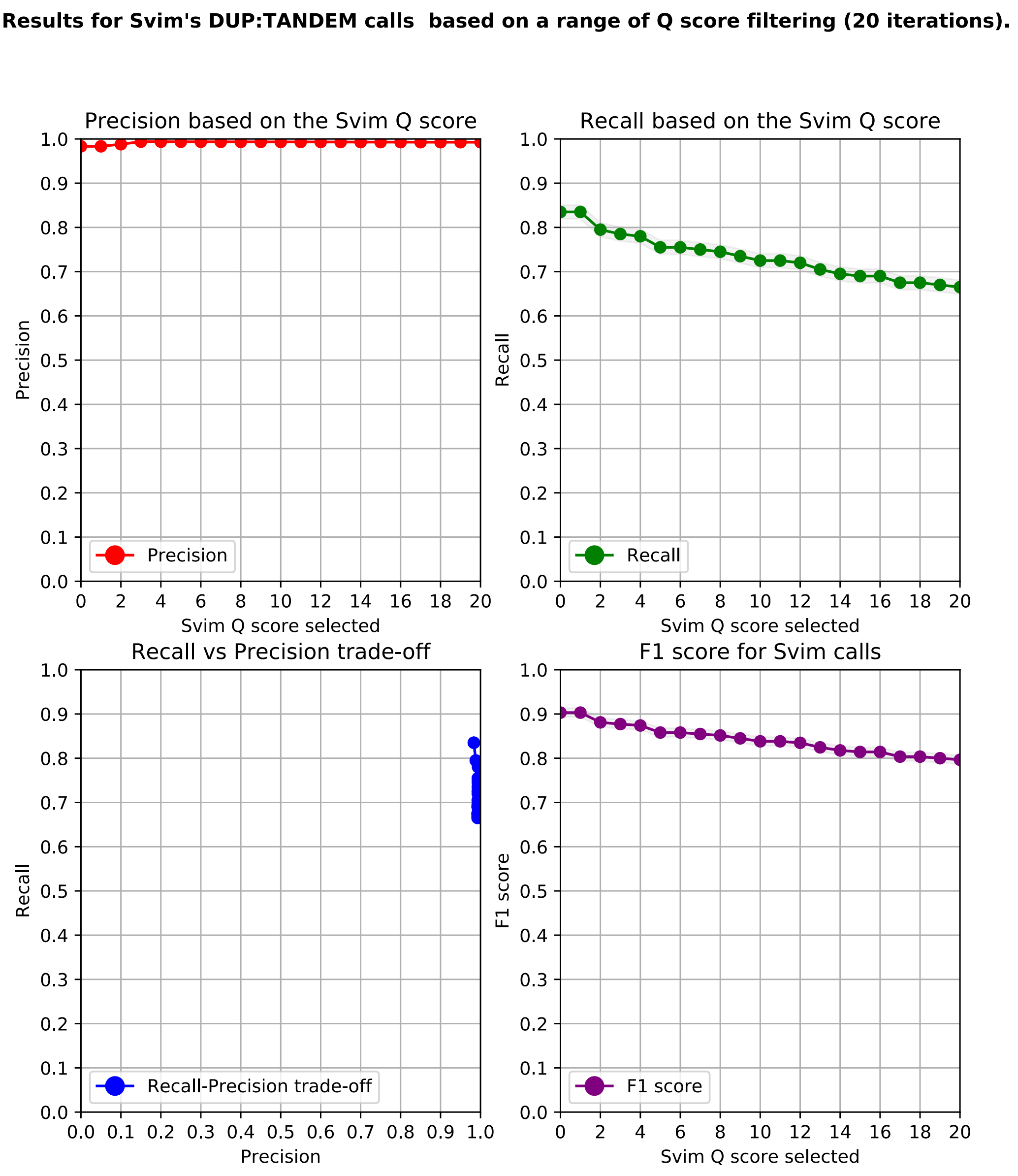

Figure B.3: **Supplementary plot: *Svim* evaluation of homozygous x52 simulated reads with default settings for DUP:TANDEM.** The optimum Q score filtering vale is 2. It must be noted however than at 3-4 the precision is almost 1 at the cost of recall who would take a significant loss.

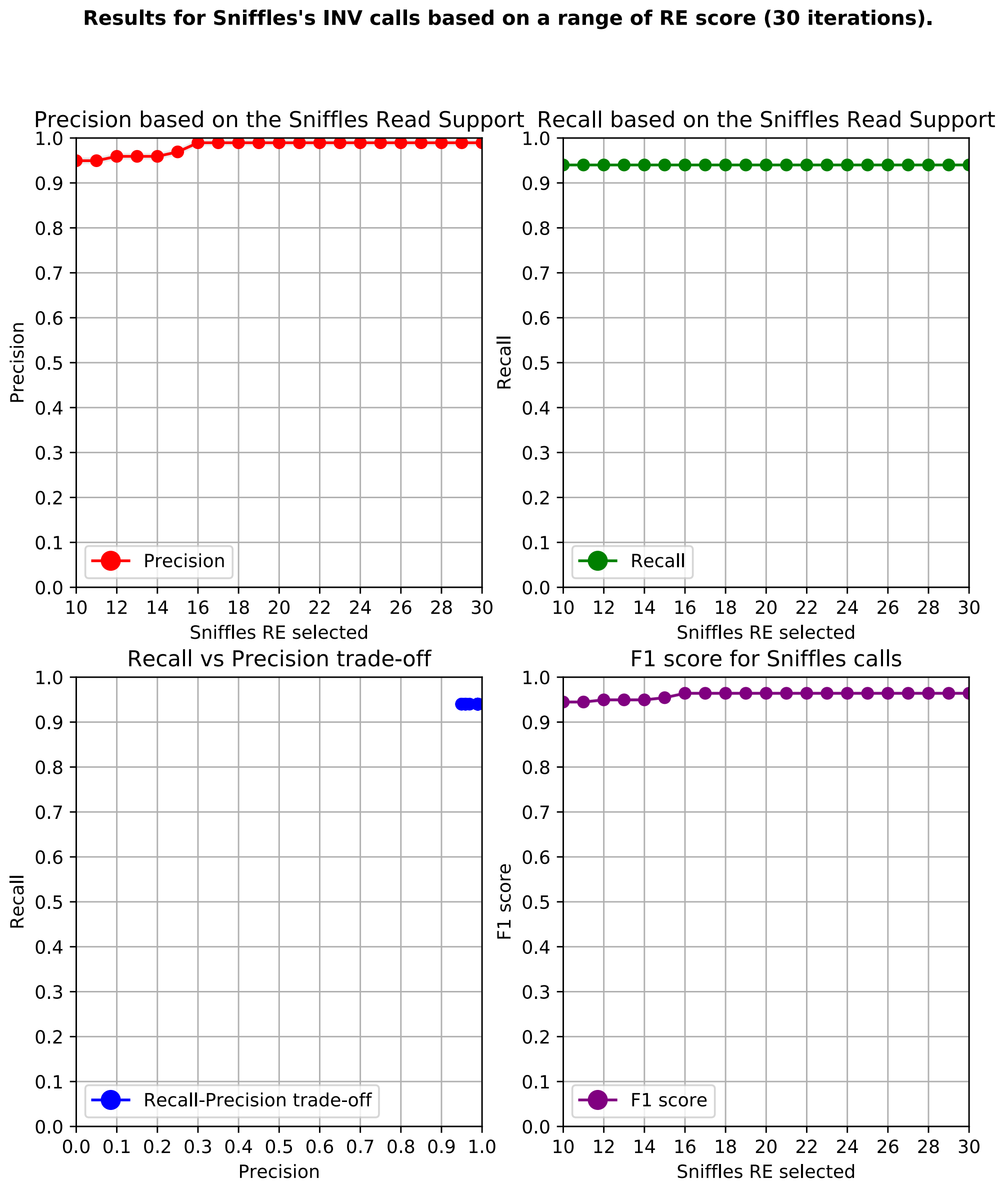

Figure B.4: **Supplementary plot: *Sniffles* evaluation of homozygous x52 simulated reads with default settings for INV.**. The optimum RE filtering value would be 16, as recall do not significantly change, but precision can be improved at that value. Please note that although the number of iterations is 30, as the default prefiltering minimum RE is 10, the first 10 are skipped so in essence is just 20 iterations (starting from 10).

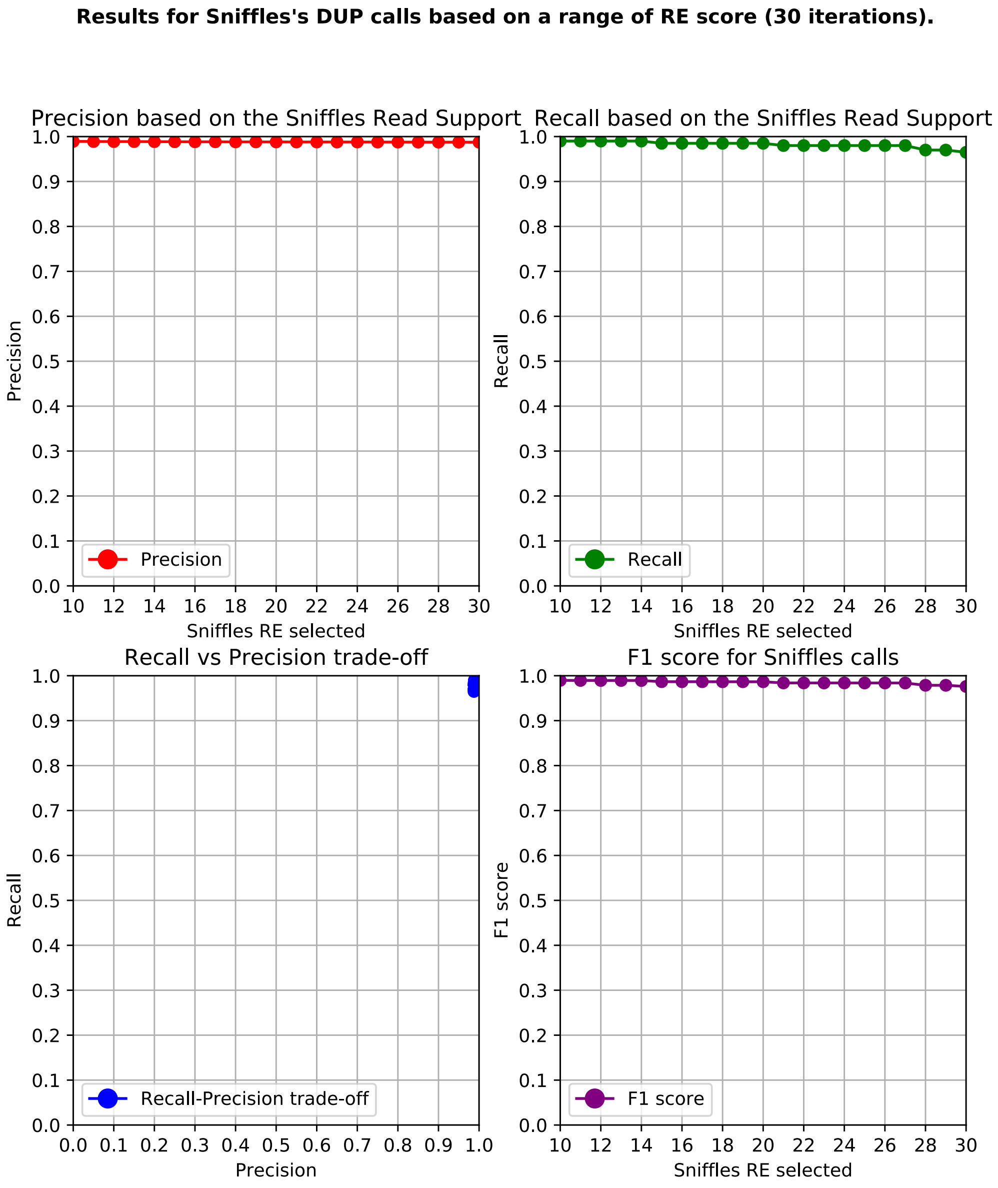

Figure B.5: **Supplementary plot: *Sniffles* evaluation of homozygous x52 simulated reads with default settings for DUP:TANDEM.** There is no optimum score, as barely for all metrics the resuls are equivalent with the ones obtained through the prefiltering phase (except for the RE values at the end of the iteration) Please note that although the number of iterations is 30, as the default prefiltering minimum RE is 10, the first 10 are skipped so in essence is just 20 iterations (starting from 10).

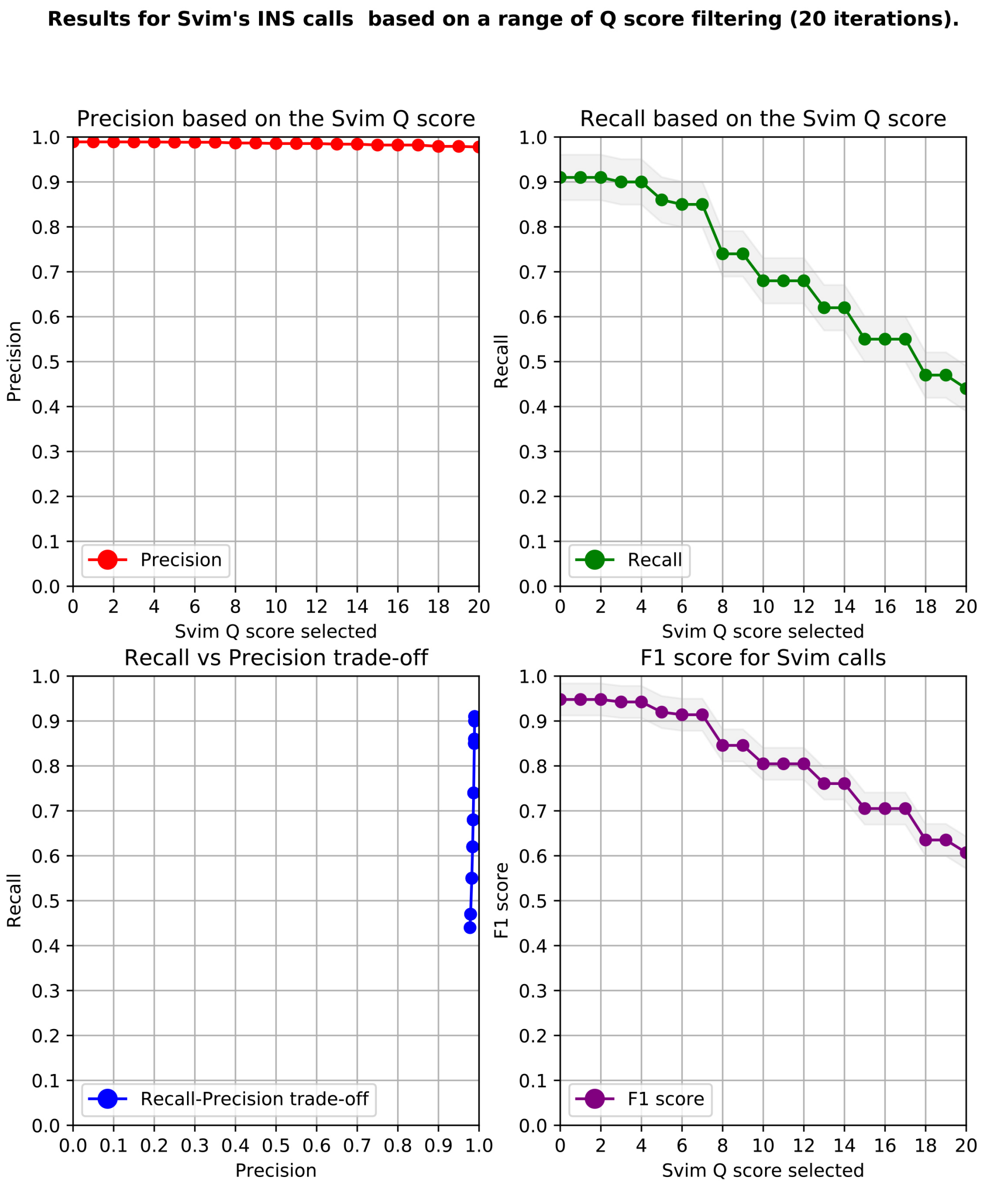

Figure B.6: **Supplementary plot: *Svim* evaluation of heterozygous x52 simulated reads with default settings for INS**. The optimum Q score is 4, as it is the maximum recall value before a signifcant drop in recall. Precision is kept near 1, although it starts to perform worse through overfiltering.

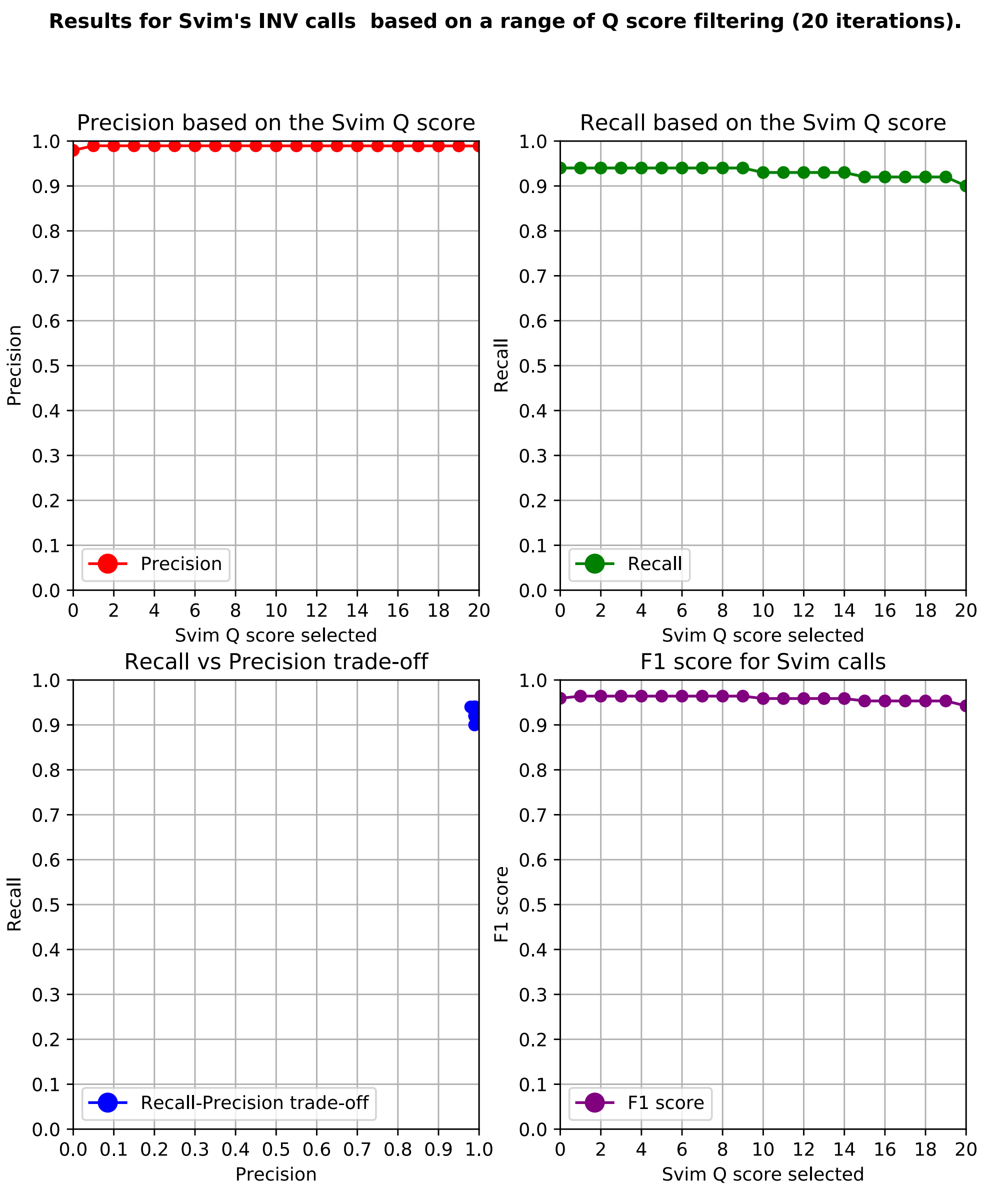

Figure B.7: **Supplementary plot: *Svim* evaluation of heterozygous x52 simulated reads with default settings for INV**. For inversions, only some sort of filtering is required to maximize the precision and recall metrics, as we can clearly see when using a filtering value of 1.

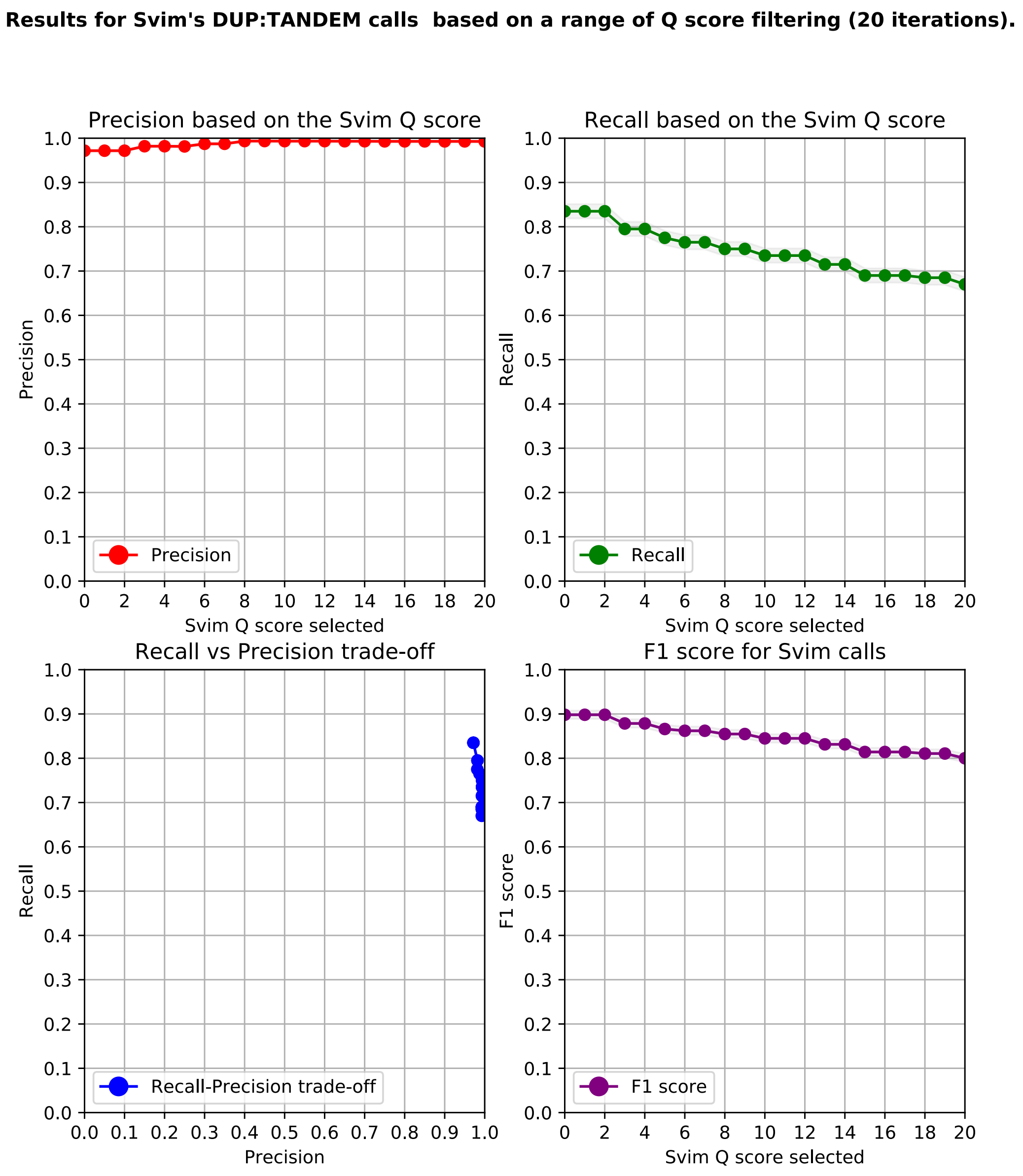

Figure B.8: **Supplementary plot: *Svim* evaluation of heterozygous x52 simulated reads with default settings for DUP:TANDEM**. For tandem duplications, the optimum score to maximize precision is 8. However recall takes a significant hit by doing so. In order to maximize both, the optimum Q score filtering is 2, as we can clearly see in the F1 plot.

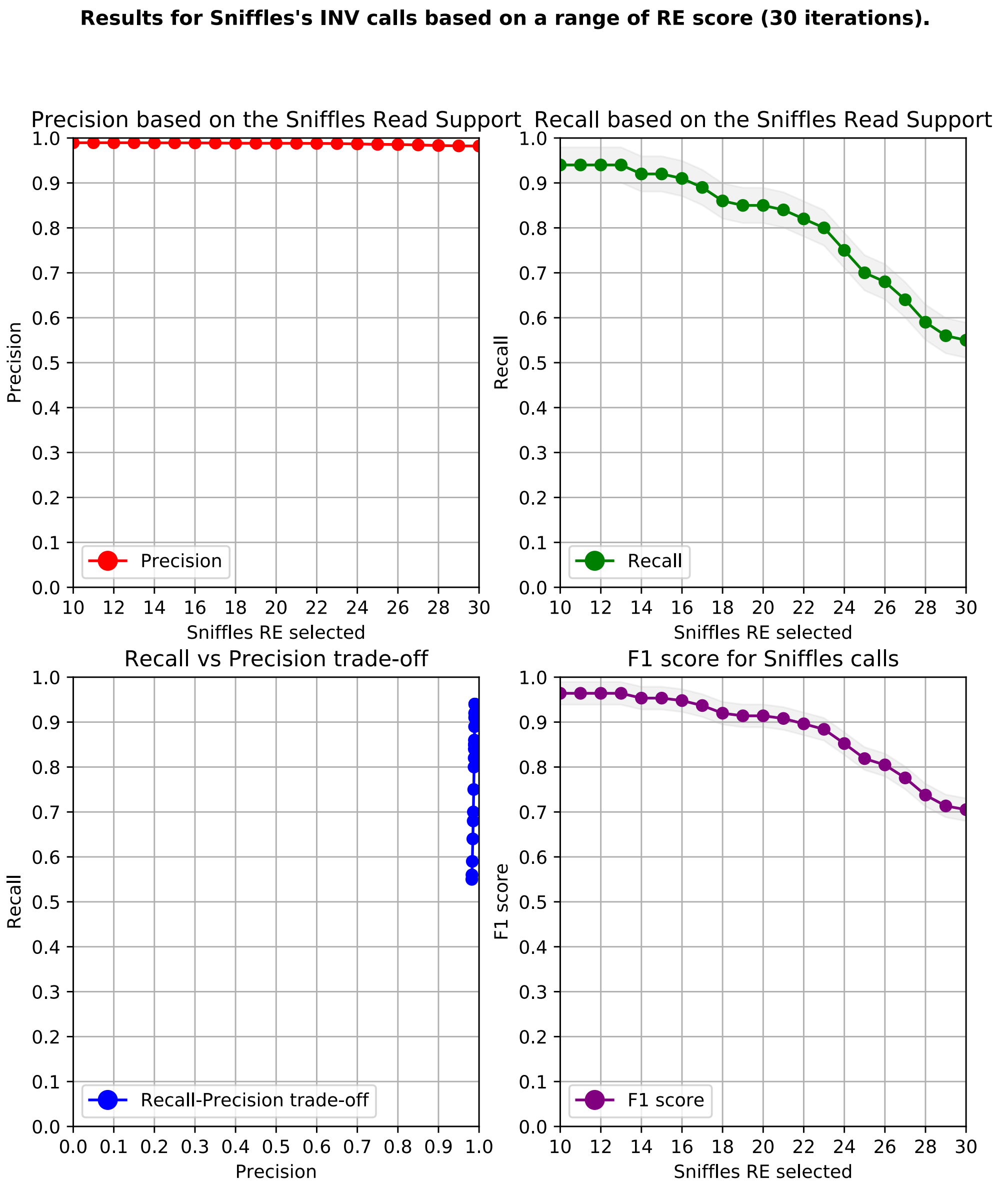

Figure B.9: **Supplementary plot: *Sniffles* evaluation of homozygous x52 simulated reads with default settings for INV.** In this case the optimum RE score would be 13-14, as it is the RE value that maximizes both recall and precision according to the F1 plot. Please note that although the number of iterations is 30, as the default prefiltering minimum RE is 10, the first 10 are skipped so in essence is just 20 iterations (starting from 10).

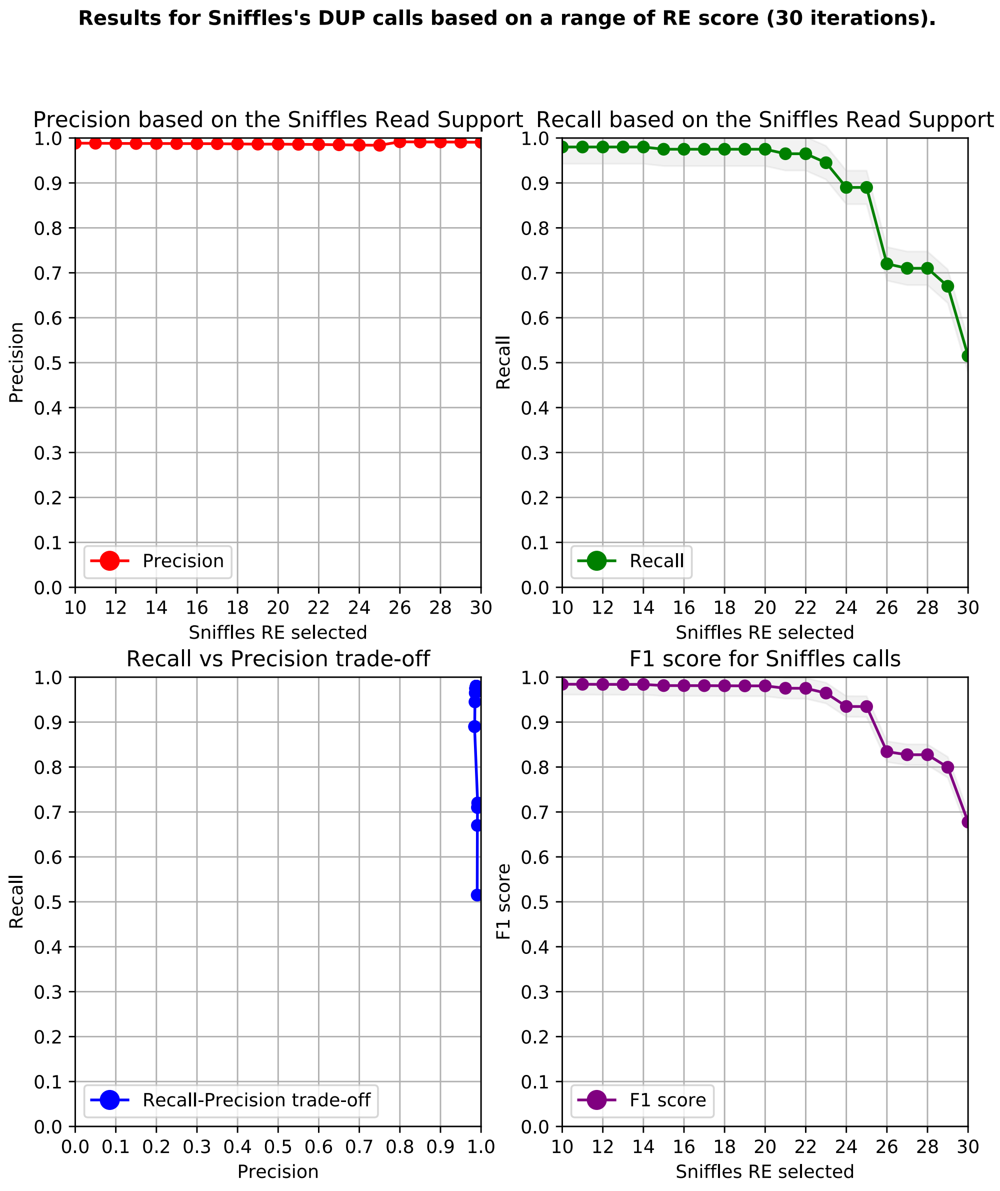

Figure B.10: **Supplementary plot: *Sniffles* evaluation of homozygous x52 simulated reads with default settings for DUP:TANDEM.** In this case, any filtering value in the range of 10-16 would suffice as optimum filtering value. Please note that although the number of iterations is 30, as the default prefiltering minimum RE is 10, the first 10 are skipped so in essence is just 20 iterations (starting from 10).