Detection and visualization of structural variants in ricefish genomes

Thesis

submitted in partial fulfilment of the requirements for the degree

Master of Science

Master programme in Organismic Biology, Evolutionary Biology and Palaeobiology (OEP-Biology)
Faculty of Mathematics and Natural Sciences

Rheinische Friedrich-Wilhelms-Universität Bonn

Presented by

Christian Bräunig

from Hamburg, Germany

Bonn, January 2021

Detection and visualization of structural variants in ricefish genomes

Thesis

submitted in partial fulfilment of the requirements for the degree

Master of Science

Master programme in Organismic Biology, Evolutionary Biology and Palaeobiology (OEP-Biology)
Faculty of Mathematics and Natural Sciences

Rheinische Friedrich-Wilhelms-Universität Bonn

Presented by

Christian Bräunig

from Hamburg, Germany

Bonn, January 2021

This work has been performed at the Center for Molecular Biodiversity Research

in the team of Lars Podsiadlowski

1. Referee: Dr. Lars Podsiadlowski

Center for Molecular Biodiversity Research

2. Referee: Dr. Alexander Donath

Center for Molecular Biodiversity Research

Affirmation for the Master's Thesis

I herewith declare, that I have written this thesis independently and myself. I have used no other sources than those listed. I have indicated all places where the exact words or analogous text were taken from sources. I assure that this thesis has not been submitted for examination elsewhere.

Bonn, _____ _____

Table of Contents

1 List of Abbreviations

bp	Base pair(s)
kb	Kilobases
mb	Megabases
IUPAC	International Union of Pure and Applied Chemistry
IUB	International Union of Biochemistry
ID	Identification
SV	Structural variant
SNP	Single nucleotide polymorphism
ITR	Inverted Terminal Repeats
GUI	Graphical User Interface
OOP	Object-oriented programming
INS	Insertion
DEL	Deletion

2 Introduction

2.1 Synteny and structural variants

Synteny describes the presence and order of a given set of genes on the same chromosome within a genome (Stein, 2013). The order of these genes is disturbed by chromosomal aberrations that occur over the course of evolution and speciation. Initially conserved segments of a given chromosome become progressively derived as intra- and interchromosomal exchanges alter the order – or even presence – of genes within said segment and/or chromosome (Sankoff and Nadeau, 1996). An important implication of this continual derivation is that the extent to which it has taken place in two related species hints at their divergence. The more a conserved chromosomal segment has been divided and thus chromosomal rearrangements have occurred, the further this segment, and by extension the species, has diverged from the ancestral state. A noteworthy phenomenon is the reconstitution of previous exchanges by later ones. As species become increasingly genetically distant, rearrangements accumulate and the size of segments not affected by rearrangements decreases. Thus, the chance grows that recent rearrangements undo the effects of earlier ones (Sankoff and Nadeau, 1996). Interestingly, between closely related species certain arrangements may be syntenic and even rearrangement hotspots have been suggested (Harewood *et al*., 2012).

The cumulative effect of rearrangements is the presence of genetic variation in general, and of different types of variants specifically. Variants have been grouped according to their size; large variants visible under a microscope are classified as microscopic variants (Feuk *et al*., 2006). These are at least several mb in length but are considered to be the lesser contributor to genetic variation as a whole. Submicroscopic variants, instead, probably account for most variations (Feuk, 2006). Among these short variants are SNPs, short repetitive elements and duplications, deletions, insertions and inversions of typically less than 1 kb length. Structural variants, while belonging to the submicroscopic variants, are usually characterized as between 1 kb and 3 mb in length, although the term has been expanded to include variants as small as 50 bp (Feuk *et al*., 2006; Merot *et al*., 2020). Insertions, deletions and duplications of lengths that incorporate complete genes are often summarized as copy number variants (CNVs). They may result in an increase or decrease of a gene’s copy number and, by extension, its expression (Feuk *et al*., 2006; Merot *et al*., 2020). For example, duplication of a gene or

exon may result in sub- or neofunctionalization (Penso-Dolfin *et al*., 2020). Hence these variants are also referred to as unbalanced structural rearrangements (Fig. 1) (Merot *et al*., 2020). No genetic material is lost in the case of balanced structural rearrangements like inversions or reciprocal translocations, yet effects on gene expression are nonetheless possible (Harewood *et al*., 2012).

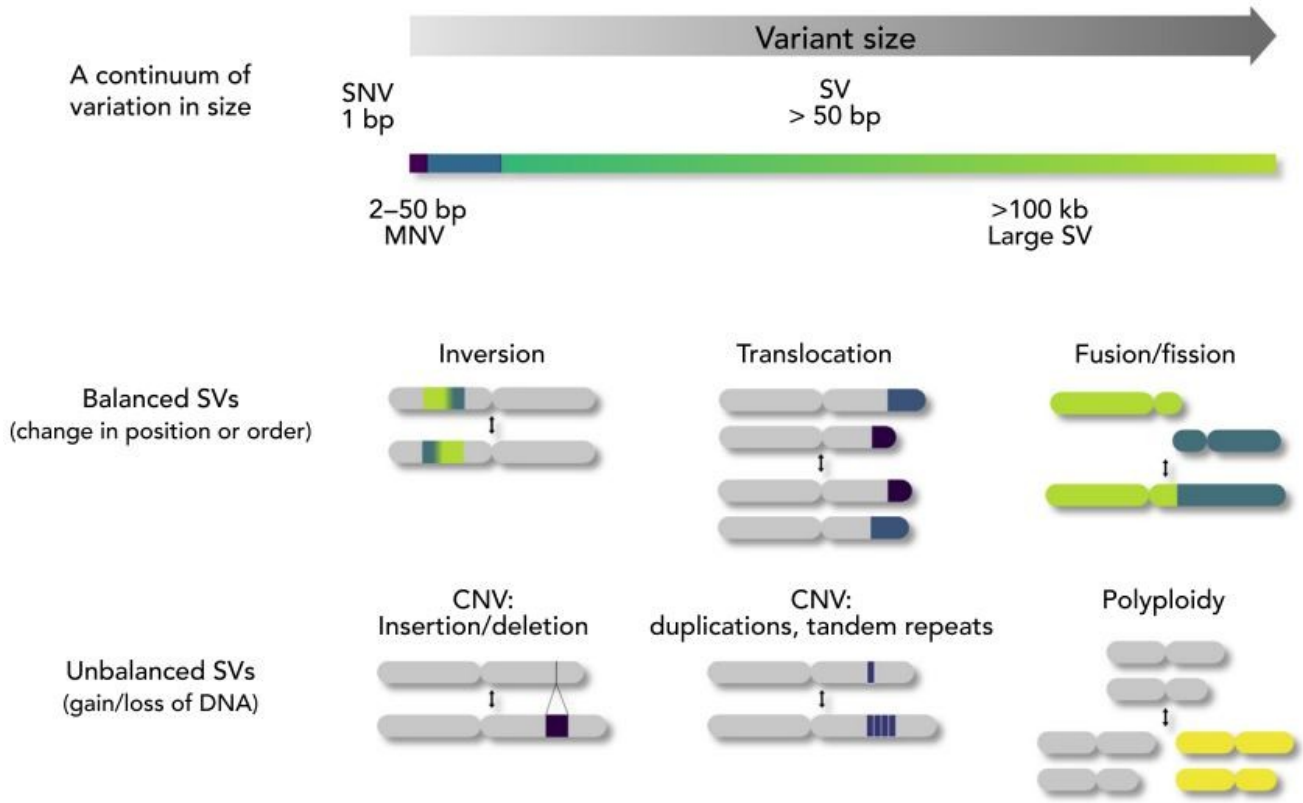

Figure 1: Classificaton of submicroscopic genetic variants according to size and underlying mechanism into single nucleotide variants (SNV), multiple nucleotide variants (MNV) and structural variants (SV). Unbalanced SVs can be further grouped into copy number variants (CNV). Taken from Merot et al., 2020.

2.2 Visualization tools and approaches

Visual representation is a powerful tool for the exploration and communication of genomic data, although at times challenging. The breadth of currently used data formats, the complexity of the data and its sheer size can all make exploring data visually complicated and time demanding. As a result, a plethora of visualization tools exists, often tailored to specified applications, data formats and questions

at hand. Exploration of variation across a genome can be associated with very large file sizes – depending on the size of the genome and question – and a variety of formats for variant calling, annotation, and metadata. Additionally, the resolution at which data is to be seen – from individual bases to regions to whole chromosomes – can further pose challenges.

One of the tools currently available for such data visualization and exploration is the Integrative Genomics Viewer (IGV) (Thorvaldsdottir *et al*., 2013). As the name suggests, a large focus for IGV lies in the integration of data for genomic studies; for example, coupling sequencing data with for example variant data, gene expression data or sample-specific information (Fig. 2).

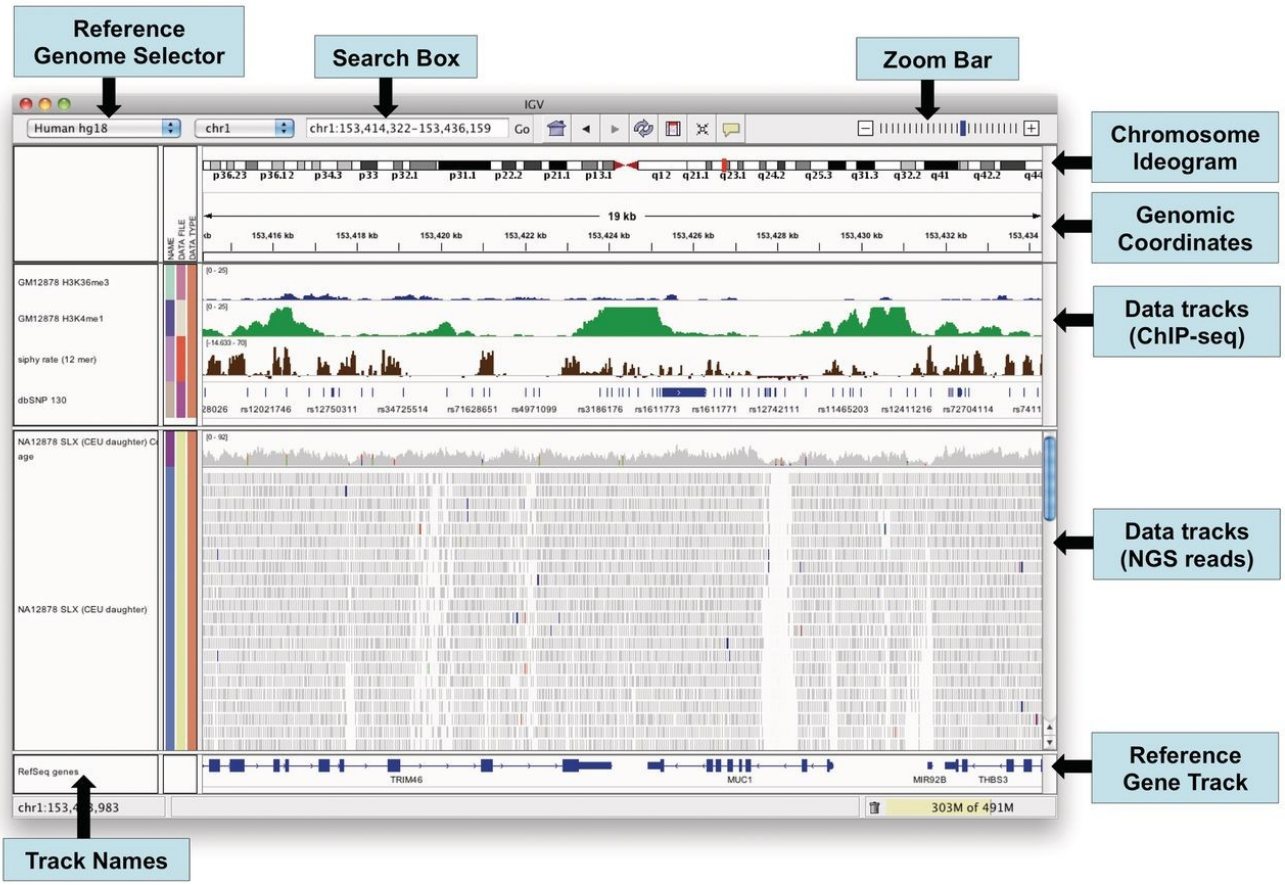

Figure 2: Exemplary view of the IGV application window. Depending on the data loaded, a variety of information is visible at once with different data separated into distinct boxes. Array-based and next generation sequencing data can be visualized side by side and along a reference genome (Thorvaldsdottir et al., 2013).

Another potent visualization software is the web-based genome browser hosted by the University of California Santa Cruz (UCSC Genome Browser, https://www.genome.ucsc.edu/index.html, Fig. 3).

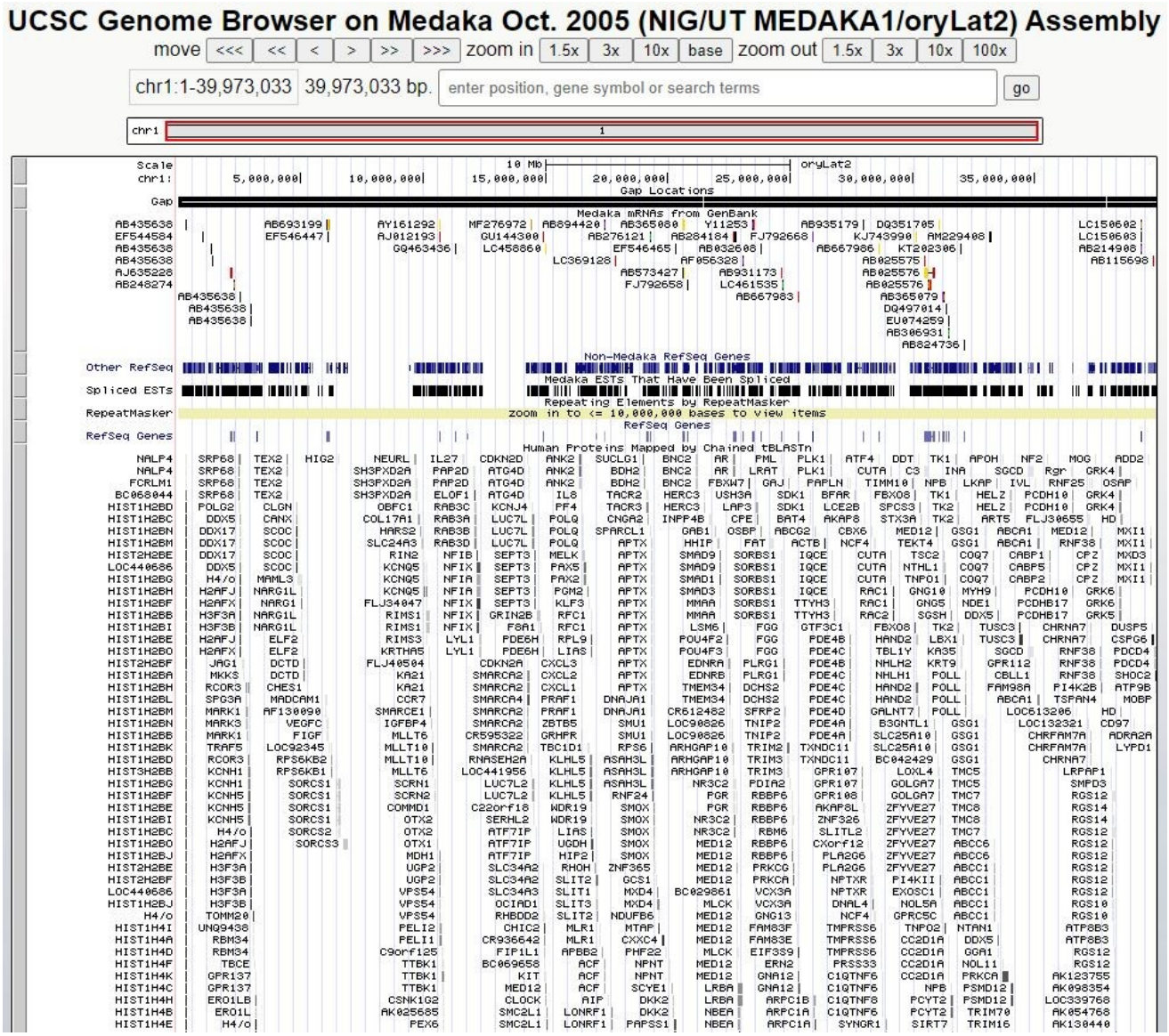

Figure 3: Exemplary view of UCSC Genome Browser featuring chr1 of a Oryzias latipes assembly. The browser provides the location of annotated genes, RefSeq genes from other species, repeat and sequence tag information, as well as alignments for related model organisms. The user can change mapping, gene and gene prediction, mRNA, variation and repeat parameters interactively and in real-time.

Created in 2000 to visualize the first sequenced and assembled human genome by the Human Genome Project consortium, the browser has since been extended by the addition of further genomes from a variety of model organisms. The power of this browser lies in its quick access to databases to add and manipulate complementary data while also featuring a variety of packages to extend functionality to different formats.

The Apollo Genome Editor is especially powerful with special regard to genome annotation (Dunn *et al*., 2019). It allows for manual addition and viewing of custom annotations to genome assemblies. The tool provides extensive functionality for manipulating and tailoring annotations, like aligning experimental sequences, as well as extracting sequence information from said annotations. A noteworthy feature is the option for several users to annotate and curate a single assembly together (Fig. 4).

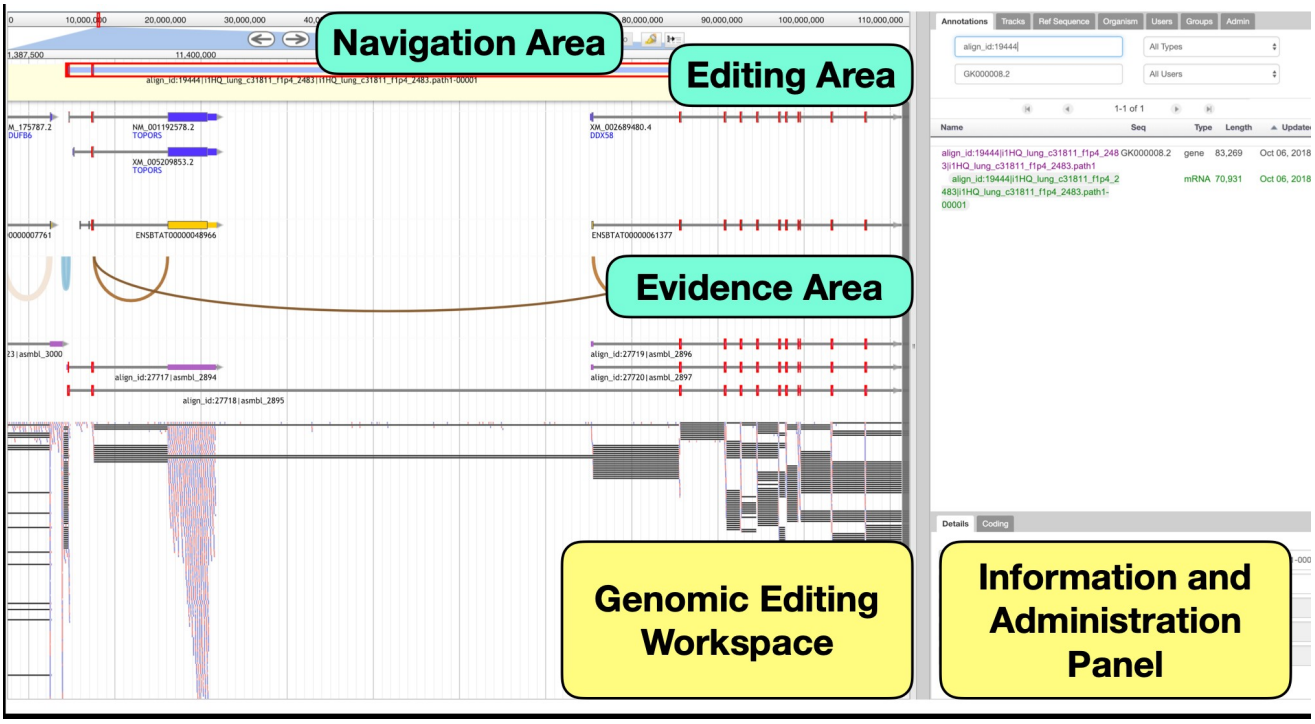

Figure 4: Exemplary view of the Apollo window. Annotation and editing is carried out in the Genomic Editing Workspace; the given genome is traversed using the various buttons of the navigation area, data for potential annotations is loaded into the evidence area and dragged into the editing to create an annotation. The information and administration panel provides search and filtering functionality for the edited annotations as well as administrative functions. Taken from Dunn et al. 2019.

This selection of visualization software is by no means extensive and a plethora of tools exists for a wide variety of specific applications. The shown tools do, however, stand among the most potent and versatile available and are a good representation of the general features visualization software should have.

2.3 Data formats

Modern molecular genetic and bioinformatic methods allow for the exploration of a vast number of genomic phenomena. A result of this exploration is the generation of data which has to be recorded, processed and made available. For this purpose, data is formatted in standardized fashions to make it usable. In the context of genomic variation analysis and visualization, sequence and annotation and variant call data is integral. Specific file formats have been designated for these different data types.

2.3.1 FASTA

The FASTA format allows for representation of nucleotide or amino acid sequence information. Each nucleotide or amino acid is represented by a single letter according to IUB/IUPAC standards with usually 60 or 80 letters per line. These standards feature unique single letter codes for each nucleotide and amino acid as well as for ambiguity cases. The actual sequence information is prefaced by a header. Each header begins with ‘>’ and typically contains an identifier by which to recognize the sequence. A header is delimited by a line break from its sequence and each header-sequence pair is separated from others by one more line breaks. Typically, ‘N’ represents a gap in the sequence, i.e. a position at which no nucleotide/amino acid was determined. FASTA files are commonly saved with ‘.fa’, ‘.fas’ or ‘.fasta’ extensions, although no standardized extension exists.

2.3.2 GFF

The *General Feature Format* is a simple text-based format for the documentation of genomic features (Stein, 2020). Each feature is characterized by information given in a single line, subdivided into nine tab-separated columns. Each column holds specific information:

1. *sequid* The name or identifier of the “landmark used to establish the coordinate system of the current feature”; this usually refers to the chromsome, contig or sequence on which the current feature is located.

2. *source* The name of the method by which the current feature was found, usually a software or database name

3. *type* The type or classification of the current feature, for example ‘Gene’, ‘Similarity’, ‘mRNA’

4. *start* The first base pair position of the current feature relative to the first position of the sequence given by ‘seqid’

5. *end* The last base pair position of the current feature, determined analogously to ‘start’. ‘end’ must be equal or larger to ‘start’

6. *score* A floating point number representing the likelihood that the feature exists; Steiner (2020) proposes E- or P-values be used depending on how the feature was found.

7. *strand* The strandedness of the feature (5’-3’ /+ or 3’-5’/-) relative to the strand of the sequence given by ‘seqid’

8. *phase* The position of the feature at which the next codon begins, required for CDS-type features; ‘phases’ may only be 0, 1 or 2.

9. *attributes* Any additional or supplementary information may be given here in a 'tag=value' format, multiple attributes separated by semicolons.

‘.’ may be used to fill empty columns. Lines with metadata begin with ‘#’ and typically preface the actual data. It should be noted that base counting for this format is one-based; the first base of the chromosome or contig is given the index ‘one’. The base count for a given feature also includes the last base; a feature of 100 bp length starting at the first position of the chromosome would thus be noted with ‘start: 1’ and ‘end: 100’.

2.3.3 BED

The *Browser Exensible Data* format is a flexible format allowing for documentation of sequence features even with little information. Like in the GFF format, a given line is dedicated to one feature and separated by tabs into at least three columns, at most twelve.

1. *chrom* The name of the chromosome, scaffold or contig on which a given feature lies

2. *chromStart* [analogous to ‘start’ for GFF]
3. *chromEND* [analogous to ‘end’ for GFF]

The contents of the nine additional optional columns are not relevant to this work but can be seen at http://genome.ucsc.edu/FAQ/FAQformat#format1. Unlike that of the GFF format, base counting is zero-based. The first base of a chromosome has the index ‘0’. The last base of a given feature is not included in the count; a feature of 100 bp length starting at the first position of the chromosome would thus be noted with ‘start: 0’ and ‘end: 100’, spanning bases ‘0’ through ‘99’.

2.3.4 VCF

The *Variant Call Format* is a text-based format primarily for the documentation of genomic variation (Danecek *et al*., 2011). Similar to GFF and BED formats, it consists primarily of data lines with eight tab-separated columns.

1. *CHROM* Name of the reference chromsome or contig
2. *POS* Reference position relative to the first nucleotide position of ‘CHROM’ with one-based base count
3. *ID* Unique identifier(s) for the currrent variant, semicolon-separated
4. *REF* Reference base(s), ‘POS’ denotes the position of the first base, given as upper-case
5. *ALT* Non-reference allele(s), comma-separated, given as upper-case
6. *QUAL* Quality score based on phred-scale. High confidence calls are indicated by high scores.
7. *FILTER* ‘PASS’ if a variant call was made at the given position and it passed all filters. If not all filters were passed, filters that were not passed are listed comma-separated.
8. *INFO* Additional information given as semicolon-separated list of 'tag=value' pairs

Data lines and headers are typically prefaced by metadata lines, each starting with '##'. Metadata might include file format version number, supplementary information for entries in the data lines and further information on the reference genome. For very detailed information see http://samtools.github.io/hts-specs/VCFv4.2.pdf. VCF files serve as central input for the visualization software.

2.4 Ricefishes

Adrianichthyidae, commonly known as ricefishes, is a family of beloniform fish native to Southeastern Asia. It consists of two genera, Adrianichthys and Oryzias, which together include 28 extant species. *Oryzias latipes*, the Japanese medaka, is the best known of the ricefishes and has been an important model organism for a wide range of disciplines, including endocrinology, embryology, toxicology and genetics, where it has been used to study transgenesis among other things (Parenti, 2008).

Neoteleosts, the taxonomic clade to which also ricefishes belong, have previously demonstrated to feature a large amount of SVs in their genomes. Feulner *et al.* (2012) showed that a substantial part (7%) of the genome of the three-spined stickleback is subject to structural variation, including more than 20000 indels, 1000 CNVs and over 40 inversions. These structural variants affected 10% of stickleback genes. When including SNVs, most genes were found to contain some form of intraspecific variation. Cichlids have also been shown to feature extensive structural variation in their genome (Penso-Dolfin *et al*., 2020). Given the adaptive radiation of cichlids within East African lakes, these variants likely contributed to the development of novel traits, for example by neo- and subfunctionalization after gene duplications or alterations of expression regulation via deletions or inversions. In the Australasian snapper, SVs may be an even more powerful phenomenon as they affect a three-times higher amount of bases than single nucleotide polymorphisms, some 18 mb (Catachan *et al*., 2019). This highlights the possibly massive contribution of SVs to genetic variation.

2.5 Aim of Thesis

The aim of this thesis was the design of application-tailored visualization tools for structural variant analyses for the scientists of the Center for Molecular Biodiversity Research of the Zoological Research Museum König. There, ricefishes from Sulawesi (Indonesia) have been found to feature highly active transposable elements and are used as model for the effect of transposon activity on structural variant density. In this context, the presented tools are intended for investigation of structural

variants of specific sizes most prevalent in this research and allow for comparison with feature data as well as local alignment data, making use of the most commonly used data formats. The functionality of these tools has been designed to be available as command-line and GUI-based programs. This allows the automation and integration of visualization into pipelines and facilitates saving option settings but also lets scientists explore data visually in real time. The presented programs thus do not comprise a Swiss army knife of functionality for each and every application but rather serve a very specific purpose.

3 Material and Methods

3.1 Python

The programs presented in this thesis were written in the programming language Python (version 3.7.6). Python is an interpreted programming language characterized by an intuitive, readable syntax and powerful flexibility. Inbuilt data structures, objects and the possibility to draw on modules and libraries allows python programs to be tailored to very specific applications, but also retain reusability. Since a separate compilation step is not necessary to execute python scripts, debugging is rapid and easy (Python Software Foundation). Python was introduced in 1991 and has since become a popular and well-maintained programming language with a large community (McKinney, 2017). The versatility and power of python is extendable with libraries that add application-specific utilities.

3.1.1 Libraries

Third party libraries allow for the extension of Python’s potency beyond its built-in functionality. In this case, visualization and graphical user interface development are greatly helped by the respective use of dedicated libraries.

Matplotlib

Matplotlib is an extensive, well-documented library specialized in 2D visualization and generation of publication-ready graphs and figures (Hunter, 2007). Graphs can be easily generated with little code but at the same time be highly modified and tailored to the application at hand. A large number of graph types can be generated and a plethora of annotation and individualization possibilities exist.

Accordingly, this library provided the entirety of the graphing functionality to create the graphs for the program output. Version 3.2.1. was used for the presented programs.

tkinter

Tkinter is dedicated Python package for GUI development, based on the programming language Tcl (‘Tool command language’, http://www.tcl.tk/)). Like Python, it is characterized by easy syntax and high versatility. Within Tcl, the Tk toolkit allows for the development of desktop applications (https://wiki.python.org/moin/TkInter). The graphical user interface of SVgui.py was created using tkinter to allow for real time visual data exploration. For the presented programs version 8.6 was used.

Numpy

Numpy is a Python library for numerical data computing. Most notably, it provides a multi-dimensional array (ndarray) data structure (McKinney, 2017). Version 1.18.3 was used here and allowed for the generation of heat maps representing SV density.

3.2 Genomic data

The *Oryzias eversi* assemblies used here are original assemblies provided by Lars Podsiadlowski and Julia Schwarzer. They were generated from Oxford Nanopore long reads (flye 2.7, CSA 2.6) and Illumina short reads for error correction using pilon 1.22. RepeatMasker and RepeatModeler were used to annotate repeat regions.

The genome assembly OLAT_genomic.fna is publicly accessible on NCBI's ftp server under https://ftp.ncbi.nlm.nih.gov/genomes/refseq/vertebrate_other/Oryzias_latipes/latest_assembly_versions/GCF_002234675.1_ASM223467v1/ as *GCF_002234675.1_ASM223467v1_genomic.fna.gz* and the associated feature file as *GCF_002234675.1_ASM223467v1_genomic.gff.gz*. These files are compressed and must be uncompressed before use.

3.3 Original scripts

For optimal usability in combination with different user software equipment, the presented programs have been written for usage on command-line terminals as well as on more common desktops by

making use of a graphical user interface. The full code for all scripts is available for viewing and download at https://github.com/c-braeunig/structural-variation-viewer.

3.3.1 Variant-mapping script

In genomic studies, sequencing is the first major step to the extraction of information that a genome harbors. With the sequencing output, commonly in FASTA or FASTQ format, a wide array of further investigations can be conducted. In the case of variant calling, several more steps are necessary. The final steps are the mapping of a query genome – the form of sequencing reads – against a reference and the subsequent determination of structural variants. Even though long reads were available for the assemblies used here and could have been mapped against a reference, a different approach was taken; Artificial long reads were simulated by extracting fragments of known size from the finished assemblies. This facilitates using uniform read length and coverage for mapping across all used assemblies and allows for the inclusion of assemblies without available original long read data. Additionally, this approach avoids scattered alignments that might result from whole-genome alignment and which would complicate visualization.

The process from assembly to artificial reads to variant calls was streamlined in a single bash script (Appendix 1). To allow for easy application of the script to many input files and to facilitate output and intermediate file naming, relevant file names are first assigned to a respective variables (Appendix 1, lines 4-10). Additionally, parameters that can be set by the user may be assigned, namely for makebed.py (Appendix 2) (Appendix1, lines 5, 6). The first step of the script is the indexing of the query genome (‘$infasta’) using samtools faidx (Appendix 1, line 13). Samtools (version 1.9) is a suite of tools for the manipulation of alignment file formats. The faidx tool indexes the FASTA file and outputs a ‘.fai’ index file (‘$infasta.fai’) that contains a table with information per sequence, including name and length (http://www.htslib.org/doc/samtools-faidx.html, https://www.htslib.org/doc/faidx.html). The next two steps (Appendix 1, line 16, 19) are needed to generate a set of “simulated” long reads as are generated by next-generation sequencing technologies. The custom script makebed.py uses the index file along with the windowsize and stepsize parameters to make a BED file (‘$outbed’) of arbitrary regions with length windowsize and each starting stepsize bases later than the previous one (see Appendix 2 for full code). A windowsize of 10000 and stepsize of 500 were chosen to allow for detection of larger structural variants and a 20x times coverage (Fig. 5).

This means that a given base position appears in twenty different reads. This BED file is then used by the bedtools getfasta tool (https://bedtools.readthedocs.io/en/latest/content/tools/getfasta.html, version v2.29.2) to extract sequence bits from the query genome FASTA file (‘$infasta’) based on the coordinates specified in the BED file (‘$outbed’) (Appendix 1, line 19) (Quinlan and Hall, 2010). Depending on the sequencing tools used and their output, one could start immediately with the mapping step (Appendix 1, line 22). The “simulated” long read data can then be mapped against the reference genome (‘$reference’) by ngmlr (Sedlazeck *et al*., 2018). A SAM alignment file is generated as output (see http://samtools.github.io/hts-specs/SAMv1.pdf for further information). Samtools view and sort tools then allow for conversion and sorting of the SAM file into the binary equivalent BAM format. With the BAM files, sniffles (Sedlazeck *et al*., 2018) can finally call variants and output the results as VCF file.

```
R6_1_pilon_pilon        0       10000
R6_1_pilon_pilon        500     10500
R6_1_pilon_pilon        1000    11000
R6_1_pilon_pilon        1500    11500
R6_1_pilon_pilon        2000    12000
R6_1_pilon_pilon        2500    12500
R6_1_pilon_pilon        3000    13000
R6_1_pilon_pilon        3500    13500
R6_1_pilon_pilon        4000    14000
R6_1_pilon_pilon        4500    14500
```

Figure 5: Exemplary BED-formatted output produced by ***makebed.py***

3.3.2 Command-line scripts

For the sake of modularity and reusability, the functionality of the program was split into two primary scripts. ***SVread.py*** reads an input VCF file, retrieves relevant information and gathers this information in a new file with BED-like format. This file can then be fed into ***SVsee.py*** which uses the contained data to generate the final graphic output. ***SVpipe.py*** acts as combination of the two primary scripts to allow for a more streamlined process from VCF and graphic output. For all of the scripts presented below, general usage always features:

```
$ python ./[script name]
```

and is then extended by adding flags, options and parameters.

SVread.py

The main function of this script is the consolidation of relevant data from the input VCF file into a more brief format. The internal workings of the program can be grouped into three main functions: reading and filtering of information is followed by sorting, if selected, and finally generating the output. Reading of the VCF file is handled by one large subroutine that uses nested subroutines for smaller operations. Essentially, each data line is separated into its columns and the relevant information bundled as an object. This allows the various data points for each variant to be conveniently handled together while still being individually accessible. If a given variant fulfills the various filtering parameters the corresponding object is appended to a list. This object list is the input for the various possible outputs. It is sorted if the according argument was provided. During the sorting step, the length filters are also applied. The list contents are then either printed or saved to a file, or further manipulated.

```
$ python ./SVread.py -h
```

prints a usage guide including descriptions to all available flags and options:

```
usage: SVread.py [-h] [--vcf VCF]
                 [--sort {asc_size,des_size,type, type_asc_size,
                  type_des_size,CR}]
                 [--ov] [-C CONTIGNAME] [-T TYPE [TYPE ...]]
                 [-S CONTTYPE [CONTTYPE ...]] [--ulim ULIM]
                 [--llim LLIM]
                 [--seq] [--nosize] [--circos] [--out OUT]

Parse and derive SV information from VCF output of variant calling
software; --ov generates a superficial overview of the listed
structural variant (SV) types and their number of occurrences. Using
--sort, --C, --T, --S, --llim and --ulim, the SVs of intereste can
be isolated and printed to stdout or to a file ( --out) in a BED-
like file format. For further information on the VCF format consult
https://samtools.github.io/hts-specs/VCFv4.2.pdf
```

```
optional arguments:
  -h, --help            show this help message and exit
  --vcf VCF             Provide variant call file in VCF format
  --sort {asc_size,des_size,type,type_asc_size,type_des_size,CR}
                        Provide sorting criterium to organize SVs;
                        SVs sorted by position on chromosome/contig
                        by default; no sorting recommended for
                        downstream SVsee use; Use "CR"
                        (chromosomal rearrangement) when filtering
                        by "BND" (complex rearrangement with
                        breakends) to sort target locations
  --ov                  Choose to display an overview of SVs in the
                        provided VCF file; --ov only will print
                        assembly-level information; using --C option
                        will print single-contig-level information
  -C CONTIGNAME, --contigname CONTIGNAME
                        Provide contig/chromosome ID by which to
                        filter SV records; multiple IDs given space-
                        separated; use only one ID for visualization
  -T TYPE [TYPE ...], --type TYPE [TYPE ...]
                        Provide type(s) by which to filter SV
                        records i.e. 'DEL'; multiple types given
                        space-separated; 'BND' for
                        complex rearrangments with breakends
  -S CONTTYPE [CONTTYPE ...], --conttype CONTTYPE [CONTTYPE ...]
                        Complementary type filter, same usage as --T
  --ulim ULIM           Provide upper length limit for SVs [bp]
                        (inclusive)
  --llim LLIM           Provide lower length limit for SVs [bp]
                        (inclusive)
  --seq                 Select to write sequence data into single
                        fasta file;
                        requires --out
  --nosize              Select to not display the end position and
                        size [bp] of each SV in line
  --circos              Circos mode; generates karyotype and link
                        files in current directory for circos for
                        BND-type SVs to show inter-chromosomal
                        events; compatible w/ --C option
  --out OUT             Provide desired name of bed-formatted output
                        file
```

The following arguments or flags can be used. An input VCF file is fed to the program by providing the file path after --*vcf*. The --*out* option takes a file path to a desired output file with BED-like format, with chromosome name, starting position, length, type and sequence given in tab-separated columns. Omitting this option will print the data onto the terminal.

The --*sort* option takes one of six arguments. By default, variants are ordered according to their position along the chromosome or contig within a VCF file. The *asc_size* argument sorts the variants according to their size from smallest to largest, while *des_size* results in sorting from largest to smallest. *type_asc_size* and *type_des_size* work analogously to the previous two arguments, but maintain a separation among the different variant types. The order of the different types follows alphanumeric order. The *CR* ('chromosomal rearrangement') argument is specific to *BND* ('complex rearrangement with breakends') type variants and orders them according to the ascending position of their target locations.

Apart from sorting, several ways to filter the contents of the input file are provided. The --*contigname/-C* option takes a single chromosome/contig name as argument to only retain variants that are located on the provided chromosome/contig. The later visualization with ***SVsee.py*** is limited to the view of a single chromosome or contig so for this downstream application, -*C* should be used. --*type/-T* and --*conttype/-S* options both influence which types of variants are to be included. If only specific types are desired, these can be specified using -*T***.** -*S* is the complementary version and any types specified using it will be excluded; notably these two options are incompatible so only one can be applied at a time. --*llim* and --*ulim* respectively refer to the lower and upper limits of of variant lengths, given in base pairs. Choosing --*seq* will gather the sequence information for all variants, if available, and save them into a single FASTA file. Headers are formatted as >[chromosome name]:[position],[length],[type]. Choosing --*circos* is specific to BND type variants and converts their information into a format usable for the circos visualization program (Krzywinski *et al*., 2009).

The --*ov* flag generates a file-level or chromosome level overview. By not specifying a chromosome with -*C*, the total number of variants for each chromosome will be determined and printed to the terminal. If a chromosome is specified using -*C*, the variants on that chromosome will be summed by type and minimum, maximum, average and median length for each type is determined.

Thus,

```
$ python ./SVread.py --vcf OEVEvOLAT_genomic.vcf--ov
```

yields

```
Overview of total SVs on 24 contigs
NC_019859.2: 1442
NC_019860.2: 417
NC_019861.2: 2444
NC_019862.2: 1588
...
```

and

```
$ python ./SVread.py --vcf OEVEvOLAT_genomic.vcf -C NC_019859.2 --ov
```

will print

```
Total number of SVs: 1442
INS: 1099  --- MIN: 35, MAX: 11399, AVG: 1181.9, MED: 712
DEL: 326  --- MIN: 31, MAX: 49240, AVG: 2916.7, MED: 2212
BND: 14  --- MIN: 1, MAX: 1, AVG: 1.0, MED: 1
DUP: 2  --- MIN: 448, MAX: 919, AVG: 683.5, MED: 684
INV: 1  --- MIN: 357, MAX: 357, AVG: 357.0, MED: 357
```

onto the terminal.

The order of the individual flags is not relevant, and with the exception of *-T* and *-S*, flags are readily combinable.

SVsee.py

This script takes the output of ***SVread.py*** and visualizes the data using matplotlib utilities. Accordingly, matplotlib must be installed to allow for proper performance. ***SVsee.py*** must first read the data from the text file produced by ***SVread.py*** and does so with the help of objects. ***SVread.py*** allows for the selection and visualization of annotating information from GFF, BED or BLAST output (in tabular format; outfmt 6) files. Annotations from these files are also read into objects, with a dedicated object class for each annotation format.

Using matplotlib utilities, an empty graph window is created and then populated with the variant and annotation elements in the form of color coded rectangles. Visualization is restricted to one chromosome/contig at a time. The x-axis represents the full length of the chosen chromosome and the position of variants along the x-axis mirrors their basepair position. The vertical position of a given variant does not have any meaning. Stacking of variants vertically allows to retain some resolution; short variants quickly become indiscernible. The addition of a heat map is also possible and can be useful to estimating variant density. The final graph is then saved in the format and under the name specified.

General usage is similar to that of ***SVread.py***;

```
$ python ./SVsee.py -h
```

will yield an overview of all available flags and options:

```
usage: SVsee.py [-h] [--IN IN] [--vcf VCF] [--gff MOD] [--bed MOD]
                [--blast MOD] [-R ANNO_TYPE [ANNO_TYPE ...]]
                [-RC ANNO_CONT [ANNO_CONT ...]] [-RL ANNO_LOWER]
                [-RU ANNO_UPPER] [--heat HEAT] [--reg REG REG]
                [--out OUT] [--outfmt {pdf,svg,png}]

Single-chromsome visualization of SV distribution based on SVread.py
output

optional arguments:
  -h, --help            show this help message and exit
  --IN IN               Provide input file (SVread.py output file)
  --vcf VCF             Provide VCF file corresponding to input file
  --gff MOD             Provide contig annotation as GFF file; must
                        have .gff or .GFF extension
  --bed MOD             Provide contig annotation as BED file; must
                        have .bed or .BED extension
  --blast MOD           Provide contig annotation as BLAST output
                        file (outfmt 6)
  -R ANNO_TYPE [ANNO_TYPE ...], --anno_type ANNO_TYPE
                        [ANNO_TYPE ...]
```

```
                        Provide type(s) of annotations to be
                        included
                        (only applicable on GFF annotation files)
  -RC ANNO_CONT [ANNO_CONT ...], --anno_cont ANNO_CONT
                        [ANNO_CONT ...]
                        Complementary type filter, selected type(s)
                        will be excluded, multiple arguments given
                        space-separated
                        (only applicable on GFF annotation files)

  -RL ANNO_LOWER, --anno_lower ANNO_LOWER
                        Provide lower length limit for annotation
                        features (in bp)
  -RU ANNO_UPPER, --anno_upper ANNO_UPPER
                        Provide upper length limit for annotation
                        features (in bp)
  --heat HEAT           Generate heatmap with given window size (in
                        bp)
  --reg REG REG         Specify region bounds (in bp) for more
                        zoomed-in look; bounds given whitespace-
                        separated i.e. 1000 1500
  --out OUT             Provide desired name of output file
  --outfmt {pdf,svg,png}
                        Select output format, default=png
```

The output of ***SVread.py*** is provided with --*IN* and the same VCF file used with ***SVread.py*** must also be given with --*vcf*, to get names and lengths of all chromosomes. The annotation file with format of choice is entered using the corresponding option (--*gff*, --*bed* or --*blast*). Annotations entered must refer to the reference genome or assembly used during mapping (see 3.3.1). ***SVsee.py*** allows for the selection of annotation features to be shown based on type and size. The options -*R/--annotype* and -*RC/--annocont* correspond to -T and -S of ***SVread.py*** respectively; based on the entered arguments, chosen feature type(s) are included or excluded. -*RL/--anno_lower* and -*RU/--anno_upper* allow to specify the length limits of desired features. While type selection is only available for GFF-formatted annotation data, length restrictions can be set for all data formats. To add a heat map to the graph, --*heat* must be chosen and a window size given (in bp) over which the density of variants is calculated. So choosing --*heat 1000* means that the density of variants in consecutive 1000-bp segments will be shown. The --*reg* flag allows to limit the extent of the chromosome shown by providing start and end

positions (in bp) of the desired region; *--reg 20000000 30000000* would thus restrict the view to the part of the chromosome between (and including) positions 20000000 and 30000000. Annotation, heat map and region restriction are readily combinable. The desired name and format of the output file are specified with *--out* and *--outfmt* respectively.

Accordingly, entering

```
$ python ./SVsee.py --IN OEVEvOLAT_genomic.txt \
                    --vcf OEVEvOLAT_genomic.vcf \
                    --out OEVEvOLAT_gen_pure.png
```

into the terminal will enter the input file *OEVEvOLAT_genomic.txt* produced by ***SVread.py***, the corresponding VCF file *OEVEvOLAT_genomic.vcf* and the GFF-formatted annotation file *OLAT_genomic.gff*. The produced graph will have the default PNG format and the name *OEVEvOLAT_gen_pure.png* (Fig. 6).

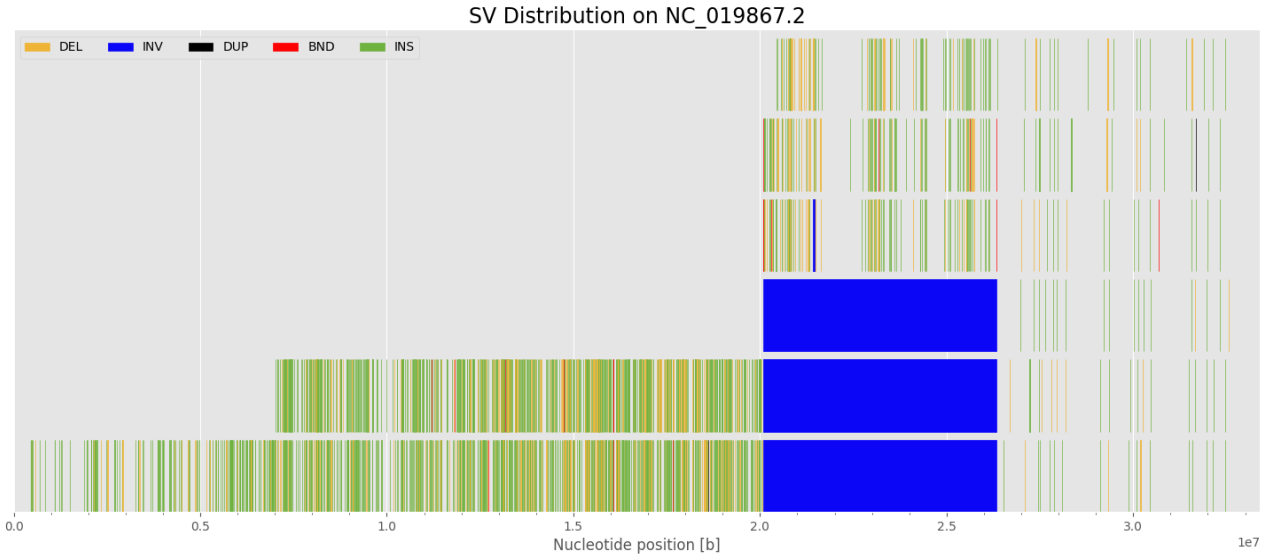

Figure 6: Exemplary plotting of all variants on contig NC_01987.2 of reference genome OLAT_genomic.fna

By providing a GFF annotation file with *--gff*, the figure is expanded by a second graph. The individual graphs share the same x-axis scaling.

Thus,

```
$ python ./SVsee.py --IN OEVEvOLAT_genomic.txt \
                    --vcf OEVEvOLAT_genomic.vcf \
                    --gff OLAT_genomic.gff \
                    --out OEVEvOLAT_gen_anno.png
```

will generate the following figure (Fig. 7).

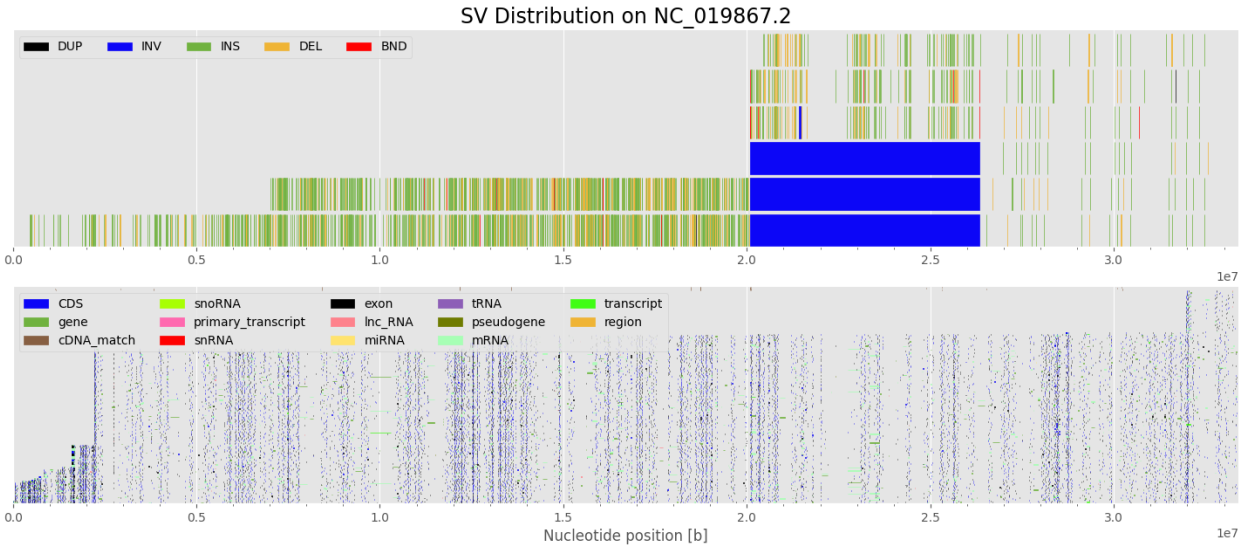

Figure 7: Exemplary variant plotting with annotations from OLAT_genomic.gff

Addition of the '--heat' option with argument 100000 to the will add a heat map of variant density over a window size of 100000 bp. Thus,

```
$ python ./SVsee.py --IN OEVEvOLAT_genomic.txt \
                    --vcf OEVEvOLAT_genomic.vcf \
                    --heat 100000 \
                    --out OEVEvOLAT_gen_heat.png
```

will generate the following figure with name *OEVEvOLAT_gen_heat.png* (Fig. 8)

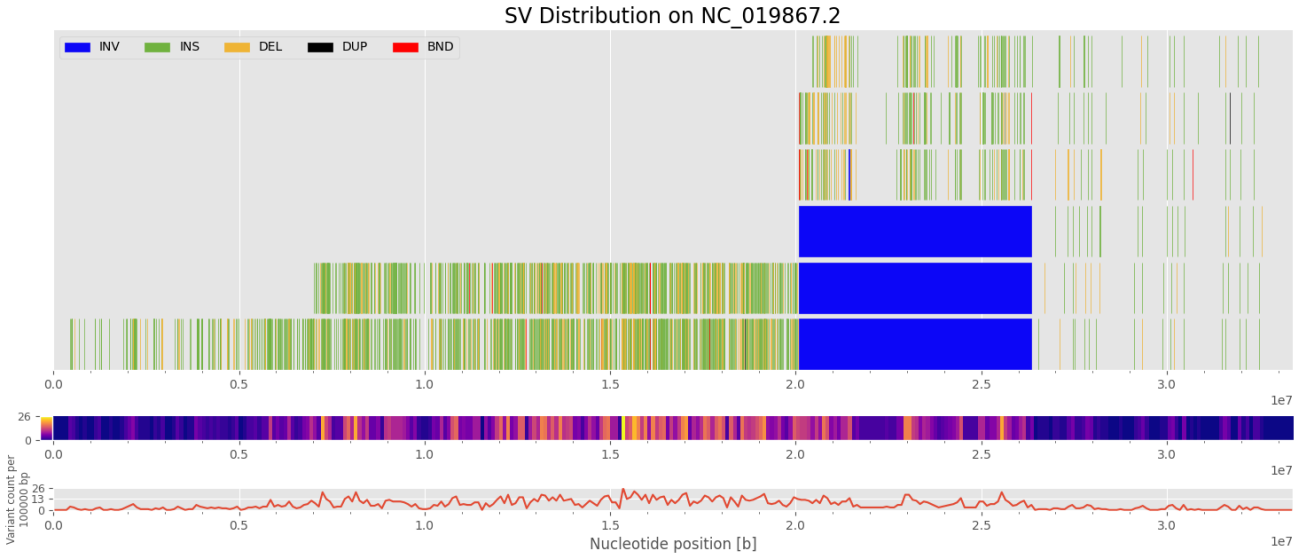

Figure 8: Exemplary variant plotting with heat map representing variant density per 100000 bp. The heat map is supplemented with a graph showing the number of variants and a colorbar.

Since the annotation and heat map features are readily combinable, a compound figure with annotation and heat map graphs (Fig. 9) is easily generated with

```
python tools/SVsee.py --IN OEVEvOLAT_genomic.txt \
                      --vcf OEVEvOLAT_genomic.vcf \
                      --gff OLAT_genomic.gff \
                      --heat 100000 \
                      --out OEVEvOLAT_gen_both.png
```

If a certain region of a given chromosome is of special interest or an interesting distribution of variants wants to be explored further, the bounds of such a region of interest can be specified with *--reg*. The x-axis will retain its length but since less data points are plotted, the horizontal resolution increases. For example, the region between 20 and 30 mb in the previous figures might be of interest. The desired region will be shown in more detail (Fig. 10) by entering

```
python tools/SVsee.py --inn OEVEvOLAT_genomic.txt \
                      --vcf OEVEvOLAT_genomic.vcf \
                      --gff OLAT_genomic.gff \
                      --heat 100000 --reg 20000000 30000000 \
                      --out OEVEvOLAT_gen_both_reg.png
```

Notably, the region restrictions are also applied to all individual graphs in a compound figure.

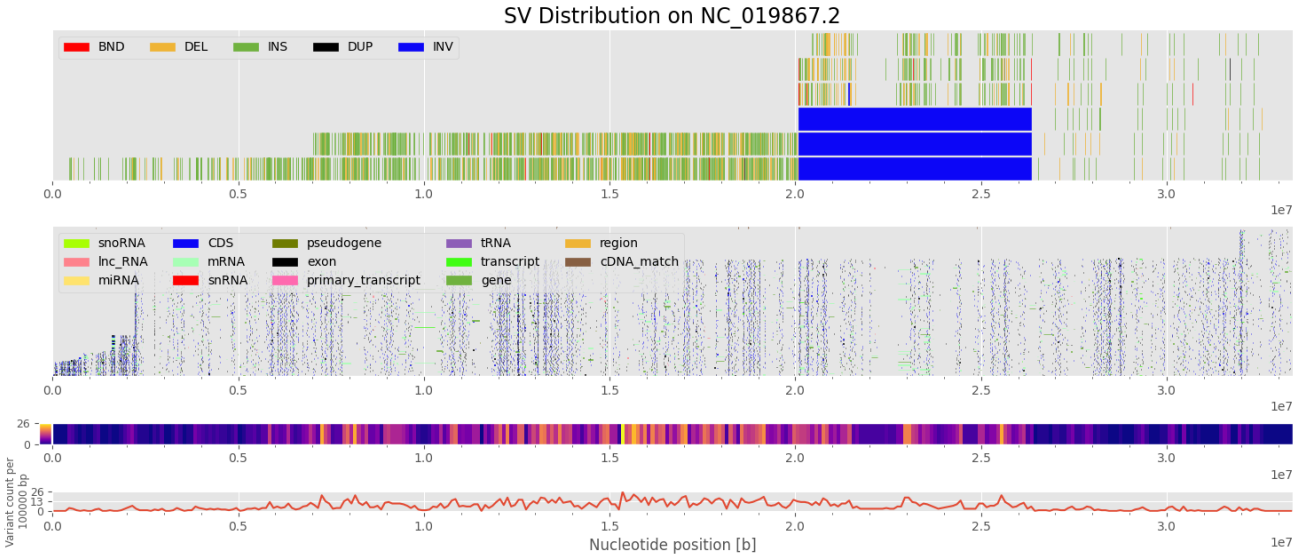

Figure 9: Compound graph with variant plot, annotation plot and heat map graphs for chromosome NC_019867.2

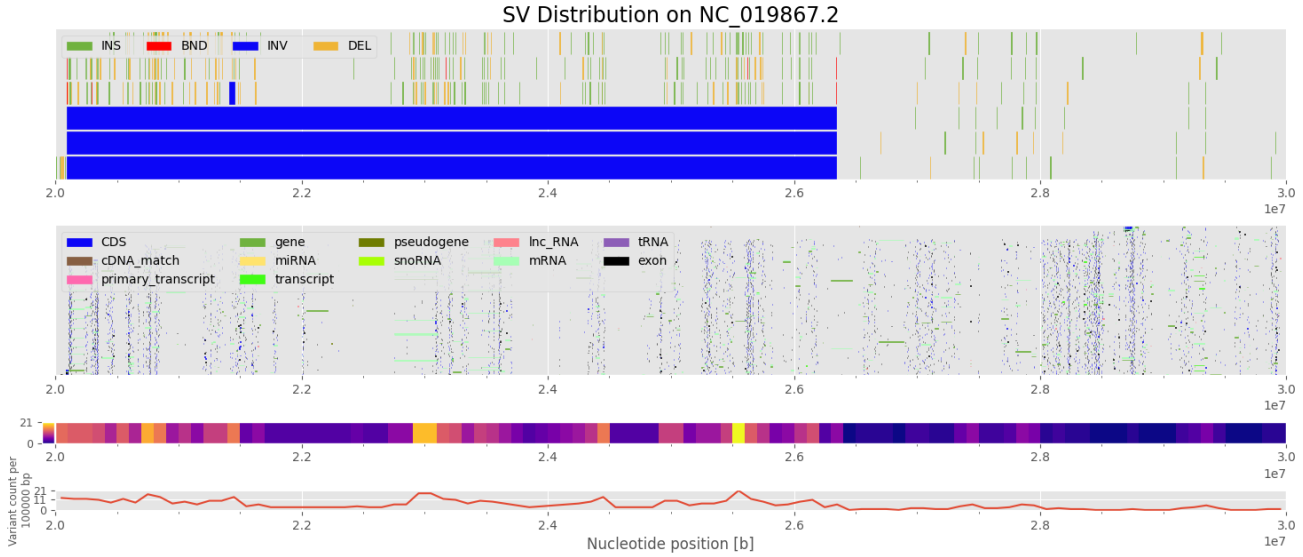

Figure 10: Compound graph with variant plot, annotation plot and heat map graphs for chromosome NC_019867.2 restricted to chromosome region between 20 and 30 mb

SVpipe.py

SVpipe.py is designed as direct path from input VCF file to graphical output and represents the combination of processes taking place in ***SVread.py*** and ***SVsee.py*** respectively. This has the advantages of quicker performance since only a single program must be run and no intermediate data format is required. Consequently, ***SVpipe.py*** also combines the options from the previous programs. Both filtering options for the input VCF file (*-C, -T, -S, --llim* and *--ulim*) and figure elements (*--gff/bed/blast, --heat, --reg*) are handled as well as *--ov* and a means to filter for certain annotation types and lengths (*-R/--annotype, -RC/--annocont, -RL/--anno_lower, -RU/--anno_upper*).

As for the other scripts,

```
$ python ./SVpipe.py -h
```

yields the usage guide:

```
usage: SVpipe.py [-h] [--vcf VCF] [-C CONTIGNAME] [-T TYPE
                 [TYPE ...]] [-S CONTTYPE [CONTTYPE ...]]
                 [--ulim ULIM] [--llim LLIM] [--gff MOD] [--bed MOD]
                 [--blast MOD] [-R ANNOTYPE [ANNOTYPE ...]]
                 [-RC ANNO_CONT [ANNO_CONT ...]] [-RL ANNO_LOWER]
                 [-RU ANNO_UPPER] [--heat HEAT] [--reg REG REG]
                 [--out OUT] [--outfmt {pdf,svg,png}] [--ov]

Full structural variant visualization from raw VCF file to figure

optional arguments:
  -h, --help            show this help message and exit
  --vcf VCF             Provide VCF file
  -C CONTIGNAME, --contigname CONTIGNAME
                        Provide contig/chromosome ID by which to
                        filter SV records; multiple IDs given space-
                        separated; use only one ID for visualization
  -T TYPE [TYPE ...], --type TYPE [TYPE ...]
                        Provide type(s) by which to filter SV
                        records i.e. 'DEL'; multiple types given
                        space-separated; 'BND' for complex
                        rearrangments with breakends
  -S CONTTYPE [CONTTYPE ...], --conttype CONTTYPE [CONTTYPE ...]
```

```
                      Verbose type filter: types entered will be
                      omitted. Same usage as --tfil
--ulim ULIM           Provide upper length limit for SVs [bp]
                      (inclusive)
--llim LLIM           Provide lower length limit for SVs [bp]
                      (inclusive)
--gff MOD             Provide contig annotation as GFF file; must
                      have .gff or .GFF extension
--bed MOD             Provide contig annotation as BED file; must
                      have .bed or .BED extension
--blast MOD           Provide contig annotation as BLAST output
                      file (outfmt 6)
-R ANNOTYPE [ANNOTYPE ...], --annotype ANNOTYPE [ANNOTYPE ...]
                      Provide type(s) of annotations to be
                      included
                      (only applicable on GFF annotation files)
-RC ANNO_CONT [ANNO_CONT ...], --anno_cont ANNO_CONT
                      [ANNO_CONT ...]
                      Complementary type filter, selected type(s)
                      will be excluded, multiple arguments given
                      space-separated
                      (only applicable on GFF annotation files)
-RL ANNO_LOWER, --anno_lower ANNO_LOWER
                      Provide lower length limit for annotation
                      features (in bp)
-RU ANNO_UPPER, --anno_upper ANNO_UPPER
                      Provide upper length limit for annotation
                      features (in bp)
--heat HEAT           Generate heatmap with given window size
                      (in bp)
--reg REG REG         Specify region bounds for more zoomed-in
                      look; bounds given whitespace-separated i.e.
                      1000 1500
--out OUT             Provide desired name of graphic output
--outfmt {pdf,svg,png}
                      Select output format, default=png
--ov                  Choose to display an overview of SVs in the
                      provided VCF file; --ov only will print
                      assembly-level information; using --C option
                      will print single-contig-level information
```

3.3.3 Graphical user interface

The visual nature of the desired final result and the trial-and-error process involved in the exploration of the visual data makes a GUI inherently useful. Using the GUI, a user may test and optimize the parameters and extent of data to be visualized in a more interactive fashion. As a result, only the optimal figure for given applications must be saved, unlike the trial-and-error process with the command-line scripts. The GUI script is available for graphics-capable terminals (such as Anaconda Prompt). All required files should be saved in the same folder. The program is started by calling

```
$ python ./SVgui.py
```

This will open the main window (Fig. 11), the basic interface from which all parameters are set and files are loaded. The menu bar (Fig. 11, red) provides the most basic functionality: pressing *Load* will open a drop-down menu with the options *Variant file* and *Annotation file* with which the respective files can be loaded, as well as *Unload annotation file* to remove previously loaded annotation data. Using *Close* is recommended when using the program on a terminal; this makes sure that tkinter and associated windows are closed and destroyed so that the terminal does not remain unresponsive. Below the menu bar is the text window (Fig. 11, orange). On it, overview information and progress

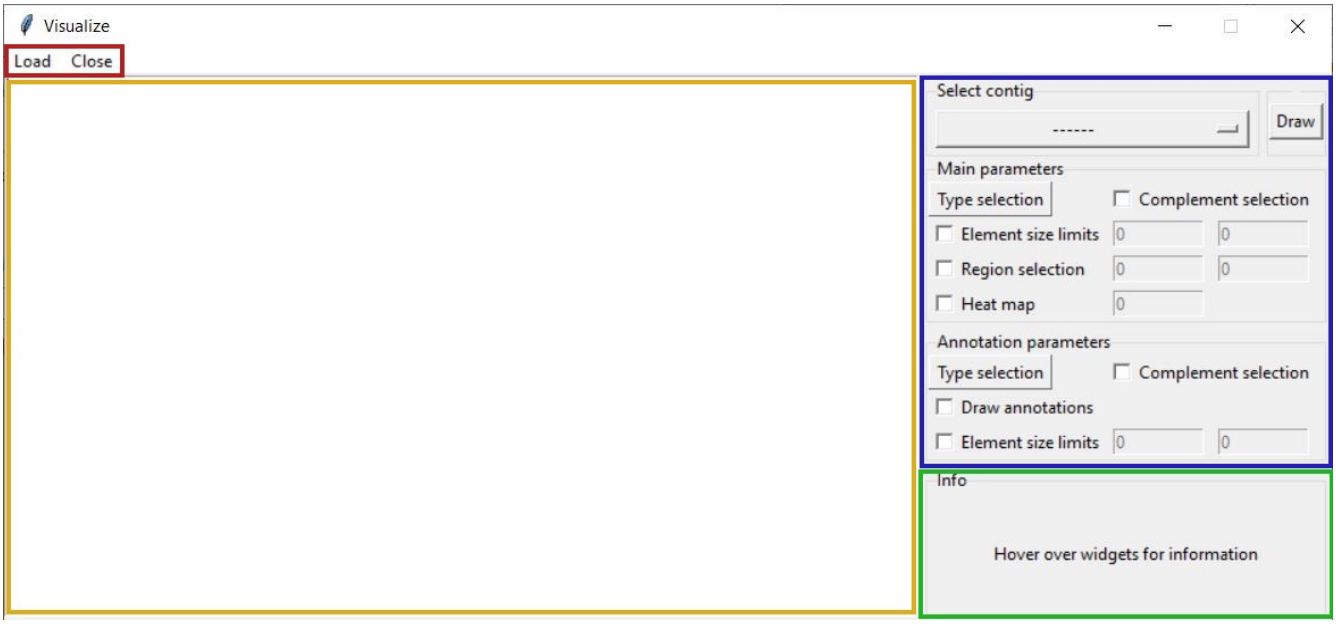

*Figure 11: Main window of **SVgui***

notifications will be shown when certain steps are taken. To the right, the control panel (Fig. 11, blue) includes all the means to change the various visualization parameters.

The *Select Contig* panel has a drop-down menu from which the chromosome/contig of interest can be chosen and that is updated dynamically as files are loaded. The choice from this drop-down menu is equivalent to that made using *-C* for ***SVsee.py*** and ***SVpipe.py***. The *Main parameters* and *Annotation parameters* panels below allow to intuitively change the respective parameters. The *Type selection* drop-down menus of both panels are updated dynamically based on the chromosome chosen and work by simply clicking on a type to select. Clicking on a selected type will unselect it. Checking *Complement selection* will result in all types being included except for those currently selected under *Type selection*. Note that when an annotation file has been loaded, annotations will be automatically included in the produced figure. To disable this, uncheck *Draw annotations*. Clicking *Draw* will trigger the figure generation and open a new window to display the figure. Finally, the main window features an info box (Fig. 11, green) which displays usage information for the buttons in the control panel as well as notifications in cases of incorrect usage.

Figure 12: Contig selection in ***SVgui.py***

To demonstrate the steps towards a figure with ***SVgui.py***, the essential steps are illustrated below.

As mentioned above, using the *Load* menu a VCF file and, if desired, an annotation file can be loaded into ***SVgui.py***. Successful loading is confirmed with a notice in the text window (Fig. 12). The number of contigs found in the VCF file is also noted. Depending on the size of the provided file, this step may be time consuming. The next step is to select a contig from the *Select contig* drop-down menu. Contigs will be listed according to descending size, with the number of variants they contain shown in parentheses (Fig. 12). Once a choice has been made, ***SVgui.py*** will summarize the number of variants in total and by type. The total number of annotations for the chosen contig is also displayed (Fig. 13). With the selection of a contig, the respective type selection choices are updated.

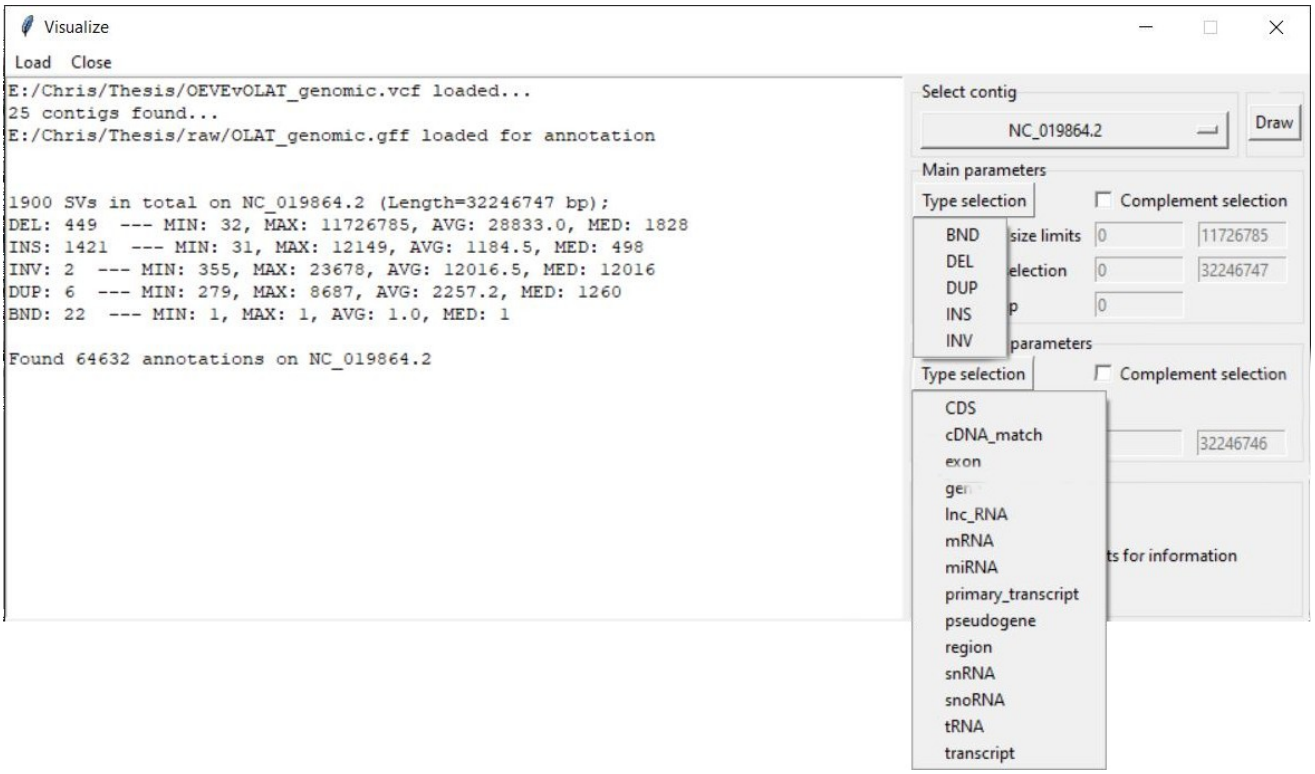

*Figure 13: Overview information and type selection possibilities in **SVgui.py** after contig selection*

The entry fields for the element size limits for variants and annotations respectively and region selection are updated as well to reflect the maximum values found for each parameter. The parameters of interest can be adapted to the user's liking and type selections made. For example, the variant size limits may be set to 100 and 10000 respective and all types but 'BND' are desired (Fig. 14). A heat map

with window size of 10000 bp may also be added and all annotation types but ‘region’ are to be included. Clicking *Draw* will then apply all these parameters to the variant and annotation feature lists and show their numbers in the text window. The produced figure will appear in a new window (Fig. 15). The figure window features the figure along with a tool bar. The buttons of the tool bar allow for interactive exploration of the figure. The *Pan* button (Fig. 15, blue) activates the panning cursor with which the axes can be dragged. To deactivate the cursor function, the button must be clicked again. Zooming can be done after clicking the *Zoom* button (Fig. 15, yellow). With the zooming cursor a

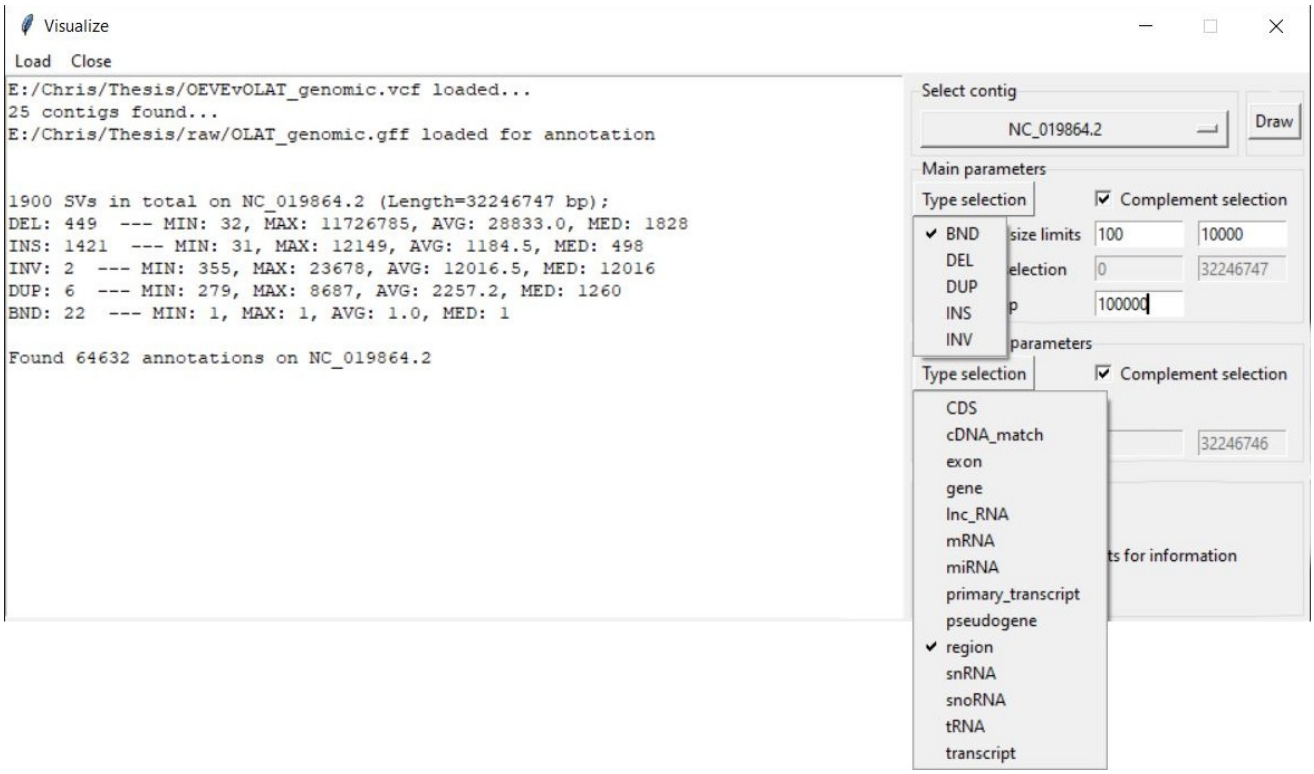

Figure 14: Exemplary parameter selection in SVgui.py

rectangular selection is made. All graphs will then be zoomed to the region of the selection and axes updated, regardless of which graph the zoom took place in. Zooming is very useful if the number of annotations is very large and individual annotation features become poorly distinguishable. Depending on the amount of variants and annotation features, panning and zooming can be very slow and ***SVgui.py*** may become unresponsive until the operations have been carried out. The *Reset View* (Fig. 15, green) will undo all previous zooming and panning operations. In case a GFF annotation file was provided for annotation, it is also possible to extract information for specific annotation features

directly from the figure. Without any special cursor equipped, clicking on a feature will spawn another window with information for that feature corresponding to the ‘attributes’ column of the provided GFF file (Fig. 16). The VCF and GFF file names are noted, as well as the chromosome selected, to allow for some traceability of the information (Fig. 16, red). The starting position, length and attribute information is displayed for each selected annotation feature (Fig. 16, green/blue). This window will remain open until closed and clicking further features will add their information to the window below the previously added attributes. The window contents can be saved as a single plain text file. The figure can finally be saved with the *Save figure* button (Fig. 15, red) as a large variety of formats including PNG, JPG, PDF, SVG and TIFF.

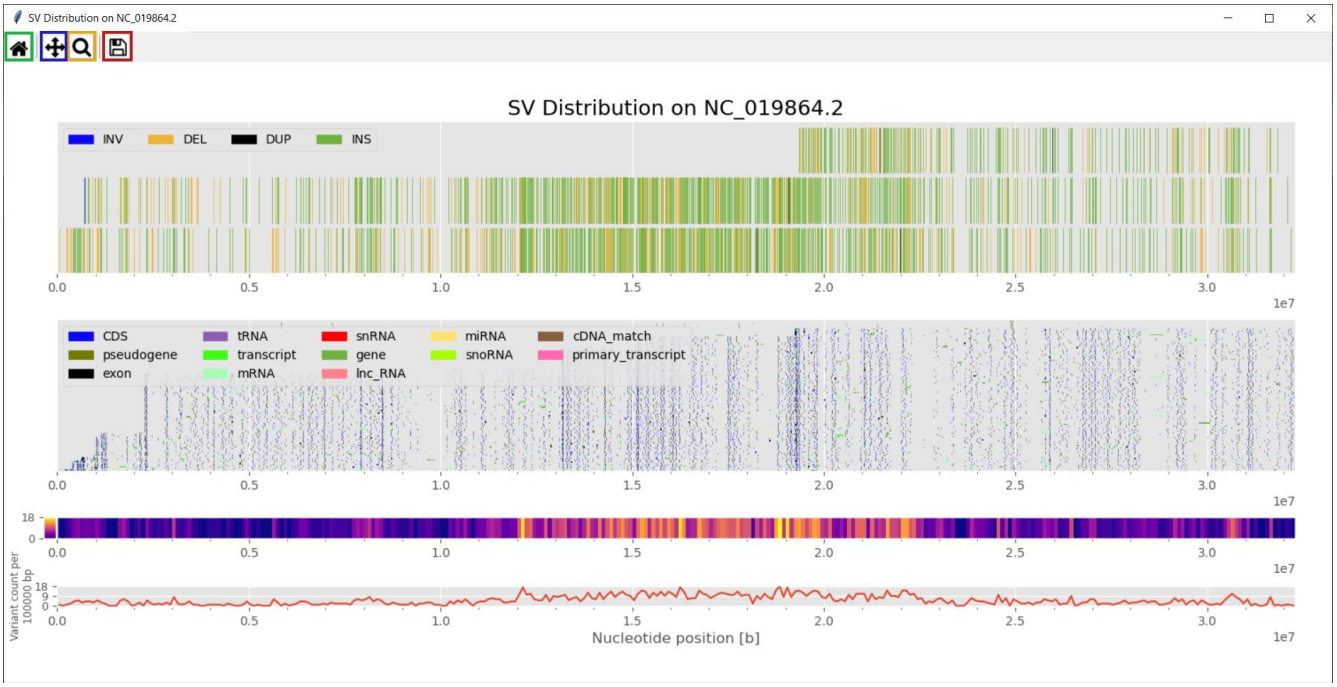

Figure 15: Figure window generated by SVgui.py for variants and annotations on contig NC_019864.2

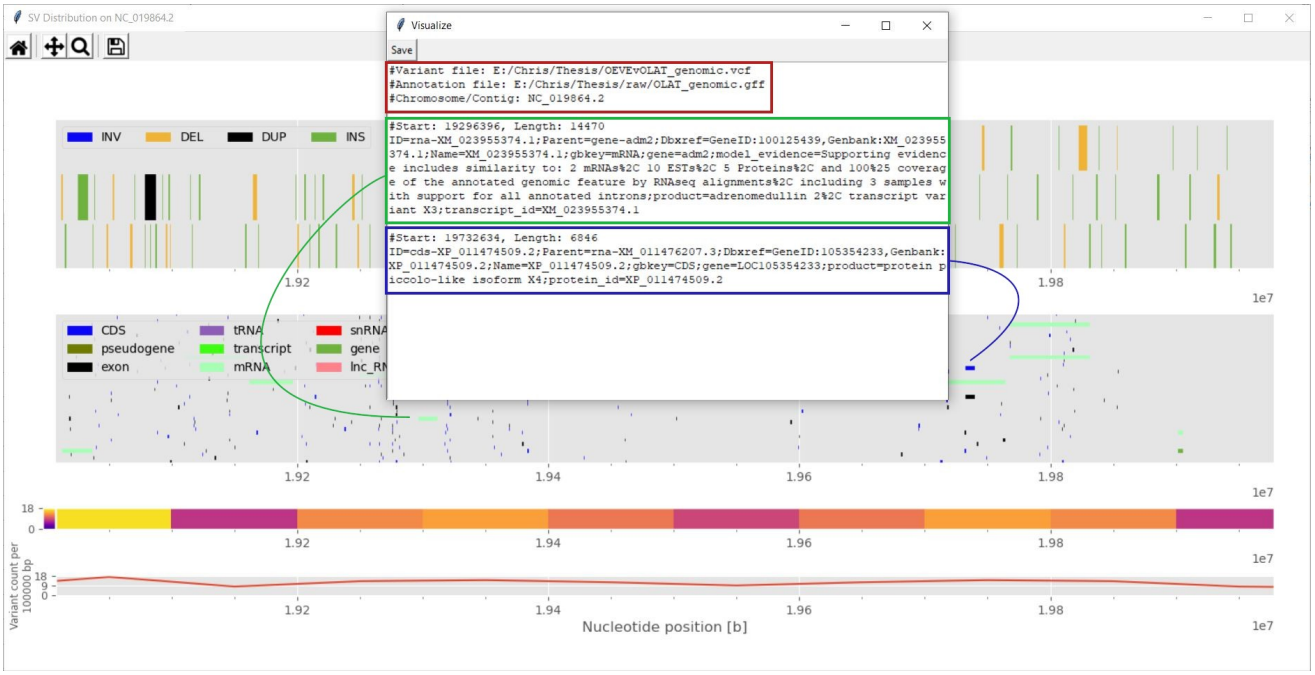

Figure 16: Individual annotation feature exploration

4 Results

To demonstrate the usage and functionality of the programs presented above, an exemplary investigation into the presence of structural variants in *Oryzias eversi* was undertaken. A wild type *O. eversi* assembly (‘OEVEW1_p1p2_csa.fasta’) served as reference against which an assembly of a laboratory-bred strain of *O. eversi* was assembled (‘OEVE_p1p2_csa.fasta’), using the variant-mapping script (3.5.). The OEVE strain has been bred in isolation for more than ten generations. To see to which extent a separation over a few generations affects the presence of structural variants within a species might give hints to their importance in speciation events and at which speed they accumulate. The four largest contigs were examined to gain a general overview of variant distribution within the assemblies. Of main interest were deletions and insertions between 500 and 10000 bp in length. Another interest was to investigate the presence of a recently found transposon belonging to the Tc1/*mariner* superfamily, dubbed TC1-celeb. Members of this superfamily have been found in fungi, plants, animals, among others, and most likely have a common origin. A single gene encoding a transposase is flanked by terminal inverted repeats, accounting for a total length of 1300 to 2400 bp. The terminal repeats range in length between less than 100 bp and more than 400 bp, but always contain at least one

binding site for the transposase. The transposase mediates a cut-and-paste transposition, allowing the transposon to move within the genome (Plasterk, 1999). The sequence of the TC1-celeb was blasted (blastn) against the OEVEW1 assembly, and the output used as annotation to the OEVE-OEVEW1 VCF file. The used sequence has a length of 1633 bp. Blasting the transposon against itself shows that it has a 5' repeat of 221 bases and a 3' repeat of 214 bases.

The presence of transposon hits and SVs was checked for in the four largest contigs of the OEVEW1 assembly. R6_7_pilon_pilon was the largest contig with a length of 66,6 mb. Deletions (DEL) and Insertions (INS) were distributed fairly regularly over the full length of the contig, with a higher density of variants in the first 2 mb (10 variants/mb) (Fig. 17).

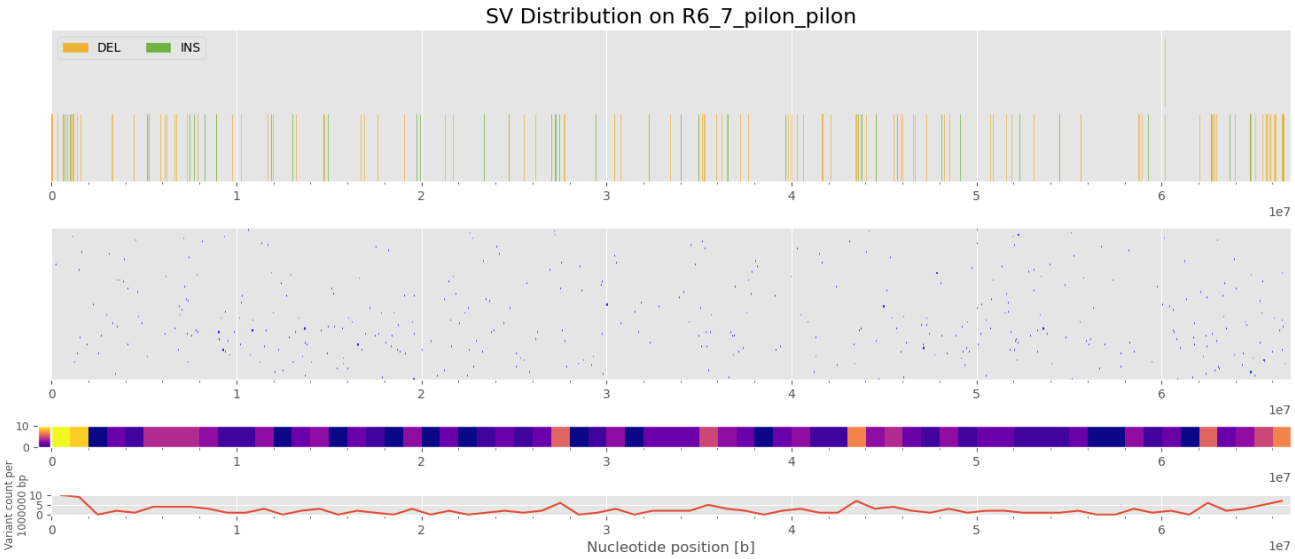

Figure 17: Variant overview of R6_7_pilon_pilon of OEVEW1. Variants restricted to deletions and insertions between 500 and 10000 bp in length, annotating blast hits limited to between 1300 and 2400 bp in length. Heat map with 1 mb increments.

The number of DELs and INSs appeared to be similar with neither outweighing the other; 153 variants were found. 771 variants were found in total, without any prior filtering. Blast hits of the transposon sequence were distributed similarly regularly along the contig, with 1524 hits in total. With the used length restriction of 1300 to 2400 b, 387 hits were found. Hits of this length presumably span the entire transposon length. 870 hits were found to correspond to terminal repeats based on length

restrictions to 200 to 250 b (Fig. 18). Summing the lengths of all hits (without length restrictions) amounted to 914573 b, corresponding to roughly 1.4% of the total length of R6_7_pilon_pilon.

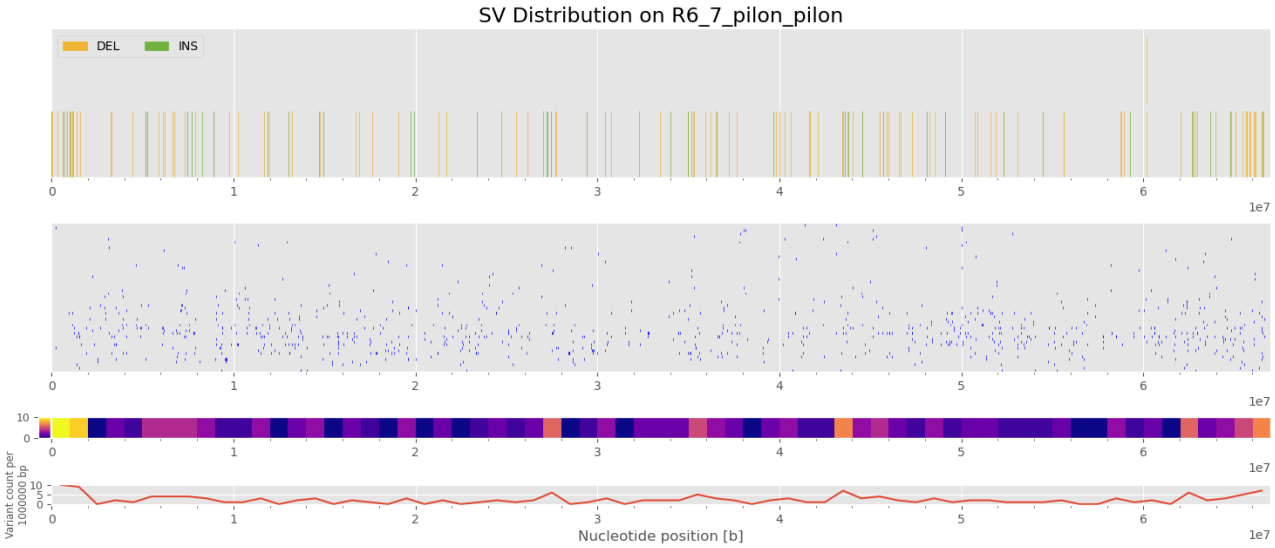

Figure 18: Variant overview of R6_7_pilon_pilon of OEVEW1. Variants restricted to deletions and insertions between 500 and 10000 bp, annotating blast hits limited to between 200 and 250 bp in length. Heat map with 1 mb increments.

The next longest contig was R7_3_pilon_pilon with a length of 64.7 mb. Total variant calls and BLAST hits were fewer than for R6_7_pilon_pilon, 734 and 1268 respective. However, more length-restricted variant calls were found (Table 1). Both short and long BLAST hits were less numerous. With exception of hotspots at 39 and 63 mb, variants were spread evenly along the contig, as were the blast hits. Total BLAST hits accounted for only 1.2% of contig length.

R7_3_pilon_pilon was followed in length by contig R7_1_pilon_pilon with a length of 62.7 mb. Although this contig was shorter than the previous ones by at least 2 mb, it featured a higher number of variants; 933 in total. After the application of the filters 154 variants remained, barely more than for R6_7_pilon_pilon (153) and less than for R7_3_pilon_pilon (162). A local peak in variant density was seen at 42 mb with 10/mb. 1315 total blast hits were found, with 332 being 1300 to 2400 b in size and 741 with a length between 200 and 250 bp. Total blast hit lengths corresponded to 1.2% of contig length.

The last of the four largest contigs was R6_11_pilon_pilon and had a length of 60.2 mb. On it, a total of 551 variants were found, as well as 1371 blast hits. Under the filters, 137 variants remained. DELs appeared to outnumber INSs, but both were spread regularly along the contig. Density was the highest at 31 mb with 7/mb. When restricted to 1300 to 2400 b length, 340 blast hits remained; 766 with 200 to 250 b size. Cumulative hit lengths amounted to 1.4% of contig length.

Overall, the number and distribution of variants along the respective contigs appeared to be uniform and with little difference between them. Total number of variants of interest ranged from 137 (R6_11_pilon_pilon) to 162 (R7_3_pilon), with little to no influence from contig length (Table 1). All contigs featured at least one region with a higher density of variants than the surrounding parts of the contig, with between 7/mb and 11/mb. As with the variants, the blast hits for the Tc1/*mariner* transposon were similar in number between the contigs. In all cases, the shorter blast hits (200-250 bp) outnumbered the longer ones (1300-2400 bp) by more than double. This was to be expected since each full transposon features two ITR that each get a hit, along with some ‘random’ hits. Still, the presence of this transposon in the wild type *O. eversi* appears widespread, at least for the presented contigs. The cumulative length of all long hits, if assumed to correspond to the full transposon, accounted for 0.89% of the genetic material across all four contigs. Due to the novel nature of the transposon, its contribution to the transposon distribution in the genome of *O. eversi* cannot be estimated. It is noteworthy, however, that a high occurence of Tc1/*mariner*-type transposons has been found in *O. latipes*, accounting for 2.91% of its genome, with seven identified families (Gao, 2017).

Table 1: Variant call and BLAST hit numbers for the four largest contigs of the OEVEW1 assembly

	Length [mb]	Total variant calls	Length-restricted variant calls	Total BLAST hits	Short BLAST hits	Long BLAST hits	BLAST hit proportion [%]
R6_7_pilon_pilon	66.6	771	153	1524	870	387	1.4
R7_3_pilon_pilon	64.7	734	162	1268	734	328	1.2
R7_1_pilon_pilon	62.7	933	154	1315	741	332	1.2
R6_11_pilon_pilon	60.2	551	137	1371	766	349	1.4

Length-restricted variant calls include deletions and insertions between 500 and 10000 bp in length only. Short BLAST hits are between 200 and 250 bp in length, long ones between 1300 and 2400 bp. BLAST hit proportion refers to how much of the contig's sequence was associated with BLAST hits and was calculated by dividing the cumulative length of all hits by the contig length.

In case of a similar situation in *O. eversi* and assuming that the found genome coverage for the contigs is representative, TC1-celeb might be a major contributor to Tc1/*mariner* transposon presence in *O. eversi*.

5 Discussion

The programs presented here have been designed as local desktop software to aid in the visualization of genomic data and associated annotations so as to give the user an overview of the annotation landscape of the investigated genome or assembly. They might be employed as first step in the visual exploration of newly sequenced assemblies to find contigs and larger regions within them that might be of interest. Deeper analyses would then likely be continued with more specialized tools, such as the ones described earlier (see 2.2). Still, the presented scripts allow to effectively visualize large amounts of data intuitively and give users a number of options for adjusting and tailoring what is to be shown in the produced figures. The example investigation into structural variants and transposons in ricefishes – albeit very brief – demonstrates the capabilities of the presented programs (see 4). Comparisons of both raw and visual data facilitate recognizing data points and trends of interest over entire contigs or small regions. This can be further expedited with heat maps as well as panning and zooming through graphs.

Depending on the size of the sequenced genome, or rather of the assembly of interest and the accompanying annotations, several gigabytes of data are to be handled by a chosen visualization program. The locally-executed nature of the here presented software – ***SVgui.py*** especially – comes with a challenges in that regard. For one, it requires that all input files, be it the assembly or annotations, must be available locally. Secondly, this means that the computational power that the program(s) have access to is limited by the user's computer. The user experience is accordingly very dependent on the power of the used computer and the available memory. The other tools considered above – to some extent – make use of the internet. All three are available as web-based applications, only IGV additionally allows for download of a desktop application. As web applications, these programs are not limited by user hardware and have access to a plethora of databases, ftp and http servers, while also allowing cooperative usage (Thorvaldsdottir *et al*., 2013; Dunn *et al*., 2019; Kent *et al*., 2002).

As only other desktop application, IGV can be more directly compared to the presented software, especially ***SVgui.py***. IGV's software architecture consists of three main layers (Thorvaldsdottir *et al*., 2013). The 'Application Layer' contains the user interface while also handling rendering of data into tracks that can then be readily combined. The application layer interacts with the underlying 'Data Layer' to demand 'tiles'. These tiles are part of pyramidally organized data and were developed to minimize memory demand while maintaining performance (Thorvaldsdottir *et al*., 2013). Atop this pyramid is a whole-genome tile, followed by an individual tile for each chromosome. Per chromosome then follows a series of exponentially growing number of tiles, as the chromosome is divided into two regions and each region divided into two further regions and so on. The genomic data corresponding to each tile is uniformly summarized and thus tile sizes do not vary. Only the relevant tiles for the current view of the genome are demanded and rendered by the application layer. The tiles are computed by the data layer from the files delivered to it by the underlying 'Stream Layer' (Thorvaldsdottir *et al*., 2013). ***SVgui.py*** uses an arguably similar yet much more basic approach of first reading the data, then processing and finally rendering it. Unlike IGV, the used data is read entirely into memory although the sequence information of the genome/assembly in question is not used. Rather, only the lengths of individual chromosomes/contigs are noted. Each structural variant of the provided VCF file is read into a memory as its own object and all variant objects are stored and passed through several containers. This approach is the same for any annotations provided. As the user interacts with the interface, these containers are manipulated and supplemented with information relevant for rendering. Thus, the memory burden is quite high. As a result, performance can be low with large VCF and annotation files and little RAM. This is also reflected in the interactivity with the rendered graph. Panning and zooming may be slow and even lead to screen freezes, limiting interactivity. Addition of further features such as including sequence data from a reference genome and/or aligned reads is not sensible for the same reason, although helpful in terms of utility. Use of powerful hardware would somewhat avoid this problem. Still, restricting the resources available to the program to those of the local desktop generally greatly limits possibilities. Thus, a restructured software architecture that does not require as much memory and extending it to make us of web resources would improve performance, utility and interactivity of ***SVgui.py***.

All of the presented applications, including ***SVgui.py,*** visualize the genomic and associated data as horizontal tracks. Given the linear nature of genomes and genetic sequences, this is arguably the

most intuitive way. Different data for the same coordinates lie parallelly to facilitate comparison. In the case of ***SVgui.py***, variant and annotation features are additionally shown staggered within the respective tracks in order to provide a quicker overview of all features at once even from the whole-chromosome view. Otherwise, features with little space in between might overlap and appear as one. This is also the case when taking a zoomed-in look at variants in the UCSC Genome Browser and IGV. Depending on the number and nature of variants shown, individual variants may be shown one per line and/or with an annotated ID and further information corresponding to the columns of a VCF file. Apart from this similarity in visualization approach, the Genome Browser and IGV allow a greater number of additional tracks to be shown at a time, with a greater variety of information encoded. Yet, part of the premise for ***SVgui.py*** was to be variant-oriented; data to be added in additional tracks should be relevant to this focus, like SNP coordinates. Program functionality could also be extended by integrating some of the required data processing steps. For example, the steps handled by the variant calling script (Appendix 1) could be executed directly by ***SVpipe.py***. However, since the produced VCF file can be used for many other applications and because the associated processing time negates the idea of a ‘quick and easy visualization’, this might increase usefulness only to a limited extent.

SVgui.py was chosen to be written in Python for its intuitive syntax and object-oriented nature. Uniting all relevant data for variant and annotation features into designated object structures greatly streamlines and facilitates any sort of manipulation and processing of said data. OOP also improves code readability and reusability. Although Python may not be as fast on runtime as compiled languages like C++, the more intuitive, readable and thus more easily maintainable code of Python outweighs the performance drawbacks. This is especially true for programs that might be released as open source software and thus see modifications from users, as is oftentimes the case for bioinformatics tools. Like Python, Java is an object-oriented programming language used to develop programs for bioinformatic application. A great advantage of Java is its independence of the user’s operating system; Java code is first compiled and then executed by Java Runtime Environment which is available for all operating systems. With bioinformatics-specific libraries like BioJava, common and specific problems can be addressed equally (Lafita *et al*., 2019). Thus it is not surprising that both IGV and Apollo have been developed using Java. Still, the popularity of both Python and Java in the bioinformatics community suggests neither is more universally useful than the other.

Apart from the real-time interactive approach in the form of ***SVgui.py***, the programs presented here also allow for command line-based processing and visualization of variant data. A graphical user interface and immediate interactivity certainly facilitate data exploration and are arguably the most intuitive way to effectively peruse data sets. Still, providing command line utilities may nonetheless be useful. In the case a user depends on a server lacking visualization capabilities, data can still be rendered on the server and viewed on the desktop. Optimizing parameters then requires a trial-and-error approach. While ***SVpipe.py*** acts as a more direct way from data to image, the combination of ***SVread.py*** and ***SVsee.py*** leaves more room for potential use case specific manipulation of data before visualization.

In conclusion, the programs presented here allow for the visualization of genomic variation and annotation data, both on command line and using a GUI. Visualization requires a VCF file encoding SV or SNP data and annotation data in the form of a GFF, BED or blast output file. Both command line and GUI versions allow for interaction with the data via a number of filtering parameters. Comparison with existing prevalent genome visualization programs shows the need for optimization and extension of functionality to some extent. However, the intended purpose of providing quick and simple means of visualizing variant data can still be fulfilled by the suite of programs presented. They could be a considered a specific solution rather than a broad-range tool like IGV or the UCSC Genome Browser for example. Still, as the potential and contribution of structural variation to overall genetic variation becomes more apparent, the importance of tools to explore and visually communicate such information will grow.

6 Summary

Structural variation has proven to have a considerable influence on intra- and interspecific differences. Balanced and unbalanced structural variants (SV) can alter gene copy numbers, manipulate the topology of regulatory regions or gene product structure, and thereby affect gene expression and the phenotype. Increasing sequencing and variant calling efforts have uncovered the widespread presence of these structural variants and have revealed the need for visualization tools to allow for effective exploration and communication of variation data. Here I present a suite of Python-based tools aimed to visualize structural variant calls in the context of single assembly contigs. This suite includes separate command line scripts for processing and rendering of VCF-formatted data as well as more succinct ‘one-step’ programs that bundle all functionality. Of the programs, ***SVgui.py*** provides the greatest extent of functionality as its graphical user interface allows for intuitive interactivity with the displayed data. Further, annotation data can be provided and visualized alongside variant call data to investigate possible intersection or coappearence of sequence features and variation. With this functionality, ***SVgui.py*** and the other programs allow for simple and intuitive exploration of variant calls without an overbearing number of features. Like in the exemplary investigation of SVs in the ricefish *Oryzias eversi*, a typical use case might be to get an overview of variation within a recently sequenced genome. Thus, this suite of programs may aid in efforts to characterize and investigate structural variation.

7 References

Catanach, Andrew; Crowhurst, Ross; Deng, Cecilia; David, Charles; Bernatchez, Louis; Wellenreuther, Maren (2019): The genomic pool of standing structural variation outnumbers single nucleotide polymorphism by threefold in the marine teleost Chrysophrys auratus. In *Molecular ecology* 28 (6), pp. 1210–1223. DOI: 10.1111/mec.15051.

Danecek, Petr; Auton, Adam; Abecasis, Goncalo; Albers, Cornelis A.; Banks, Eric; DePristo, Mark A. et al. (2011): The variant call format and VCFtools. In *Bioinformatics (Oxford, England)* 27 (15), pp. 2156–2158. DOI: 10.1093/bioinformatics/btr330.

Dunn, Nathan A.; Unni, Deepak R.; Diesh, Colin; Munoz-Torres, Monica; Harris, Nomi L.; Yao, Eric et al. (2019): Apollo: Democratizing genome annotation. In *PLoS computational biology* 15 (2), e1006790. DOI: 10.1371/journal.pcbi.1006790.

Feuk, Lars; Carson, Andrew R.; Scherer, Stephen W. (2006): Structural variation in the human genome. In *Nature reviews. Genetics* 7 (2), pp. 85–97. DOI: 10.1038/nrg1767.

Feulner, Philine G. D.; Chain, Frédéric J. J.; Panchal, Mahesh; Eizaguirre, Christophe; Kalbe, Martin; Lenz, Tobias L. et al. (2013): Genome-wide patterns of standing genetic variation in a marine population of three-spined sticklebacks. In *Molecular ecology* 22 (3), pp. 635–649. DOI: 10.1111/j.1365-294X.2012.05680.x.

Gao, Bo; Chen, Wei; Shen, Dan; Wang, Saisai; Chen, Cai; Zhang, Li et al. (2017): Characterization of autonomous families of Tc1/mariner transposons in neoteleost genomes. In *Marine genomics* 34, pp. 67–77. DOI: 10.1016/j.margen.2017.05.003.

Harewood, Louise; Chaignat, Evelyne; Reymond, Alexandre (2012): Structural variation and its effect on expression. In *Methods in molecular biology (Clifton, N.J.)* 838, pp. 173–186. DOI: 10.1007/978-1-61779-507-7_8.

Hunter, John D. (2007): Matplotlib: A 2D Graphics Environment. In *Comput. Sci. Eng.* 9 (3), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

Kent, W. James; Sugnet, Charles W.; Furey, Terrence S.; Roskin, Krishna M.; Pringle, Tom H.; Zahler, Alan M.; Haussler, David (2002): The human genome browser at UCSC. In *Genome research* 12 (6), pp. 996–1006. DOI: 10.1101/gr.229102.

Krzywinski, Martin; Schein, Jacqueline; Birol, Inanç; Connors, Joseph; Gascoyne, Randy; Horsman, Doug et al. (2009): Circos: an information aesthetic for comparative genomics. In *Genome research* 19 (9), pp. 1639–1645. DOI: 10.1101/gr.092759.109.

Lafita, Aleix; Bliven, Spencer; Prlić, Andreas; Guzenko, Dmytro; Rose, Peter W.; Bradley, Anthony et al. (2019): BioJava 5: A community driven open-source bioinformatics library. In *PLoS computational biology* 15 (2), e1006791. DOI: 10.1371/journal.pcbi.1006791.

Maloy, Stanley; Hughes, Kelly (Eds.) (2013): Brenner's Encyclopedia of Genetics. 2nd ed.: Elsevier.

McKinney, Wes (2017): Python for data analysis. Data wrangling with Pandas, NumPy, and IPython. Second edition. Sebastopol, CA: O'Reilly Media. Available online at http://proquest.tech.safaribooksonline.de/9781491957653.

Mérot, Claire; Oomen, Rebekah A.; Tigano, Anna; Wellenreuther, Maren (2020): A Roadmap for Understanding the Evolutionary Significance of Structural Genomic Variation. In *Trends in ecology & evolution* 35 (7), pp. 561–572. DOI: 10.1016/j.tree.2020.03.002.

Parenti, Lynne R. (2008): A phylogenetic analysis and taxonomic revision of ricefishes, Oryzias and relatives (Beloniformes, Adrianichthyidae). In *Zoological Journal of the Linnean Society* 154 (3), pp. 494–610. DOI: 10.1111/j.1096-3642.2008.00417.x.

Penso-Dolfin, Luca; Man, Angela; Mehta, Tarang; Haerty, Wilfried; Di Palma, Federica (2020): Analysis of structural variants in four African cichlids highlights an association with developmental and immune related genes. In *BMC evolutionary biology* 20 (1), p. 69. DOI: 10.1186/s12862-020-01629-0.

Plasterk, Ronald H.A; Izsvák, Zsuzsanna; Ivics, Zoltán (1999): Resident aliens: the Tc1/ mariner superfamily of transposable elements. In *Trends in Genetics* 15 (8), pp. 326–332. DOI: 10.1016/s0168-9525(99)01777-1.

Python Software Foundation (Ed.): What is Python? Executive Summary. Available online at https://www.python.org/doc/essays/blurb/, checked on 11/14/2020.

Quinlan, Aaron R.; Hall, Ira M. (2010): BEDTools: a flexible suite of utilities for comparing genomic features. In *Bioinformatics (Oxford, England)* 26 (6), pp. 841–842. DOI: 10.1093/bioinformatics/btq033.

Sankoff, David; Nadeau, Joseph H. (1996): Conserved synteny as a measure of genomic distance. In *Discrete Applied Mathematics* 71 (1-3), pp. 247–257. DOI: 10.1016/S0166-218X(96)00067-4.

Sedlazeck, Fritz J.; Rescheneder, Philipp; Smolka, Moritz; Fang, Han; Nattestad, Maria; Haeseler, Arndt von; Schatz, Michael C. (2018): Accurate detection of complex structural variations using single-molecule sequencing. In *Nature methods* 15 (6), pp. 461–468. DOI: 10.1038/s41592-018-0001-7.

Stein, Lincoln (2020): Generic Feature Format Version 3 (GFF3) (Version 1.26). Available online at https://github.com/The-Sequence-Ontology/Specifications/blob/master/gff3.md, updated on 8/18/2020, checked on 11/3/2020.

Stein, Nils (2013): Synteny (Syntenic Genes). In Stanley Maloy, Kelly Hughes (Eds.): Brenner's Encyclopedia of Genetics, vol. 4. 2nd ed.: Elsevier, pp. 623–626.

Thorvaldsdóttir, Helga; Robinson, James T.; Mesirov, Jill P. (2013): Integrative Genomics Viewer (IGV): high-performance genomics data visualization and exploration. In *Briefings in bioinformatics* 14 (2), pp. 178–192. DOI: 10.1093/bib/bbs017.

8 Appendix

```
#!/bin/bash

#assign in/out and params
infasta=../raw_data/OEVEW1_p1p2_csa.fasta
windowsize=10000
stepsize=500
outbed=../OEVEW1_p1p2_csa.fasta.bed
outfasta=../OEVEW1_p1p2_csa.LR
reference=../raw_data/OEVE_p1p2_csa.fasta
outtemp=../OEVEW1vOEVE


#index infasta file
samtools faidx $infasta

#generate bedfile for simulated long reads
python makebed.py --fi $infasta.fai \
                  --sz $windowsize \
                  --st $stepsize \
                  --bed $outbed

#use bedtools to generate long read fasta file
    w/ bed file and infasta
bedtools getfasta -fi $infasta -bed $outbed -fo $outfasta

#align reads against reference w/ NGMLR
ngmlr -t 4 -r $reference -q $outfasta -o $outtemp.sam

#delete intermediates
rm $infasta.fai
rm $outbed
rm $outfasta

#convert sam to bam, sort
samtools view -S -b $outtemp.sam > $outtemp.bam
samtools sort $outtemp.bam -o $outtemp.sorted.bam

#variant calling w/ sniffles
```

```
35 sniffles -m $outtemp.sorted.bam -v $outtemp.vcf
36
37 #delete intermediate files
38 rm $outtemp.sam
39 rm $outtemp.bam
40 rm $outtemp.sorted.bam
```

Appendix 1: Full script of the variant-calling script to generate a VCF-formatted file from the query and reference assemblies. The query assembly is aligned against the reference genome by ngmlr and variant calling is done based on the alignment data with sniffles.

```
1 #!/usr/bin/env python3
2
3 import argparse
4
5 parser=argparse.ArgumentParser(description='Generate artificial
                                long reads from existing assembly
                                with sliding window approach')
6 parser.add_argument('--fi',help='Provide index of input fasta
                      file (generate w/ samtools faidx)')
7 parser.add_argument('--sz',type=int,help='Size of artifical long
                      reads [bp]')
8 parser.add_argument('--st',type=int,help='Step size')
9 parser.add_argument('--bed',help='Output file name')
10 args=parser.parse_args()
11
12 def _global(sz,st):
13     global size
14     global step
15     size=sz
16     step=st
17
18 def _splits(line):
19     columns=line.split()
20     return [columns[0],int(columns[1])]
21
22 def _slide(cont_name,cont_len,outhandle):
23     start=0
24     stop=size
25     while stop<cont_len:
```

```
26           outhandle.write(f'{cont_name}\t{start}\t{stop}\n')
27           start+=step
28           stop=start+size
29       stop=start+(cont_len-start)
30       outhandle.write(f'{cont_name}\t{start}\t{stop}\n')
31
32 _global(args.sz,args.st)
33 with open(args.bed,'w') as outhandle:
34       for line in open(args.fi,'r'):
35           _slide(*_splits(line),outhandle)
36 outhandle.close()
```

*Appendix 2: Full **makebed.py** script*