Fusion Gene Visualizer

Lynn Dotrang, Joanna Harris, Mary Friedman, Nana Anguah Dei, and Sam Freund

**Executive Summary**

Our group worked to design a system for visualizing aligned sequencing reads which contain fusion gene transcripts. Many fusion genes are known to be associated with specific diseases, and if detected, can allow for a more accurate diagnosis and even targeted therapy in some cases (Dupain, et al. 2017). Integrative Genomics Viewer (IGV) does not readily lend itself to this application, as it displays read alignments according to the order of the reference sequence being used, rather than splicing the reference to match the order of each sequencing read. Our group sought to address this issue by creating a viewer which arranges the references for fusion partner genes according to the order of the fusion gene sequencing read, allowing for a continuous view of the read and breakpoint identification. Our company is working in collaboration with Wolfgang Rumpf who suggested that such a tool would be helpful for analyzing sequence data related to pediatric cancers, many of which are known to be associated with the presence of fusion genes and/or proteins (Dupain, et al. 2017). Our company developed a program which reads into a .bam file and identifies fusion gene alignments and the reference sequence locations of their associated gene partners and has begun development of a visualization tool which displays each fusion gene alignment in a continuous fashion in regard to the read.

**Article Reviews**

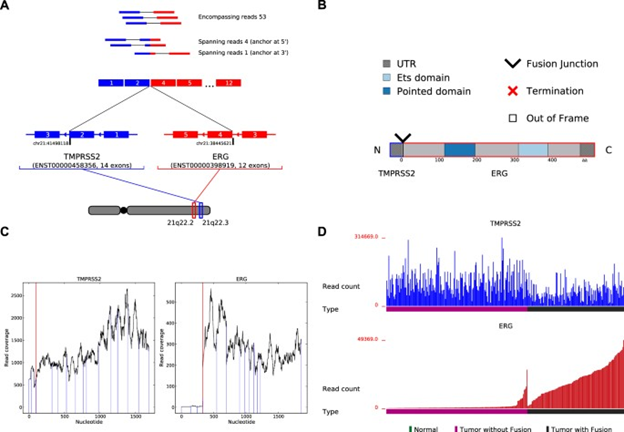
Finding the foundation and the direction for any project is important. An online search for articles, posts, or programs that had keywords “fusion gene visualizer” was performed. There were numerous results that appeared. Narrowing down the results to ones that would give an idea of the direction to move in was difficult. Through group meetings and discussions, our group decided the best choice of programming language was Python. There were three articles which were presented to the group. They were fusion gene visualizers called Clinker, INTEGRATE-Vis, and FGviewer. Other closely related fusion gene visualizers found were AGFusion and chimeraviz.

Clinker was the best of all the articles and results found. Clinker is a bioinformatics tool written in Python, R, and Bpipe that leverages the superTranscript method to visualize fusion genes. Clinker obtains interpretable visualizations of the RNA-seq data that lead to fusion calls. (Schmidt et al., 2018). SAMtools, which was a tool the group already needed in converting the Sequence Alignment/Map (SAM) files to Binary Alignment/Map (BAM) files. Other tools such as STAR and Gviz were mentioned and presented to the group if they were useful. Clinker used Python as one of the main languages with the inclusion of R and Bash. This article presented information for the use of Python in reading bam files and some of the tools used in providing greater insight of the view of fusion genes. The Clinker pipeline takes output from any fusion calling software, providing that it includes the hg19 or hg38 genomic coordinates of fusion gene breakpoints. From using the program, the downside was it just zoomed into the fusion gene but there was no opportunity to interact with the results, meaning users could not click or highlight specific breakpoints. It also only worked on 64-bit Linux or Mac OS X operating systems, not Windows.



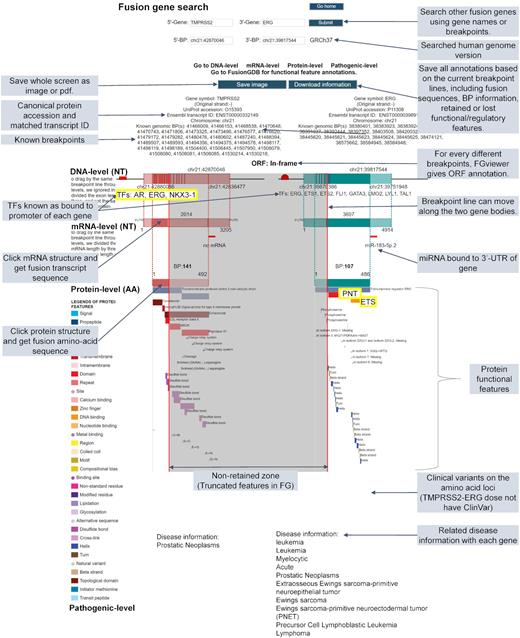
**(A)***KMT2A-MLLT3* fusion gene visualized in IGV after alignment to the human genome. **(B)** Clinker output of the *KMT2A-MLLT3* gene fusion, visualized in IGV and **(C)** the GViz visualization.

INTEGRATE-Vis is a gene visualization tool that generates comprehensive, highly customizable, publication-quality graphics focused on annotating each gene fusion at the transcript- and protein-level and assessing expression within an individual sample or across a patient cohort (Zhang et al., 2017). This tool functions using Python with other programs that were unknown such as CMake, GCC, Matplotlib, and gtfToGenePred. This program is great to visualize results from fusion genes, but for the group project this was complex and did not allow user interaction in which the group could use information from it to develop our project.



INTEGRATE-Vis output illustrated using the *TMPRSS2-ERG* gene fusion in prostate cancer. INTEGRATE-Vis outputs four visualizations including: (**A**) gene fusion transcript isoforms, (**B**) the predicted protein structure of the gene fusion, (**C**) RNA-Seq read coverage across each gene fusion partner to reveal changes in exon expression, (**D**) expression of each gene fusion partner across the TCGA PRAD cohort.

FGviewer is a tool for visualizing functional features of human fusion genes. FGviewer gets the input of fusion gene symbols, breakpoint information, or structural variants of the VCF (v4.3) file from Whole Genome Sequencing (WGS) data. For any combination of gene pairs/breakpoints to be involved in gene fusion, the users can search the functional aspect of fusion genes at the three bio-molecular levels (DNA-, mRNA-, and protein-levels) and one clinical level (Kim, 2020). FGviewer was aesthetically pleasing to look at, but was very complex and web-based yielding enormous data output, like a database retrieval tool. It used Python’s Tornado web server and MySQL database. One group member tested FGviewer and did not receive results until longer than 15 minutes. FGviewer, although very attractive and informative, was too complicated and complex for the group. It also did not allow for user interaction of the visual output. Reviewing these tools gave us an idea on the direction we needed our project to go. These reviews will hopefully aid the next group in their background research of gene visualizer tools.



Overview of the FGviewer result page of the TMPRSS2-ERG query search. FGviewer is composed of four levels including DNA-, mRNA-, protein- and pathogenic-levels. Details on individual functions are described with shaded text.

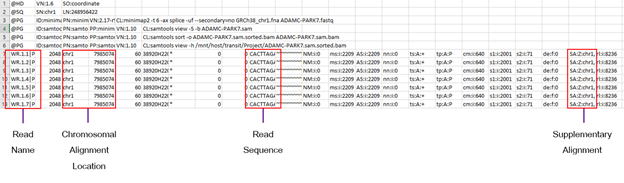
**Converting from SAM/BAM file types**

SAM files store sequence read alignments provided by various sequencing platforms against reference sequences and SAMtools allows for post-processing of these alignments as well as conversion into BAM format, a compressed version of a SAM file (Li et al., 2009). It was helpful to learn about the package and go through the steps of installing it and trying it out through Anaconda all with Unix commands. A quick guide on SAMtools usage was provided by Dave Tang on GitHub via their learning\_bam\_file repository where examples of usage for several of the SAMtools utilities were shown (Tang, 2020). Some key features of SAMtools are the sort tool, which allows for easy indexing of alignments, and the view command, which converts between SAM and BAM file types (Li et al, 2009).

**Parsing SAM/BAM files**

A primary task within this project is the parsing of pertinent information from Sequence Alignment/Map (SAM) or Binary Alignment/Map (BAM) files that will indicate fusion gene products. Before any visualizations can be formed, the information needs to be compiled, filtered, then delivered to the user in a comprehensible format. Within a SAM/BAM file, relevant reads with fusion genes will contain the Supplementary Alignment (SA) tag which signifies other canonical alignments in a chimeric alignment (jkbonfield, 2020). The goal is to parse these alignment files for query names, flags, reference sequence name , mapping position, and other relevant information each read contained within. Additionally, the SA tag will contain information for the corresponding alignment, so it needs to be parsed as well.

For this project, we were provided with three sample BAM files to use throughout the development process, one containing only fusion gene reads, one containing only non-fusion gene reads, and one containing a combination of both. We used the SAMtools view command to convert the sample BAM files into SAM files, which are easily viewable in notepad or excel. This allowed us to identify the location of fields which might be important for the fusion gene viewer, as shown below.



Python is a flexible language that all members of the group have experience with. When researching existing tools for handling SAM/BAM files, the group decided to see what kind of tools were written in Python to compliment the collective group skillset. PyBam was found as a viable option to retrieve information from SAM or BAM files. The PyBam repository, by user JohnLongiotto, includes explanations of what each of the parse codes will pull from the SAM/BAM file (JohnLongiotto, 2019). Within PyBam, the “Read” class handles the parsing based on which parse function is called for each object (alignment) contained within the entire alignment file.

for alignment in pybam.read('/my/data.bam'):

print alignment.sam\_seq

This block demonstrates a dynamic parser that will print the DNA sequence for each read in the alignment. Keywords in green represent items from the PyBam package.

Using this functionality provided by PyBam, relevant values, "Name", "Chr", "Pos", "SuppAl", "ReadSeq", "ReadLen", and "Cigar" were stored as tuples for each read in the alignment using the namedTuple function (Python collections library).

Read = namedtuple("Read",["Name","Chr","Pos","SuppAl","ReadSeq","ReadLen","Cigar","SA\_Chr", "SA\_Pos", "SA\_MAPQ"])

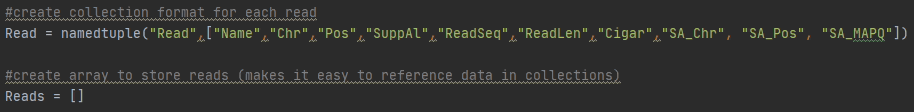
The Supplementary alignment tag requires additional parsing because the tag’s string contains information regarding the corresponding chimeric alignment. In this case, it is the relevant information needed to indicate the location of the fusion gene in the format below:

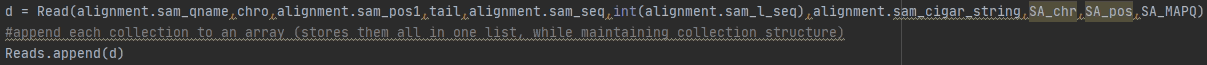
(rname ,pos ,strand ,CIGAR ,mapQ ,NM ;)

Because PyBam parses this line as a string of tags, additional parsing is required to retrieve only the pertinent information following the “SA” tag. To perform this parse, “SA” is searched via regular expression. Positionally within the list of tags, the SA tag is not in a fixed position based on the presence or absence of other tags. So, a parsing code should not be based on fixed list indices; however, within the list of items within the SA tag as shown above is fixed and can be parsed easily as list indices. From the SA tag, “rname”, “pos”, and “mapQ” are stored in the tuple.

While parsing through each read, filtering is concurrently performed. The first measure for filtration is if the string tag contains “SA” at all. If not, the read does not need to be stored. Additionally, the Mapping Quality (MAPQ), a measure of quality of alignment, is assessed so that only alignments with a MAPQ over 50 are accepted to be stored. Subsequently, SA tag MAPQ values, or the value of the corresponding alignment, must also be over 50 to be stored as a read.

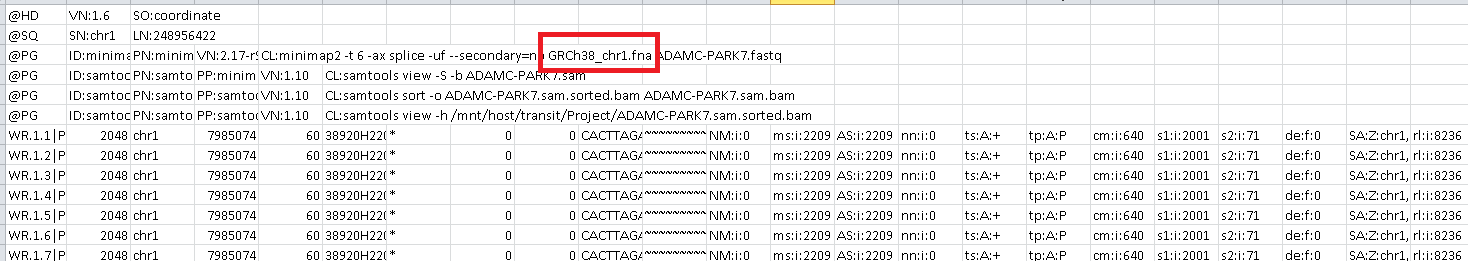
After looping through each read alignment in a .bam file, the end product of this initial parsing is an array containing named tuples for each read which passed the criteria. (see screenshots below)



/\*

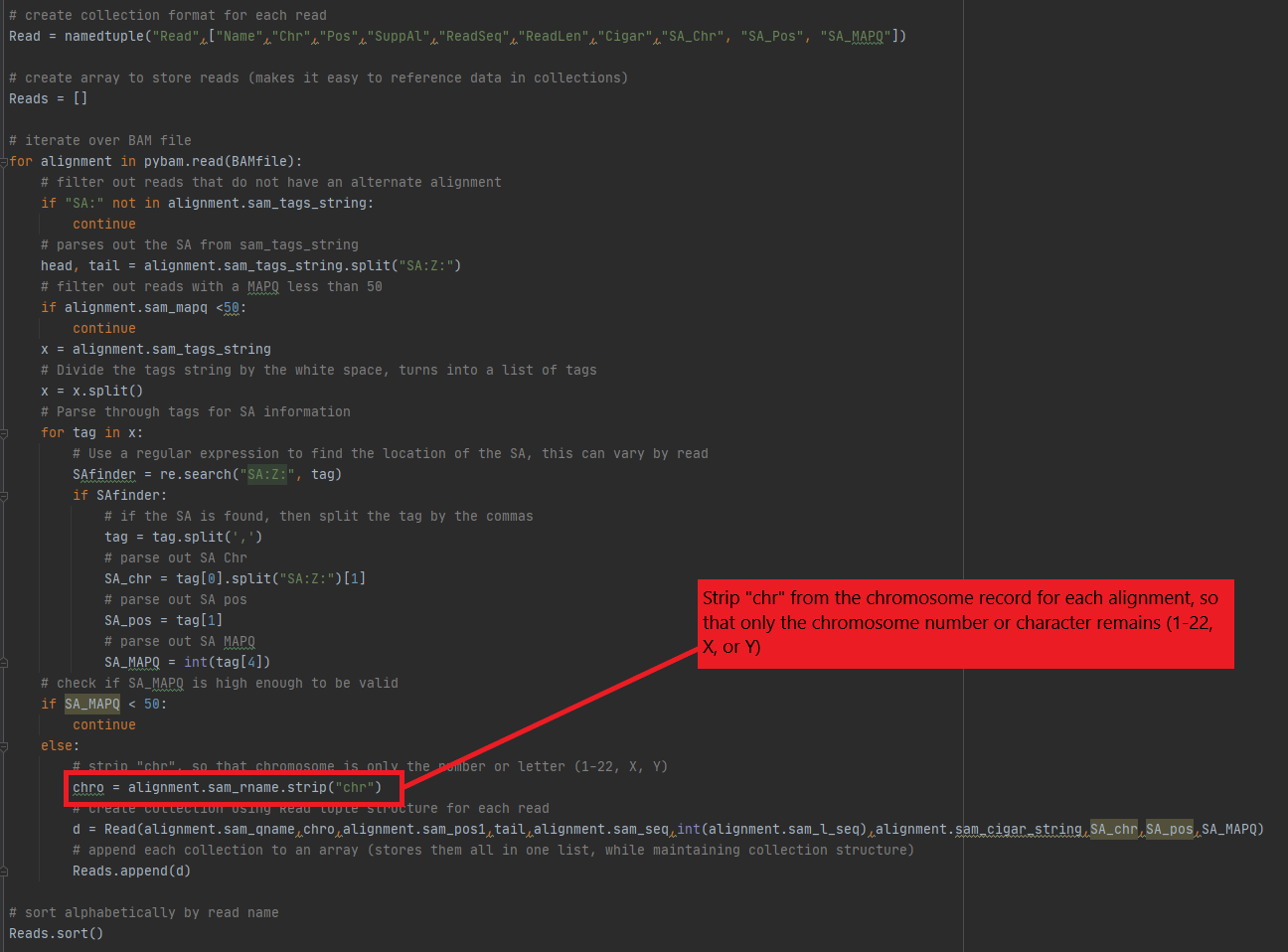
The data elements stored for each read can be called with terms designated when the named tuple structure was created, such as “Name” and “ReadSeq.

Another important aspect to consider is the display of the reference sequence(s) below each sequencing read. The reference sequence is not stored in the .bam, but must be pulled from a separate file. A .bam will specify which reference sequence was used for alignment when the .bam was created, as shown in the example below.



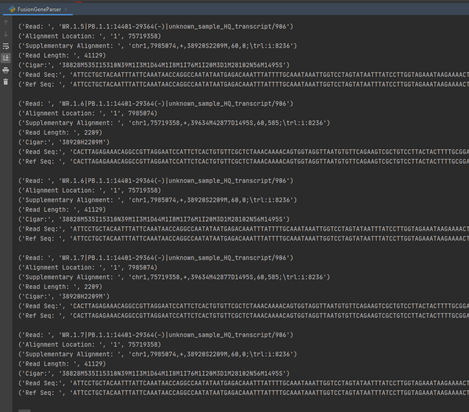
The fusion partners for the gene in this example file (ACADM-PARK7) are both located on chromosome 1, so it makes sense that a chromosome 1 reference was used for alignment. However, not all fusion genes have partners located on the same chromosome, so it made sense that our program should have the ability to pull reference sequences from any chromosome as needed.

To start, GRCh38 chromosome reference FASTA files were downloaded from NCBI for all 24 chromosomes. A code was written in python to reformat these files, such that sequences could be pulled based on the chromosome location listed for each read alignment. This reformatting process involved renaming the file as the chromosome character only (1-22, X, or Y), and removing the FASTA header and all newline characters. With all 24 modified chromosome references stored in the same folder, the .bam parsing code can then use the chromosome number in a read alignment to identify the correct reference file. The chromosomal alignment location can then be used to point to the start location of the read alignment within the reference file, and the read length can be used to extract the correct reference sequence length. (see details below)

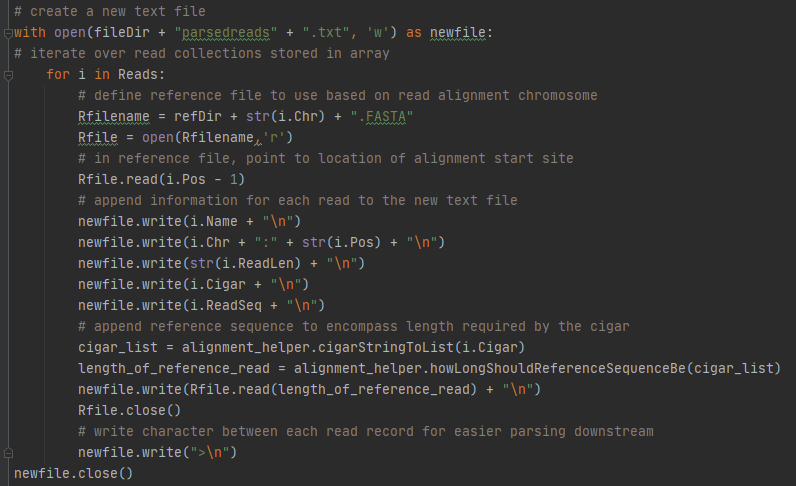


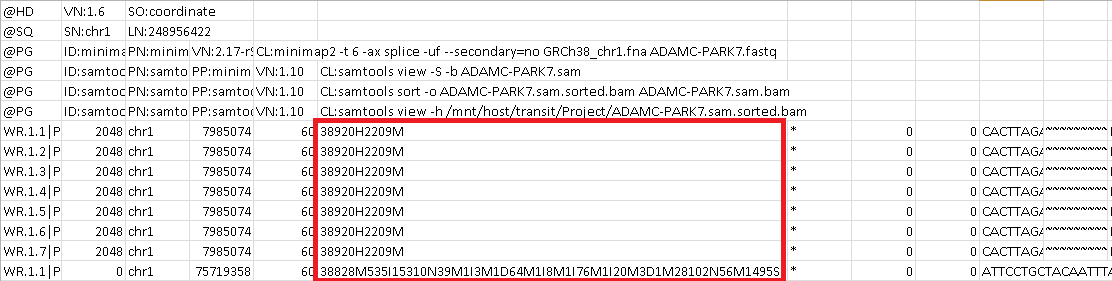


This code results in an output that looks like this:



While this read and reference data output indicates that data is being filtered and stored correctly for each read, it does not bridge the gap between .bam processing and our viewer interface. Because the GUI is still currently in the development phases, we opted to bridge this gap by having the .bam parser write a text file with pertinent information for each read, which can then be used by the GUI code.



One field included in this text file is the cigar string, a series of numbers and characters which describes the alignment of the read to the reference file. (see example below) 

The cigar is essential when attempting to visualize aligned read and reference sequences, as it accounts for events such as deletions and insertions. The shorthand for deciphering these strings is as follows:

M = match (does not necessarily indicate matching base pairs - just matching alignment)

D = deletion (gap in read sequence)

I = insertion (gap in the reference sequence)

S = soft clipping

H = hard clipping

For example, a cigar such as “20M2I” would indicate an alignment with 20 matches followed by 2 insertions. Because the alignment between a read and reference will rarely be 1 to 1, in regard to base pairs, the cigar must be considered when attempting to determine the proper length of reference sequence needed for alignment. (see example below)

**Read: ATAGG - - - CATGGTA**

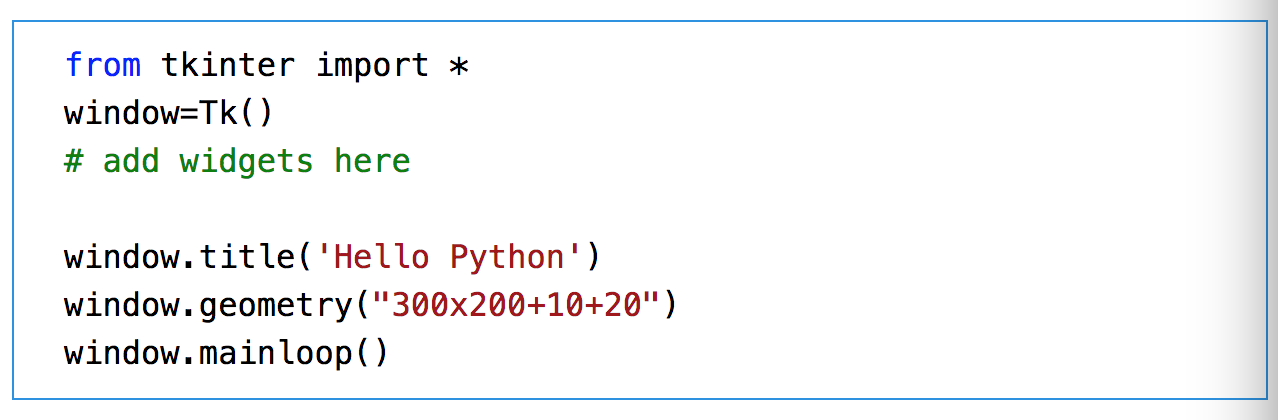
**Ref: ATACGGCTAGGCC**

Because there is a deletion in the read sequence, the overall length of the read is decreased from 15bp to 13bp. If using the static read length, this would cause a 13bp sequence to be pulled from the reference file. However, when properly aligned, the reference sequence needs to be 15bp to cover the length of the read sequence. The function howLongShouldReferenceSequenceBe(), addresses this problem, since it is based on the CIGAR string which thereby determines out how many nucleotides the user needs to pull from the reference sequence. This function is present in the alignment\_helper.py and is explained in the development portion of this paper. The FusionGeneParser\_final.py calls this function and an example is shown below.



**GUI Construction**

Tkinter is a GUI (graphical user interface) widget set for Python. Tkinter is an open source GUI library that is known for its simplicity and flexibility. It comes pre-installed in Python 3 and is a great starting point for beginner and intermediate programming levels interested in writing code for GUIs. Tkinter in Python comes with a lot of good widgets which are standard GUI elements like different kinds of buttons and menus. The following code demonstrates the steps in creating a user interface using Tkinter:



First, the user must import the Tkinter module and set up the application object by calling the Tk() function. This creates a top level window (root) having a frame with a title bar, control box with the minimize and close buttons and a client area to hold other widgets (Shipman, 2013). The geometry() method defines the width, height, and coordinates of the top left corner of the frame. You can set the following properties to customize widgets: text, bg (background color), fg (foreground color), font, image, and command.

We investigated using Tkinter to create a GUI for our gene visualizer tool. The goal of which is to import a .bam file and be able to view fusion genes against the reference sequence. Code for our GUI can be found on Github under the following repository link: <https://github.com/srfreund/BIOT670/blob/master/Fusion%20Gene%20GUI>.

Our GUI for the gene visualizer includes label and button widgets. The button widget allows for import of the .bam file from the user (Figure 1).

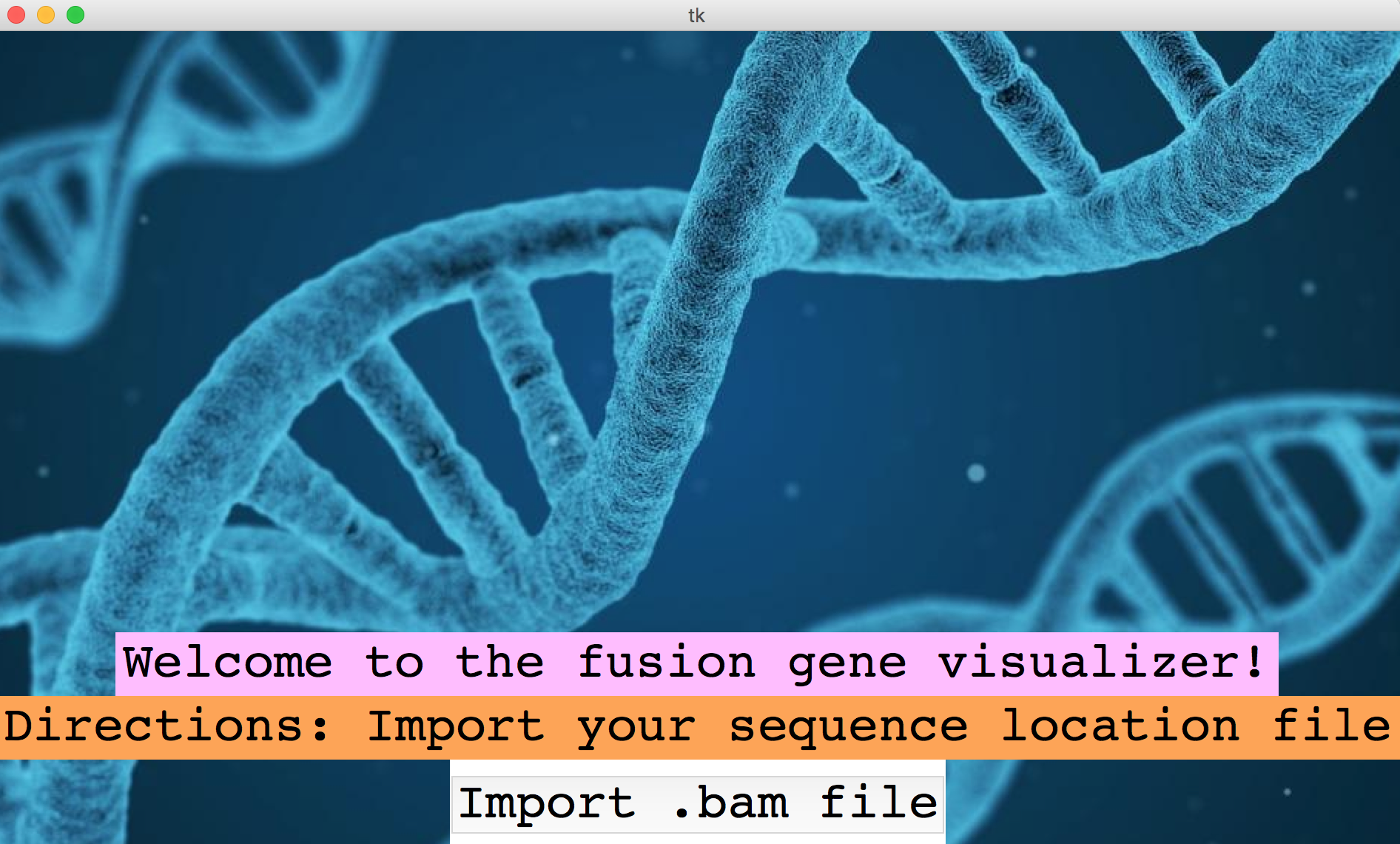
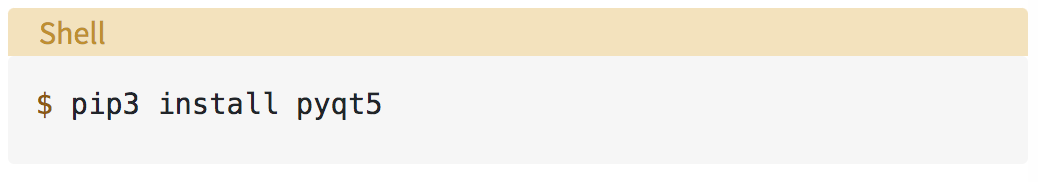


Figure 1. Fusion Gene Visualizer created with Tkinter in Python

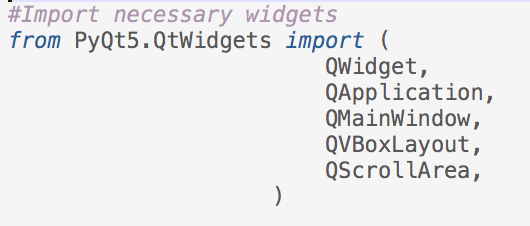
**PyQt**

PyQt is a Python linkage of the GUI toolkit Qt implemented as a Python plugin. Qt is a set of C++ libraries and development tools that include platform-independent abstractions for BUI, as well as networking, threads, and regular expressions. The combination of Python and QT (PyQt) makes it possible to develop applications on any supported platform and run them unchanged on all the supported platforms- for instance all modern versions of Windows, Linux, Mac OS X, and most Unix-based Systems (Summerfield, 2007). PyQt is developed by RiverBank Computing Ltd and is available in two editions: PyQt4 and PyQt5. PyQt is a free software which implements over 440 classes and over 6,000 functions and methods including GUI widgets, classes for accessing SQL databases, an XML parser, and more. If using Python 3.5 or later, one can install PyQt5 from PyPI by running the following command:



This will install PyQt5 in the base system and one can start using the library immediately after the installation finishes.

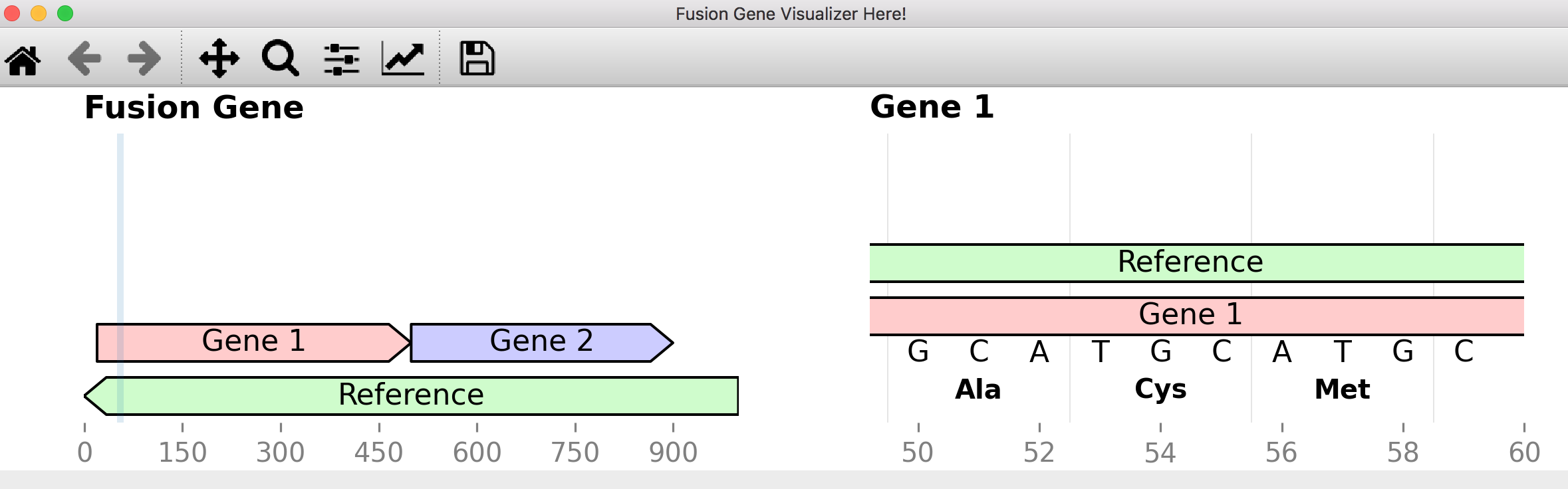
We decided to use PyQt5 to aid in developing the gene visualizer. In order to do this, we first had to install PyQt and the necessary widgets and used the following code to do so:

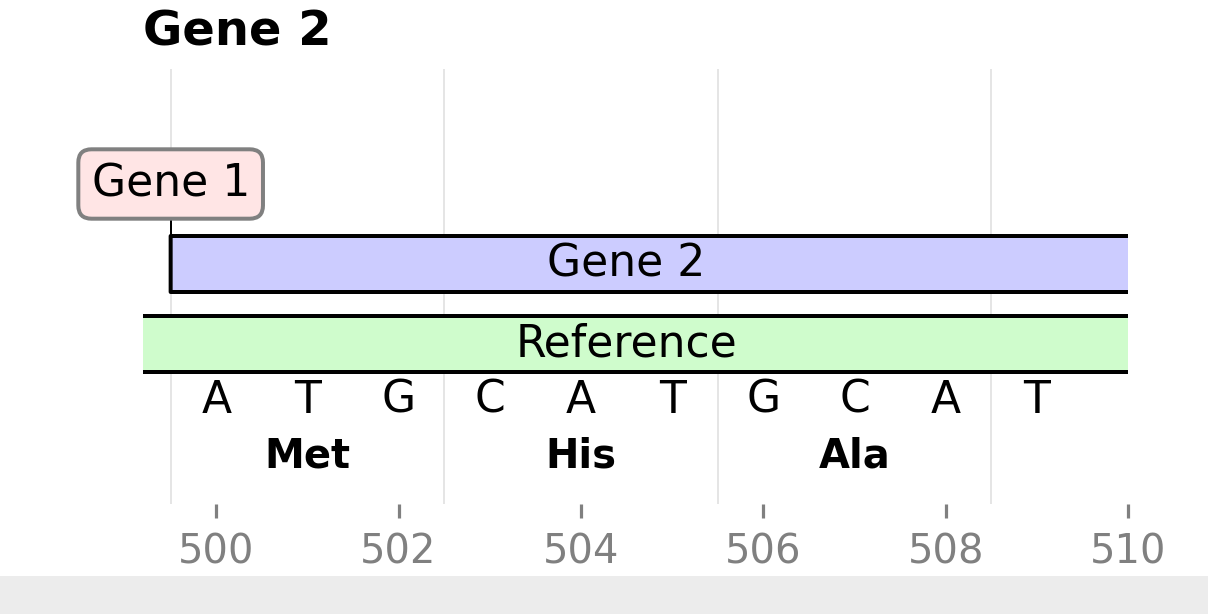


The QApplication object deals with common command line arguments. Our GUI is based on Qwidget which is the base class of all user interface objects in PyQt.

**DNA Features Viewer**

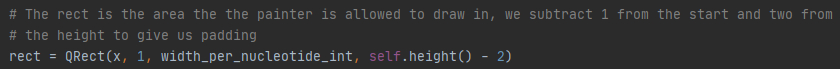
DNA Features Viewer is a Python library to visualize DNA features from a variety of files such as GenBank, Gff, or BioPython SeqRecords. The DNA Features Viewer creates simple and clear plots for sequences with many features and long labels. The library works well with Matplotlib and Biopython and the plots can be exported in different formats such as PNG, JPEG, SVG, and PDF. This would be useful for our group’s goal of creating a tool that allows researchers to create nice fusion gene images for report generation and article figures. In order to install this tool, we first had to type in the terminal “pip install dna\_features\_viewer”. We were able to create a couple of basic plots that could be applied to our fusion gene visualizer. We also used PyQt to create a separate window for the DNA features viewer that enables the user to scroll between plots. For future work on this product, the next group could work to connect the DNA features viewer to the parser tool. It is important to note that DNA Features Viewer provides static images that do not have dynamic zoom capabilities.





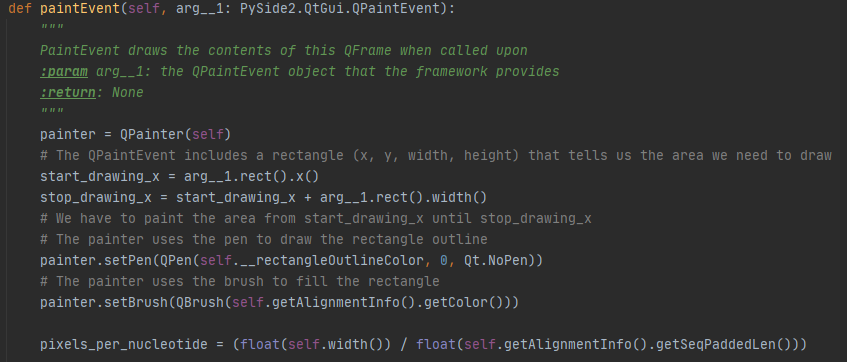
**PySide2**

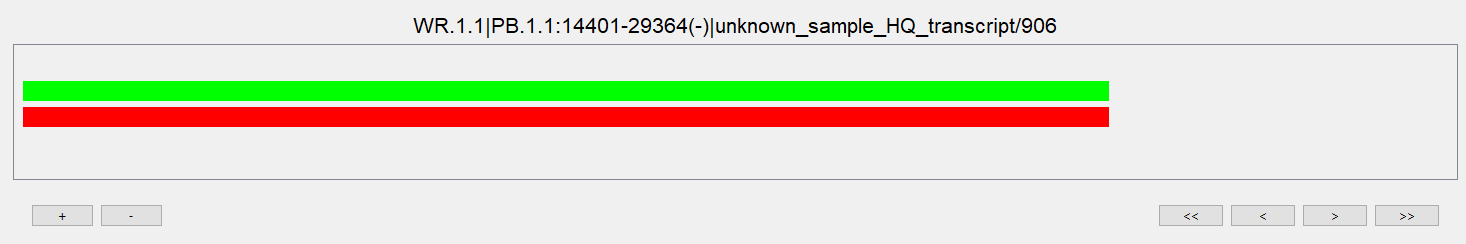
Python has been a keystone in interdisciplinary research, specifically crossing the bridge between computer science and biology. However, one development issue with Python is that it does not have strong, built in functions for GUI building. PySide2 (V5.15.1) hopes to bridge that gap, as it is the official Python module for the Qt for Python project (PySide2). To install PySide2, use the pip installer, pip install PySide2 in the command line. It is important to note that PySide2 V5.12+ uses a C++ parser that is based on the Clang library (PySide2).

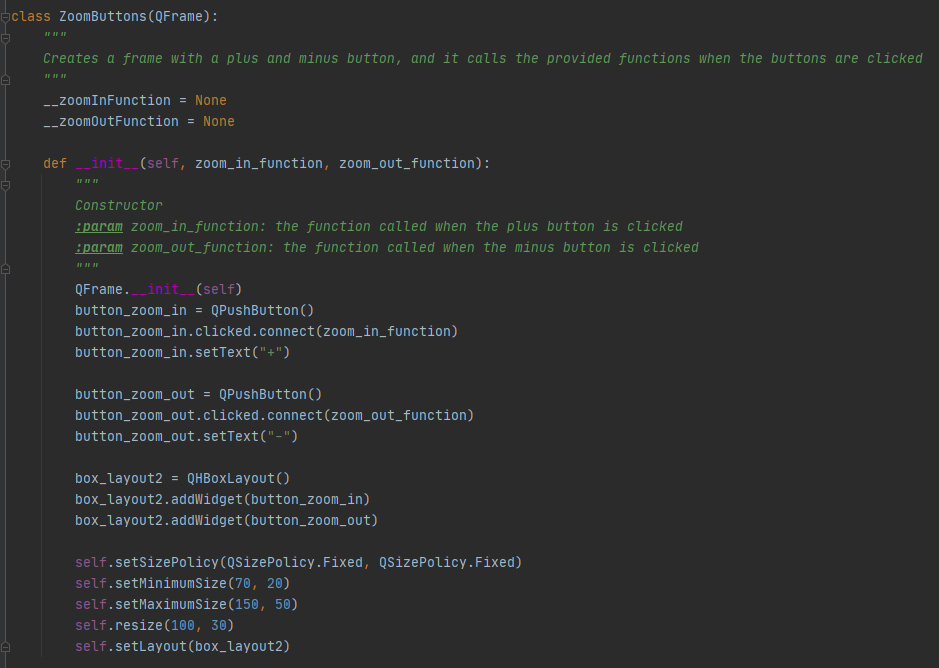
PySide2 is implemented heavily in the alignment\_gui.py and imports QtCore-QRect, QtGui-QPainter, QBrush, QPen, QFont, QT, and QtWidgets- QMainWindow, QFrame, QVBoxLayout, QHBoxLayout, QSizePolicy, QPushButton, QScrollArea, and QLabel. QRect allows the user to draw in a rectangle area with painter, and an example of its use is shown below.

It is important to note that x is the sequence index multiplied by the width per nucleotide, which helps provide the outline for the rectangle that will be used to represent the reference and read sequences.

QPainter is used to perform low-level painting on widgets, QBrush is used to define the fill patterns of shapes (in our case rectangles) drawn by QPainter, QPen is used to define how the QPainter needs to draw lines and outlines of shapes (Qt GUI C++ Classes). QPainter, QBrush, and QPen work together in alignment\_gui.py to visually display the nucleotide sequences, and an example is shown below.



 The green rectangle is the visual representation of the reference sequence and the red rectangle is the visual representation of the read sequence. The class QT is used to call different class attributes like AlignVCenter, AlignTop, and AlignBottom which helps the user choose their alignment preference of the display (Qt GUI C++ Classes).

QtWidgets module lets the user call classes like QMainWindow, QFrame, QVBoxLayout, QHBoxLayout, QSizePolicy, QPushButton, QScrollArea, and QLabel in alignment\_gui.py. The QMainWindow is the main application window, the QFrame is a base class of widgets that has a frame, QVBoxLayout will line up the widgets vertically, whereas the QHBoxLayout will lign up the widgets horizontally (Qt GUI C++ Classes). QSizePolicy is what allows the GUI window to be resized, since this class will describe the horizontal and vertical resizing policy (Qt GUI C++ Classes). QPushButton is a command button, and an example of this is seen with the zoom in button, where when clicked, will call the zoom in function and zoom into the GUI. 

QScrollArea allows the scrolling view onto another widget, and QLabel provides the text or image display (Qt GUI C++ Classes). A screenshot of the final output of alignment\_gui.py is shown below.

****

It is important to note that the GUI also takes into consideration wheel events. This allows the user to zoom into and out of individual alignment frames, as well as scroll through the global frame by using the mouse wheel instead of the zoom buttons or the scroll bar. Shown below is a detailed GUI window, with multiple alignments.



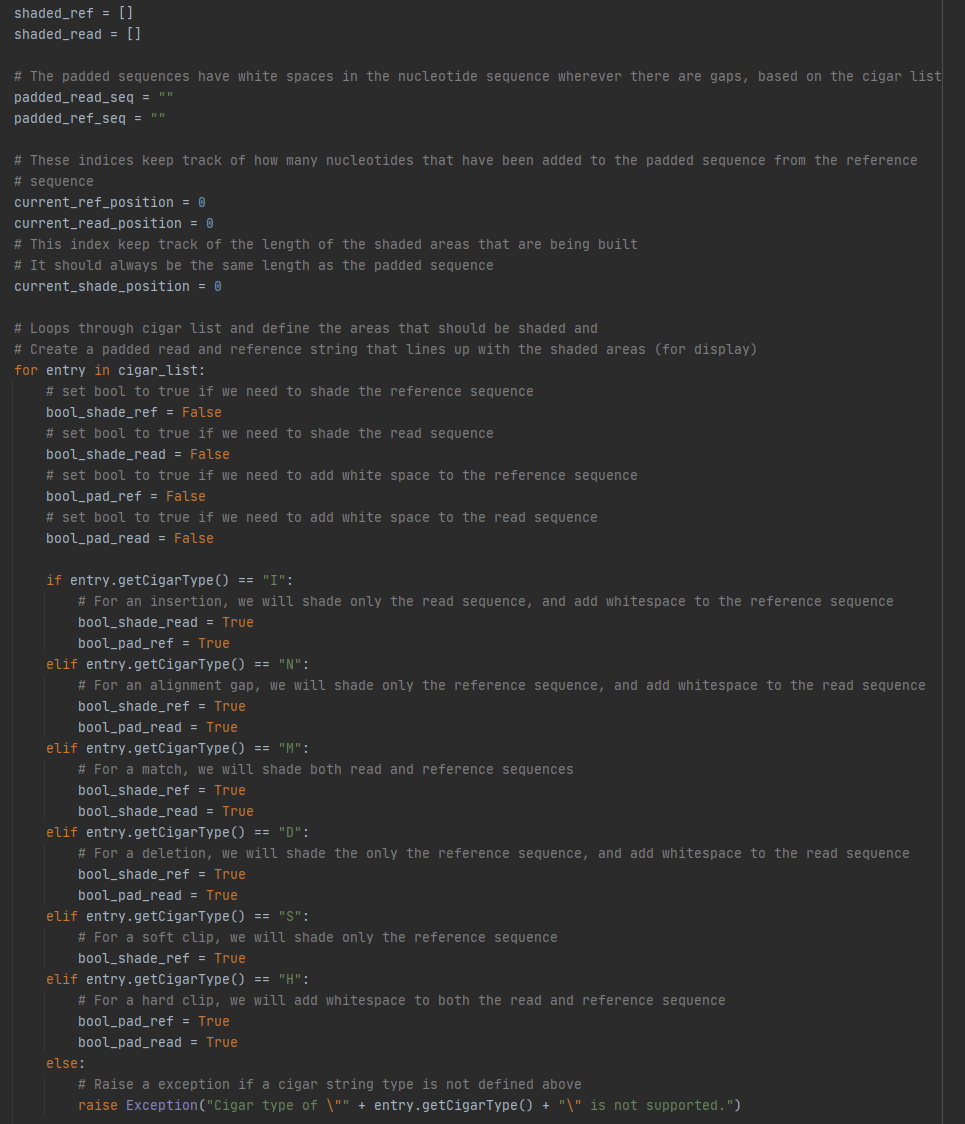
**Development**

GUI development implements the use of three scripts, alignment\_tester, alignment\_helper.py, and alignment\_gui.py. Alignment\_tester.py is used to interface with the parser code (FusionGeneParser\_final.py), and handles parsed data from the raesedreads.txt with ParsedRead File(), including name, read start position, reference start position, length of read sequence, cigar string read sequence, reference sequence and returns a list of alignment information to be handled later.The ParsedReadFile() function is shown below. 

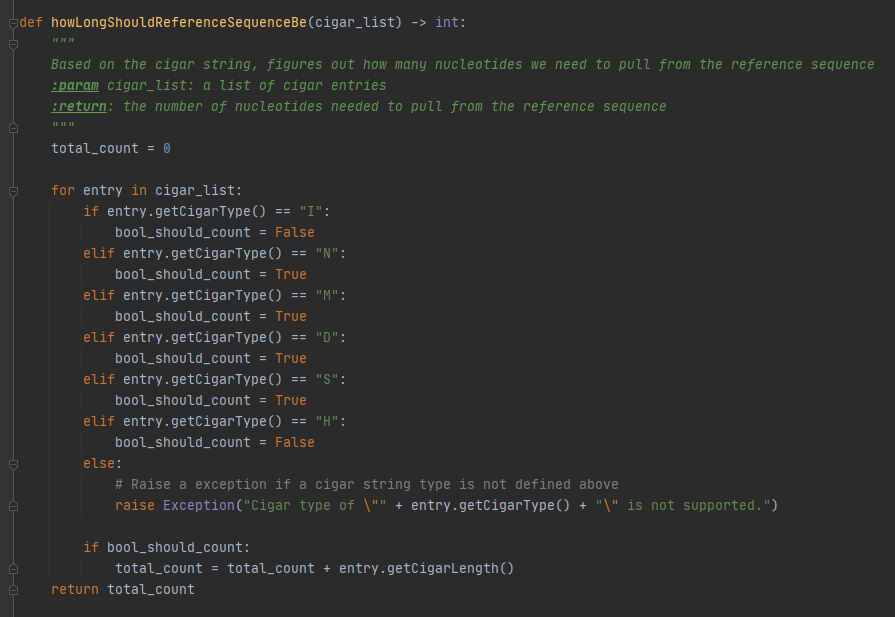
Lastly, alignment\_tester.py will display the alignment results from a file path provided as an argument inorder to create an instance of GraphTester which creates the main window that displays multiple alignments.

Alignment\_helper.py contains classes and definitions to help with non-graphical processing and logic. Inorder for the GUI to display accurate alignment information the parsed data needed to be processed so it could take into consideration common modifications in the nucleotide read such as insertions, deletions, soft and hard clips, not just the match case. The CIGAR is used to give the alignment information. Class CigarEntry() is a struct that provides the length and cigar type, for example 16 M, which is 16 matches. The function cigarStringToList, converts the cigar string into a list of class CigarEntry. This function uses re.findall and the regular expression ‘\\d+[A-Z]’ to break the cigar into its individual strings, and then stores the results of the cigar parse into the list my\_cigar as struct objects. The function trimHardClips() removes the hard clipped CIGAR entries from the front and back of the list, since the .bam file will not provide the nucleotides that were omitted due to the hard clip in the read sequence. The class ShadedAreasInfo is a struct that provides shaded areas (for drawing boxes) and padded sequences (for displaying next to boxes). This class is needed so that reference sequence and read sequence can be visualized with the alignment according to the CIGAR.

The function cigarListToShadedAreas() is based on the cigar list, and defines the areas that should be shaded and creates a padded read and reference string that lines up with the shaded areas for the display and graphical alignment. The three parameters this function requires is the read sequence, the reference sequence, and the cigar list. This function returns a ShadedAreasInfo that contains four functions. GetShadedref() which is the area that would be shaded on the reference sequence([0] the star position of the shaded area, and [1] the length of the shaded area (end pos = start + length)). GetShadedRead() is the area that would be shaded on the read sequence. GetPaddedRefSeq is the nucleotide sequence padded with white space where there should be gaps based on the cigar\_list. GetPaddedReadSeq() is the nucleotide sequence padded with white space where there should be gaps based on cigar\_list. This function will handle cigars as followed. A match (M) is the exact match of x positions, the reference bar will be shaded, the reference sequence will be present, the read bar will be shaded, and the read sequence will be present. An alignment gap (N) is the next x positions on the reference sequence do not match, the reference bar will be shaded, the reference sequence will be present, the read bar will not be shaded, and the read sequence will not be present. A deletion (D) is where the next x positions on the reference sequence do not match, the reference bar will be shaded, the reference sequence will be present, the read bar will not be shaded, and the read sequence will not be present. An insertion is where the next x positions on the read sequence do not match, the reference bar will not be shaded, the reference sequence will not be present, the read bar will be shaded, and the read sequence will be present. A soft clip (S) has nucleotides present in the reference sequence, but are not present in the read sequence, the reference bar will be shaded, the reference sequence will be present, the read bar will not be shaded, but the read sequence will be present since it is a soft clip. A hard clip (H) is not present in either the read or reference sequence, so the reference bar will not be shaded, the reference sequence will not be present, the read bar will not be shaded, and the read sequence will not be present. An example of how the alignment\_helper.py will handle CIGAR results logically is shown below.



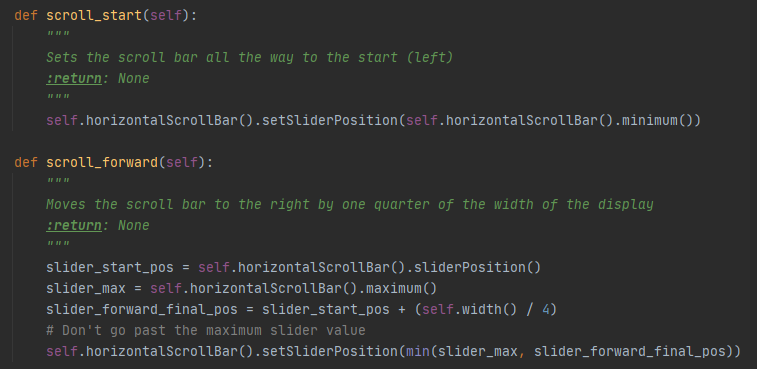
The function howLongShouldReferenceSequenceBe() is based on the CIGAR string and figures out how many nucleotides we need to pull from the reference sequence. It takes the cigar\_list and returns the number of nucleotides needed to pull from the reference sequence. This function counts the number of nucleotides involved in an alignment gap, match event, deletion event, and soft clip. This function will not take into consideration nucleotides on the reference sequence that are insertions or hard clips. The code for this function is shown below.



The class HalfAlignmetnInfo contains enough information to draw one nucleotide sequence, and has the functions getShadedAreas(), getSeqPadded, getSeq, getStartPos, getColor, getSeqLen(), and getSeqPaddedLen(). Whereas the class AlignmentInfo contains enough information to show the comparison of two nucleotide sequences, with the functions getName(), getCigarList(), getRefInfo(), getRead.

Alignment\_gui.py contains classes and definitions for components of GUI building. The class NucleotideFrame() is used to show the nucleotide sequence in the QFrame, it will automatically adjust the size of the sequence based on the QFrame. The class BarFrame() is a QFrame that draws a single nucleotide alignment bar. Two BarFrame() classes are typically used together to display both the reference and the read sequence. The class BarFrameStacker() will stack four items vertically. This is used to display the two nucleotide frames and the two bar frames.

The class BarFrameStackerScrollArea() contains a bar frame stacker and handles all zooming functionality. The functions for zoom options are getCurrentZoomLevel() which provides the current zoom level, zoomIn() which zooms in to make the displayed nucleotide comparison larger. This function ensures that the nucleotide at the location x\_position\_to\_lock remains at that location after zooming. The zoomOut() function zooms out to make the displayed nucleotide comparison smaller and will ensure that the nucleotide at the location x\_position\_to\_lock. It is important to note that the parameter x\_postion\_to\_lock is a location in pixels on the display that should remain fixed when zooming into and out of the sequence frame. The function scroll\_start() will set the scroll bar all the way to the start of the sequence (left), and the function scroll\_end will set the scroll bar all the way to the end of the sequence (right). The function scroll\_forward() will move the scroll bar to the right by one quarter of the width of the display where the function scroll\_backward() will move the scroll bar to the left by one quarter of the display. The functions scroll\_start() and scroll\_forward() in the class BarFrameStackerScrollArea() are shown below.



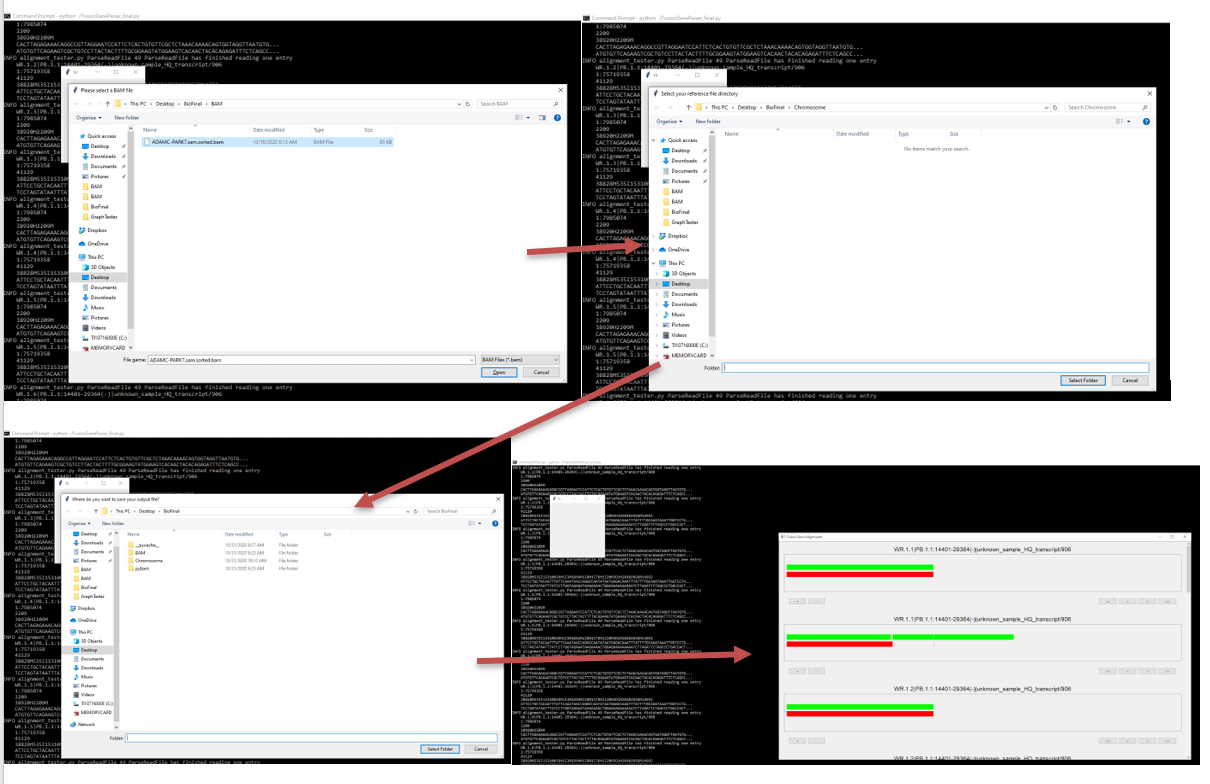
It is also important to note the function \_\_fixPositionAndInitiateResize() as this function adjusts the width of the BarFrameStacker to match the current zoom level, and the user must adjust the objects zoom level before calling this function inorder for it to have an effect. The last function that is built into this class is wheelEvent(), which handles the mouse wheel event and calls the bar frame stacker zoom event. These functions give the user the option to use the zoom buttons, scroll bar, or the mouse wheel to interact with the GUI.

The class ZoomButtons() creates a frame with a plus and minus button, and it calls the provided functions (zoom\_in\_function, zoom\_out\_function) when the buttons are clicked. The class ScrollButtons() creates a frame with start, backward forward, and end buttons, and calls the provided functions when they are clicked (start\_fucntion, backward\_function, forward\_function, and end\_function). An example of these buttons are shown below.



The class CompleteComparisonFrame() creates a QFrame that shows an alignment based on the information provided to the constructor. It is important to note that this class combines the BarFrameStackerScrollArea, the Zoom Buttons, the ScrollButtons, and a title label. The class CompleteComparisonFrameStacker() creates a QFrame that holds multiple CompleteComparisonFrames in a vertical box layout. The class CompleteComparisonFrameStackerScrollArea() holds a CompleteComparisonFrameStacker in a QScrollArea. Lastly the GraphTester() creates a window that displays multiple alignments.

GUI building in python can be a bit daunting at first, however with PySide2 it is easy to build, stack, and nest the different GUI components. Shown below are the four displays users will see after running FusionGeneParser\_final.py in command prompt. Be sure to install Python3.9, PySide2, PyQT5 before running as these are the system requirements.



**Future Considerations**

**Gene Annotation**

The current BAM parsing code outputs a read and reference sequence in simple string format with no additional information about the chromosomal location. This is useful for simple display in a viewer, as the sequences simply have to be aligned properly based on the cigar string and require no further processing. However, a meaningful association must be identified for the alignment to be useful in clinical applications. Though we have not begun to approach this challenge as of yet, a Python package which could be used is PyEnsembl (<https://pypi.org/project/pyensembl/0.5.4/>), which includes a variety of functions related to genes, transcripts, and exons. (see examples below)

* **genes\_at\_locus(contig, position, end=None, strand=None)** : returns list of Gene objects overlapping a particular position on a contig, optionally extend into a range with the end parameter and restrict to forward or backward strand by passing strand='+' or strand='-'.
* **gene\_names\_at\_locus(contig, position, end=None, strand=None)** : names of genes overlapping with the given locus (returns a list to account for overlapping genes)

**Breakpoints and Read Depth**

Because our GUI is still currently under development, it currently only displays a single read and reference sequence per window pane. This was a good starting point for tinkering with different display settings, such as zooming and scrolling. However, BAM files tend to contain hundreds, if not thousands of reads, so displaying each one individually is less than ideal. It is common for multiple breakpoints to be detected when a fusion gene is present, so it would be ideal to group the read alignments in a BAM by breakpoint. From there, each breakpoint and read depth could be displayed in its own pane, with vertical scrolling enabled for viewing individual reads that have variations.

**Putting It All Together**

Our approach to this project involved the two challenges of BAM processing and GUI development proceeding in a parallel manner, with both being worked on simultaneously from the very beginning by two separate, smaller teams of people. This allowed us to achieve the maximal amount of progress on the project as a whole given the limited time constraint. A future challenge for groups seeking to continue this project from where we left off will be linking the two parts together. Currently, the BAM parsing code outputs a text file with the following information for each read alignment:

Read Name

Chromosome: Chromosome Position

Read Length

Cigar String

Read Sequence

Reference Sequence

These specific pieces of data were requested by the GUI development team as starting sample data, but other pieces of data are available from the BAM parsing code should they be needed for future testing iterations, including:

Supplemental Alignment Chromosome

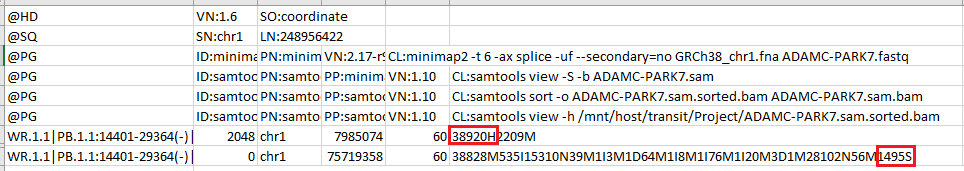
Supplemental Alignment Chromosome Position

Supplemental Alignment MAPQ score

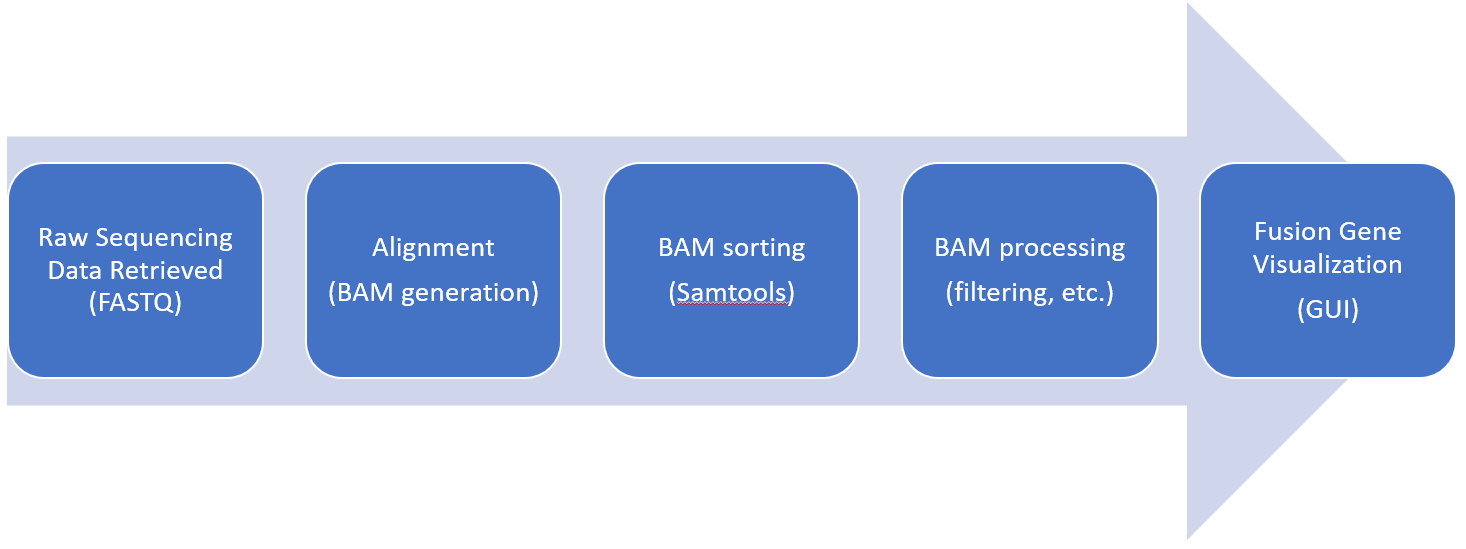
Once the GUI code is finalized, the two codes can be linked in a more integrated fashion using variables rather than an external text file.

**Project Scope Expansion**

The parsing code works on sorted .bam files, so it is assumed that the user will provide a file in this format. To circumvent this restraint, it would be helpful to have SAMtools integrated within the pipeline. From there, once a user selects a file, it can be sorted as needed. Then, the file will be parsed for reads that contain fusions and fed into the GUI. It is certainly possible that the GUI needs to be flexible because not all files will contain the same number of reads to display and the finished pipeline should account for this.

An issue we encountered with the sample BAM files provided for this project is that the read sequences for each alignment are trimmed, (hard/soft clipping) which results in the absence of a full fusion transcript read. (see example below)

The two read alignments shown are for the same sequence read, one being for the primary alignment and the other being for the supplementary alignment. We can see from the cigar that the first alignment starts with a hard clipping of 38,920bp, meaning that this part of the read will not be displayed in the read sequence for this record. The cigar for the second record indicates that the alignment ends with a 1495bp short clipping, meaning that this part of the read will not be displayed. These types of clippings are a function of the aligner settings, and would have to be considered if this fusion gene viewer is to be used with BAMs which were generated using different aligners and/or settings. The most ideal solution would be to include a standardized alignment as part of the finished product, which would also for standardization of the reference file used for alignment. Ideally, a raw data file (FASTQ, FAST5, etc.) could be imported, aligned (BAM generation), sorted, processed, and displayed in the fusion gene viewer seamlessly.



**References**

User jkbonfield. (2020, May). *Sequence Alignment/Map Optional Fields Specification.* SAMtools repository. <https://github.com/samtools/hts-specs/blob/master/SAMtags.pdf>

User JohnLonginotto. (2019, February). PyBam. Github Repository. <https://github.com/JohnLonginotto/pybam>

Kim, P., Yiya, K., Zhou, X. (2020). FGviewer: an online visualization tool for functional features of human fusion genes, *Nucleic Acids Research*, 48( W1), W313 - W320. <https://doi.org/10.1093/nar/gkaa364>

Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., Durbin, R., & 1000 Genome Project Data Processing Subgroup (2009). The Sequence Alignment/Map format and SAMtools. *Bioinformatics (Oxford, England)*, *25*(16), 2078–2079. <https://doi.org/10.1093/bioinformatics/btp352>

PySide2. (n.d.). Retrieved November 01, 2020, from https://pypi.org/project/PySide2/

Qt GUI C++ Classes: Qt GUI 5.15.1. (n.d.). Retrieved November 01, 2020, from

https://doc.qt.io/qt-5/qtgui-module.html

Shipman, J. W. (2013). Tkinter 8.4 reference: a GUI for Python. *New Mexico Tech Computer Center*. Retrieved from: <http://alexandre.benoit.83.free.fr/archives/1415/adumas/ISN/tkinter.pdf>

Schmidt, B. M., Davidson, N. M., Hawkins, A.D.K., Bartolo, R., Majewski, I. J., Ekert, P. G., Oshlack, A. (2018). Clinker: visualizing fusion genes detected in RNA-seq data, *GigaScience*, 7(7), giy079. <https://doi.org/10.1093/gigascience/giy079>

Summerfield, M. (2007). *Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming (paperback)*. Pearson Education.

Tang, D. (2020). *Learning\_BAM\_file*. Github Repository. https://github.com/davetang/learning\_bam\_file

Zhang, J., Gao, T. & Maher, C. A. (2017). INTEGRATE-Vis: a tool for comprehensive gene fusion visualization. *Sci Rep* **7,** 17808.<https://doi.org/10.1038/s41598-017-18257-2>